Johannes Sebastian Grohmann

Model Learning for Performance Prediction of Cloud-native Microservice Applications

Dissertation, Julius-Maximilians-Universität Würzburg
Fakultät für Mathematik und Informatik, 2022

Gutachter: Prof. Dr. Samuel Kounev, Julius-Maximilians-Universität Würzburg, Germany
Prof. Dr. Dr. Stefan Wesner, Universtität Ulm, Germany
Prof. Dr. Weiyi Shang, Concordia University, Canada

Datum der mündlichen Prüfung: 16.03.2022

Abstract

One consequence of the recent coronavirus pandemic is increased demand and use of online services around the globe. At the same time, performance requirements for modern technologies are becoming more stringent as users become accustomed to higher standards. These increased performance and availability requirements, coupled with the unpredictable usage growth, are driving an increasing proportion of applications to run on public cloud platforms as they promise better scalability and reliability.

With data centers already responsible for about one percent of the world's power consumption, optimizing resource usage is of paramount importance. Simultaneously, meeting the increasing and changing resource and performance requirements is only possible by optimizing resource management without introducing additional overhead. This requires the research and development of new modeling approaches to understand the behavior of running applications with minimal information.

However, the emergence of modern software paradigms makes it increasingly difficult to derive such models and renders previous performance modeling techniques infeasible. Modern cloud applications are often deployed as a collection of fine-grained and interconnected components called microservices. Microservice architectures offer massive benefits but also have broad implications for the performance characteristics of the respective systems. In addition, the microservices paradigm is typically paired with a DevOps culture, resulting in frequent application and deployment changes. Such applications are often referred to as cloud-native applications. In summary, the increasing use of ever-changing cloud-hosted microservice applications introduces a number of unique challenges for modeling the performance of modern applications. These include the amount, type, and structure of monitoring data, frequent behavioral changes, or infrastructure variabilities. This violates common assumptions of the state of the art and opens a research gap for our work.

In this thesis, we present five techniques for automated learning of performance models for cloud-native software systems. We achieve this by combining machine learning with traditional performance modeling techniques. Unlike previous work, our focus is on cloud-hosted and continuously evolving microservice architectures, so-called cloud-native applications. Therefore, our

contributions aim to solve the above challenges to deliver automated performance models with minimal computational overhead and no manual intervention. Depending on the cloud computing model, privacy agreements, or monitoring capabilities of each platform, we identify different scenarios where performance modeling, prediction, and optimization techniques can provide great benefits. Specifically, the contributions of this thesis are as follows:

- *Monitorless: Application-agnostic prediction of performance degradations.* To manage application performance with only platform-level monitoring, we propose *Monitorless*, the first truly application-independent approach to detecting performance degradation. We use machine learning to bridge the gap between platform-level monitoring and application-specific measurements, eliminating the need for application-level monitoring. *Monitorless* creates a single and holistic resource saturation model that can be used for heterogeneous and untrained applications.

  Results show that *Monitorless* infers resource-based performance degradation with 97% accuracy. Moreover, it can achieve similar performance to typical autoscaling solutions, despite using less monitoring information.

- *SuanMing: Predicting performance degradation using tracing.* We introduce *SuanMing* to mitigate performance issues before they impact the user experience. This contribution is applied in scenarios where tracing tools enable application-level monitoring. *SuanMing* predicts explainable causes of expected performance degradations and prevents performance degradations before they occur.

  Evaluation results show that *SuanMing* can predict and pinpoint future performance degradations with an accuracy of over 90%.

- *SARDE: Continuous and autonomous estimation of resource demands.* We present *SARDE* to learn application models for highly variable application deployments. This contribution focuses on the continuous estimation of application resource demands, a key parameter of performance models. *SARDE* represents an autonomous ensemble estimation technique. It dynamically and continuously optimizes, selects, and executes an ensemble of approaches to estimate resource demands in response to changes in the application or its environment.

  Through continuous online adaptation, *SARDE* efficiently achieves an average resource demand estimation error of 15.96% in our evaluation.

- *DepIC: Learning parametric dependencies from monitoring data. DepIC* utilizes feature selection techniques in combination with an ensemble regression approach to automatically identify and characterize parametric dependencies. Although parametric dependencies can massively improve the accuracy of performance models, *DepIC* is the first approach to automatically learn such parametric dependencies from passive monitoring data streams.

  Our evaluation shows that *DepIC* achieves 91.7% precision in identifying dependencies and reduces the characterization prediction error by 30% compared to the best individual approach.

- *Baloo: Modeling the configuration space of databases.* To study the impact of different configurations within distributed Database Management Systems (DBMSs), we introduce *Baloo*. Our last contribution models the configuration space of databases considering measurement variabilities in the cloud. More specifically, *Baloo* dynamically estimates the required benchmarking measurements and automatically builds a configuration space model of a given DBMS.

  Our evaluation of *Baloo* on a dataset consisting of 900 configuration points shows that the framework achieves a prediction error of less than 11% while saving up to 80% of the measurement effort.

Although the contributions themselves are orthogonally aligned, taken together they provide a holistic approach to performance management of modern cloud-native microservice applications. Our contributions are a significant step forward as they specifically target novel and cloud-native software development and operation paradigms, surpassing the capabilities and limitations of previous approaches. In addition, the research presented in this paper also has a significant impact on the industry, as the contributions were developed in collaboration with research teams from Nokia Bell Labs, Huawei, and Google. Overall, our solutions open up new possibilities for managing and optimizing cloud applications and improve cost and energy efficiency.

Zusammenfassung

Eine der Folgen der weltweiten Coronavirus-Pandemie ist die erhöhte Nachfrage und Nutzung von Onlinediensten in der gesamten Welt. Gleichzeitig werden die Performanceanforderungen an moderne Technologien immer strenger, da die Benutzer an höhere Standards gewöhnt sind. Diese gestiegenen Performance- und Verfügbarkeitsanforderungen, gepaart mit dem unvorhersehbaren Nutzerwachstum, führen dazu, dass ein zunehmender Anteil der Anwendungen auf Public-Cloud-Plattformen läuft, da diese eine bessere Skalierbarkeit und Zuverlässigkeit versprechen.

Da Rechenzentren bereits heute für etwa ein Prozent des weltweiten Stromverbrauchs verantwortlich sind, ist es von größter Bedeutung, den Ressourceneinsatz zu optimieren. Die gleichzeitige Erfüllung der steigenden und variierenden Ressourcen- und Performanceanforderungen ist nur durch eine Optimierung des Ressourcenmanagements möglich, ohne gleichzeitig zusätzlichen Overhead einzuführen. Dies erfordert die Erforschung und Entwicklung neuer Modellierungsansätze, um das Verhalten der laufenden Anwendungen mit möglichst wenigen Informationen zu verstehen.

Das Aufkommen moderner Softwareparadigmen macht es jedoch zunehmend schwieriger, solche Modelle zu lernen und macht bisherige Modellierungstechniken unbrauchbar. Moderne Cloud-Anwendungen werden oft als eine Sammlung von feingranularen, miteinander verbundenen Komponenten, sogenannten Microservices, bereitgestellt. Microservicearchitekturen bieten massive Vorteile, haben aber auch weitreichende Auswirkungen auf die Performance der jeweiligen Systeme. Darüber hinaus wird das Microserviceparadigma häufig in Verbindung mit einer DevOps-Kultur eingesetzt, was zu häufigen Änderungen am Deployment oder der Anwendung selbst führt. Solche Anwendungen werden auch als cloud-native Anwendungen bezeichnet. Zusammenfassend lässt sich sagen, dass der zunehmende Einsatz von sich ständig ändernden und in der Cloud gehosteten Microservice-Anwendungen eine Reihe von besonderen Herausforderungen für die Modellierung der Performance von modernen Anwendungen mit sich bringt. Darunter sind die Menge, Art und Struktur der Monitoringdaten, häufige Änderungen am Verhalten oder Veränderungen der zugrundeliegenden Infrastruktur. Das verstößt gegen gängige Annahmen des

aktuellen Stands der Technik und eröffnet eine Forschungslücke für unsere Arbeit.

In der vorliegenden Arbeit stellen wir fünf Techniken zum automatisierten Lernen von Performancemodellen für cloud-native Softwaresysteme vor. Wir erreichen dies durch die Kombination von maschinellem Lernen mit traditionellen Performance-Modellierungstechniken. Im Gegensatz zu früheren Arbeiten liegt unser Fokus auf in der Cloud gehosteten und sich ständig weiterentwickelnden Microservice-Architekturen, sogenannten cloud-nativen Anwendungen. Daher zielen unsere Beiträge darauf ab, die oben genannten Herausforderungen zu lösen, um automatisierte Performancemodelle mit minimalem Rechenaufwand und ohne manuellen Aufwand zu erzeugen. Abhängig vom jeweiligen Cloudmodell, eventuellen Datenschutzvereinbarungen oder den Möglichkeiten des Monitoringsframworks der jeweiligen Plattform, identifizieren wir verschiedene Anwendungsszenarien, in denen Techniken zur Modellierung, Vorhersage und Optimierung der Performance große Vorteile bieten können. Im Einzelnen sind die Beiträge dieser Arbeit wie folgt:

- *Monitorless: Anwendungsagnostische Vorhersage von Performanceverschlechterung.* Um die Performance einer Anwendung ausschliesslich mittels Monitoring auf Plattformebene zu verwalten, schlagen wir *Monitorless* vor, den ersten wirklich anwendungsunabhängigen Ansatz zur Erkennung von Performanceverschlechterungen. Wir verwenden maschinelles Lernen, um die Lücke zwischen Monitoring auf Plattformebene und anwendungsspezifischen Messungen zu schließen, wodurch das Monitoring auf Anwendungsebene überflüssig wird. *Monitorless* erstellt ein einziges und ganzheitliches Modell der Ressourcensättigung, das auch für heterogene und nicht im Training enthaltene Anwendungen verwendet werden kann.

  Die Ergebnisse zeigen, dass *Monitorless* ressourcenbasierte Performanceverschlechterungen mit einer Genauigkeit von 97% erkennt. Darüber hinaus zeigt es ähnliche Leistungen wie typische Autoscalinglösungen, obwohl es weniger Monitoringinformationen verwendet.

- *SuanMing: Vorhersage von Performanceverschlechterung mithilfe von Tracing.* Wir führen *SuanMing* ein, um Performanceprobleme zu entschärfen, bevor sie sich auf das Benutzererlebnis auswirken. Dieser Beitrag wird in Szenarien angewendet, in denen Tracing-Tools das Monitoring auf Anwendungsebene ermöglichen. *SuanMing* sagt erklärbare Ursachen für erwartete Performanceeinbußen voraus und verhindert diese, bevor sie auftreten.

Evaluationsergebnisse zeigen, dass *SuanMing* zukünftige Performanceeinbußen mit einer Genauigkeit von über 90% vorhersagen und lokalisieren kann.

- *SARDE: Kontinuierliche und autonome Schätzung des Ressourcenbedarfs.* Wir stellen *SARDE* vor, um Performancemodelle für hochvariable Anwendungen zu lernen. Dieser Beitrag konzentriert sich auf die kontinuierliche Schätzung des Ressourcenbedarfs von Anwendungen, einem wichtigen Parameter in Performancemodellen. *SARDE* ist ein autonomes Ensembleverfahren zum Schätzen. Es wählt dynamisch und kontinuierlich aus einem Ensemble von Ansätzen, optimiiert diese, und führt sie aus, um den Ressourcenbedarf als Reaktion auf Änderungen in der Anwendung oder ihrer Umgebung zu schätzen.

  Durch kontinuierliche Online-Anpassung erreicht *SARDE* in unserer Evaluation effizient einen durchschnittlichen Fehler bei der Schätzung des Ressourcenbedarfs von 15,96%.

- *DepIC: Lernen parametrischer Abhängigkeiten aus Monitoringdaten. DepIC* nutzt Techniken zu Featureauswahl in Kombination mit einem Ensemble-Regressionsansatz, um parametrische Abhängigkeiten automatisch zu identifizieren und zu charakterisieren. Obwohl parametrische Abhängigkeiten die Genauigkeit von Performancemodellen deutlich verbessern können, ist *DepIC* der erste Ansatz, der solche parametrischen Abhängigkeiten automatisch aus passiven Monitoringdatenströmen lernt.

  Unsere Evaluation zeigt, dass *DepIC* eine Genauigkeit von 91,7% bei der Identifizierung von Abhängigkeiten erreicht und den Fehler bei der Charakterisierungsvorhersage um 30% im Vergleich zum besten individuellen Ansatz reduziert.

- *Baloo: Modellierung des Konfigurationsraums von Datenbanken.* Um die Auswirkungen verschiedener Konfigurationseinstellungen in verteilten Datenbankmanagementsystemen zu untersuchen, führen wir *Baloo* ein. Unser letzter Beitrag modelliert den Konfigurationsraum von Datenbanken unter Berücksichtigung der Messungsvariabilitäten der Cloud. Genauer gesagt, schätzt *Baloo* dynamisch die erforderliche Anzahl der Benchmarkmessungen und baut automatisch ein Konfigurationsraummodell eines gegebenen Datenbankmanagementsystems auf.

  Unsere Evaluation von *Baloo* auf einem aus 900 Konfigurationspunkten bestehenden Datensatz zeigt, dass das Framework einen Vorhersage-fehler von weniger als 11% erreicht und gleichzeitig bis zu 80% des Messaufwands einspart.

Obwohl die Beiträge an sich orthogonal zueinander ausgerichtet sind, bilden sie zusammengenommen einen ganzheitlichen Ansatz für das Performancemanagement von modernen cloud-nativen Microservice-Anwendungen. Unsere Beiträge sind ein bedeutender Schritt, da sie speziell auf neuartige und cloudnative Paradigmen für Softwareentwicklung und Betrieb abzielen, sowie die Fähigkeiten bisheriger Ansätze übertreffen. Darüber hinaus hat die in dieser Arbeit vorgestellte Forschung auch einen bedeutenden Einfluss auf die Industrie, da die Beiträge in Zusammenarbeit mit Forschungsteams von Nokia Bell Labs, Huawei und Google entwickelt wurden. Insgesamt eröffnen unsere Lösungen neue Möglichkeiten für die Verwaltung und Optimierung von Cloudanwendungen und verbessern so die Kosten- und Energieeffizienz.

Acknowledgments

This thesis is a result of great support guidance from many people, of which I can only name a few. First and foremost, I thank my advisor Prof. Dr. Samuel Kounev, for the opportunity to be part of his research group. He always supported me with advice and encouragement during the last five years.

Furthermore, I had the pleasure of working at and being part of the Software Engineering Chair at the University of Würzburg. I want to thank my current and former colleagues with whom I enjoyed collaborating on many projects: André Bauer, Lukas Beierlieb, Vanessa Borst, Bohdan Dovhan, Simon Eismann, André Greubel, Marius Hadry, Christoph Hagen, Nikolas Herbst, Stefan Herrnleben, Lukas Iffländer, Dennis Kaiser, Jóakim von Kistowski, Christian Krupitzer, Robert Leppich, Veronika Lesch, Maximilian Meißner, Thomas Prantl, Piotr Rygielski, Norbert Schmitt, Maximilian Schwinger, Florian Spieß, Simon Spinner, Martin Sträßer, Jürgen Walter, and Marwin Züfle. I also want to thank Fritz Kleemann, Susanne Stenglin, and Erika Littmann, who have always been very supportive. In addition, I would like to thank Vanessa Ackermann, Sven Elflein, Stefan Herrnleben, Lydia Kaiser, Christian Knaub, Torsten Krauss, Nico Ruck, Karsten Schaefers, and Martin Sträßer for supporting me as research students or as part of their theses.

Additionally, I would like to thank all of the numerous collaborators I met and worked with during my journey through the academic world. In particular, I thank Marco Cello, Jesus Omana Iglesias, Diego Lugones, and Patrick Nicholson for my time at Nokia Bell Labs, Yair Arian, Avi Chalbani, Idan Nagar, and Noam Peretz for the productive collaboration with Huawei, and Arif Merchant for the fruitful discussions and insightful comments. Further, the collaborations with the SPEC RG DevOps Performance and SPEC RG Cloud, as well as my participation at GI Dagstuhl Seminar "Software Engineering for Intelligent and Autonomous Systems" sparked inspiring and enlightening discussions, for which I am very thankful.

Finally, I want to thank my parents and my friends for their emotional support and encouragement throughout the years.

Contents

Peer-Reviewed Publications

Journal and Magazine Articles

[Gro+21a] Johannes Grohmann, Simon Eismann, André Bauer, Simon Spinner, Johannes Blum, Nikolas Herbst, and Samuel Kounev. "SARDE: A Framework for Continuous and Self-Adaptive Resource Demand Estimation". In: *ACM Transactions on Autonomous and Adaptive Systems* 15.2 (2021), pp. 1–31. DOI: 10.1145/3463369.

[Gro+20a] Johannes Grohmann, Nikolas Herbst, Avi Chalbani, Yair Arian, Noam Peretz, and Samuel Kounev. "A Taxonomy of Techniques for SLO Failure Prediction in Software Systems". In: *Computers* 9.1 (2020), p. 10. DOI: 10.3390/computers9010010.

[Eyk+19] Erwin van Eyk, Johannes Grohmann, Simon Eismann, André Bauer, Laurens Versluis, Lucian Toader, Norbert Schmitt, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. "The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms". In: *IEEE Internet Computing* 23.6 (2019), pp. 7–18. DOI: 10.1109/mic.2019.2952061.

[Spi+19] Simon Spinner, Johannes Grohmann, Simon Eismann, and Samuel Kounev. "Online model learning for self-aware computing infrastructures". In: *Journal of Systems and Software* 147 (2019), pp. 1–16. DOI: 10.1016/j.jss.2018.09.089.

[Eis+21a] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina Abad, and Alexandru Iosup. "The State of Serverless Applications: Collection, Characterization, and Community Consensus". In: *IEEE Transactions on Software Engineering* (2021). DOI: 10.1109/TSE.2021.3113940.

[Eis+21b] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. "Serverless Applications: Why, When, and How?" In: *IEEE Software* 38.1 (2021), pp. 32–39. DOI: 10.1109/ms.2020.3023302.

Full Conference Papers

[Gro+21b] Johannes Grohmann, Martin Straesser, Avi Chalbani, Simon Eismann, Yair Arian, Nikolas Herbst, Noam Peretz, and Samuel Kounev. "Suan-

Ming: Explainable Prediction of Performance Degradations in Microservice Applications". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ACM, 2021. DOI: 10.1145/3427921.3450248.

[Gro+20b] Johannes Grohmann, Daniel Seybold, Simon Eismann, Mark Leznik, Samuel Kounev, and Jörg Domaschka. "Baloo: Measuring and Modeling the Performance Configurations of Distributed DBMS". In: *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. MASCOTS '20. IEEE, 2020, pp. 1–8. DOI: 10.1109/MASCOTS50786.2020.9285960.

[Gro+19a] Johannes Grohmann, Patrick K. Nicholson, Jesus Omana Iglesias, Samuel Kounev, and Diego Lugones. "Monitorless: Predicting Performance Degradation in Cloud Applications with Machine Learning". In: *Proceedings of the 20th International Middleware Conference*. Middleware '19. ACM, 2019, pp. 149–162. DOI: 10.1145/3361525.3361543.

[Gro+19b] Johannes Grohmann, Simon Eismann, Sven Elflein, Manar Mazkatli, Jóakim von Kistowski, and Samuel Kounev. "Detecting Parametric Dependencies for Performance Models Using Feature Selection Techniques". In: *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. MASCOTS '19. IEEE, 2019, pp. 309–322. DOI: 10.1109/mascots.2019.00042.

[Her+20a] Stefan Herrnleben, Johannes Grohmann, Pitor Rygielski, Veronika Lesch, Christian Krupitzer, and Samuel Kounev. "A Simulation-Based Optimization Framework for Online Adaptation of Networks". In: *Simulation Tools and Techniques*. SIMUtools 2020. Springer International Publishing, 2020, pp. 513–532. DOI: 10.1007/978-3-030-72792-5_41.

[Eis+20a] Simon Eismann, Johannes Grohmann, Erwin van Eyk, Nikolas Herbst, and Samuel Kounev. "Predicting the Costs of Serverless Workflows". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ICPE '20. ACM, 2020, pp. 265–276. DOI: 10.1145/3358960.3379133.

[Kis+19a] Jóakim von Kistowski, Johannes Grohmann, Norbert Schmitt, and Samuel Kounev. "Predicting Server Power Consumption from Standard Rating Results". In: *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. ICPE '19. ACM, 2019, pp. 301–312. DOI: 10.1145/3297663.3310298.

[Eis+19] Simon Eismann, Johannes Grohmann, Jürgen Walter, Jóakim von Kistowski, and Samuel Kounev. "Integrating Statistical Response Time Models in Architectural Performance Models". In: *2019 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2019, pp. 71–80. DOI: 10.1109/icsa.2019.00016.

[Bau+18] André Bauer, Johannes Grohmann, Nikolas Herbst, and Samuel Kounev. "On the Value of Service Demand Estimation for Auto-scaling". In: *Lecture Notes in Computer Science*. Vol. 10740. Lecture Notes in Computer Science. Springer International Publishing, 2018, pp. 142–156. DOI: 10.1007/978-3-319-74947-1_10.

[Eis+21c] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and Samuel Kounev. "Sizeless: Predicting the Optimal Size of Serverless Functions". In: *Proceedings of the 22nd International Middleware Conference*. Middleware '21. ACM, 2021, pp. 248–259. DOI: 10.1145/3464298.3493398.

[Züf+21] Marwin Züfle, Joachim Agne, Johannes Grohmann, Ibrahim Dörtoluk, and Samuel Kounev. "A Predictive Maintenance Methodology: Predicting the Time-to-Failure of Machines in Industry 4.0". In: *2021 IEEE 19th International Conference on Industrial Informatics (INDIN)*. IEEE, 2021, pp. 1–8. DOI: 10.1109/INDIN45523.2021.9557387.

[Her+20b] Stefan Herrnleben, Rudy Ailabouni, Johannes Grohmann, Thomas Prantl, Christian Krupitzer, and Samuel Kounev. "An IoT Network Emulator for Analyzing the Influence of Varying Network Quality". In: *Simulation Tools and Techniques*. SIMUtools 2020. Springer International Publishing, 2020, pp. 580–599. DOI: 10.1007/978-3-030-72795-6_47.

[Maz+20] Manar Mazkatli, David Monschein, Johannes Grohmann, and Anne Koziolek. "Incremental Calibration of Architectural Performance Models with Parametric Dependencies". In: *2020 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2020, pp. 23–34. DOI: 10.1109/icsa47634.2020.00011.

[Her+20c] Stefan Herrnleben, Piotr Rygielski, Johannes Grohmann, Simon Eismann, Tobias Hossfeld, and Samuel Kounev. "Model-Based Performance Predictions for SDN-Based Networks: A Case Study". In: *Lecture Notes in Computer Science*. MMB 2020. Springer International Publishing, 2020, pp. 82–98. DOI: 10.1007/978-3-030-43024-5_6.

[Bau+21] André Bauer, Marwin Züfle, Simon Eismann, Johannes Grohmann, Nikolas Herbst, and Samuel Kounev. "Libra: A Benchmark for Time Series Forecasting Methods". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ACM, 2021. DOI: 10.1145/3427921.3450241.

[Züf+20] Marwin Züfle, Christian Krupitzer, Florian Erhard, Johannes Grohmann, and Samuel Kounev. "To Fail or Not to Fail: Predicting Hard Disk Drive Failure Time Windows". In: *Lecture Notes in Computer Science*. MMB 2020. Springer International Publishing, 2020, pp. 19–36. DOI: 10.1007/978-3-030-43024-5_2.

[DAn+19] Mirko D'Angelo, Simos Gerasimou, Sona Ghahremani, Johannes Grohmann, Ingrid Nunes, Evangelos Pournaras, and Sven Tomforde. "On Learning in Collective Self-Adaptive Systems: State of Practice and a 3D Framework". In: *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. SEAMS '19. IEEE, 2019, pp. 13–24. DOI: `10.1109/seams.2019.00012`.

[Her+21] Stefan Herrnleben, Maximilian Leidinger, Veronika Lesch, Thomas Prantl, Johannes Grohmann, Christian Krupitzer, and Samuel Kounev. "ComBench: A Benchmarking Framework for Publish/Subscribe Communication Protocols Under Network Limitations". In: *Performance Evaluation Methodologies and Tools*. Springer International Publishing, 2021, pp. 72–92. DOI: `10.1007/978-3-030-92511-6_5`.

[Kis+18] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research". In: *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. MASCOTS '18. IEEE, 2018, pp. 223–236. DOI: `10.1109/mascots.2018.00030`.

Short Conference Papers

[Gro+17] Johannes Grohmann, Nikolas Herbst, Simon Spinner, and Samuel Kounev. "Self-Tuning Resource Demand Estimation". In: *2017 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 2017, pp. 21–26. DOI: `10.1109/icac.2017.19`.

[Bau+20] André Bauer, Marwin Züfle, Johannes Grohmann, Norbert Schmitt, Nikolas Herbst, and Samuel Kounev. "An Automated Forecasting Framework based on Method Recommendation for Seasonal Time Series". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ICPE '20. ACM, 2020, pp. 48–55. DOI: `10.1145/3358960.3379123`.

[Bez+19] Cor-Paul Bezemer, Simon Eismann, Vincenzo Ferme, Johannes Grohmann, Robert Heinrich, Pooyan Jamshidi, Weiyi Shang, André van Hoorn, Mónica Villavicencio, Jürgen Walter, and Felix Willnecker. "How is Performance Addressed in DevOps?" In: *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. ICPE '19. ACM, 2019, pp. 45–50. DOI: `10.1145/3297663.3309672`.

Workshop Papers

[GEK19] Johannes Grohmann, Simon Eismann, and Samuel Kounev. "On Learning Parametric Dependencies from Monitoring Data". In: *Proceedings of the 10th Symposium on Software Performance 2019 (SSP'19)*. 2019.

[Gro+19c] Johannes Grohmann, Simon Eismann, Andre Bauer, Marwin Zuefle, Nikolas Herbst, and Samuel Kounev. "Utilizing Clustering to Optimize Resource Demand Estimation Approaches". In: *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. IEEE, 2019, pp. 134–139. DOI: `10.1109/fas-w.2019.00043`.

[Ack+18] Vanessa Ackermann, Johannes Grohmann, Simon Eismann, and Samuel Kounev. "Black-box Learning of Parametric Dependencies for Performance Models". In: *Proceedings of 13th International Workshop on Models@run.time (MRT), co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018)*. CEUR Workshop Proceedings. 2018.

[Von+20] Sonya Voneva, Manar Mazkatli, Johannes Grohmann, and Anne Koziolek. "Optimizing Parametric Dependencies for Incremental Performance Model Extraction". In: *Communications in Computer and Information Science*. Vol. 1269. Communications in Computer and Information Science. Springer International Publishing, 2020, pp. 228–240. DOI: `10.1007/978-3-030-59155-7_17`.

[Bau+19] André Bauer, Simon Eismann, Johannes Grohmann, Nikolas Herbst, and Samuel Kounev. "Systematic Search for Optimal Resource Configurations of Distributed Applications". In: *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. IEEE, 2019, pp. 120–125. DOI: `10.1109/FAS-W.2019.00040`.

[DAn+20] Mirko D'Angelo, Sona Ghahremani, Simos Gerasimou, Johannes Grohmann, Ingrid Nunes, Sven Tomforde, and Evangelos Pournaras. "Learning to Learn in Collective Adaptive Systems: Mining Design Patterns for Data-driven Reasoning". In: *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. IEEE, 2020, pp. 121–126. DOI: `10.1109/acsos-c51401.2020.00042`.

Vision, Position, and Tutorial Papers

[GEK18] Johannes Grohmann, Simon Eismann, and Samuel Kounev. "The Vision of Self-Aware Performance Models". In: *2018 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2018, pp. 60–63. DOI: `10.1109/icsa-c.2018.00024`.

[Gro+18] Johannes Grohmann, Nikolas Herbst, Simon Spinner, and Samuel Kounev. “Using Machine Learning for Recommending Service Demand Estimation Approaches - Position Paper”. In: *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018)*. INSTICC. SCITEPRESS - Science and Technology Publications, 2018, pp. 473–480. DOI: 10.5220/0006761104730480.

[Wal+18] Jürgen Walter, Simon Eismann, Johannes Grohmann, Dušan Okanovic, and Samuel Kounev. “Tools for Declarative Performance Engineering”. In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE ’18. ACM, 2018, pp. 53–56. DOI: 10.1145/3185768.3185777.

[Eyk+18] Erwin van Eyk, Alexandru Iosup, Cristina L. Abad, Johannes Grohmann, and Simon Eismann. “A SPEC RG Cloud Group’s Vision on the Performance Challenges of FaaS Cloud Architectures”. In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE ’18. ACM, 2018, pp. 21–24. DOI: 10.1145/3185768.3186308.

[Dom+21] Jörg Domaschka, Mark Leznik, Daniel Seybold, Simon Eismann, Johannes Grohmann, and Samuel Kounev. “Buzzy: Towards Realistic DBMS Benchmarking via Tailored, Representative, Synthetic Workloads”. In: *Companion of the ACM/SPEC International Conference on Performance Engineering*. ACM, 2021. DOI: 10.1145/3447545.3451175.

Software Contributions

[SGK19] Simon Spinner, Johannes Grohmann, and Samuel Kounev. *LibReDE: Library for Resource Demand Estimation*. Standard Performance Evaluation Corporation Research Group (SPEC RG) accepted Tool. https://research.spec.org/tools/overview/librede.html. 2019.

[Kis+19b] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. *TeaStore*. Standard Performance Evaluation Corporation Research Group (SPEC RG) accepted Tool. https://research.spec.org/tools/overview/teastore.html. 2019.

Chapter 1

Introduction

The recent coronavirus pandemic impacted the lives of billions of people around the world. Due to travel restrictions and lockdowns, many conferences, classes, meetings, and conversations were online. Therefore, the demand for video conferencing services, such as Zoom, spiked from 10 to over 300 million daily meeting participants during early 2020 [Iqb20; Zoo20; Hof20]. Similarly, we saw an increase in other application usages. For example, the streaming service Netflix attracted nearly 16 million additional subscribers during the first quarter of 2020, marking the largest three-month increase in the company's history so far [Aro20], reaching a total of over 200 million subscribers by the end of that year [Sta21b]. Additionally, Amazon.com, the largest online marketplace, has increased its revenue by 38% during 2020, reaching a total of $386 billion [Koh21] with the online retail sector accounting for $197 billion in 2020 [Sta21a; Ama21b].

At the same time, the performance requirements for modern technology are becoming increasingly important, as users are accustomed to improved standards [DB13; Ein19; Gaj+20]. For example, Zoom attributes its success over other video conferencing solutions to its improved user experience, especially its latency requirement of fewer than 150 ms [Pie20; Iqb20; Hof20]. For Netflix, studies also show that the bit rate and the buffering time have a high impact on user engagement for the delivered streaming content [Gov14; Dob+13]. Furthermore, it is reported that a response time increase of only 100 ms costs around 1% in lost sales for Amazon.com [Ela18; Ein19], and a page slowdown of one second could amount to $1.6 billion lost sales a year [Eat12].

These increased performance and availability requirements, paired with the discussed unprecedented user growth, are only possible if the applications rely on public cloud platforms, as these promise higher scalability and reliability [Jud16; Fle20; Gar20]. For example, Zoom migrated to using both Amazon Web Services (AWS) and the Oracle cloud platforms [Jud20a; Jud20b; Hof20], while both Amazon.com and Netflix rely on the AWS cloud [Jud16; Ama17; Ama21a]. Consequentially, AWS is one of the strongest-growing business segments of Amazon [Sta21a; Koh21]. An industry survey by Flexera shows that 59% of respondents expect their cloud use to exceed their plans due to the pandemic [Fle20]. Similarly, Gartner, Inc. projects the worldwide spending on public cloud services to grow to over $362 billion until 2022 in response to the pandemic [Gar20]. It is furthermore expected that by 2025, 80% of production applications will run in the cloud [Lot17].

Cloud computing is already a significant contributor to climate change, as data centers are responsible for about one percent of the world's electricity consumption [Mas+20; Jon18], which is expected to grow to eight percent by 2030 [AE15; Jon18]. The continuing growth of cloud services implies increasing energy consumption and, therefore, calls for raising the efficiency of systems [Ber+11; Mas+20]. It is estimated that currently, roughly 30% of cloud spending is wasted, making optimizing usage and cost the top cloud initiative of many companies [Fle20]. This resource waste is also exacerbated by the above discussed rising demands for performance and availability, usually negotiated via Service Level Agreements (SLAs), as these are usually met by provisioning more resources [Her18; JS14; DB13].

It is, therefore, of paramount importance to optimize the resource usage while fulfilling the increased user demands and performance requirements [Ber+11; Fer+12b]. This helps to save both operating costs and energy consumption as well as to minimize the carbon footprint of the global cloud infrastructure. These optimizations can only be achieved by closely monitoring and modeling the applications in order to understand their behavior [JS14].

1.1 Problem Statement

Modern cloud applications often employ microservice architectures [San16; Kwi19; Gaj+20]. Microservice architectures are based on a collection of fine-grained and interconnected services that should focus on a single responsibility [Dra+17; Gan+19b; LF14]. Developing microservice applications offers huge advantages to developers [Lin16; Gan+19b; Jam+18; LF14; Faz+16], which is why it is expected to become the de-facto standard for software development [Gaj+20]. The global microservice cloud market is projected to reach over $3 billion by 2026, growing at a compound annual growth rate of over 21% [Ver21]. Both Netflix and Amazon are early adopters and strong advocates of microservice architectures [Coc16; Ful15].

However, despite the advantages, microservice architectures represent a significant departure from the way cloud applications are traditionally designed and therefore have broad implications on their performance properties, rendering many previous approaches for performance management infeasible [Gan+19b; Fow15; Eis+20a]. The number of services in complex applications can easily reach several hundred services (e.g., Netflix: 700, Spotify: 810, Uber: 500) [Kwi19; Rud19], which complicates the process of finding and fixing performance problems. While production monitoring tools are usually capable of capturing enough data points, the amount of gathered information is too large to oversee manually. For example, Netflix collects over 1.2 billion measurements per minute [HR14].

In addition, the microservice paradigm is usually paired with DevOps culture adopting a Continuous Integration and Continuous Deployment (CI/CD) pipeline, which leads to frequent code and deployment changes as applications are constantly under development [Gaj+20; Coc16; Rud19; BHJ16; Eis+20a]. Most companies deploy new code weekly, daily, or even hourly [Nov16]. Large companies like Amazon and Netflix deploy code thousands of times per day, which renders manual tracking of performance changes infeasible [Nul21; Nov16].

To summarize, we observe that modern software applications are often microservice applications developed based on the DevOps paradigm and operated in cloud environments. These applications are sometimes also referred to as cloud-native applications [Red18]. Cloud providers aim to optimize their resource usage in order to save costs and conserve energy. These trends pose a set of distinct challenges for the modeling and performance analysis of modern applications:

- We are able to collect vast amounts of monitoring data from different services, possibly spread over different data centers; however, the analysis of this data is becoming increasingly complex.
- The deployment, as well as the monitored applications themselves, are subject to continuous changes due to frequent CI/CD cycles.
- Even for static applications, the performance of the underlying infrastructure may be constantly changing as cloud infrastructures generally experience higher degrees of performance variabilities.
- In addition, cloud providers usually have no prior knowledge about the running applications and must infer all information from generic monitoring tools.
- Finally, as monitoring information is gathered from production systems, ensuring a low overhead of the monitoring tools is imperative.

In this thesis, we present different techniques for the automated performance model learning and performance prediction of modern software systems. In contrast to previous work, our focus is on cloud-hosted and continuously evolving microservice architectures. Therefore, our contributions aim at solving the above-mentioned challenges in order to deliver automated software performance models using minimal computational overhead and fully avoiding the need for manual interference.

1.2 Shortcomings of the State of the Art

Depending on the cloud computing model, privacy agreements, or monitoring capabilities of the respective platform, we see four main application scenarios where performance modeling, prediction, and optimization techniques can provide major benefits. In this section, we identify the key scenarios towards this task and outline the shortcomings of current state-of-the-art approaches. While there exist several related approaches in the literature, most of them are not applicable in the defined scenarios.

The first scenario is when a cloud provider only has black-box and platform-level information. For example, the platform itself can be instrumented via monitoring tools by the cloud provider, but the hosted applications (usually deployed within containers on that platform), as well as any monitoring within these applications, stays opaque for the cloud provider. This can be due to privacy and data protection concerns of customers, but also due to the overhead of application-level monitoring tools or the lack of a proper monitoring interface. This black-box view and the lack of any application-level information necessitate application-agnostic approaches that solely rely on platform-level data. While previous works attempted to enable performance predictions based on platform-level monitoring [HT11; Yan+15b; Yan+15a; Coh+04; Ngu+13; KKR10; Kun+12; Eme+10; Eme+12; Woo+07; Woo+09; Cor+17; Bia+20], no approach currently exists that provides a single holistic and application-agnostic model.

The second scenario is the opposite case, where application-level monitoring is available via application-level tracing tools (see Section 2.2.1 on page 19), but actual platform-level monitoring (including deployment or utilization metrics) is unavailable. As before, approaches need to be applicable without prior knowledge or tuning to the specific application and must not rely on any platform-level information as the underlying infrastructure is abstracted away or is subject to continuous variabilities or redeployments. The state of the art in this scenario [JPG19; LCZ18; Wan+18; Wu+20b; Zho+18; SP19; Lou+18;

Sha+13; vOI19; LLG18; Cor+17; Bia+20; Gan+19a] is usually not capable of providing actionable performance predictions, that is, predictions that are: (i) explainable and (ii) reasonably far in the future. Both properties are required in order to be able to mitigate a performance issue before it affects the user experience.

In some scenarios, it is possible to combine both monitoring types from the previous two scenarios. We have, on the one hand, platform-level monitoring of all resources and deployments; on the other hand, application-level tracing of the different application microservices. This scenario corresponds to the most developed area of related work, as it has a strong overlap with classical (component-based) performance engineering scenarios. Hence, classical performance modeling techniques [Hub+17; Hoo14; BKR09] are also applicable in this context. However, the advent of DevOps practices, CI/CD pipelines, and opaque cloud platforms renders the common assumption of a static application model with fixed parameters, resources, and deployments infeasible [Bez+19]. Therefore, previous approaches [Wal+17; Hri+99; Isr+05; MF11; BKK09; BHK11; BVK13; Wil+15a; SWK16; Spi+19] are not able to continuously keep the model parameters updated. In addition, no approach to automatically identify the impact of input parameters on these model parameters has been proposed.

Finally, almost all cloud-native applications still rely on backend services treated as black-box components. For example, databases still play a huge role in microservice applications, as each service usually needs to maintain its data [Rud19; LF14]. The respective, possibly distributed, DBMSs cannot be modeled by classical (i.e., white-box) modeling approaches and furthermore offer only limited monitoring information. In addition, distributed DBMSs offer a large set of configuration parameters that also massively impact the performance as well as the occupied resources and operating costs of the respective system. Previous approaches for modeling the configuration spaces of DBMSs [DTB09; YNM16; Van+17; Mah+17; Far+18; DCS17; Zhe+19] do not consider the measurement variabilities in the cloud or do not cover distributed setups.

The described scenarios were identified in cooperation with our industrial partners (see Section 1.4). Chapter 3 on page 35 provides a more detailed overview of the state of the art and the individual shortcomings of respective approaches.

1.3 Goals and Research Questions

In the previous section, we introduce the four main scenarios of interest for this thesis and the problem with the current state of the art with regard to our problem statement. We identify a set of research gaps that arise due to the way modern software systems are designed and operated. This thesis aims to close these gaps. In the following, we present the four main goals of this thesis. Each goal is designed in a way that closes the described research gap and therefore advances the field with its respective contribution. In addition, we list a set of Research Questions (RQs) that will be answered during the pursuance of each goal.

Goal I: *Design an application-agnostic approach for the detection of resource saturation based on platform-level monitoring data.*

- **RQ I.1**: *How can platform-level measurements be utilized to detect resource saturation?*
- **RQ I.2**: *How can we generalize the results to create a generic and holistic prediction model?*

This first goal mainly addresses the first scenario, that is, the question of how we can infer information about the running applications without explicitly monitoring them. As discussed in the previous sections, this would only be possible by generating a generic and holistic prediction model that is capable of detecting application-agnostic performance problems based on monitoring resource saturation. Hence, the second RQ addresses the issue of generalization.

Goal II: *Develop an approach for the prediction of performance degradation using application-level tracing.*

- **RQ II.1**: *How can tracing data be utilized to predict the future performance of a system?*
- **RQ II.2**: *How can we pinpoint the root cause service of a performance problem?*

The second goal is geared towards the second application scenario. As application-level tracing contains more detailed information, our goal in this scenario is to not only detect performance problems but also to develop an approach for predicting and mitigating performance problems before they occur in the system. In order to do so, we require predictions of the future state of the system and a way to identify the root cause of the anticipated performance problem. Therefore, **RQ II.1** and **RQ II.2** target these properties, respectively.

Goal III: *Enable the continuous estimation and improvement of performance model parameters using production monitoring data.*

- **RQ III.1**: *How can we combine different estimation approaches to efficiently produce continuous resource demand estimations?*
- **RQ III.2**: *How can the impact of parameters on resource demands be identified and characterized?*

As the third scenario allows for the application of several proven techniques, we do not aim at developing similar or competing approaches to the already established techniques. Instead, the third goal identifies two specific research gaps in the area of architectural modeling and respective RQs addressing these gaps. First, resource demands are central parameters of architectural performance models and, therefore, need to be measured and updated continuously in order to maintain an accurate performance model. Second, parametric dependencies describing the relation between input parameters and the resource demand specifications need to be integrated into performance models, as they improve the models' expressiveness and their variability in changing situations. Nevertheless, there currently exist no approaches addressing these issues while relying solely on monitoring data.

Goal IV: *Develop a workflow for modeling configurable, cloud-based, and distributed DBMSs.*

- **RQ IV.1**: *How can the influence of performance variabilities during benchmark measurements be mitigated?*
- **RQ IV.2**: *How can we analyze a configuration space that is too large to measure exhaustively?*

As already introduced, the fourth scenario presents an orthogonal challenge to the three other scenarios. As modern software architectures still increasingly rely on cloud-hosted data storage, modeling, configuring, and optimizing the underlying distributed DBMSs is still a primary concern. Since previous work does not exhaustively cover the topic, the fourth goal targets two additional RQs that address two major challenges when modeling the configuration space of these systems.

1.4 Contribution Summary

In accordance with the scenarios identified in Section 1.2 and the research goals defined in Section 1.3, we now present a summary of the five individual contributions of this thesis.

Contribution 1: *Monitorless*: Detection of resource saturation using platform-level metrics Our first contribution was conducted during a research visit at the cloud research lab of Nokia Bell Labs in Dublin, Ireland. Software operation usually relies on application Key Performance Indicators (KPIs) for sizing and orchestrating cloud resources dynamically. In this work, we leverage machine learning to bridge the gap between platform-level monitoring and application-specific KPIs. Inspired by the way the *Serverless Computing* paradigm delegates the entire management of the execution environment to the cloud provider, we propose a *Monitorless* approach to application performance management. We show that training a machine learning model with platform-level data collected from the execution of representative containerized services allows inferring application KPI degradation at runtime. This is an opportunity to simplify operations as engineers can rely solely on platform metrics while still fulfilling application KPIs, removing the need for all application-level monitoring. This contribution shows that one generic resource saturation model can be used to detect performance degradation of several complex applications, even when such applications are unknown to the trained model. We note that this represents a significant divergence between *Monitorless* and other solutions based on KPIs as we propose a generic approach with a single resource saturation model employed for a heterogeneous set of applications.

We evaluate *Monitorless* using three different scenarios. A multitier web service application and two representative microservice applications that are not included in the training phase. Results show that *Monitorless* infers KPI degradation with an accuracy of 97%. Furthermore, we use the inferred KPI degradations as a basis for a simple autoscaling mechanism which, notably, can achieve similar performance like typical autoscaling solutions, even though it uses less monitoring information. This contribution was published at the 20th ACM/IFIP International Middleware Conference (MIDDLEWARE 2019) [Gro+19c] and addresses **Goal I**.

Contribution 2: *SuanMing*: Prediction of performance degradations using application-level tracing This contribution was made in the context of a research project with the Huawei Research Center in Tel Aviv, Israel. In this work, we focus on Application Performance Monitoring (APM) or tracing tools. APM is normally purely reactive; that is, it can only report current or past performance degradations. Although some approaches capable of predictive application monitoring have been proposed, they can only report a predicted degradation but cannot explain its root cause, making it hard to prevent the expected degradation. In order to mitigate this issue, we present *SuanMing*, a framework for predicting performance degradation of microservice applications running in cloud environments. *SuanMing* is able to predict future root causes for anticipated performance degradations and therefore aims at preventing performance degradations before they actually occur.

We evaluate *SuanMing* on two realistic microservice applications, and we show that our approach predicts and pinpoints performance degradations with an accuracy of over 90%. This contribution addresses **Goal II** and was published at the 12th ACM/SPEC International Conference on Performance Engineering (ICPE 2021) [Gro+21d].

Contribution 3: *SARDE*: Continuous and self-adaptive resource demand estimation In this contribution, we focus on the estimation of application resource demands. Resource demands are crucial parameters for modeling and predicting the performance of software systems. Currently, resource demand estimators are usually executed in an offline scenario for system analysis. However, the monitored system, as well as the resource demands themselves, are subject to constant changes, as discussed in the previous sections. These changes impact the applicability, the required parameterization, as well as the resulting accuracy of available estimation approaches. Over time, this leads to invalid or outdated performance models. To address this issue, we propose *SARDE*, a framework for *S*elf-*A*daptive *R*esource *D*emand *E*stimation. *SARDE* dynamically and continuously optimizes, selects, and executes an ensemble of resource demand estimation approaches to adapt to changes in the environment. This creates an autonomous and unsupervised ensemble estimation technique, providing reliable resource demand estimations in dynamic environments. The research towards this topic was awarded a Google Faculty Research Award, demonstrating its significance to the industry.

We evaluate *SARDE* using two realistic datasets. One set of different microbenchmarks reflecting different possible system states and one dataset consisting of a continuously running microservice in a changing environment. Our results show that by continuously applying online optimization, selection, and estimation, *SARDE* efficiently adapts to the online trace and achieves an average estimation error of 15.96% using the resulting ensemble technique.

This contribution addresses **RQ III.1** of **Goal III** and was published in ACM Transactions on Autonomous and Adaptive Systems (ACM TAAS) [Gro+21b].

Contribution 4: *DepIC*: Learning parametric dependencies from monitoring data Our fourth contribution addresses **RQ III.2** of **Goal III**. Parametric dependencies describe the relationship between the input parameters of a component and its performance properties. Therefore, they significantly increase the prediction accuracy of architectural performance models. However, manually modeling parametric dependencies is time-intensive and requires expert knowledge, while existing automated extraction approaches require source code or dedicated performance tests. Therefore, in collaboration with researchers from the Karlsruhe Institute of Technology in Karlsruhe, Germany, we propose *DepIC*, a new approach to derive parametric dependencies solely from monitoring data available at run-time. The *DepIC* approach for Dependency Identification and Characterization works by utilizing feature selection techniques for identification combined with an ensemble regression approach for the characterization of dependencies.

Our evaluation shows that *DepIC* achieves an identification precision of 91.7% on a microservice application. In addition, a proposed meta-selector reduces the prediction error of the characterization compared to the best individual machine learning approach by 30%. The contribution in this area was published at the 27th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2019) [Gro+19b]. This contribution addresses **RQ III.2** and, taken together with the previous contribution, completes **Goal III**.

Contribution 5: *Baloo*: Measuring and modeling performance configurations of distributed DBMSs As already discussed in Section 1.2, the modeling and optimization of distributed DBMSs is still a vital part of tuning modern cloud-native applications. Correctly configuring a distributed DBMS deployed in a cloud environment for maximizing performance poses many challenges to operators. Even if the entire configuration spectrum could be measured directly, which is often infeasible due to the multitude of parameters, single cloud measurements are subject to random variations and need to be repeated multiple times. Therefore, it is not trivial to decide how many repetitions need to be done for each measurement. Our fifth contribution addresses these issues by proposing *Baloo*, a framework for systematically measuring and modeling different performance-relevant configurations of distributed DBMSs in cloud environments. *Baloo* dynamically estimates the required number of measurement configurations, as well as the number of required measurement repetitions per configuration based on the desired target accuracy.

We evaluate *Baloo* based on a dataset consisting of 900 DBMS configuration points, measured together with a research group from the University of Ulm, Germany. Our evaluation shows that the framework achieves a prediction error of under 11% while saving up to 80% of the measurement effort. This contribution addresses **Goal IV** and was published at the 28th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2020) [Gro+20b].

Summary The presented contributions are aligned with the four scenarios and research goals that we introduced in Sections 1.2 and 1.3. Taken together, they represent a significant step forward compared to the state of the art in the respective fields. Our collaborations with the research teams of Nokia Bell Labs and Huawei allowed us to identify open research challenges in the industry and helped to increase the research impact of our contributions. In addition, we collaborated with researchers from the Karlsruhe Institute of Technology while working on the third scenario and proposed *Baloo* together with colleagues from the University of Ulm.

These collaborations allowed us to present different solutions tailored specifically to the different scenarios that cloud providers nowadays typically find themselves in. As such, although the contributions themselves are orthogonally aligned, taken together, they provide a holistic approach for the performance management of modern microservice applications in the cloud. Overall, our framework helps to overcome the limitations of existing approaches to performance management of microservice applications in cloud environments and provides a solution to solve the critical issues of usage growth and resource waste described previously.

1.5 Thesis Outline

In the following, we describe the remaining structure of this thesis. Part I introduces the fundamental notions and concepts and gives a broad overview of the current state of the art. In Chapter 2 on page 15, we give quick definitions of various important terms from the areas of software engineering, performance analysis, and machine learning. Furthermore, we survey the related work on the targeted goals in order to point out shortcomings and differences in relation to the contributions of this thesis in Chapter 3 on page 35.

Following, Part II presents the main contributions of this thesis. Chapter 4 on page 53 describes the *Monitorless* approach for the application-agnostic detection of performance degradation (**Goal I**), while Chapter 5 on page 73 delineates the automatic prediction of performance problems (**Goal II**) with *SuanMing*. *SARDE*, our approach for continuous estimation of resource demands (**Goal III**), is introduced in Chapter 6 on page 91, followed by Chapter 7 on page 109, focusing on learning parametric dependencies with *DepIC*. Our fifth contribution, *Baloo* (**Goal IV**), is described in Chapter 8 on page 137.

Part III then continues to evaluate the proposed solutions by following the same structure in Chapters 9 to 13. Chapter 9 on page 151 shows our evaluation of *Monitorless* using three different applications. Second, Chapter 10 on page 171 analyses the performance of *SuanMing* in two different evaluation scenarios. Chapter 11 on page 187 focuses on *SARDE* and its results for the estimation of resource demands. Then, we evaluate *DepIC* and its capabilities to identify and characterize parametric dependencies in Chapter 12 on page 209. In the last chapter of Part III, Chapter 13 on page 227 evaluates *Baloo* in terms of modeling accuracy and measurement cost. In addition, all evaluation chapters of Part III further contain a discussion on the threats to the validity of the evaluation as well as the conceptual assumptions and limitations of each approach.

Finally, Part IV concludes this thesis by summarizing our achievements in Chapter 14 on page 241. We give an outlook and describe the possibilities for future work in Chapter 15 on page 247.

Part I

Fundamentals and Related Work

Chapter 2

Fundamentals

In this section, we quickly outline the fundamental concepts that lay the foundation for our research. In Section 2.1, we quickly define notions and paradigms that we utilize throughout this work. Section 2.2 focuses on different types and levels of performance evaluation of software systems. Finally, we introduce the most relevant machine learning concepts for this work in Section 2.3.

2.1 Current Paradigms for Software Development and Operation

First, we outline recent trends in software engineering and operation as these paradigms strongly influence the problem statement and the environment for our solutions. Therefore, we aim at explaining and defining the most relevant concepts to contextualize them for the scope of this work.

2.1.1 Microservice Applications

The microservice architecture can be seen as a specific variant of traditional service-oriented architecture [LF14]. The idea of microservice architecture is to develop an application out of a set of fine-grained, self-contained, independent, and loosely coupled smaller application components, the so-called microservices [Coc16; Eis+20a]. These components can be independently architected, developed, and deployed. Communication is handled via lightweight messaging protocols, e.g., via Hypertext Transfer Protocol (HTTP) and Representational State Transfer (REST) [LF14; Faz+16; Kis+18]. Well-known pioneers of this paradigm were, among others, Amazon and Netflix with their large-scale cloud applications [Rud19; Coc16; Ful15; LF14]. Depending on the size and the business case of the application, microservice architectures can consist of several hundred different microservices (e.g., Netflix: 700, Spotify: 810, Uber: 500) [Kwi19; Rud19]. In practice, microservice application development is often paired with DevOps processes, CI/CD, and containerized deployment [LF14; Coc16; Ful15; BHJ16; Gaj+20; Eis+20a] in order to develop so-called cloud-native applications [Red18; BHJ16]. In addition, the serverless paradigm has also attracted a lot of interest for operating microservice applications [Gaj+20; Red18; Gan+19b].

2.1.2 DevOps

The DevOps paradigm is an ongoing development towards tighter integration between the development (*Dev*) and the operation (*Ops*) of an application as well as the respective responsible teams [Bru+15; Bez+19]. The idea behind this is to enable the teams to react more flexibly to changes in requirements or the environment, which also integrates well with modern agile development techniques [BHJ16]. In conjunction with microservice architectures, this often means that the same team is responsible for developing, running, and maintaining one specific microservice, following Amazon's principle of "you build, you run it" [LF14; OHa06; BHJ16]. One key goal of the DevOps movement is to allow for shorter development cycles, enabling to roll out new features and bug fixes more quickly and generally achieving a faster time to market while maintaining the overall software quality [Bru+15; BHJ16; Nul21]. Therefore, large companies nowadays deploy new code weekly, daily, or even hourly, while high-performing organizations report 30 times more deployments with 200 times shorter lead times [Nov16]. For example, Amazon deploys code every 11.7 seconds, on average [Nul21; Nov16].

2.1.3 Continuous Integration and Continuous Deployment

Tightly coupled with the concepts of DevOps is the implementation of continuous integration, delivery, or deployment pipelines [Bru+15; Red18; BHJ16; Eis+20a]. These pipelines automate the processes of building, testing, deploying, and sometimes also monitoring the developed code. The acronym *CD* is sometimes used to denote continuous delivery, as opposed to continuous deployment. The difference between the two is that continuous delivery produces deployable products, while continuous deployment goes one step further and automatically deploys the built products [Bru+15]. In this work, we define CI/CD as the whole process of Continuous Integration and Continuous Deployment, as it is increasingly becoming the norm [Ful15; Coc16; Nul21].

2.1.4 Virtualization

In today's data centers, virtualization is the de-facto standard method to deploy and run applications, as already around 80% of x86 server workloads in data centers are virtualized [Spi17; BDW16]. Although there exist different related concepts, like network virtualization [RK13] or desktop virtualization [MP07], we only refer to server virtualization in this work. Virtualization enables the dynamic allocation of hardware resources to different Virtual Machines (VMs) or containers. This is done by using a host Operating System (OS) or a *hypervisor*, which controls the actual hardware resources and dynamically manages the access of the deployed virtualized guest instances to those resources [Spi17; Faz+16]. Currently, server virtualization technology comes in two flavors: VM-based and container-based. While VM-based virtualization starts new OSs and hosts them as independent entities, container-based virtualization uses the same OS kernel for all running containers [Faz+16]. The advantage of this is that containers are more lightweight and flexible than VMs, while VMs offer better resource isolation and are less dependent on the host OS [Faz+16; Nad+16]. It is not uncommon to combine both approaches by hosting containers inside VMs [Eis+20a; Nad+16]. However, the flexibility and size of containers lend themselves nicely for the fine-granular deployments of microservice applications, which is why container-based virtualization is the preferred flavor in that context [Fle20; Gan+19b; Nad+16].

2.1.5 Docker

One popular way to organize and manage container virtualization is using Docker[1] containers [Fle20; Faz+16; Eis+20a]. Docker images are packaged container snapshots that bundle the required software together with all of its dependencies and requirements [Nad+16]. This massively improves the portability of containerized applications and also simplifies scaling or migrating containers. Furthermore, Docker enables sharing these images using registries. Currently, the most popular container registry is `Dockerhub`[2], with 6 million repositories and 8 billion monthly image pulls [New21; Kre20]. Due to these simplifications, Docker images are a popular product of CI/CD pipelines, where each microservice can be bundled in its own image, pushed to a public or private registry, and then deployed as one or many service instances [Nad+16; Gan+19b; BHJ16; Eis+20a].

[1] `https://docker.com`
[2] `https://hub.docker.com`

2.1.6 Container Orchestration

As manually overseeing applications consisting of hundreds or thousands of container instances is infeasible, container orchestration tools can take care of managing containerized applications. The most common representative container orchestration tool is Kubernetes[3] [Gaj+20; Nad+16; New21; Eis+20a]. Container orchestration tools are capable of automatic deployment, scaling, and management of containerized applications, including service discovery, load balancing, and storage management. Therefore, there exist many cloud offerings with already integrated container orchestration [New21; Fle20].

2.1.7 Cloud-native

Another popular paradigm is the concept of *cloud-native* applications [Red18]. Cloud-native application development is focused on building and running software that takes full advantage of the offered cloud computing capabilities. This usually includes the use of DevOps and CI/CD processes, container-based infrastructure, as well as Application Programming Interface (API)-driven and service-based application architectures, like the microservice paradigm [Red18; BHJ16]. Therefore, many cloud-native applications utilize microservice architectures and follow the techniques we introduced in this section [BHJ16].

2.1.8 Serverless Computing

The term *Serverless Computing* refers to a cloud computing paradigm that abstracts most operational infrastructure concerns away from the customer, enables granular billing and event-driven interactions [Eyk+17; Eis+21b; Eis+21c]. One type of serverless offerings are so-called Function-as-a-Service (FaaS) platforms, capable of managing resources, lifecycle, communication, and execution of single-responsibility, stateless, and on-demand services [Eyk+19; Eyk+18b]. Note that serverless only refers to the view of the cloud customer, where the actual server and hardware resources are no longer observable [Eis+21a]. However, in the end, computations are still executed on regular cloud server infrastructure [Eyk+19]. As cloud functions hosted by FaaS platforms are conceptually not different from a microservice with a single endpoint [Eyk+18a; Eyk+19], the serverless paradigm is also a popular deployment option for microservice or cloud-native applications [Gaj+20; Red18; Gan+19b].

[3] `https://kubernetes.io`

2.2 Performance Analysis of Software Systems

There exist several ways of evaluating the performance of a system or an application. In this work, we are focusing on the two main areas of (i) monitoring the actual system under study in Section 2.2.1 and (ii) analyzing the performance of a system using a model representation in Section 2.2.2. While monitoring has the advantage that it delivers more accurate results than analyzing a model, it requires, relies on, and possibly disturbs the actual system under study [Hoo14; Wal19; Wal+18]. Furthermore, monitoring can only record events that happened on the system and is therefore not suited for hypothetical scenarios or forecasting [Koz08; Hub+17]. Recent approaches also aim at combining both techniques [WHK17; Wal19; Wal+18].

2.2.1 Monitoring

There exist several different ways to measure the performance of a system or an application. Note that other notions refer to APM as Application Performance Management [Men02b; Hoo14]. While this definition includes both monitoring and management, we define the acronym solely as Application Performance Monitoring (APM). We restrict ourselves to the monitoring aspect for now, as application management and the insights gained from the monitoring data are the focus of later sections.

2.2.1.1 Measurement Strategies

We distinguish four different ways of obtaining a metric of interest from a running application, following the definitions of Lilja [Lil00] and Kounev et al. [KLK20]:

- **Event-driven** measurements are obtained and stored only when specific events of interest occur, where an event is a change in the system state. Therefore, the respective measurement overhead is dependent on the frequency of the observed events.
- **Tracing** is similar to event-driven monitoring. However, while event-driven measurements rely on simple counting, tracing records further information about each event. Depending on the amount of additional stored information, the overhead is significantly higher than with counting.
- **Sampling** records system state in equidistant time intervals. Sampling does not observe every event occurrence and only reports on statistical

summaries of the system behavior. Therefore, the introduced overhead is only dependent on the configured sampling frequency.

- **Indirect measurement** can be used when the metric of interest cannot be measured directly by observing certain events or introduces too much overhead. In such cases, the metric of interest is derived based on other metrics that can be measured directly. For example, the estimation of resource demands presented in Chapter 6 on page 91 is exactly such a case, where the metric of interest, the resource demand, is indirectly measured through other metrics, for example, the response time or the resource utilization. As indirect measurement does not require additional information (in addition to the already observed metrics), the overhead is usually the smallest.

One important type of tracing is the so-called *call path tracing*. Call path tracing is a type of application-level tracing where the hierarchy of method calls or requests is stored in order to be able to reconstruct the call path or the call chain. We utilize call path tracing in Chapters 5 and 7 on page 73 and on page 109, respectively. In Chapter 7, we additionally store the parameter types and values of each call.

2.2.1.2 Measurement Levels

In addition to the different strategies, we also distinguish between different levels of system monitoring in this thesis. We introduce the following terminology:

- **System-level** monitoring is the measurement of hardware metrics, like Central Processing Unit (CPU), Random Access Memory (RAM), or Input-Output (I/O) usage, usually done from the perspective of the OS. The majority of metrics are usually captured using sampling or event-driven counts. Therefore, depending on the sampling rate and the number of required metrics, the overhead of system-level monitoring is usually comparatively small.

- **Platform-level** monitoring is similar to system-level monitoring, but also virtualization layers. For example, in addition to the total CPU utilization of the host system, a hypervisor can report on the distribution of CPU usage between the different VMs. Similarly, the control groups (`cgroups`[4]) feature on Linux can be utilized to account for the usages of many container virtualization techniques.

- **Application-level** monitoring is only concerned with measurements and logs produced by the running software application. In contrast to the previous levels, application-level monitoring is, therefore, less generic, and generated logs are usually specific to the respective application. Consequentially, several APM solutions have been proposed that tackle this issue in order to define and maintain unified interfaces. In addition, application-level monitoring tools usually employ significantly more tracing strategies to capture relevant information about the application behavior.

All three introduced monitoring types can be applied on single hosts or distributed systems. Collecting, aggregating, and compiling the respective monitoring data is the task of the applied monitoring tool.

2.2.1.3 Monitoring tools

Several different tools for APM have been developed both in industry and academia. Most of them specialize in application-level monitoring, but some include the means for platform- or system-level monitoring as well. Commercial representatives are, for example, Dynatrace[5], New Relic[6], or AppDynamics[7]. Open-source solutions include for example, inspectIT Ocelot[8], Zipkin[9], Jaeger[10], Pinpoint[11], or Kieker[12] [HWH12]. One example of a platform-level monitoring tool is Performance Co-Pilot (PCP)[13]. Although it is also able to connect application-specific monitoring agents, the default capabilities are limited to a platform-level view. A general overview of the APM landscape can be found at the OpenAPM website[14].

[4]`https://kernel.org/doc/Documentation/cgroup-v1/cgroups.txt`
[5]`https://dynatrace.com`
[6]`https://newrelic.com`
[7]`https://appdynamics.com`
[8]`https://inspectit.rocks`
[9]`https://zipkin.io`
[10]`https://jaegertracing.io`
[11]`https://pinpoint-apm.github.io/pinpoint`
[12]`https://kieker-monitoring.net`
[13]`https://pcp.io`
[14]`https://openapm.io/landscape`

2.2.2 Modeling

Next to measuring the performance of an application, one can also evaluate the performance of a system using a model representation of the system itself. The main advantage of model-based predictions is that they allow for exploration of different implementations, deployments, architectures, or configurations without the need for having a potentially costly live system [Eis+18; Wal19]. Therefore, model predictions are usually faster and more convenient compared to measurements, given that the accuracy of the respective predictions is sufficient [Wal19; WHK17]. There exist multiple approaches for modeling and predicting the performance of a software system. We distinguish between *predictive* (or analytic) and *descriptive* (or architecture-level) models [KBH14; Hoo14]; however, models can also be both at the same time [Kou+17; Hoo14].

2.2.2.1 Predictive Models

Predictive performance models usually focus on producing the actual performance prediction. These models are usually highly abstract, treat the services as black-box, and make strong assumptions. Such models include different variants from queueing theory, for example, Queueing Networks (QNs) [MG00; VSS12], Layered Queueing Networks (LQNs) [Li+09; Woo21], Queueing Petri Nets (QPNs) [Kou06], as well as statistical regression models [EFH04; ZCS07; Eis+19] or other machine learning techniques [Kun+12; Cor+17; Bia+20; Gro+19c]. As each of those approaches has its own benefits and drawbacks, various approaches have been developed to enable dynamically switching between different methods [EFH04; Hub+17; WHK17; RKT15], or alternatively altering the prediction model itself [GEK18; Eis+19]. These approaches accomplish this by working with a descriptive or architecture-level performance model that focuses on describing the system and its performance properties [KBH14; Koz10].

2.2.2.2 Descriptive Models

Next to dynamically selecting or tailoring the actual prediction model, descriptive performance modeling approaches generally have the advantage that they decouple the description from the prediction and analysis process. This has several advantages, including improved readability and reusability [Koz10; KBH14; Hub+17]. One representative of such a modeling formalism is the Descartes Modeling Language (DML) [Kou+16; KBH14].

The Descartes Modeling Language (DML) is a descriptive, architecture-level performance modeling formalism that is specifically targeted at online performance and resource management of applications running in data centers. The performance prediction itself is done using specialized model-to-model transformations into predictive models, such as QNs, LQNs, or QPNs [Hub+17].

Central parameters of DML and other architectural modeling formalisms are service demands or resource demands [Spi+15; Spi+19; Wal+17]. *Resource demands* are the average time a unit of work (e.g., request or transaction) spends obtaining service from a resource (e.g., CPU, Hard Disk Drive (HDD), or Solid State Drive (SSD)) in a system, aggregating all visits but excluding any waiting times [Laz+84; MDA04]. Hence, the correct configuration of resource demands is a crucial factor for an accurate performance prediction of a system [Spi17; Bau+18].

Additionally, DML puts an explicit focus on *parametric dependencies* of model variables [Eis+18; KBH14]. Parametric dependencies describe the relationship between model parameters, such as resource demands or loop counts, with input parameters [Koz08; Hub+17]. Hence, including and modeling parametric dependencies increases the accuracy and the predictive power of performance models [Koz10; Koz08; Eis+18].

Other architectural performance modeling formalisms that are used include ROBOCOP [Bon+04], KLAPER [GMS07], SLAstic [Hoo14], and the Palladio Component Model (PCM) [BKR09]. An extended list and comparison of different approaches are given in the survey by Koziolek [Koz10]. However, none of these architectural approaches is currently widely used in industry [Koz10; Bez+19]. Hence, there is a need for automated approaches in order to foster adoption in the industry.

2.3 Machine Learning

Throughout this thesis, we utilize several techniques from the area of machine learning. In this section, we quickly introduce the basic concepts as well as the most important algorithms we utilize in this thesis. Machine Learning is generally defined as using computers to solve problems without explicitly programming them. Instead, example data or past experience is used to train the system [Alp20]. The field is traditionally classified into the three areas of supervised, unsupervised, and reinforcement learning, based on the type of feedback from the training system [RN20]. As most of this work utilizes supervised learning techniques, we mainly focus on this area.

The task of supervised machine learning can be defined as follows [RN20]. A set of labeled training data $T = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ containing n training examples (x_i, y_i) for $i \in 1, \ldots, n$ is given. Each sample assigns a (potentially multidimensional) feature vector $x_i \in \mathbb{R}^l$ to a (potentially multidimensional) label $y_i \in \mathbb{R}^m$. The dimensionality of the input space l is also referred to as the number of features. The distinction of the concrete task can be made depending on the properties of the labels y_i.

If the set of possible target values is finite, we speak of a *classification* problem and refer to the possible target values as classes. We further distinguish between *binary* classification and multi-class classification depending on whether there are two or more classes [RN20]. If, instead, the target variables are continuous, we speak of a *regression* problem. If the target values are multi-dimensional, that is, if $m > 1$, we speak of a multi-label classification or multi-target regression problem [Pic12]. In this work, we concentrate on single-target learning. That is, we assume $m = 1$.

The machine learning task is to find a function $h : \mathbb{R}^l \mapsto \{1, 2, \ldots, c\}$ for classification or $h : \mathbb{R}^l \mapsto \mathbb{R}$ for regression, mapping from x_i to y_i. This function h is sometimes also called a machine learning *model* [Pic12; RN20]. The model h can then be used to predict the label y_j for new and unseen samples of x_j with $j > n$. For simplicity, we assume all features to be from the domain of $\mathbb{R}$ in this work.

2.3.1 Algorithms

In the following, we quickly introduce the machine learning algorithms that we use throughout this work.

2.3.1.1 k-Nearest Neighbors

The k-Nearest Neighbors (kNN) algorithm [Alt92] is a non-parametric approach that can be used for regression and classification tasks. For classification, the sample vector is assigned to the class that occurs most frequently among its k closest neighbors. Similarly, for regression, a weighted average of the target values of the nearest neighbors is returned. As the name suggests, k-Nearest Neighbors (kNN) can be parameterized by defining the number of neighbors k to be considered [Alp20].

2.3.1.2 Linear Modeling

An intuitive variant of regression modeling is Linear Regression (LR) [Alp20; DS98]. Linear Regression (LR) models are functions that describe the value of a target variable using a linear combination of the feature values. Common ways to find such functions are using the Least Squares (LSQ) or Least Absolute Deviation (LAD) criterion [DS98]. A (binary) Logistic Regression (Logit) model is a linear model that enables binary classification by assigning probabilities to each class [Cox58]. The *multinomial* Logistic Regression (Logit) model is a generalization of this approach, allowing for more than two classes [The92].

One approach aiming to provide robust regression and hence a model that is less susceptible to outliers in the training data is Huber Regression (HR) [Hub64]. As the name suggests, Huber Regression (HR) relies on the Huber loss function, a combination of L_1 and L_2 loss functions (see Section 2.3.2.1 on page 28).

Further improvements to the generated models can be achieved using model *regularization*. Regularization modifies the used loss functions to penalize the size of the coefficients, that is, the model complexity [RN20]. For example, Ridge Regression (Ridge) is an LR technique that minimizes the Sum of Squared Errors (SSE) penalized with the squared L_2 norm, that is, the sum of the squared feature weights [HK70b; HK70a]. Similarly, the Least Absolute Shrinkage and Selection Operator (LASSO) regression penalizes the SSE with the L_1 norm, that is, the sum of the absolute weights [Tib96]. Elastic Net Regression (ElasticNet) combines Ridge and LASSO as it uses both regularization terms [ZH05].

2.3.1.3 Bayesian Ridge Regression

Bayesian Ridge Regression (BRR) is another variant of linear regression. In contrast to the previous approaches, Bayesian models estimate probability distributions for the model parameters instead of giving single-point estimates [Mac92]. This is done by starting with prior distributions based on experience or best guesses about the model parameters and updating these distributions using the available training examples. The posterior distributions become increasingly precise with increasing amounts of available measurement data. BRR is a Bayesian linear regression that additionally penalizes the model complexity with a regularization term, similar to the procedure of the Ridge Regression (Ridge) introduced above [Mac92].

2.3.1.4 Support Vector Machines

This technique works on binary classification problems by searching for a hyperplane in the l-dimensional feature space separating the two classes with maximal margin to each of the two classes [CV95]. A new data point is then assigned to the class depending on which side of the hyperplane they lie. If there is no linear separation between the two groups, a kernel function can be utilized to transform the feature space into a higher dimensional space [BGV92]. Classification using Support Vector Machines (SVMs) is also referred to as Support Vector Classification (SVC), where a multiclass classification problem can be broken into multiple binary classifications [DK05]. SVMs can also be adapted to support regression problems via Support Vector Regression (SVR) [Dru+96].

2.3.1.5 Decision Tree Learning

Decision trees group training samples into several so-called leaf nodes. Starting from the root of the tree, the data is split into two or more groups based on the value of certain features [RN20]. This is recursively repeated until a stopping criterion is reached, creating a leaf node with all data samples of the respective group. For classification, each leaf node is associated with a prediction class. For regression, the value of each leaf node is a combination of all samples of the respective node.

The goal of the learning algorithm is to minimize the classification error, which can be achieved by maximizing the homogeneity in each leaf node. Popular algorithms for decision tree learning include Classification and Regression Trees (CART) by Breiman et al. [Bre+84], Iterative Dichotomiser 3 (ID3) [Qui86], or C4.5 [Qui93]. An extension to Classification and Regression Trees (CART) is the M5 model tree (M5) algorithm [Qui92]. M5 also builds models based on decision trees; however, in contrast to CART, where a constant value is returned for every leaf node, M5 builds LR models at each leaf node [Qui92]. Therefore, M5 is specifically targeted at regression problems but can also be adapted for classification problems [Fra+98].

2.3.1.6 Boosting

There exist several techniques for improving the performance of single estimators, based on combining the predictions of multiple ones [Alp20]. One of these *ensemble* techniques is Boosting [Sch90], where weak estimators are iteratively combined to form a strong estimator. The weights of the weak estimators are determined with respect to their individual learning accuracy. Popular approaches include Adaptive Boosting (AdaBoost) [FS97; Has+09], Gradient Boosted Decision Trees (GBDT) [Fri01], and eXtreme Gradient Boosting (XGBoost) [CG16].

2.3.1.7 Bagging

Bootstrap Aggregating or Bagging is another ensemble technique that works by combining the individual model predictions of various estimators trained on different subsets of the training data [Bre96]. A popular representative is the Random Forest (RF) algorithm [Ho95; Bre01]. Random Forest (RF) learns by combining multiple decision trees and randomly selecting a different feature subset for each tree. Both Bagging and Boosting techniques can also be applied to other (non-decision-tree-based) approaches [RN20; Alp20].

2.3.1.8 Neural Networks

Neural Networks (NNs) are a class of machine learning algorithms inspired by the functionality of biological brains [MP43]. A Neural Network (NN) is a network of neurons connected through edges, which transfer signals (usually in the form of real-valued numbers) from one neuron to another [Gur97]. Furthermore, each neuron is associated with an activation function, controlling the output of each neuron. One central part of the training process of NNs is the configuration of edge weights, that is, the weighting functions for each incoming edge of a neuron. Usually, neurons are grouped into layers. We denominate the first layer as the *input layer*, the last layer as the *output layer*, and all other layers as *hidden layers* [Gur97].

Due to the general applicability, there exist various architectures for the application of NNs [RN20]. In this work, we focus on *feed-forward* networks. Feed-forward NNs only allow links in one direction and therefore form a directed and acyclic graph [Gur97]. A neuron can only have input links from the previous layer and output links to the succeeding one. If every node from one layer has edges to every node of the following layer, we refer to the respective layers as *fully connected*. NNs can be used for classification and regression tasks [RN20].

2.3.2 Model Evaluation

There exist multiple ways to analyze the accuracy of a given machine learning model h [Bot19; RN20; Alp20]. Generally speaking, we can judge the accuracy of h by comparing the prediction $\hat{y}_i = h(x_i)$ with the real label y_i for $i \in 1, \dots, n$. During model learning, most algorithms repeatedly judge the accuracy of the current predictions on the set of training samples T via some *error function* (or *loss function*) in order to find h [RN20]. However, note that an evaluation should always utilize testing samples from a test set $(x_i, y_i) \notin T$, which are not contained in the training set. This way, it is possible to evaluate the achieved abstraction and generalization of the machine learning model [Alp20; RN20].

Therefore, the *training set* refers to the training samples T, that are available during model training. A *validation set* is a sub-set of T, that can be used for tuning the accuracy and the generalization of h, for example, by tuning its hyperparameters. We discuss the k-fold cross-validation, a common technique to split T into different folds of validation sets. The *test set* consists of examples that are used for analyzing the *out-of-sample* accuracy of the model, that is, the performance of the model on data that has not been used for learning [BCB14]. Finally, the *evaluation set* is any set on which a defined accuracy metric is calculated. This can be the training, the validation, or the test set.

There exists no single established metric, as all metrics each have their benefits and drawbacks [Bot19; Cha07]. Therefore, we utilize different metrics throughout this work, which we introduce in the following. We distinguish between regression and classification metrics. In Section 2.3.2.1, we focus on metrics for regression models, while Section 2.3.2.2 focuses on classification. Note that we reuse the notation introduced to describe the training set T, but explicitly speak of an evaluation set, as the defined metrics can be executed on training, validation, or test set, as described above. Therefore, in the following, we consider the set of n examples (x_i, y_i) for $i \in 1, \ldots, n$ to be any evaluation set on which our metrics are applied.

2.3.2.1 Regression

Absolute Error Our first error measure is the Mean Absolute Error (MAE). The Mean Absolute Error (MAE) calculates the mean of all deviations between label and predictions [Bot19]:

$$MAE = \frac{1}{n} \sum_{i=1}^{n} |\hat{y}_i - y_i| , \tag{2.1}$$

where n is the size of the evaluation set, $\hat{y}_i = h(x_i)$ is the predicted value, and y_i is the correct label of each sample. Note that although we assume y_i and $\hat{y}_i$ to be one-dimensional (i.e., $m = 1$), these error definitions can also be adapted to evaluate multi-dimensional labels or distributions, as they are basically distance metrics [Cha07]. MAE is generally easy to interpret, is dependent on the scale of the samples, and weighs outliers less strongly than Mean Squared Error (MSE) [Bot19].

One variant of MAE that aims at scale independence is the Mean Absolute Percentage Error (MAPE). This has the advantage that the accuracy of different approaches can be compared, although the scale of the sample sets is different [Bot19].

$$MAPE = \frac{100\%}{n} \sum_{i=1}^{n} \frac{|\hat{y}_i - y_i|}{|y_i|}, \tag{2.2}$$

where n is the size of the evaluation set, $\hat{y}_i = h(x_i)$ is the predicted value, and y_i is the correct label of each sample. The Mean Absolute Percentage Error (MAPE) is still intuitive to interpret, and its relative measure allows comparing the performance between different scales [Bot19].

When used for model training, the averaging of MAE can be omitted; hence, the Sum of Absolute Errors (SAE) or L_1 loss, for example, used in LAD regression, is defined as [RN20]:

$$SAE = \sum_{i=1}^{n} |\hat{y}_i - y_i|\,, \tag{2.3}$$

where n is the size of the evaluation set, $\hat{y}_i = h(x_i)$ is the predicted value, and y_i is the correct label of each sample.

Squared Error Another common family of error functions is the squared error, such as the Mean Squared Error (MSE) [Bot19; RN20]. In contrast to MAE, the MSE squares all deviations before aggregating them. This gives a higher weight to outliers, as one large deviation quickly dominates several smaller ones. The MSE is defined as [Bot19]:

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2, \tag{2.4}$$

where n is the size of the evaluation set, $\hat{y}_i = h(x_i)$ is the predicted value, and y_i is the correct label of each sample. While the MSE is still scale-dependent, it is much harder to interpret as squaring the errors transforms the MSE to a different scale than the original data. Hence, the Root Mean Squared Error (RMSE) aims at reverting this transformation by taking the square root of the MSE [Bot19]:

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2}, \tag{2.5}$$

where n is the size of the evaluation set, $\hat{y}_i = h(x_i)$ is the predicted value, and y_i is the correct label of each sample.

Analogously to the Sum of Absolute Errors (SAE), we can define the Sum of Squared Errors (SSE) or the squared L_2 loss, which aggregates all squared errors and can be used in LSQ regression [RN20]:

$$SSE = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2, \tag{2.6}$$

where n is the size of the evaluation set, $\hat{y}_i = h(x_i)$ is the predicted value, and y_i is the correct label of each sample.

2.3.2.2 Classification

In contrast to regression, classification problems have only a set of c possible classes. Therefore, during the evaluation of such models, there also exist a bounded set of different outcomes based on the label and the prediction of the model. For binary classification (i.e., $c = 2$), there are only two possible classes, namely a *positive* and a *negative* class. We define a True Positive (TP) as a positive sample that was correctly classified as such and a False Positive (FP) as a positive sample that was not detected. Analogously, we speak of a True Negative (TN), when a negative sample was correctly classified as negative, and of a False Negative (FN), when a negative sample was falsely classified as positive [RN20; Alp20].

Metrics Using these basic definitions, a set of accuracy metrics can be used to evaluate the performance of a classifier. The first one is the *Precision*, defined as [Alp20]:

$$\text{Precision} = \frac{TP}{TP + FP}, \tag{2.7}$$

where TP is the number of TPs, and FP is the number of FPs of the algorithm on the respective evaluation set. The precision expresses the share of correct classifications from all given positive classifications.

Next, we define the *Recall*, as the share of positive labels that were correctly found [Alp20]:

$$\text{Recall} = \frac{TP}{TP + FN}, \tag{2.8}$$

where TP is the number of TPs, and FN is the number of FNs of the algorithm on the respective evaluation set.

As both metrics are important, when analyzing the performance of a classifier, they are usually aggregated by the *F1* score. The F1 score is the harmonic mean of precision and recall and can be calculated as follows [RN20]:

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2TP}{2TP + FP + FN}, \tag{2.9}$$

where Precision and Recall are the above-defined metrics, TP is the number of TPs, FP is the number of FPs, and FN is the number of FNs of the algorithm on the respective evaluation set.

So far, none of the above metrics considers the number of TN of a classifier. This is useful when the number of TN is hard to determine or if it is too large so that all other metrics would be dominated by the number of TNs. One metric that considers the number of TNs is the *Accuracy*, defined as [Met78]:

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}, \tag{2.10}$$

where TP is the number of TPs, FP is the number of FPs, TN is the number of TNs, and FN is the number of FNs of the algorithm on the respective evaluation set. Therefore, the denominator is equal to the total number of samples $n =$ TPs + FPs + TNs + FNs.

Similarly, we also include the False Positive Rate (FPR) to analyze the performance of different models [Alp20; Met78]:

$$\text{FPR} = \frac{FP}{FP + TN}, \tag{2.11}$$

where FP is the number of FPs and TN is the number of TNs of the algorithm on the respective evaluation set. The FPR can be seen as the percentage of negative labels that were incorrectly classified as positive or the probability of reporting a false alarm [Alp20].

Receiver Operating Characteristic Curve Many classification algorithms can be tuned via a threshold [RN20; Alp20]. That is, every sample gets an assigned score on whether or not it belongs to a respective class. Therefore, by tuning the threshold of the respective algorithm starting at which threshold a sample is classified as positive, the classification accuracy can be tuned.

The Receiver Operating Characteristic (ROC) curve analyses the behavior of the recall and the FPR for different thresholds [Alp20; Met78]. Hence, it can be used to analyze the internal prediction ranking of the evaluation samples. One important metric for this analysis is the Area Under Curve (AUC). The AUC score can be seen as the integral of the ROC curve; a bigger number (with a maximum of 1) is considered to be better, as it resembles that more positive samples are ranked before negative ones [Alp20].

2.3.3 Model Optimization

Next to the set of available algorithms presented in Section 2.3.1, there exist several meta-heuristics and techniques aimed at improving or optimizing the performance of a machine learning model. In this work, we employ techniques from the area of hyperparameter tuning and feature engineering. Therefore, we will give a short introduction to the used techniques in Sections 2.3.3.1 and 2.3.3.2, respectively.

2.3.3.1 Hyperparameter Tuning

Most machine learning algorithms (or statistical estimation techniques) have a set of configurable parameters that strongly influence the performance of the respective approach [RN20; Alp20]. These parameters are commonly referred to as *hyperparameters*, as they are not the parameters of the learned model but rather of the model learning algorithm itself [Alp20]. However, choosing these hyperparameters is not trivial as the effect of the respective parameter setting is also highly specific to the underlying dataset used for training and validation.

If the number of features and their respective configuration settings is comparatively small, we can apply a *grid search* by simply evaluating all possible value combinations of each feature. This is necessary, as the effects of many configuration options interact with each other, making it necessary to try all possible combinations [Alp20].

One remaining problem of this procedure is that we need a representative evaluation of how well the model performs on the training data. This can be done using the functions defined in Section 2.3.2. However, as also discussed in Section 2.3.2, evaluating the generalization power of a trained model is only possible by analyzing its performance on samples that have not been used in the training set [RN20].

A common technique to solve this problem is *k-fold cross-validation* [RN20; Alp20]. The method works by splitting the training data into k different groups or folds. For each of the k folds, we then train a machine learning model on the $k - 1$ other available folds (training set) and evaluate the achieved model on the remaining one (validation set). The overall score is then aggregated from all k runs to assess the performance of the algorithm. However, one of the main disadvantages of this technique is the increased computational demand [RN20].

2.3.3.2 Feature Selection

For many machine learning problems, a large set of potential features is available for model training, many of which only contain limited or redundant information [RN20; Alp20; GE03]. In such instances, *feature selection* can be a valid tool for reducing the dimensionality of the problem [GE03]. One way of reducing the dimensionality is to apply subset selection, that is, to select only a subset of the available features while discarding others.

Feature subset selection techniques can be generally grouped into three different types [GE03]: First, *Filter* techniques select subsets of variables during prepossessing and before the model itself is trained. These methods are therefore independent of the chosen machine learning algorithm. Second, *Wrapper* approaches treat the machine learning models as a black-box and evaluate the achieved model prediction quality (possibly by means of k-fold cross-validation) of different feature subsets. Third, *Embedded* methods perform variable selection as part of the training process of some machine learning algorithms. Therefore, embedded methods are only applicable for specific types of learning algorithms that support selection during model training.

Another way of dimensionality reduction is by combining multiple features into a single feature, e.g., by transforming the feature space [GE03]. Similarly, Principle Component Analysis (PCA) applies a linear transformation on the feature space [Pea01] that itself does not reduce the dimensionality of the feature space. However, one main advantage of PCA is that the first principal components now account for larger amounts of variability in the dataset. Therefore, it is possible to reduce the dimensionality of the feature space while losing minimal information [Alp20; GE03].

Chapter 3
State of the Art

We split the related works into three different chapters in alignment with the research field of the respective contributions. Section 3.1 covers the prediction of performance degradation. The section is split into two subsections, aligned with the respective contributions. The first part focuses on the application-agnostic degradation prediction in Section 3.1.1, while the second emphasizes explainable degradation predictions in Section 3.1.2. Note that although the sections describe distinct aspects, there is still some overlap between the discussed works.

Next, in Section 3.2, we discuss the learning of model parameters for software performance models. This section is also subdivided into the essential areas aligned with our contributions. First, we cover the area of extracting parametric dependencies for performance models in Section 3.2.1. Second, we present the field of resource demand estimation in Section 3.2.2. Finally, as it is relevant for both of the presented contributions, we include a short discussion about other approaches on the topic of algorithm optimization and selection in Section 3.2.3.

Finally, Section 3.3 also focuses on learning performance models but pays specific attention to the area of DBMSs, as our contribution aims at modeling distributed and cloud-hosted DBMS. Therefore, we split the section into three parts: (i) benchmarking and measuring the performance of DBMSs (Section 3.3.1), (ii) modeling and optimizing DBMSs performance (Section 3.3.2), and (iii) predicting the performance of configurable systems (Section 3.3.3). While the latter section does not specifically focus on DBMSs, the presented techniques are still relevant as they can be seen as a somewhat more general approach to modeling these systems.

3.1 Predicting Performance Degradation

Regarding the prediction of performance degradations or generally Service Level Objective (SLO) failures in software systems, we published a survey about the current state of the art during the course of this thesis [Gro+20a].

The publication presents a systematic mapping of 67 scientific articles, eight theses, three technical reports, and two patents covering the topic of SLO failure prediction in software systems and introduces a taxonomy in order to group the found works. The taxonomy classifies related work along the dimensions of the prediction target (e.g., anomaly detection, performance prediction, or failure prediction), the time horizon (e.g., detection or prediction, online or offline application), and the applied modeling type (e.g., time series forecasting, machine learning, or queueing theory). Based on the given categories, we identify research gaps that we attempt to close with our contributions.

However, due to the lack of space, we can not go into detail about the results and instead refer to the publication itself. Instead, we will only focus on the two most interesting subgroups relevant to the contributions in this work. The *Monitorless* approach presented in Chapter 4 on page 53 and the *SuanMing* approach introduced in Chapter 5 on page 73 both fall in the category of online prediction of SLO failures as they are intended for detecting or predicting performance degradation in an online cloud environment.

However, the presented taxonomy groups *Monitorless* as a black-box approach, while *SuanMing* classifies as an architectural approach. The main reason for this is that *SuanMing* has explicit trace information from which the application architecture can be inferred, while the application-agnostic platform-level approach of *Monitorless* forbids the application of architectural knowledge during the phase of performance detection. More details on the design decisions can be found in the respective Chapters 4 and 5 on page 53 and on page 73.

The following two sections present a detailed view of the related works of the respective sub-fields. Note that the classifications presented in Sections 3.1.1 and 3.1.2 are not targeted towards following or replicating the introduced taxonomy [Gro+20a]. Instead, we believe the intricacies of the respective approaches usually lean towards a more fine-grained or accentuated differentiation of the related work. For the same reason, we discuss the related work for the two approaches in two different sections. Although there exist similarities or even overlaps between the two fields, different approaches solicit different viewpoints on the fields.

3.1.1 Application-agnostic Degradation Prediction

There are several approaches in both industry and academia for augmenting cloud operation by detecting performance saturation of software systems. While the goals of the approaches are comparable, we distinguish them by analyzing the type and amount of information that they require as the key idea of our contribution is to avoid application-level monitoring as much as possible (see Chapter 4 on page 53).

3.1.1.1 KPI-driven Solutions

The first group of approaches relies on application-specific metrics. For example, Satopaa et al. [Sat+11] require KPIs to find valid operating ranges without resource saturation. In many cases, additional metrics are required, for example, the workload type or intensity [Gan+12; Gan+14; Tru+11; Gan+19a; YUK06], or an indication of SLO fulfillment [Gma+08; FPK14]. Other proposals [Bod+09; MF10; SP19] need KPIs as input to machine learning models to predict performance degradation. Our own *SuanMing* approach introduced in Chapter 5 on page 73 can also be grouped into this category. Instead, the idea of *Monitorless* (introduced in Chapter 4 on page 53) is to infer KPI degradation without actually measuring it.

3.1.1.2 Black-box Techniques

The second group of approaches treat applications as a black-box and use either online (e.g., resource pressure models [Ngu+13]) or offline analysis (e.g., bytecode benchmarking [KKR10]) to infer performance. Kundu et al. [Kun+12] use machine learning to create performance models. Yet, these approaches only work for the trained software and need to be recreated for each target application. Emeakaroha et al. [Eme+10; Eme+12] rely on manually created mappings between low-level metrics and high-level SLAs. Wood et al. [Woo+07; Woo+09] achieve bottleneck detection based on fixed threshold rules build from a limited set of platform metrics. Cortez et al. [Cor+17] and Bianchini et al. [Bia+20] present Resource Central, their generic machine learning system able to predict different properties of VM for cloud optimization.

Similarly, commercial (e.g., Amazon Web Services (AWS)[1], Google Cloud[2], or Microsoft Azure[3]) and open-source (e.g., Cloudstack[4], or OpenStack[5]) autoscaling solutions rely only on platform metrics but require expertise to manually combine the scaling triggers properly.

[1] `https://aws.amazon.com/autoscaling`
[2] `https://cloud.google.com/compute/docs/autoscaler`
[3] `https://azure.microsoft.com/features/autoscale`
[4] `https://cwiki.apache.org/confluence/display/CLOUDSTACK/Autoscaling`
[5] `https://wiki.openstack.org/wiki/Heat/AutoScaling`

3.1.1.3 Automated and Application-agnostic Approaches

The third group consists of more sophisticated proposals to apply machine learning to application-level metrics in order to predict performance. Hence, they do not need to measure application KPIs at runtime. These proposals are applied, for example, to a web server for performance modeling during VM migrations [HT11], to a video streaming service for KPI prediction [Yan+15b; Yan+15a], or for training Bayesian networks to classify SLO compliance of web servers [Coh+04]. In contrast to *Monitorless*, there is no evidence on how generic these models are and whether they can be used autonomously across various applications and platforms.

To summarize, approaches from literature are usually tuned towards a specific application. This is done either by specifically monitoring KPIs for that application or by analyzing, adapting, or training a specific use case of that application. There is no evidence that the trained model can be transferred to prior unseen applications. In contrast, our work aims at predicting application performance without measuring KPIs or any application-level metrics while creating one prediction model transferable to a large set of different applications.

3.1.2 Explainable Degradation Prediction

In the previous section, our focus was on the type and amount of monitoring data that the respective approaches require, as this is the main limitation of our *Monitorless* approach. As the focus of our other contribution, the *SuanMing* framework, is to deliver explainable predictions, the focus of this section is now on the degree of explainability that the respective works offer.

3.1.2.1 Black-box Approaches

The first group relies on black-box techniques, for example, machine learning techniques, to detect or predict performance degradations [Lou+18; Sha+13; vOI19]. Lou et al. [Lou+18] predict the failures of cloud services with relevance vector machines using different flavors of quantum-inspired binary gravitational search algorithms. Li et al. [LLG18] postulate to incorporate information from the network, the hardware, and the software to detect failures. Sharma et al. [Sha+13] apply a multi-layered online learning mechanism for distinguishing cloud-related anomalies from application faults. Others focus on predicting CPU contentions with the use of different regression models [vOI19]. The Resource Central approach by Cortez et al. [Cor+17] and Bianchini et al. [Bia+20], as already introduced in the previous section, can also be classified into this category. Bianchini et al. [Bia+20] also acknowledge the need for debugging and explainability of machine learning models. Another work includes Seer, an online cloud performance debugging system that uses deep learning for performance prediction [Gan+19a]. Our own *Monitorless* approach can be grouped into this category as well. However, as all these techniques rely on black-box techniques, they can not deliver sufficient explanations for their predictions.

3.1.2.2 Architectural Solutions

The second group of works applies similar techniques as *SuanMing*, targeting degradation predictions using either rule-based [CH10; Gu+08; Pit+14], or architectural models [Pit+18; Moh12; Cap+13; OY16; Mar+20].

An example of a rule-based system is NETradamus, a framework for forecasting failures based on event messages proposed by Clemm and Hartwig [CH10]. The tool mines critical event patterns from logs based on past failures in order to create rules for the detection of failures in an online context. The work of Gu et al. [Gu+08] and Pitakrat et al. [Pit+14] are based on a similar idea, that is, creating rules based on log events of past failures for predicting new upcoming failures of the same type. However, as the creation of these rules requires available log data, which can not be assumed to be included in general cloud monitoring, these approaches are not viable for the application in the *SuanMing* framework.

Architectural models, for example, Pitakrat et al. [Pit+18], develop failure propagation models for incorporating the probabilities of cascading failures. Mohamed [Moh12] uses the so-called error spread signature to derive propagating failure models. Other approaches [OY16; Cap+13] propose to combine measurement from hardware and from inside the software in order to overcome the downsides of black-box failure models while keeping the monitoring overhead acceptable. Mariani et al. [Mar+20] aim at predicting failures in distributed multi-tier environments; however, they do not address microservice environments. Similar to above, these approaches generally require more intrusive monitoring infrastructures than the *SuanMing* framework.

3.1.2.3 Root Cause Analysis Techniques

The third group of works is concerned with root cause analysis and failure search for microservice applications. Jindal et al. [JPG19] propose a regression modeling technique for analyzing the capacity of each microservice, which could be used for root cause analysis, similar to our work. However, they require a test-bed for analyzing the performance of the running services. Microscope relies on causality graphs to pinpoint root causes for failures in microservice applications [LCZ18]. In contrast to our work, they introduce the concept of non-communicating dependency in their dependency graph, for example, via co-location. Other root case localization algorithms for microservices based on attributed call or dependency graphs were proposed by Wang et al. [Wan+18] and Wu et al. [Wu+20b; Wu+20a]. Zhou et al. [Zho+18] present an approach for fault analysis in microservice applications using tracing tools, while Samir and Pahl [SP19] use Hierarchical Hidden Markov Models for analyzing root causes.

While these works can deliver root causes and therefore the explanation of respective performance problems, none of these works is able to forecast a future performance degradation. Instead, these works focus on the explanation of currently occurring degradations. However, the applied techniques are still related to and relevant for the *SuanMing* architecture.

3.2 Learning Performance Model Parameters

There already exist a lot of works covering the topics of learning or extracting performance models from monitoring data. For example, Walter et al. [Wal+17], Hrischuk et al. [Hri+99], Israr et al. [Isr+05], Mizan and Franks [MF11], Brosig et al. [BKK09; BHK11], Brunnert et al. [BVK13], Willnecker et al. [Wil+15a], and Spinner et al. [SWK16; Spi+19] all propose different frameworks or approaches for the dynamic extraction of performance models from monitoring or tracing data. However, while there was already extensive work done on this topic, two areas that require specific attention are (i) the extraction of parametric dependencies and (ii) the continuous learning of resource demands.

Our first aspect of attention, parametric dependencies, describe the relationship between the input parameters of a component and its performance properties. Additionally, the importance of including such influences in performance models has been discussed by a variety of works, for example, by Woodside et al. [Woo+95], Pozzetti et al. [Poz+95], Menascé [Men97], Menascé and Gomaa [MG98], and Koziolek [Koz10]. Nonetheless, none of the above approaches is capable of extracting parametric dependencies.Section 3.2.1 discusses the related works in more detail.

Resource demands are the second central parameter of performance models that we will focus on. Extracting resource demands has been the target of many different extraction approaches. Nevertheless, there is a lack of holistic approaches, as we discuss in Section 3.2.2.

Thirdly, we will cover the topic of optimizing and selecting a set of base approaches, sometimes also referred to as meta-learning [Smi09; Ker+19]. The techniques proposed throughout this work are originating or closely related to approaches proposed in the field of machine learning and self-adaptive systems [Kou+17]. Therefore, we give a short overview of the current state of the art in Section 3.2.3. However, none of the previously developed techniques focus on learning the parameters of performance models as we do. Hence, our work serves as a proof-of-concept that these techniques can be applied in the area of learning performance models as well.

3.2.1 Parametric Dependencies

We first focus on the area of learning parametric dependencies. As already stated, the extraction of parametric dependencies has only received limited attention in research, although the importance of parametric dependencies for accurate performance predictions is widely known [Woo+95; Poz+95; Men97; Koz10; Koz08] and the manual modeling of parametric dependencies is time-intensive and error-prone [KKR10].

As such, we include all approaches explicitly dealing with the extraction of parametric dependencies in Section 3.2.1.1. In addition, we discuss the area of modeling software using statistical functions in Section 3.2.1.2. Although works from this area do not explicitly include the notion of parametric dependencies, the applied techniques are still related and are therefore also influential to our work.

3.2.1.1 Extracting Parametric Dependencies

Krogmann et al. [KKR10] perform dedicated performance experiments after instrumenting the application to monitor method call parameters and the number of executed byte-code instructions. The number of executed byte-code instructions are later mapped to resource demands for a specific system using byte-code benchmarks. Therefore, the proposed approach can not be applied at run-time, as the used byte-code instrumentation causes monitoring overheads of up to 250% [KKR10].

Courtois and Woodside [CW00] propose to use regression splines to extract parametric dependencies. They perform dedicated performance tests to obtain the data on which they fit the regression splines. This approach is not applicable to monitoring data from a running system, as there is no way to influence what monitoring data will be collected next.

The approach of Mazkatli and Koziolek [MK18] updates architectural performance models based on source code change analysis and dynamic monitoring [Mon+21]. In addition, it calibrates models incrementally with parametric dependencies [Maz+20] that can be optimized using a genetic search algorithm [Von+20]. In contrast, we do not assume source code information to be available.

Brosig et al. [BHK11] propose an approach to extract PCM [BKR09] instances based on monitoring data collected by Oracle WebLogic Server. Their approach requires information about parameter tuples for which a dependency exists as input.

To summarize, approaches from literature either require to run preliminary experiments in a testing environment [CW00; KKR10], available source code [Maz+20; Von+20], or require already detected parametric dependencies as input [BHK11]. To the best of our knowledge, there is currently no approach that can automatically identify parametric dependencies using only monitoring data from the production environment.

3.2.1.2 Modeling Software Performance Using Statistical Functions

There exists a set of approaches that model the software dependencies, like we do, using statistical functions. These functions aim to model the performance of a system (usually response time) as a function over the configuration or the input parameters of the system. Therefore, although they aim at a sufficiently different granularity than parametric dependencies, the proposed techniques are still influential for the design of our approach.

Kwon et al. [Kwo+13] derive the response time of Android applications from parameters calculated early on in the application execution. During an offline stage, an instrumented version of the application is benchmarked to determine the influence of parameters such as branch counts, loop counts, or variable values on the application response time. Thereska et al. [The+10a; The+10b] predict the performance of several Microsoft applications based on configuration and input parameters from data collected from several hundred thousand real users. The authors apply CART to filter relevant attributes, followed by a similarity search to derive performance predictions. Westermann et al. [Wes+12] analyze the suitability of Multivariate Adaptive Regression Splines (MARS), CART, Genetic Programming (GP), and Gaussian process regression for the construction of software performance models. Additionally, three different measurement point selection algorithms are evaluated, which reduce the required number of dedicated performance measurements. Noorshams et al. [Noo+13] investigate the prediction accuracy of LR, MARS, CART, and cubist (an extension of M5) for virtualized storage systems. The authors propose a general heuristic search algorithm to optimize the parameters of these regression techniques. Faber and Happe [FH12] derive software performance models with the use of genetic programming and conduct a thorough parameter optimization. The optimized GP algorithm outperformed MARS in their case study. In contrast, Velez et al. [Vel+21] present Comprex, a white-box approach to capture configuration-specific performance behavior to generate both local and global performance-influence models. Weber et al. [WAS21] present a technique for method-level white-box modeling based on profiling results.

To summarize, the presented software prediction techniques are a powerful tool to predict the response time of a system for different workloads. However, unlike architectural performance models, they can not be used to analyze the impact of changes to the system itself, such as scaling, redeployment or system evolution. We use the existing work on software performance prediction as aid to select the regression approaches we apply in Chapter 7 on page 109.

3.2.2 Resource Demands

The second important topic that we want to focus on in this work is the estimation of resource demands. As resource demands are a crucial parameter for many modeling approaches, the topic of extraction resource demands for performance models received a lot of attention in the literature. Although there exist approaches trying to directly measure the resource demand [KKR09], measuring resource demands during system operation is not feasible in most realistic systems [Spi+15] due to instrumentation overheads and possible measurement interferences. Furthermore, Willnecker et al. [Wil+15b] show that statistical estimation approaches can provide comparable accuracy to direct measurements. Therefore, many different solutions for the estimation of resource demands based on standard monitoring tools have been proposed.

3.2.2.1 Resource Demand Estimation Techniques

Spinner et al. [Spi+15] present a literature survey covering the most prominent approaches. In this work, we just give a brief overview of the most relevant techniques.

Approaches using operational laws The first idea is to approximate the resource demands using the measured response time, as the response time is usually easy to obtain. The works of Urgaonkar et al. [Urg+07], Nou et al. [Nou+09], and Brosig et al. [BKK09] propose ideas towards this approach, called Response Time Approximation (RTA). However, RTA assumes that the queuing delay is sufficiently small, which can be problematic in practical scenarios.

Therefore, other approaches propose the Service Demand Law (SDL), based on the operational utilization law [MDA04]. While Menascé et al. [MDA04] and Lazowska et al. [Laz+84] proportion the utilization for each workload class by collecting additional per-class data, Brosig et al. [BKK09] try to estimate it using the response times of the different classes.

Approaches using statistical techniques Other approaches rely on LR to perform resource demand estimation. LSQ regression can be used to estimate the unknown resource demand based on the utilization law [MDA04], called Utilization Regression (UR), as done by Pozzetti et al. [Poz+95], Rolia and Vetland [RV95; RV98], and Pacifici et al. [Pac+08].

Kraft et al. [Kra+09] also use LR but estimate the resource demands based on response times and queue length on arrival. We call this approach Response time Regression (RR). They use LSQ regression for estimating the resource demands based on response times and queue lengths on arrival. Pérez et al. [PPC13] extend this approach to other processor scheduling techniques. Others use LAD regression [Kel+06; ZCS07; SKZ07] other regression types [CC07; CCT08].

Zheng et al. [Zhe+05; ZWL08; Zhe+15] and Kumar et al. [KTZ09] apply Kalman Filters (KFs) for resource demand estimation. Kumar et al. [KTZ09] utilize the measured utilization and the arrival rate, while Wang et al. [Wan+11; Wan+12] use an alternative KF based on the utilization law [MDA04].

Approaches using optimization The third group of approaches tries to estimate the resource demands by formulating an optimization problem and minimizing the error respective error function.

If the chosen constraints are linear, the problem is solvable by quadratic programming as used by Zhang et al. [Zha+02]. Incerto et al. [INT18] also propose a linear programming technique to estimate service demands and routing probabilities by utilizing deterministic approximation [INT21]. Other approaches using non-linear optimization to minimize the error function, include Liu et al. [Liu+03], Kumar et al. [KZT09], Wynter et al. [WXZ04], and Liu et al. [Liu+06]. Liu et al. [Liu+03] and Kumar et al. [KZT09] adapt the error function to use relative instead of absolute errors. Wynter et al. [WXZ04] and Liu et al. [Liu+06] propose an approach that increases robustness against noisy measurements.

Another approach using optimization is proposed by Menascé [Men08]. Due to the non-linear structure of the objective function, the problem is not solvable with linear or quadratic programming. Therefore, Menascé [Men08] proposes a recursive non-linear optimization solver strategy.

Other approaches Finally, other approaches include the works of Sharma et al. [Sha+08] using Independent Component Analysis (ICA) and the works of Kalbasi et al. [Kal+11] using SVMs to estimate resource demands. Other machine learning techniques like clustering or change-point regression are applied by Cremonesi et al. [CDS10] and Cremonesi and Sansottera [CS12; CS14b], while Garbi et al. [GIT20] utilize recurrent NNs. Kraft et al. [Kra+09] and Pérez et al. [PPC13] propose the use of maximum likelihood estimation, while Sutton and Jordan [SJ11], Wang and Casale [WC13], and Wang et al. [WCS16] estimate resource demands with Gibbs sampling. Rolia et al. [Rol+10] and Kalbasi et al. [Kal+12] are able to estimate aggregate resource demands of a given workload mix. For more details on the individual algorithms, we again refer to the survey by Spinner et al. [Spi+15].

3.2.2.2 Approaches Combining Resource Demand Estimation Techniques

While all introduced approaches mainly focus on presenting a single approach, there already exist works aiming at providing libraries for a set of techniques. These include the Filling-the-Gap tool by Wang et al. [WPC15] and the Library for Resource Demand Estimation (LibReDE) by Spinner et al. [Spi+14]. Filling-the-Gap [WPC15] provides and compares implementations of the complete information method [PCP15], a variant of Gibbs sampling with queue lengths [Wan+12], one approach using maximum likelihood estimation based on a Markov chain representation [PCP15], one technique based on maximum likelihood estimation using a fluid approximation [PCP15], two regression-based approaches [PCP15; ZCS07], and one approach based on optimization [WXZ04; Liu+06].

Similarly, the publicly available tool Library for Resource Demand Estimation (LibReDE) [Spi+14] offers open-source implementations of currently eight different estimators:

- Approximation with response times: Response Time Approximation (RTA) [BKK09].
- Approximation using the SDL: Service Demand Law (SDL) [BKK09].

- LSQ regression based on queue lengths and response times: Response time Regression (RR) [Kra+09].
- LSQ based on utilization law: Utilization Regression (UR) [RV95].
- A KF based on utilization law: Wang Kalman Filter (WKF) [Wan+11; Wan+12].
- A KF based on response times and utilization: Kumar Kalman Filter (KKF) [ZWL08; KTZ09].
- Recursive optimization based on response times [Men08].
- Recursive optimization based on response times and utilization [WXZ04; Liu+06].

The results obtained for comparing these approaches are published in the respective study [Spi+15; Spi17]. The results show that no approach outperforms the others in all scenarios. This is in accordance with the no-free-lunch theorem [WM97] and supports the claim as well as the motivation behind our approach presented in this work. To the best of our knowledge, the only approach for the automatic selection of resource demand estimation techniques is proposed by Spinner [Spi17]. However, this approach is not designed as a continuous and self-improving learning activity. Instead, its focus is on estimating the resource demands once, using the available estimation data.

3.2.3 Optimization and Selection in Self-adaptive Systems

Although no works with a focus on resource demand estimation or dependency extraction have been proposed, the idea of continuously adapting and optimizing a system in a changing environment is not new. For example, the communities of self-aware, self-adaptive, or self-organizing systems tackle challenges of monitoring, managing, and optimizing intelligent systems in continuously changing environments [Kou+17]. In addition, hyperparameter tuning and algorithm selection is an interesting topic in the area of machine learning [Ker+19; Bie+18]. (Also refer to Section 2.3.3).

3.2.3.1 Algorithm Optimization

As such, the ideas presented throughout this work have been successfully applied to other domains. For example, Porter et al. [Por+16] present Rex, a development platform that is also able to apply online learning and optimization based on a linear bandit model. Others define self-organization or self-assembly to achieve a similar goal [RP17; EM14; KW14]. Fredericks et al. [Fre+19a; Fre+19b] present an overview of different optimization techniques in self-adaptive systems. They divide works into techniques using probabilistic, combinatorial, evolutionary, stochastic, or mathematical optimization. Additionally, we also contributed to a survey and a taxonomy for online learning of collective self-adaptive systems [DAn+19; DAn+20], where the focus is on learning, adaptation, and optimization.

Algorithm optimization is also a common topic in machine learning, either through hyperparameter tuning is also a common topic in machine learning. Therefore, a set of algorithm configuration approaches, like Sequential Model-based Algorithm Configuration (SMAC) [HHL11], or Stepwise Sampling Search (S3) [Noo+13; Noo15] have been proposed, as well as analysis and visualization tools [Bie+18]. A sub-field is also neural architecture search [EMH19], which goal is to automatically find neural network architectures and which techniques could also be applied in future work.

However, while all of the presented approaches demonstrate the feasibility of applying the proposed techniques in practice, none of these works explicitly focuses on the area of resource demand estimation or parametric dependency extraction as we do. Therefore, our contribution in respect to this field is to demonstrate and verify the applicability of continuous algorithm optimization in the specific domain of resource demand estimation.

3.2.3.2 Algorithm Selection

An orthogonal field in the context of continuous optimization is algorithm selection [Bis+16; Ker+19]. Algorithm selection [Ric76] (closely related to the field of hyper-heuristic selection [Bur+13; SHP15] or meta-learning [Smi09; Rij+14]) is defined as choosing from a set of algorithms the best for a specific problem instance, and has found many application areas in prior research [Xu+08; HRK11; Mal14; Geb+11; Lin+15; Bis+16].

However, the creation of features for the selection process is a critical task influencing the performance [Bis+16; Ker+19]. Hence, by tailoring our features to the specific task at hand, we can provide better results than generic optimization and selection frameworks. The application presented in Chapter 6 on page 91 is different from most of the proposed techniques as it offers the possibility to perform selection on continuously incoming data streams, which currently only a few works consider [Rij+14; Rij+17; Ker+19]. In addition, Chapters 6 and 7 on page 91 and on page 109 provide an application for online algorithm selection [Arm+06; GS10]. Both areas have been identified as specific research challenges by prior works [Ker+19].

Again, as no works concentrate on resource demand estimation or dependency learning, the focus of this work is to demonstrate the feasibility algorithm selection in our specific domain. However, similar to the previous section, many of the proposed techniques can be applied to our task as well in order to further improve the results presented in this work.

3.3 Modeling Distributed Cloud Database Performance

In Chapter 8, we will also cover the topic of measuring and modeling configurable and distributed Database Management Systems (DBMSs) in the cloud. Therefore, in addition to the related approaches listed in the previous section, we target three more areas in this section: (i) performance measurement of distributed DBMSs in Section 3.3.1, (ii) performance optimization of DBMSs in Section 3.3.2, and (iii) more generally the performance prediction of configurable software systems in Section 3.3.3.

3.3.1 Benchmarking and Measurement

DBMSs benchmarking is a common process to determine the performance of an operational model for DBMSs. Therefore, there exists a multitude of different benchmarks that support diverse workload models and evaluation objectives [SD17; Ren+17]. In consequence, there are numerous supportive performance studies that focus on cloud-hosted DBMS available [Coo+10; KKR14; Hen+18]. Yet, each of these studies covers only a small part of our scope.

The *Mowgli* framework by Seybold [Sey17] and Seybold et al. [Sey+19] enables a holistic approach to automated benchmark setup, measurement, and tear-down while enabling evaluation of non-functional features under runtime and resource constraints, specifically targeted at cloud providers. We, therefore, also build upon the *Mowgli* tool in our contribution. However, all works in this section solely focus on benchmarking or measurement of distributed DBMS and therefore do not consider the modeling perspective.

3.3.2 Modeling and Optimization

DBMSs performance optimization approaches such as ITuned [DTB09], DB-Sherlock [YNM16], and OtterTune [Van+17] target single instance relational DBMSs that are operated on dedicated resources. These approaches have a special focus on the workload by considering trace-based workloads [DTB09; YNM16] or unknown workload types [Van+17]. Rafiki [Mah+17] targets the performance optimization of single instance Not only SQL (NoSQL) DBMSs for different workload types by automatically determining DBMS runtime parameters and deriving their optimal configuration. While these approaches provide comprehensive performance prediction mechanisms for single node DBMSs, they neither consider aspects of distributed DBMSs nor the volatility of cloud resources.

Performance models for distributed DBMSs are, for example, presented by Farias et al. [Far+18], Dipietro et al. [DCS17], and Xiong et al. [XYD19], considering the performance prediction impact of different cluster sizes [Far+18] and DBMS specific runtime parameters [DCS17; XYD19].

Finally, the URSA framework [Zhe+19] targets the automated capacity planning of a single node DBMS operated on cloud resources. Thus, the focus of URSA lies on the cloud resources, while the aspects of distributed DBMSs are not considered.

In summary, existing approaches provide comprehensive performance prediction mechanisms for single node DBMS on dedicated resources [DTB09; YNM16; Van+17; Mah+17], focus on distribution aspects [DCS17; Far+18; XYD19] without considering cloud resources or consider only cloud resources without considering DBMS distribution aspects [Zhe+19]. In contrast, our work aims at specifically addressing the modeling and optimization challenges of distributed and cloud-hosted DBMSs and therefore tries to close this research gap. The respective contribution is described in Chapter 8 on page 137.

3.3.3 Prediction of Configurable Systems

The third group of related approaches is generally focused on modeling and predicting the performance of general configurable software systems.

Zhu and Liu [ZL19] propose ClassyTune, a system for tuning software systems in cloud environments. Singh et al. [Sin+16] optimize the performance of object-relational mapping frameworks, mapping database operations onto high-level APIs, by using a multi-objective genetic algorithm. Zhang et al. [Zha+15] propose the application of Fourier learning to predict the performance of configurable systems with theoretical accuracy guarantees. Siegmund et al. [Sie+15] combine machine learning and sampling heuristics to build performance-influence models for highly configurable systems. Sarkar et al. [Sar+15] compare different sampling techniques for CART-based performance models and introduce a novel heuristic for the selection of the initial samples. Guo et al. [Guo+17] improve the CART-based performance model by resampling the training data to determine the accuracy of the resulting model and automated hyperparameter tuning. Ha and Zhang [HZ19] propose a deep sparse neural network architecture and hyperparameter optimization approach for the performance prediction of configurable systems. Nardi et al. [NKO19] introduce HyperMapper 2.0, a multi-objective optimization framework with support for unknown feasibility constraints, as well as categorical and ordinal parameters. Westermann et al. [Wes+12] compare the accuracy of MARS, CART, GP, and Kriging for the construction of software performance models. Similarly, Noorshams et al. [Noo+13] evaluate the accuracy of LR, MARS, CART, M5 Trees, and Cubist Forests for the performance modeling of storage systems. Other white-box approaches [Vel+21; WAS21] build models based on detailed profiling of the software artifact to predict the configuration-specific performance behavior. The latter approaches have also been discussed in Section 3.2.1.2.

However, none of these approaches are specifically considering distributed and cloud-hosted DBMSs. To the best of our knowledge, no approach exists that addresses measurement variability during sampling or specifically targets distributed DBMSs as we do.

Part II

Contributions

Chapter 4

Detecting Resource Saturation

In this chapter, we present *Monitorless*, an approach for the application-agnostic prediction of resource-based performance saturation. In the same way that the Serverless Computing [Eis+21b] paradigm allows the execution environment to be fully managed by the cloud provider, we discuss a *Monitorless* model to take care of application monitoring. We show that training a machine learning model with platform-level data, collected from the execution of representative containerized services allows inferring application KPI degradation. This is an opportunity to simplify operations, as engineers can rely solely on platform-level metrics to configure portable and application-agnostic rules to automatically trigger actions such as autoscaling, instance migration, network slicing, etc.

Recent research work aimed at automating performance analysis and orchestration typically relies on specific application KPIs measurements [Igl+17; TAL15; Ven+16; Yan+15b] (see Section 3.1 on page 35). However, we argue that application-specific metrics limit the generality of solutions and their applicability across applications and platforms as application-level metrics are not always available [Bia+20; Cor+17]. That is, operation teams need to proficiently evaluate and control critical KPIs and keep track of their specific target operating range for each running application, which reduces the agility of deployment.

Therefore, we leverage machine learning to loosen the dependencies between operation tasks and application-specific KPIs. Intuitively, the idea is to use historical data from various and commonly used services, labeled with information about bottlenecked resources and hardware configuration, to infer service degradation without the need to specifically monitor or analyze any KPIs during production. Thus, our approach uses only a standard set of application-agnostic, hypervisor-level, platform metrics (e.g., CPU and RAM utilization) to infer KPI degradation. These metrics are autonomously processed by a machine learning algorithm to detect performance issues without the need to explicitly monitoring them. Therefore, we call our approach "*Monitorless*", as we avoid the monitoring effort for application-level metrics.

The reasoning behind *Monitorless* is motivated by 1) new software architectural patterns such as microservices that allow building applications from a collection of loosely coupled and fine-grained services; 2) cloud commodity hardware that allows for a more concrete and homogeneous set of platform-level metrics; and 3) the fact that application KPIs are directly related to the usage of underlying platform resources (physical or virtual), and finding this relation is a solvable machine learning task. This way, we solve **RQ I.1** (*"How can platform-level measurements be utilized to detect resource saturation?"*). In our proof-of-concept, *Monitorless* relies on a binary classifier that works with a large set of features derived from combinations of platform-level metrics. The classifier is trained with data obtained during the execution of typical microservices, widely used by application developers (e.g., databases, load balancers, messaging, etc.), monitored as they experience resource saturation as well as in normal operation.

We introduce the *Monitorless* framework and show that one resource saturation model allows for detecting performance degradation of several complex applications, even when such applications are unknown to the trained model. Therefore, we answer **RQ I.2** (*"How can we generalize the results to create a generic and holistic prediction model?"*). We note this represents a significant divergence between *Monitorless* and other solutions based on KPIs as we propose a generic approach with a single resource saturation model working for a heterogeneous set of different applications. In contrast, other machine learning solutions are based on training models with specific application data known beforehand or rely on KPIs that are not transferable and generic. This is the target of **Goal I** (*"Design an application-agnostic approach for the detection of resource saturation based on platform-level monitoring data."*).

In summary, the key contributions are the following:

- A methodology for creating the appropriate feature set for the robust operation of cloud environments driven by generic platform data. Features are selected according to the Utilization, Saturation, Error (USE) method [Gre13] and extracted with the Performance Co-Pilot[1] tool.
- A pipelined architecture for model training and validation, as well as the components required to infer performance degradation without monitoring application KPIs, and to integrate the *Monitorless* model into the cloud orchestrator.

[1]`https://pcp.io`

Moreover, *Monitorless* can be used as a building block that enables black-box services and applications deployed on a cloud platform to be autonomously managed. In particular, since our framework can be used to detect performance degradation in a service-agnostic way based on generic platform-level metrics, it can be used as a basis for autoscaling and consolidation decisions, as well as performance bottleneck analysis. Thus, we posit that in many cases *Monitorless* can eliminate the need for (i) dedicated tests to analyze performance behavior and (ii) specialized online service-level monitoring of KPIs. The long-term goal is to have one globally applicable model for predicting resource-based service degradation, applicable for the vast majority of operating service instances, without the need to retrain the model for every application. This work was conducted during a research visit at Nokia Bell Labs in Dublin, Ireland. We later also published the developed approaches and the respective results [Gro+19c].

The rest of this chapter is organized as follows. Section 4.1 presents an overview of the *Monitorless* architecture and design, whereas Section 4.2 elaborates on the feature selection and the methodology used to create a robust machine learning model. Finally, we conclude in Section 4.3. Experimental results for *Monitorless* are discussed later in Chapter 9 on page 151.

4.1 *Monitorless* Design

Figure 4.1 illustrates the components of *Monitorless* in a simple deployment with two physical nodes running three applications (indicated with different colors), each composed of one or many instances of different microservices. *Monitorless* introduces two key components: (i) a monitoring agent deployed on each cloud node and (ii) the orchestrator function serving as a centralized repository for data collection and training. Both components can be integrated into an existing cloud environment to augment its functionality.

The monitoring agents run on each physical host to collect a set of predefined platform-level metrics using standard monitoring tools. Examples of these metrics include CPU usage, RAM usage, and throughput of I/O-devices; all of them are measured at the OS-level and include hypervisor information, for example, namespace and cgroups interfaces for Linux containers. The orchestrator periodically receives metrics from the agents. First, the data collected is used to make performance predictions at each service instance to detect resource saturation; see Section 4.2 for more details on the prediction model. Second, the orchestrator infers the overall application performance from the predictions at individual containers composing the application. Third, based on the inference, the orchestrator can decide to change the current deployment

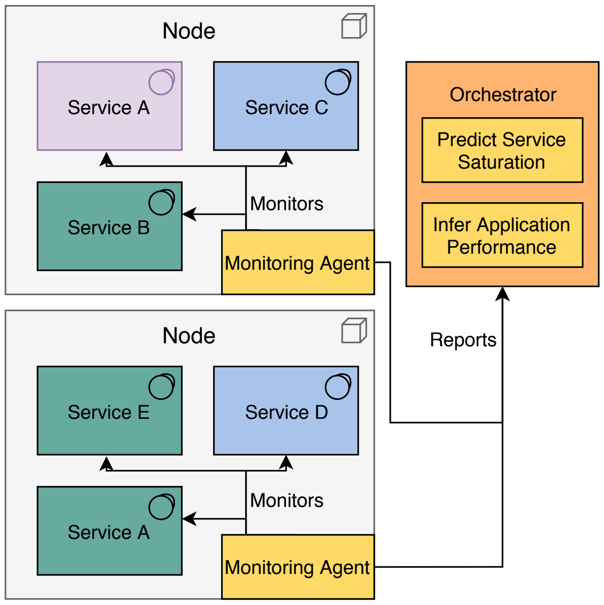

Figure 4.1: Overview of the *Monitorless* components.

by migrating or scaling applications automatically as remediation for performance problems. The orchestrator operates online, repeating the steps above at regular intervals.

Next, we describe the nomenclature used throughout this chapter in Section 4.1.1, the methodology proposed for detecting saturation in Section 4.1.2, and a formal definition of the machine learning problem *Monitorless* is designed to solve in Section 4.1.3.

4.1.1 Nomenclature and Definitions

A *cloud* is a set of connected nodes $\mathcal{C} = \{c_1, \dots, c_{|\mathcal{C}|}\}$ in which multiple applications are executed, and where each *node* $c \in \mathcal{C}$ is a computing entity able to host service instances in a virtualized environment (as VMs or Linux containers). Consequently, $|\mathcal{C}|$ is the total number of nodes in the cloud $\mathcal{C}$. An *application* $\mathcal{A} = \{\mathcal{S}_1, \dots, \mathcal{S}_{|\mathcal{A}|}\}$ is composed of interconnected services running in the cloud, where each *service* $\mathcal{S} = \{\mathcal{I}_1, \dots, \mathcal{I}_{|\mathcal{S}|}\}$ consists of one or more service instances. These *service instances* are functionally identical but can differ in terms of which node they are assigned to, effectively determining the type and amount of available resources. Thus, each service instance $\mathcal{I}$ is assigned to run inside its own virtual environment on exactly one node c of the cloud environment. The parameter $|\mathcal{A}|$ describes the number of services within application $\mathcal{A}$ and $|\mathcal{S}|$ is the total number of service instances of service $\mathcal{S}$.

At each point in time t, the monitoring agents collect a set of $k_{\mathcal{H}}$ *host metrics*, $\mathcal{H}_{c,t}$, as well as a set of $k_{\mathcal{V}}$ *virtual metrics*, $\mathcal{V}_{\mathcal{I},t}$, for each service instance $\mathcal{I}$ running on node c. Moreover, at each point in time t, it is possible to observe a KPI value $\mathcal{P}_{\mathcal{A}}(t)$, which represents the performance of application $\mathcal{A}$ at time t. Each service instance $\mathcal{I}$ is associated with one set of host metrics $\mathcal{H}_{c,t} \in \mathbb{R}^{k_{\mathcal{H}}}$ as well as one set of virtual metrics $\mathcal{V}_{\mathcal{I},t} \in \mathbb{R}^{k_{\mathcal{V}}}$ for a given time t, assuming that $\mathcal{I}$ was running at time t. We denote the vector representing the concatenation of metrics $\mathcal{H}_{c,t}$ and $\mathcal{V}_{\mathcal{I},t}$, as $\mathcal{M}_{\mathcal{I},t}$.

4.1.2 Labeling Resource Saturation

We consider *service* instances (i.e., microservices composing the application) to be either *saturated* or *non-saturated* at a given time and define the saturation state of an *application* $\mathcal{A}$ at time t according to its $\mathcal{P}_{\mathcal{A}}(t)$. We use application KPIs only for labeling data to train our models. Note that such KPIs are not required for using the resulting model. Depending on the application, examples of KPIs can be response time or throughput of a web service, jitter in a video streaming application, or availability indicators in communication systems. In the following, we present a methodology for labeling throughput. However, this step can be analogously applied to similar KPIs; for other more sophisticated KPIs, manual modeling might be necessary. Note that this labeling is only required during the training of different applications and implies the abstraction from the KPIs of the used application. Therefore, this abstraction of different KPIs enables *Monitorless* to act application-agnostically. The actual training is done using labeled (and therefore KPI-independent) historical data from a set of selected representative applications without assuming that the target application is also used for training. Although for common use cases (e.g., autoscaling), binary classification is enough, note that one can also apply more complex state descriptions based on multiple classes.

In practice, finding resource saturation is difficult as metrics can have non-deterministic or noisy behavior (see observed throughput (blue dots) in Figure 4.2), important points in the dataset can be missing, or the sampling frequency is too low. Usually, an application serving increasing workloads shows a proportional increase in throughput until a saturation point is reached, from which a non-linear behavior is expected (see the elbow around 700 requests/sec in the smoothed curve (orange line) of Figure 4.2). This behavior can be correlated to other KPIs; for example, the response time would rapidly increase

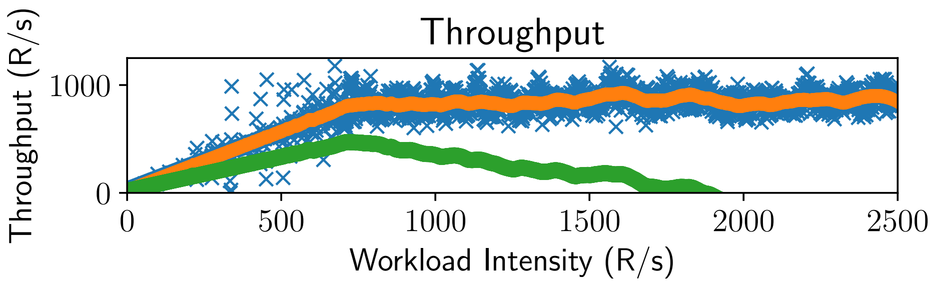

Figure 4.2: Observed throughput, smoothed curve, and calculated differences of an example load test.

when the throughput reaches the nonlinear region. Alternatively, there would be an increase in the number of dropped requests.

To label the dataset properly, it is critical to ensure that the non-linear behavior of KPIs is due to resource saturation when detecting elbows or knees in KPIs. Although there is no mathematically unique "elbow" in an exponential curve, we use the definition of Satopaa et al. [Sat+11] based on curvature and employ their Kneedle approach to find it.

Therefore, to create training and test data, we linearly increase the workload of a given target application $\mathcal{A}$. While applying the workload, we monitor the KPI $\mathcal{P}_{\mathcal{A}}(t)$. Relating this KPI to the workload intensity for time step i defines a discrete function f with $f(\alpha_i) = \beta_i$, where α_i is the workload intensity and β_i the corresponding KPI. Our implementation of Kneedle to label the training and test data is outlined below:

1. Smooth f by applying a Savitzky-Golay filter [SG64] (orange curve in Figure 4.2) or any similar filter.
2. We normalize the points of f to the unit square by setting:
   $\alpha_i \leftarrow (\alpha_i - \min_j\{\alpha_j\})/(\max_j\{\alpha_j\} - \min_j\{\alpha_j\})$, and
   $\beta_i \leftarrow (\beta_i - \min_j\{\beta_j\})/(\max_j\{\beta_j\} - \min_j\{\beta_j\})$.
3. We calculate the differences between β_i and α_i by setting: $\beta_i \leftarrow \beta_i - \alpha_i$. This yields the green curve in Figure 4.2.
4. The set of candidate saturation points are the local maxima of this curve defined by the differences above, and we manually choose the local maximum (such as in Figure 4.2), defining the corresponding β_i as our threshold Υ.

Hence, we obtain a function $\tilde{\mathcal{P}}_{\mathcal{A}} : \mathbb{R} \to \{0, 1\}$, with

$$\tilde{\mathcal{P}}_{\mathcal{A}}(t) = \begin{cases} 0 & \text{if } \mathcal{P}_{\mathcal{A}}(t) \leq \Upsilon\,, & (no\ saturation) \\ 1 & otherwise. & (saturation) \end{cases}$$

The function f above is assumed to have positive concavity. If the opposite is true, the same technique can be applied by setting $\beta_i \leftarrow \max_j(\beta_j) - \beta_i$ and $\alpha_i \leftarrow \max_j(\alpha_j) - \alpha_i$. We note that Savitzky-Golay filters have tunable parameters that should be adjusted depending on input data. However, its purpose is to provide a smoothed curve from which we can identify a knee or elbow in a reproducible manner, as we are applying machine learning algorithms to the function f. Thus, the procedure above need not be fully automated for arbitrary input data, and, indeed, we recommend that f is visually inspected as a sanity check.

4.1.3 A Machine Learning Problem

The training dataset $\mathcal{T}$ is built from multiple instances running various services on different deployments and serving diverse workloads to increase the model robustness. We define the machine learning problem as follows. $\mathcal{T}$ is partitioned in many subsets $\mathcal{T}_{\mathcal{I}}$ for each instance $\mathcal{I}$ from which we generated labeled data. Each $\mathcal{T}_{\mathcal{I}}$ is a set of pairs $\mathcal{T}_{\mathcal{I},t} = \{(\mathcal{M}_{\mathcal{I},t}, \tilde{\mathcal{P}}_{\mathcal{A},t})\}$. The vector $\mathcal{M}_{\mathcal{I},t}$ represents the system state of service instance $\mathcal{I}$ at time t, and $\mathcal{H}_{c,t} \subset \mathcal{M}_{\mathcal{I},t}$ are the platform metrics obtained from each host. Instead, $\mathcal{V}_{\mathcal{I},t} \subset \mathcal{M}_{\mathcal{I},t}$ are metrics specific to the service running in the instance $\mathcal{I}$. In the case of a Linux container, this could be the CPU time relative to the allocated maximum. Thus, multiple containers running on machine c at time t share the same feature values for $\mathcal{H}_{c,t}$ but have different values for $\mathcal{V}_{\mathcal{I},t}$. Note that this architecture enables the model to handle any variable number of service instances on different hosts.

Then, solving the machine learning problem consists of (i) processing $\mathcal{M}_{\mathcal{I},t}$ first to extract feature vectors $x_{\mathcal{I},t}$ and (ii) training a binary classifier mapping $x_{\mathcal{I},t}$ to $y_{\mathcal{I},t} = \tilde{\mathcal{P}}_{\mathcal{A},t}$.

4.2 Modeling Process

This section provides implementation details of the prototyped *Monitorless* binary classifier.

4.2.1 Metric Collection

The USE method [Gre13] allows for detecting performance bottlenecks by examining the **U**tilization, **S**aturation, and **E**rrors of each relevant platform resource. We include as much of the USE metrics as possible to detect resource bottlenecks without incurring monitoring overhead.

For collecting such metrics, we use the PCP monitoring tool[0], which provides a wide set of platform metrics while being lightweight. PCP gathers usage and saturation indicators of processors (incl. virtual processors), memory, network and storage devices, controllers and buses, as well as interrupts and network errors. As a preprocessing step, metrics reporting counters must be converted into rates, and utilization metrics to a relative scale, for example, a percentage value. These steps are necessary to avoid overfitting our model to a particular hardware configuration or system state. See Section 4.2.3 for more details. Measurements are extracted every second, a default interval of PCP at the time of writing. This is a reasonable sampling time as it enables the algorithms to react quickly and start/stop Linux containers that usually instantiate within a few seconds [XFJ16]. As the framework is not conceptually dependent on PCP, other platform-level monitoring agents could be applied as well.

4.2.2 Training Data

The idea of the *Monitorless* model is to classify service performance without any prior knowledge. For this purpose, it is necessary to create training data using services with different resource utilization patterns, performance demands, and traffic intensity. In this work, we create training data using a small set of services widely used by application developers. Ideally, we would choose a few dozen different services, in different configurations and deployments, running on different machines, and with many hours of measurement data. However, as our goal is to present a proof-of-concept rather than a mature software product, we believe that this small set is enough to provide a convincing proof-of-concept. However, note that adding a set of more diverse services during the model training can help to further improve the robustness and accuracy of the prediction model.

4.2.2.1 Training Services and Applications

We use three different applications for training: Solr[2], Memcached[3], and Cassandra[4]. We select these applications for two reasons: i) they are representative services in Cloudsuite [PSF16], and ii) they show different resource usage profiles and, therefore, different resource bottlenecks. To improve the reproducibility of our results, we rely on the Cloudsuite benchmarking tools to run these services [Fer+12a; PSF16]. Generally, the goal of the training application set is to include a wide variety of different and representative resource usage scenarios applied in practice.

Solr Apache Solr is an enterprise search platform. We use an index size of 12 GB of content crawled from the Internet that comes with Cloudsuite and the HTTPLoadGenerator [KDK18] to generate HTTP varying workloads as specified by Load Intensity Modeling Tool (LIMBO) [Kis+17]. The individual response times and failed request rates are logged every second to label the training data.

During the load generation, clients send search requests with a range between one and five terms. Each term is randomly selected from the top 10,000 most frequent words in the index. The server outputs a list of the ten most relevant documents according to these terms. Our hardware configuration allows the index to fit into the main memory, thus eliminating page faults and minimizing disk activity [Jan+10]. With this configuration, the benchmark is mostly CPU-bound. In addition, however, we also conduct experiments with different configurations in which the container resources were limited in the server to alter such CPU-bound behavior.

Memcached Memcached is a distributed memory object system, usually used to alleviate database loads via caching. Again, we used the load generator provided by Cloudsuite that uses a 10 GB Twitter dataset to populate the cache. It then applies a constant target throughput with a configurable get/set rate parameter. Memcached is configured to be memory-bound. Thus, in our setup, we constrained CPU resources in the containers just for one dataset and changed the memory configuration to either 8 GB, 4 GB, or unlimited.

Cassandra Apache Cassandra is a scalable NoSQL database system. We use Yahoo! Cloud Serving Benchmark (YCSB) [Coo+10] to generate database

[2]`https://solr.apache.org`
[3]`https://memcached.org`
[4]`https://cassandra.apache.org`

workloads. The database is populated with 30 million records, consuming about 30 GB (plus additional indexing and log files). Several constant target loads are applied by changing the number of client threads and the required target throughput. As Cassandra stresses several resources, we can tune it to be either CPU-bound or disk-bound. One setting with 20 cores and a 30 GB RAM limit creates an I/O bottleneck, while a setting with six cores and unlimited memory creates a CPU bottleneck.

4.2.2.2 Generated Datasets

Each application is monitored under different workloads and resource constraints. They run both in isolation, as the only instance running on the host, as well as in combination with other instances. The purpose of the latter is to create a model robust to interference caused by resource sharing in the physical host. Table 4.1 lists all the configurations used for training, the benchmarked services, including CPU and RAM limits (CPU, RAM), whether they ran in parallel (Par), the traffic pattern (Traffic), and the resource bottleneck (Bottleneck). A dash indicates no container limitation for the respective resource or no parallel executions. All training experiments are conducted on HP ProLiant DL380 Gen9 servers provisioned with a 48-core Intel® Xeon® CPU E5-2680 v3 @ 2.50 GHz processor and 125 GB of memory, connected by a 10 GBit/s switch and running CentOS 7.3. with Docker 17.06.1-ce and PCP 3.12.1.

The master orchestrator collects throughput, response times, and platform metrics generated throughout the execution of the specified workloads. An additional experiment with linearly increasing load is conducted in order to determine the threshold value Υ, defined in Section 4.1.2. After acquiring this threshold, the labeling (saturated or non-saturated) of samples is performed as described in Section 4.1.3.

If a test runs in parallel with other tests to create interference, we indicate this in the Par column. For example, Solr (test 3) was run in parallel with Cassandra (test 18). For Memcached, we show the minimum and maximum request rates used. Cassandra workloads can be divided into four different classes: A, B, D, and F. These correspond to core workloads available in YCSB. A is an update-heavy workload (Read/Write: 0.5/0.5), B is read-heavy (Read/Write: 0.95/0.05), D is constantly inserting records and reading the most recent, and F reads a record, modifies it, and then writes it. For Solr, we have two workload curves generated by LIMBO. The first (`sin-1000`) is a simple sine function with a minimum request rate of 1 and a maximum request rate of 1 000 requests per second. The second (`sin-noise-1000`) has the same base structure but was massively modified by adding random noise to increase variability. For the

Table 4.1: List of created training datasets.

#	Service	CPU, RAM	Par	Traffic	Bottleneck
1	Solr	3/–	–	sin-1000	Container-CPU
2	Solr	–/–	–	sin-1000	Host-CPU
3	Solr	–/8 GB	18	sin-noise-1000	I/O-Bandwidth
4	Solr	–/8 GB	19	sin-noise-1000	I/O-Bandwidth
5	Solr	3/8 GB	20	sin-noise-1000	I/O-Bandwidth
6	Solr	1.5/8 GB	22	sin-noise-1000	Container-CPU
7	Memcached	–/–	–	2K – 50K R/s	RAM-Bandwidth
8	Memcached	1/–	–	20K – 85K R/s	Container-CPU
9	Memcached	–/8 GB	–	39K – 45K R/s	I/O-Queue
10	Memcached	–/4 GB	23	10K – 65K R/s	I/O-Queue
11	Cassandra	–/–	–	A: 30K – 100K R/s	Network-Util.
12	Cassandra	–/–	–	B: 20K – 70K R/s	Host-CPU
13	Cassandra	–/–	–	D: 40K – 90K R/s	Network-Util.
14	Cassandra	20/30 GB	–	A: 300 – 1200 R/s	I/O-Bandwidth
15	Cassandra	20/30 GB	–	B: 100 – 900 R/s	I/O-Bandwidth
16	Cassandra	20/30 GB	–	B: 700 – 1000 R/s	I/O-Bandwidth
17	Cassandra	20/30 GB	–	B: 100 – 1000 R/s	I/O-Bandwidth
18	Cassandra	6/–	3	A: 15K – 25K R/s	Container-CPU
19	Cassandra	6/–	4	B: 10K – 15K R/s	Container-CPU
20	Cassandra	6/–	5	D: 10K – 25K R/s	Container-CPU
21	Cassandra	6/–	–	A: 5K – 20K R/s	Container-CPU
22	Cassandra	6/–	6	B: 5K – 20K R/s	Container-CPU
23	Cassandra	6/–	10	B: 10K R/s	Container-CPU
24	Cassandra	1/–	–	F: 200 R/s	I/O-Wait
25	Cassandra	1/–	–	F: 20 R/s	I/O-Wait

sake of simplicity, we describe the used workload patterns in a very concise matter. However, we configured the different runs such that the training sets capture as many different contention scenarios as possible.

Furthermore, we give an indication of the limiting factors and critical metrics in each dataset. If two datasets run in parallel, the goal is to learn also the isolation effects, that is, how resource sharing impact multiple running applications.

4.2.2.3 Iterative Improvement of the Training Set

We repeated the tests outlined in Table 4.1 multiple times, iterating over the various stack configurations to improve the datasets with more representative use cases stressing most of the resources in the platform. As a result, we devised the following structure to create a robust and accurate model.

1. Normalize our training data and save the normalizing instance. We use the `MinMaxScaler` in Scikit-learn [Ped+11].
2. Use the normalizing instance to analyze with a validation dataset. This is done by scaling the validation set with the known scaler instance. If any feature has a maximum or minimum value outside the scaling range of the trained scaler, we know that this feature was not sufficiently trained.
3. Analyze the features not covered by the training set and decide if they are critical for the model performance.
4. Design additional training cases that include other feature values in the training set and perform the additional measurements.
5. Add the training set to the others and repeat from step 1 in order to validate that the training has now improved.

4.2.3 Feature Selection and Optimization

We collect 1 040 platform metrics using the PCP monitoring tool as described in Section 4.2.1. Of these collected platform-level metrics, 952 consider the host and 88 are specific to service instances (i.e., containers) running on the host. As expected, not all the metrics are relevant for the machine learning model, and in many cases, metric preprocessing is required such that they can be useful or leveraged by the algorithm. Next, we describe the preprocessing performed on these raw metrics.

4.2.3.1 Binary Features

CPU and RAM utilization are important indicators of saturation; thus, in order to improve the accuracy of the model, we introduce three additional boolean (i.e., hot-encoded) features for both of these metrics. Namely, `LOW` indicates whether utilization is under 50%, `MED` indicates whether utilization is in the 50% – 80% range, and `HIGH` indicates whether it is above 80%. For CPU utilization, we also introduced two other boolean features called `VERYHIGH`, indicating utilization above 90%, and `EXTREME` indicating utilization above 95%. These new metrics are inserted both for the host and for the container-specific metrics resulting in a total of 16 additional binary features.

4.2.3.2 Scaling

We scale all metrics having units in byte-values like KB or MB, and which are not convertible to a relative scale. For example, the number of bytes read by an I/O-device where the maximum capacity is not known is transformed to a logarithmic scale. The goal of this transformation is to improve accuracy by emphasizing the magnitude rather than a specific value and thus reduce hardware dependency. Nevertheless, these metrics are still prone to overfit to a specific hardware configuration as they are not on a normalized, and therefore portable, scale and need to be handled carefully.

4.2.3.3 Normalization

As the maximum values of some features are undefined, we opt to artificially limit the possible values to a range. We used the `StandardScaler` of Scikit-learn to transform feature values such that their distribution has a mean value of zero and a standard deviation of one. Hence, each monitored value would have the sample mean value subtracted and then divided by the standard deviation of all existing samples.

4.2.3.4 Filtering with Random Forest or PCA

We use the Random Forest (RF) algorithm [Bre01] to filter the most relevant metrics, as it allows for simple computation of the information gain of each feature and ranks them by importance. We trained the RF on each of the datasets shown in Table 4.1 and took the union of the top 30 most important features of each dataset. Below the top 30, the algorithm assigns features a weight lower than $1/\#Features$. This union set consists of 117 unique features.

We make three observations about this filtering step. The first is that the filtering does not decrease the cross-validation accuracy, which implies that resource saturation is detectable by looking at a small set of platform metrics; this finding is consistent with related work [Coh+04]. Second, while there is overlap between most top 30 feature lists (e.g., CPU utilization), there are also metrics that are specific to one dataset. This encourages the inclusion of many different training applications to stress different platform resources in the future. Third, the binary features for memory were filtered out, whereas the binary features for CPU were selected as highly important. The reasons for this can be: (i) the training set has not enough memory saturation samples, (ii) the binary labels are not actually required for training because memory problems are likely to be preceded by other symptoms (e.g., disk usage because of page thrashing), or (iii) the binary labels are not required because memory measurements are more insightful with the real values (i.e., non-binary).

An alternative method for feature selection is the PCA [Pea01], which provides a linear transformation to obtain orthogonal features (see Section 2.3.3.2 on page 33). PCA can reduce the number of features in large spaces. The side effect is that the transformed features no longer correspond to physical magnitudes, which makes it difficult to fully interpret the resulting model. Using the Scikit-learn implementation of PCA, we reduced the number of features to 50, which accounts for 99.99% of the variability in the data.

4.2.3.5 Time-dependent Features

Metrics are collected for a fixed time window of one second. However, time-dependent features can give some insight and reflect certain dynamics of performance. For example, having a low CPU utilization during the past 15 seconds and a peak at the present time might not actually imply that the resource is saturated. On the contrary, if CPU utilization is high for the last 15 seconds, then the platform is more likely to be experiencing resource saturation.

Based on this intuition, we create the X-AVERAGED and X-LAGGED variants of each metric. X-AVERAGED takes the average over the last $X+1$ samples (seconds), including the current one, while X-LAGGED contains the value of the metric X samples ago. This helps to include some context in an otherwise isolated one-second snapshot of the platform. We include X-AVERAGED and X-LAGGED for each metric with values $X = 1, 5, 15$. A window of 15 seconds proved to be sufficient in our experiments regarding the applications we considered.

4.2.3.6 Combining Features

It is a common practice in data science to create new features by multiplying existing features, as this often improves the models significantly by revealing relationships that are not visible by simply analyzing linear combinations of the features separately. In our case, we multiply all pairs of features from different domains (e.g., CPU and RAM), but in order to prevent an explosion of the size of the feature set, we omit all time-dependent features from this step. This refinement turned out to be crucial to capture performance problems across features, that is, problems that are detectable by observing combinations of metrics (see Section 4.2.5 for more details).

4.2.3.7 Pipeline Optimization

To guarantee the right order and configuration, as well as to streamline the execution of the aforementioned steps, we have implemented a pipeline. The pipeline has five steps:

1. Create binary features and scale required features.

2. Normalize the features.

3. Apply a first reduction step (either filtering or PCA).

4. Create new time-dependent and multiplicative features.

5. Apply a second reduction step (either filtering or PCA).

6. Remove features with a variance of 0 (provide no information).

We perform a grid search on steps 2 to 5 in order to find the best combination of features. Step 3 and step 5 can either do filtering, PCA, or none of them, while step 2 and step 4 can perform optional normalization and feature addition.

The grid search is performed with the training set described in Section 4.2.4, applying the same cross-validation scheme. We again use a Random Forest (RF) algorithm with default parameters as a prediction algorithm to evaluate the individual feature engineering steps. We exclude the combination of not applying a first feature selection in step 3 and then adding the multiplicative features in step 4 since this is practically unfeasible due to the huge number of resulting features.

4.2.4 Training the *Monitorless* Model

Combining all datasets from Section 4.2.2 and the features described in Section 4.2.3, we get a total training set of 63 086 samples with 4 492 features each. The portion of saturated examples in the training set is 26%. For algorithm training, we choose to compare six different classifiers.

1. Binary Logistic Regression (Logit) modeling [Cox58] using the Stochastic Average Gradient (SAG) solver [SLB17],
2. Support Vector Classification (SVC) based on LIBLINEAR [Fan+08],
3. AdaBoost [FS97] with CART [Bre+84],
4. XGBoost [CG16],
5. a three-layer (106 neurons), fully connected, feed-forward Neural Network (NN) [Gur97], and
6. RF [Bre01].

Note that we use a linear kernel for SVC as any other kernel function increased the algorithm training time significantly.

We use the Python implementations of XGBoost [CG16], Keras [Cho+15] with a TensorFlow backend [Aba+16] for the NN, and Scikit-learn [Ped+11] for all other algorithms. We perform five-fold cross-validation with a grid search optimizing the $F1$ score to select the hyperparameters of each algorithm. The five-fold cross-validation partitions the training sets from Table 4.1 using 20 sets for training and 5 sets for validation in the fold. By partitioning the training sets in this way, rather than using the union of all training sets, we aim to avoid overfitting. Table 4.2 lists the parameters considered during the hyperparameter grid search, with bold parameter values as the chosen ones for all considered algorithms. The naming of the parameters follows the convention of Scikit-learn.

Table 4.3 lists the training times, the per-sample classification time, and the $F1_2$ score of our first validation set. Note that detailed results, including a proper definition of the used metrics and a description of the used scenario, are included later in Chapter 9 on page 151. The results presented here just provide a preliminary analysis of the individual performances of each algorithm. We utilize the results in order to define the Random Forest (RF) approach as our machine learning algorithm of choice.

We observe that Random Forest (RF) outperforms all other approaches with an $F1$ score of 0.99. There is a clear trade-off between training and prediction

Table 4.2: Examined grid search parameter space for each applied algorithm.

Algorithm	Parameters	Values
Logit	C	**0.01**, 0.1, 1
	tol	**0.1**, 0.01, 0.001, 0.0001
	class_weight	**balanced**, None
SVC	C	0.1, 1, **10**
	tol	**0.01**, 0.0001, 0.00001
	penalty	**l1**, l2
	class_weight	**balanced**, None
AdaBoost	n_estimators	**50**, 250, 500
	algorithm	**SAMME**, SAMME.R
	DT_criterion	gini, **entropy**
	DT_splitter	random, **best**
	DT_min_samples_split	**5**, 10, 20
XGBoost	min_child_weight	1, **4**, 16, 64
	max_depth	1, 4, 16, **64**
	gamma	**0**, 1, 4, 16
NN	activation_function1	**softmax**, relu, sigmoid, linear
	activation_function2	**softmax**, relu, sigmoid, linear
	activation_function3	**softmax**, relu, sigmoid, linear
RF	n_estimators	**250**, 500, 1000
	min_samples_leaf	**5**, 10, 20, 30
	min_samples_split	5, 10, **20**, 30
	criterion	gini, **entropy**
	class_weight	**balanced**, subsample, None

time, on the one hand, and prediction accuracy, on the other hand. However, even for the slowest algorithm (RF), the prediction time was still only 40 ms per prediction on average. This is sufficiently fast to make such predictions online in production. Interestingly, we found that Logit, Neural Network (NN), and AdaBoost only predict the majority label and, hence, a classifier that predicts all samples to be saturated would receive the same score. While SVC does make some correct predictions, it does poorly overall and achieves an even lower score than simply predicting the majority label. The high $F1$ score of 0.997 can be attributed to the number of training samples, including many clearly saturated or non-saturated examples. Note that all training and classification times given in Table 4.3 exclude the feature extraction process described in

Table 4.3: Performance of the applied algorithms.

Algorithm	Training Time	Classification Time	$F1_2$
SVC	837.8 s	0.2 ms	0.579
Logit	3.1 s	0.2 ms	0.858
AdaBoost	250.7 s	0.4 ms	0.858
NN	76154.9 s	4.1 ms	0.858
XGBoost	5140.3 s	6.3 ms	0.944
RF	68.3 s	40.6 ms	0.997

Section 4.2.3 since the required time is the same for all algorithms and only accounts for roughly 28 ms per prediction on average.

Based on these results, we select the model created by Random Forest (RF) for the evaluation of *Monitorless* in the next section. The hyperparameter tuning of RF resulted in 250 trees, trained with 20 samples in a leaf node using the information gain as the splitting criterion and applying no weights to the different classes (see Table 4.2).

4.2.5 Comparing the Model Behavior with Expert Decisions

Finally, in this section, we analyze the trained model focusing on the filtered features to provide insight from a system perspective and compare the outcome to expert-made decisions. For this, we use the feature importances as given by the trained RF model as RF provided the best results and is comparatively easy to interpret. Table 4.4 shows the 30 most important features based on the trained RF model. The $\times$-symbol denotes a multiplication of two features. The features S-MEM-U, C-CPU-MEDIUM, C-CPU-HIGH, C-CPU-VERYHIGH are derived relative utilizations, the suffix HIGH and VERYHIGH are binary features. An AVERAGED-k or LAGGED-k denotes that the feature was averaged or lagged by k seconds. All other parameters follow the nomenclature of the PCP monitoring tool.

First, we observe that the combination of features is a beneficial step as almost all used features are multiplications of two original metrics. Most of them are the multiplication of CPU-level metrics with a metric of another resource, for example, CPU-HIGH multiplied with various network or RAM-related features. Different metrics like RAM utilization are also included but with a lower ranking and are therefore not contained in the table. Intuitively, this means that service saturation is captured by analyzing more than one resource type at the same time. Additionally, lagged and average features are occasionally

Table 4.4: Top 30 features sorted by importances assigned by Random Forest.

Feature name
network.tcp.currestab × C-CPU-HIGH
hinv.ninterface × C-CPU-VERYHIGH
kernel.all.pswitch-AVERAGED14
mem.vmstat.nr_inactive_anon × C-CPU-VERYHIGH
network.tcp.currestab × C-CPU-VERYHIGH
network.tcpconn.established × C-CPU-HIGH
C-CPU-HIGH
network.sockstat.tcp.inuse × C-CPU-VERYHIGH-AVERAGED14
C-CPU-VERYHIGH × C-CPU-VERYHIGH
network.sockstat.tcp.inuse × C-CPU-VERYHIGH
cgroup.cpusched.periods × C-CPU-HIGH
C-CPU-VERYHIGH
C-CPU-SUPERHIGH-AVERAGED14
C-CPU-HIGH-AVERAGED4
mem.vmstat.nr_kernel_stack × C-CPU-VERYHIGH
cgroup.cpusched.throttled × C-CPU-VERYHIGH
kernel.all.nprocs × C-CPU-HIGH
hinv.ninterface × C-CPU-MEDIUM
C-CPU-SUPERHIGH-LAGGED15
S-MEM-U-mapped × C-CPU-VERYHIGH
C-MEM-U-usage × C-CPU-HIGH
cgroup.cpusched.throttled × C-CPU-HIGH
C-CPU-VERYHIGH-AVERAGED4
C-CPU-HIGH × C-CPU-VERYHIGH
vfs.inodes.free × C-CPU-VERYHIGH
mem.vmstat.pgpgin × C-CPU-HIGH
mem.vmstat.nr_inactive_file × C-CPU-VERYHIGH
vfs.inodes.free × C-CPU-HIGH
disk.all.aveq-AVERAGED4
S-MEM-U-active_file × C-CPU-VERYHIGH

used. In fact, raw (un-engineered) metrics are rather low-rated and are used seldom. However, the chosen features heavily depend on the CPU utilizations (in various levels), the number of network connections, the disk queue, the memory utilization, the number of throttled cgroup periods, etc. All of these metrics intuitively make sense to a system engineer. Hence, we conclude that the *Monitorless* model itself does not behave very differently from any human performance engineer regarding the set of analyzed metrics. Nonetheless, there are also some interesting combinations chosen by the system which are surprising and not directly obvious. Furthermore, no human can monitor all of the considered metrics in parallel. Therefore, although the performance predictions of the RF algorithm are reasonable and comprehensible to a human expert, they are not reproducible by a human at the given scale and rate.

4.3 Summary

In this chapter, we introduce *Monitorless*, a method for inferring application KPI degradation in the cloud. The differentiating feature of *Monitorless* is that only platform-level metrics are required as input instead of application-specific metrics, allowing for accelerated software onboarding and feature releases. Therefore, one defining RQ of this chapter is **RQ I.1** (*"How can platform-level measurements be utilized to detect resource saturation?"*). We address **RQ I.1** by proposing a binary classifier that leverages machine learning to bridge the gap between platform-level and application-level monitoring. Furthermore, *Monitorless* answers **RQ I.2** (*"How can we generalize the results to create a generic and holistic prediction model?"*), by proposing diversified applications and feature engineering methods to generalize the training dataset. Using this generalized dataset, a holistic model can be created that works on the individual service level, and therefore is independent of the application structure.

With *Monitorless*, operation engineers can rely on a generic and stable set of metrics to streamline testing without application expertise or handcrafted configurations. Hence, *Monitorless* represents our solution that achieves **Goal I** (*"Design an application-agnostic approach for the detection of resource saturation based on platform-level monitoring data."*). This information can be used for fast resource provisioning and allocation as *Monitorless* can detect such bottlenecks in a matter of seconds. Our model is designed to be compatible with current cloud automation practices and easy to integrate. In Chapter 9 on page 151, we evaluate the performance of *Monitorless* on different microservices applications with varying complexity and levels of interference.

Chapter 5
Predicting Performance Degradation

In this chapter, we present *SuanMing*, our solution for predictive APM for microservice applications in the cloud. We already established that microservice applications [Fow15; LF14] are increasingly seen as the main architectural paradigm for developing medium and large cloud applications [Lin16; Dra+17; Gaj+20]. While a microservice architecture offers clear advantages for developing and operating an application, the increase in the number of individual components also increases the perceived complexity of the respective system [Gan+19b; Fow15; Eis+20a]. Therefore, operators and performance engineers increasingly rely on APM tools to supervise the operation of an application [Jam+18; Faz+16]. There exist several APM tools and services that allow the collection, processing, and analysis of performance metrics of cloud-based applications (see Section 2.2.1 on page 19).

However, such tools are limited to reactive performance management; that is, performance degradations can only be detected and addressed after they occurred in the system. This leads to unavoidable quality of service degradation, which negatively impacts user experience and revenue [Gan+19a; Ela18; Ein19]. Therefore, we envision a proactive APM tool capable of predicting performance degradations before they occur.

To that end, we surveyed approaches targeted towards online failure prediction [Gro+20a]. We found that previous approaches either require intrusive monitoring not available in low-overhead APM tools or lack the means to provide sufficient explanations of the prediction. More details on the respective approaches and their shortcomings are discussed in Section 3.1.2.

Therefore, we target **Goal II** (*"Develop an approach for the prediction of performance degradation using application-level tracing."*) by formulating the research questions **RQ II.1** (*"How can tracing data be utilized to predict the future performance of a system?"*) and **RQ II.2** (*"How can we pinpoint the root cause service of a performance problem?"*) to address these properties. We introduce *SuanMing*, our framework for enabling explainable performance predictions for microservice applications running in cloud environments.

SuanMing utilizes tracing data commonly available in APM tools to learn three different models: (i) a forecasting model predicting the user behavior; (ii) a propagation model inferring the behavior of each user request in an application; and (iii) a performance prediction model predicting the performance of services and back-propagating its effect on other dependent services. Based on the model predictions, *SuanMing* is able to forecast the future state of the application and provide an explanation by pinpointing the respective root cause service. For this, *SuanMing* utilizes the increased granularity of microservice applications, as this enables a divide-and-conquer strategy for the performance predictions without the need for extensive and intrusive monitoring.

The contribution of this chapter is two-fold:

- We introduce *SuanMing*, an approach for predicting performance degradation of microservice applications based on the propagation of internal requests and the back-propagation of service performance, enabling the pinpointing of a root cause service.

- We present an abstract formalization of the involved prediction tasks to enable a modular architecture of the proposed approach.

Operators can use *SuanMing* as a plugin to their already configured and running monitoring stack to augment the reactive capabilities of their APM tools with a predictive and proactive component that is able to determine and consequentially avoid performance degradations before they occur. Due to the modular and highly configurable approach of the framework, operators can fine-tune the sensitivity of the approach based on their specific needs. In contrast to related work, *SuanMing* requires no additional application data for delivering predictions. Instead, all information is extracted from the APM tool, and models are continuously updated. We published our contribution together with our colleagues from the Huawei Research Center in Tel Aviv, Israel [Gro+21d]. Furthermore, we published the analysis of the related work as a separate taxonomy [Gro+20a] and created a replication package of our experimental results [Gro+21e].

In the remainder of this chapter, we introduce the general *SuanMing* framework in Section 5.1. We detail the individual components of the architecture in Section 5.2 and summarize our contribution in Section 5.3. We provide an experimental evaluation in Chapter 10 on page 171.

5.1 Overview

Our approach is based on two key observations in microservice architectures. First, we assume that the performance of microservices is only dependent on the type and amount of requests arriving at each particular service instance. This is based on the assumption that microservice designs should be mostly stateless [Eis+20a; KDK18] and that all state information is transmitted using the request itself.

Second, a service usually has a limited set of responsibilities [Jam+18; LF14]. Subsequently, a single user request usually causes a sequence of internal requests to other microservices in order to realize complex application behaviors. This enables us to split the prediction of large and complex applications into multiple smaller tasks that are individually solvable as the small-scoped services have a predictable performance behavior. The propagation of requests and their resulting performance are also predictable by tracing request call trees.

We use a divide-and-conquer approach. First, we predict the user requests and their propagation through the application. Then, we use the fine-granular analysis of the individual services to infer the whole application performance and additionally pinpoint the location of the performance degradation without the need for in-depth monitoring data.

5.1.1 Terminology

Before we describe our approach in more detail, we introduce some important definitions which we use during the individual component descriptions. In the following, the term *service* represents an application component, which is stateless and has a clear-scoped functionality, for example, a microservice. A service always consists of one or multiple *endpoints*. An endpoint is an interface (or workload class) of a service, which can be called by users or other services, for example, REST endpoints. *User requests* are requests that enter the system from outside the monitored environment. On the contrary, *requests* are calls that were issued by other entities (i.e., other services) from inside the system. A *backend service* is a service, which does not issue requests to services and responds to incoming requests only. A *frontend service* is a service responding to user requests.

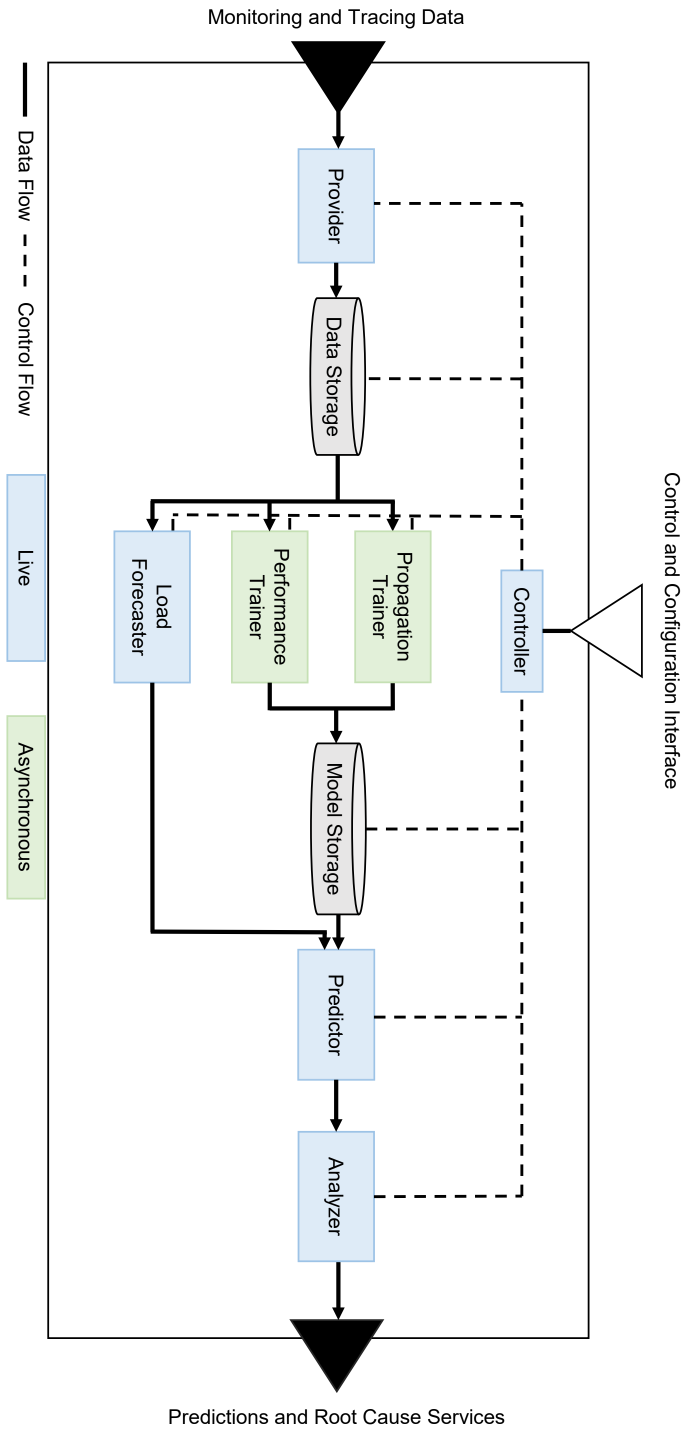

Figure 5.1: Architecture overview of the *SuanMing* framework.

5.1.2 Architecture

Figure 5.1 presents our reference architecture containing the different components of *SuanMing*. The structure enables us to separate the learning of the models and the actual prediction into parallel executable online processes. Additionally, due to the modular structure, it is possible to modify and exchange individual algorithms of the framework without affecting the performance of the other components.

The *Controller* serves as a central synchronization component, responsible for updating times, configurations, and activities. Next, the *Provider* collects and parses incoming data and stores it into a uniform format using the data storage component. Subsequently, the gathered monitoring information is fed into the *Propagation Trainer* and *Performance Trainer* components. Both modules train a prediction model and store it in the model storage. In contrast, the *Load Forecaster* directly produces a forecast of the expected number of incoming user requests, which is forwarded to the Predictor. The *Predictor* is the central component responsible for predicting the performance of each service using the load forecast together with the trained propagation and performance models. Finally, the *Analyzer* compares the predictions with the user-given goals in order to alert and pinpoint any anticipated performance degradation.

Figure 5.1 depicts all components in blue that are required for live execution. As the propagation and performance model trainers store the finalized models using the model storage, the creation of these model instances can happen asynchronously, depicted as green. In the following, Section 5.2 explains each component in more detail.

5.2 Components

In this section, we provide a detailed description of all individual components involved in the architecture depicted in Figure 5.1. Section 5.2.1 details the Provider component, while Section 5.2.2 focuses on the Forecaster. Following, we introduce the propagation and performance trainers in Sections 5.2.3 and 5.2.4, respectively. The Prediction component combines all trained models to predict actual predictions in Section 5.2.5. Finally, we introduce the Analyzer component in Section 5.2.6.

5.2.1 Provider

SuanMing requires two types of training data. First, we need information about the chain of internal requests issued to process an incoming user request. This is required to extract the application architecture as well as to forecast the number of requests arriving at every service using the propagation model.

Second, performance metrics for every service must be available in order to train the performance models and to conduct predictions for these monitored values. Depending on the availability, more metrics influencing the service performance, including parameter values, deployment, and co-locations, or hardware specifications, can be added optionally. However, the target performance metrics (e.g., response times) are required as labels for the performance model training algorithms.

Both types of information are usually contained in call traces, call stacks, or call trees. Therefore, the required monitoring and tracing data is obtainable in many state-of-the-art tracing tools, like Jaeger, Zipkin, Pinpoint, Dapper [Sig+10], or Kieker [HWH12] (see Section 2.2.1 on page 19). The current implementation supports the formats of Zipkin, Pinpoint, and the proprietary format used in Huawei Cloud (see the evaluation in Chapter 10 on page 171).

5.2.2 Load Forecaster

The Load Forecaster component is responsible for forecasting the number of user requests in the next prediction period. The forecast result is represented by a tuple $U \in \prod_{i=1}^{s} \mathbb{R}_{\geq 0}^{m_i}$, where each entry $u_i \in \mathbb{R}_{\geq 0}^{m_i}$ represents the number of user requests to the m_i endpoints of service i. As our list S of observable services contains $s = |S|$ services, U contains s vectors. Summed over all elements, U contains m entries, where $m = \sum_{i=1}^{s} m_i$ is the total number of application endpoints.

However, the problem can be simplified by isolating each request type of U as a univariate time series and forecasting the expected requests independently. As there already exist many works capable of forecasting user behavior (e.g., Herbst et al. [Her+17], Bauer et al. [Bau+20c], and Bauer [Bau21]), we rely on GluonTS [Ale+20] in this work as it offered the best prediction performance in our prior analysis. However, due to the modular approach of *SuanMing*, this component can be seamlessly exchanged for another forecasting approach. As a side contribution, we also helped develop an automated forecasting framework [Bau+20a] and a benchmark suite for forecasting techniques [Bau+21], which can be utilized in future work.

5.2.3 Propagation Trainer

The Propagation Trainer component is responsible for learning the propagation model describing how many additional internal inter-service requests are needed to process an incoming user request.

5.2.3.1 Formalization

We formalize the request propagation using the *propagation matrix* D.

$$D = \begin{pmatrix} d_{1,1} & \cdots & d_{1,s} \\ \vdots & \ddots & \vdots \\ d_{s,1} & \cdots & d_{s,s} \end{pmatrix},$$

where each entry $d_{i,j} : \mathbb{R}^{m_i}_{\geq 0} \rightarrow \mathbb{R}^{m_j}_{\geq 0}$ in D represents a function mapping a vector of incoming requests at service i to a vector of requests to service j sent by service i. Hence, both input and output of each propagation function $d_{i,j}$ are vectors containing scalars greater or equal to zero. The dimension of each vector is determined by the variable m_i, which represents the number of endpoints of service i.

Note that our model is able to assess the distribution of calls across the different endpoints of each service. This is useful for performance predictions, as different endpoints might have different performance metrics. For example, a web server might take longer to show the dynamic login page than to deliver a static index page. In total, the matrix D has s^2 entries overall, where s represents the number of services in the application.

The application topology modeled by D can also be visualized as a directed graph. Each node represents a service, and an edge from node i to node j represents service i sends requests to service j. In a graph representation, the edges between the nodes are associated to the propagation functions stored in D. Consequently, we can associate paths in this graph with composite propagation functions. For example, a path $\gamma = (i, j, k)$ from i over j to k corresponds to the composite function $d_\gamma = d_{j,k} \circ d_{i,j}$ using the composition operator[1] $\circ$. A propagation model is called *acyclic*, if for every cycle $\gamma = (i, j, k, \ldots, z, i)$ at least one of the associated component functions $d_{i,j}, d_{j,k}, \ldots, d_{z,i}$ is the null function. We define a *null function* as a function that always returns zero. Hence, the topology graph omitting all edges associated with null functions must be acyclic. Analogously, a propagation model is called cyclic, if at least one cycle exists, where none of the associated component functions is the null function.

A propagation model is called *regular*, if for every cycle γ and every input vector x the sequence

$$d_\gamma(x)^n = \underbrace{d_\gamma \circ d_\gamma \circ ... \circ d_\gamma}_{n},$$

[1]Let f and g be functions, then the composite function $g \circ f$ is defined as $g \circ f = g(f(x))$.

converges to zero for n tending to infinity. Such models represent topologies, where limited cycles but no infinite loops exist. This means that, after a certain number of iterations, the application always terminates. All acyclic models are regular by definition. Our framework assumes that all valid propagation models are regular models.

5.2.3.2 Model learning

The task of the Propagation Trainer is to learn the propagation matrix D using historical data. To build this model iteratively, we initialize all functions $d_{i,j}$ as null functions, representing no dependency between the services i and j. After each observation period, we update each function $d_{i,j}$, where a request between service i and service j has been recorded. This step can be done by analyzing the average behavior of the call tree of each request. The model learning still continuously updates the propagation model whenever new monitoring data becomes available in order to react to changes in application or user behavior.

5.2.4 Performance Trainer

As the final learning component, the performance trainer utilizes the information of the individual requests at each service endpoint to predict the performance of the individual endpoints and the resulting performance of the overall system. We assume that the performance is mainly dependent on the number and type of incoming requests to the service. This is a simplification made possible by the assumption of statelessness of modern microservice architectures [Eis+20a; KDK18]. If all microservices follow this requirement and are stateless, the performance of each request only depends on the availability of resources, which is dependent on the number and type of other requests arriving at the same processing resource and the parameters of each request itself.

5.2.4.1 Formalization

The performance of a service i is described using the performance vector $p_i \in \mathbb{R}^{m_i \cdot r}$. To describe the performance of a service or endpoint, we use r different performance metrics of interest. These could be, for example, the average response time or the number of exceptions. These metrics are calculated for all m_i endpoints of service i. Consequentially, the service performance vector p_i of service i has $m_i \cdot r$ entries overall.

We obtain the requests arriving at i from the tuple of requests $X \in \prod_{i=1}^{s} \mathbb{R}_{\geq 0}^{m_i}$ calculated by the forecasting and the propagation model. Additionally, the framework allows adding additional features of arbitrary type α, for example, the number of incoming requests on co-located or influencing services, the parameter distribution of the given requests, the measured resource utilization, a priority vector, etc. It is then dependent on the chosen modeling type, how the given auxiliary information is utilized. These auxiliary metrics α can be forecast using the standard load forecasting component also utilized for forecasting the request numbers (see Section 5.2.2) or other specialized forecasting or prediction engines. As the type and number of the auxiliary features are dependent on each service, α_i refers to the set of auxiliary features for service i.

For adding additional important factors of the service performance, we consider the position of the service in the topology graph. We define z_i as a variable, which represents for a given service i the sum of all endpoints of all services, receiving calls from service i. A backend service only responds to incoming requests. Therefore, for a backend service b, $z_b = 0$. We assume that its performance p_b only depends on the number of incoming requests x_b and the set of auxiliary metrics α_b. Therefore, the performance function $f_b : \mathbb{R}_{\geq 0}^{m_b} \times \mathbb{R}^{|\alpha_i|} \to \mathbb{R}^{r \cdot m_b}$ maps $m_b + |\alpha_b|$ input values to $r \cdot m_b$ performance metrics. For all other services i, we assume that the performance p_i additionally depends on the performance measures of all endpoints answering calls from i. Note that we do not make any distinction between synchronous or asynchronous calls. It is up to the performance function to determine the actual impact of each influencing service.

To summarize, for each service i there is a performance function $f_i : \mathbb{R}_{\geq 0}^{m_i} \times \mathbb{R}^{r \cdot z_i + |\alpha_i|} \to \mathbb{R}^{m_i \cdot r}$ that maps the requests tuple $x_i \in \mathbb{R}_{\geq 0}^{m_i}$, $r \cdot z_i$ additional influencing service performances, and $|\alpha_i|$ auxiliary input values to the predicted service performance $\tilde{p}_i$. We define $F = (f_1, \ldots, f_s)$ to be the tuple containing all s performance functions of an application.

5.2.4.2 Model learning

For learning performance models, historical training data is required. Hence, we have a training set L_i, consisting of t individual measurement periods for each service i. The combination of all L_i is the set of the total available training data L. Each scenario contains a tuple as one input of $m_i + r \cdot z_i + |\alpha_i|$ performance-relevant features, together with r measurable performance metrics for each of the m_i endpoints of service i. Therefore, L_i is a tuple of two matrices, each consisting of t measurement rows. The first matrix contains $(m_i + r \cdot z_i + |\alpha_i|)$

performance-relevant features, while the second matrix contains $(r \cdot m_i)$ target performance measurements in each row.

Due to the formalization and the modular design of *SuanMing*, we support multiple possible performance modeling and learning techniques. In this work, we focus on black-box machine learning algorithms as they offer the most transferable and domain-independent performance. However, other approaches can be integrated as well, should the need arise for different or more expressive models. For example, one could integrate the presented resource demand estimation techniques presented in Chapter 6 or the dependency detection mechanisms of Chapter 7 to model the performance of the system.

In the current implementation, we approximate the performance functions in F using supervised machine learning algorithms to predict the r measurable performance metrics. Consequentially, we can train regression models, which approximate the performance function list F (see Section 2.3). It is also possible to train different prediction models for each performance metric. As the performance functions might be subject to change, we update F on a regular basis using measured performance data. Using this black-box modeling type, the additional performance indicators α, like deployment information or hardware specifications, can be easily added as no semantic meaning needs to be provided. This is advantageous, as we can not assume to have prior knowledge about the application available. Hence, all performance indicators that may seem relevant for the prediction of any of the r targeted performance metrics and that can be reliably monitored can be easily integrated into the performance predictor functions f_i.

5.2.5 Predictor

The Predictor can be seen as the main component of the *SuanMing* framework. It combines the user requests forecast U with the propagation model D and the performance prediction model F for predicting the future state of the system.

5.2.5.1 Request propagation algorithm

The first step is to calculate the number of requests arriving at each service in the given time period and, hence, propagate the user requests through the system. Given the service dependencies captured by D and the predicted user requests U, we want to predict how the requests are forwarded through the application. We propose an iterative algorithm that works with any regular model D.

Algorithm 5.1: Request propagation.

```
Input  : Propagation matrix $D$, user requests $U$, threshold $\epsilon$, number of
         services $s$, number of endpoints per service $(m_1, \ldots, m_s)$
Output : Total requests $X$.
1  $X = U$
2  $\bar{X} = U$
3  while $\bar{X} \neq (0_{m_1}, \ldots, 0_{m_s})$ do
4      $\hat{X} = (\hat{x}_1, \ldots, \hat{x}_s) = (0_{m_1}, \ldots, 0_{m_s})$
5      foreach $\bar{x}_i$ in $\bar{X}$ do
6          if $\bar{x}_i \neq 0_{m_i}$ then
7              foreach $d_{i,j}$ in $i$-th row of $D$ do
8                  $\hat{x}_j = \hat{x}_j + d_{i,j}(\bar{x}_i)$
9      $X = X + \hat{X}$
10     $\bar{X} = \hat{X}$
11     Set all numerical entries in $\bar{X}$ lower than $\epsilon$ to 0
12 return $X$
```

Algorithm 5.1 takes the user request tuple U, generated by the forecasting engine, as an input and returns the request tuple $X \in \prod_{i=1}^{s} \mathbb{R}_{\geq 0}^{m_i}$, which contains the total number of predicted incoming requests in the next prediction period for all services and endpoints. The tuple X can be seen as the sum of the incoming user requests U and their generated internal calls. Hence, at the start of Algorithm 5.1, the values of U are assigned to X. Then, the algorithm iteratively calculates the resulting internal calls starting in line 3. The tuple $\bar{X}$ represents the requests that need to be forwarded in the current iteration of the algorithm. Once the loop terminates, there are no more requests to forward, and we can return the total tuple of requests X.

In the inner loops of line 5 and line 7, line 8 evaluates the propagation functions $d_{i,j}$ for each service i and all target services j if i forwards requests. The newly generated internal requests are stored in the temporary support variable $\hat{X}$, which gets filled with zeros at the beginning of each iteration (line 4). After iterating all services, the newly generated requests $\hat{X}$ are added to the total numbers of requests X and need to be considered for the next iteration. Hence, $\bar{X}$ gets set as $\hat{X}$. As already stated earlier, requests do not need to be integers, as a service can also call another service with a probability of, for example, 80%. Therefore, functions $d_{i,j}$ are defined on non-negative real numbers for input and output.

To prevent an infinite loop, the threshold parameter ϵ is used. It represents a lower bound, which is applied to $\bar{X}$. All numerical entries which are smaller than ϵ are set to 0. This guarantees termination of Algorithm 5.1 for regular propagation models as it can deal with cyclic topologies as long as cycles send fewer requests than received per iteration. Additionally, Algorithm 5.1 is easy to parallelize, as the only synchronized calls are the commutative addition in line 9 and the assignment in line 10. This enables high scalability, even for increasing topology sizes, as they can be processed independently.

However, if D is linear and acyclic, we can improve the scalability of the request propagation algorithm even more. A propagation model D is considered *linear*, if every propagation function $d_{i,j}(x)$ within D can be written in the form of $d_{i,j}(x) = c_{i,j} \cdot x$ with $c_{i,j} \in \mathbb{R}^{m_j \times m_i}$ being a constant matrix. As the composition of linear functions is itself linear, it follows that for every path γ, the associated composite propagation function is also linear.

Additionally, if a service i receives calls from multiple origins, the total number of incoming requests is the sum of all inbound request flows. Hence, the vector of incoming requests x_i can be written as the sum of multiple linear propagation functions. We further know that the resulting request tuple X is the sum of the user requests U and the internal calls, while every internal call is originated by a user request. From these properties combined with the acyclicity of the propagation model, it follows that every vector x_i can be written as a linear combination of the entries u_i of the user requests U. With that, we are able to calculate every x_i using s matrix multiplications:

$$x_i = \sum_{j=1}^{s} A_{j,i} u_j,$$

where $A_{j,i}$ is the matrix of all summarized request propagations from service j to service i and u_j is the j-th element of the user requests tuple U. Therefore, for linear and acyclic propagation models, the iterative calculations from Algorithm 5.1 can be summarized into s^2 independent matrix multiplications, followed by s^2 (asynchronous) additions. Depending on the size of an application topology, this might further improve the computation effort and, hence, the scalability of the request propagation model learning.

5.2.5.2 Performance inference algorithm

Finally, after we predicted the number of requests at each service i, the performance prediction algorithm infers $\tilde{p}_i$ for the given state. Algorithm 5.2 iteratively calculates the performance $\tilde{p}_i$ for each service i by starting with all backend services and going backward through the application topology. It requires the performance model $F = (f_1, \dots, f_s)$, containing all performance inference functions learned in Section 5.2.4, the total requests for each service $X = (x_1, \dots, x_s)$, and the list of additional metrics for each service $A = (\alpha_1, \dots, \alpha_s)$. It is assumed that A is known using forecasting techniques or historical measurements of the available variables and can be treated analogously to X.

Algorithm 5.2: Performance inference.

```
Input  : List of services $S$, performance model $F$, tuple of request
         vectors $X$, list of additional metrics $A$.
Output : Predicted application performance $\tilde{P}$.
1  $F, S, X, A = \text{resolveCycles}(F, S, X, A)$
2  $\tilde{P}, S' = \emptyset$
3  while $S' \subsetneq S$ do
4      foreach $i \in S \setminus S'$ do
5          Let $d_i$ be the set of dependencies of service $i$
6          if $d_i \subseteq S'$ then
7              $\tilde{P}_i = \{\tilde{p}_j \in \tilde{P} \mid j \in d_i\}$
8              $\tilde{p}_i = f_i(x_i \cup \alpha_i \cup \tilde{P}_i)$
9              $\tilde{P} = \tilde{P} \cup \tilde{p}_i$
10             $S' = S' \cup i$
11 return $\tilde{P}$
```

In line 1, Algorithm 5.2 first ensures that the application is acyclic and then continues to initialize the performance vector P and the set of processed services S'. Following, it iterates through all services until S' is no longer a proper subset of S. That is, until every service has an associated performance prediction. For an unprocessed service i, we calculate the list of required performance values $\tilde{P}_i$ that are necessary for processing that service's performance prediction function f_i in line 7 based on the set of services d_i that i depends on. In line 6, it is determined whether $\tilde{P}_i$ can be calculated, that is, whether all influencing service metrics are already available. If so, the performance inference model is queried and stored in line 8 and line 9, and the service i is added to the list of processed services. Similar to Algorithm 5.1, the calculation of the individual performance metrics is highly parallelizable for large topologies, improving the scalability of *SuanMing*.

Note that for any backend service b, $d_b = \emptyset$. Therefore, as $\emptyset \subseteq S'$ is always true, including $\emptyset \subseteq \emptyset$ in the first iteration, all backend services are added in the first iteration of the loop, as their performance only depends on the input requests X and the additional metrics A. Hence, for acyclic graphs, Algorithm 5.2 is guaranteed to terminate. For regular and other cyclic models, Algorithm 5.2 does not terminate as all services of the cycle can not be calculated as long as their influencing services are not known.

Consequentially, it is necessary to refer to a heuristic capable of resolving cycles in the application model in line 1 of Algorithm 5.2. One possible heuristic sets the performance values of all services of the cycle to ∞. This results in false positives; however, all affected services will be post-processed in Algorithm 5.3, which is capable of solving cycles and, therefore, filtering false positives. Other possible heuristics include ignoring the dependency generating the fewest calls for a given input list or contracting all affected services into one hyper-service. While the former might reduce the prediction accuracy of individual services, the latter lacks the granularity to pinpoint a specific service.

5.2.6 Analyzer

The final component focuses now on the analysis of the predictions produced in the previous steps. Therefore, the target of the Analyzer component is to classify the severity of the predicted performance problems and to deliver explanations for the performance prediction in order to foster actionable insights. For example, two services a and b might experience performance degradations as their response times increase. However, as service a calls service b while answering its request, its performance is only degraded due to the time service a waits for service b. Therefore, by addressing the performance degradation of b, we automatically also solve the performance degradation of a.

In our case, the system operator defines target ranges for every performance metric of user endpoints in advance. *SuanMing* is then tasked with supervising these ranges and alerting the operator if one value is predicted to exceed its target range in the near future. This translates into a binary classification problem. The classification is done using the predicted performance vector $\tilde{p}_i$ for every service i and comparing it to the defined target threshold t_i for each of the r performance metrics. The approach of binning performance metrics into different classes was already shown to be suitable for related performance problems [Bia+20].

However, as *SuanMing* focuses on explainable predictions, a simple problem classification does not suffice. Therefore, if a service is expected to perform worse than the defined threshold, the analysis component needs to pinpoint the service responsible for the anticipated performance problem in order to offer solutions on how to avoid it. Hence, Algorithm 5.3 calculates and returns the list of root cause services R, responsible for the predicted performance problem. Based on the computation of R, an operator can specifically target all necessary services, for example, by up-scaling all services in R to avoid the predicted performance problem while still using minimal resource effort.

Algorithm 5.3: Root cause inference.

```
Input  : Predicted performance $\tilde{P}$, performance thresholds $T$,
         performance model $F$, input requests $X$, additional metrics $A$.
Output : Predicted application performance $\tilde{P}$, List of root cause
         services $R$.
1  $R = \emptyset$
2  foreach $\tilde{p}_i \in \tilde{P}$ do
3      if not $\text{satisfies}_{t_i}(\tilde{p}_i)$ then
4          Let $\tilde{P}_i$ be the set of predicted dependent performance metrics of
           service $i$
5          $\tilde{P}_i' = \bigcup_{\tilde{p}_j \in \tilde{P}_i} \min(\tilde{p}_j, t_j)$
6          if not $\text{satisfies}_{t_i}(f_i(x_i \cup \alpha_i \cup \tilde{P}_i'))$ then
7              $R = R \cup i$
8  if not $\text{confident}(\tilde{P}, R)$ then
9      return $\emptyset, \emptyset$
10 return $\tilde{P}, R$
```

Similar to Algorithm 5.2, Algorithm 5.3 requires the performance model F, the tuple of input request vectors X, and the list of additional metrics A. In addition, Algorithm 5.3 utilizes the performance predictions $\tilde{P} = (\tilde{p}_1, \ldots, \tilde{p}_s)$, that is, the output of Algorithm 5.2, and the defined performance thresholds $T = (t_1, \ldots, t_s)$ for each service.

Algorithm 5.3 utilizes the helper function $\text{satisfies}_{t_i}(\tilde{p}_i)$ to check whether a performance prediction $\tilde{p}_i$ of service i violates any of its defined thresholds t_i. The function returns `false` if any of the r components in the predicted performance vector $\tilde{p}_i$ is higher than its defined threshold t_i, otherwise it returns `true`. That is, $\text{satisfies}_{t_i}(\tilde{p}_i)$ performs an element-wise greater-than comparison.[2] If

[2]Without loss of generality, we assume that a threshold is always an upper bound for what values are acceptable. If a threshold is set as a lower bound, we negate the value and all of its predictions.

a service i is detected to violate any of its performance thresholds t_i in line 3, we calculate the *all-fine* performance vector $\tilde{P}_i'$ for service i in line 5. This is done by lowering all threshold-exceeding values to the defined threshold and therefore simulating a normal behavior of all influencing services. If i still violates its threshold after all influencing services respond normally, it is added to the list of root cause services R as i itself is responsible for the performance problem. On the other hand, if all performance predictions in $\tilde{p}_i$ fall below their respective threshold in t_i after all influencing services respond within their given boundaries, then it can be concluded that the performance problems of i can be fixed by fixing the performance problems of its influencing services. Hence, i is not considered to be a root cause service.

After the calculation of R, the Analyzer finally conducts a confidence check of all its predictions in line 8. This step is necessary to avoid inaccurate performance predictions, especially due to a lack of training data or model inaccuracies. Therefore, *SuanMing* enables the Analyzer component to scrap all performed calculations based on a configurable confidence function. If the confidence check fails, Algorithm 5.3 does not return any prediction. If the check succeeds, Algorithm 5.3 returns the application performance prediction $\tilde{P}$, as well as the list of responsible root cause services R.

In this work, our confidence value is based on the accuracy of the forecaster, as all model predictions are dependent on the forecasting accuracy and as it gives good insight into the general model accuracy. We use the coefficient of variation of the output distribution of the GluonTS [Ale+20] forecast as our measure of confidence. Hence, in the following evaluation, *SuanMing* starts to deliver root causes and ratings after the described coefficient of variation falls under 0.15. This value was chosen empirically after preliminary analysis in our test environment. However, in future work, we plan to extend the confidence analysis to compare the respective prediction with actual measurements from the last periods in order to rate the confidence of all component predictions.

As Algorithm 5.3 iterates over every performance prediction only once, it is guaranteed to terminate in linear time on both cyclic and acyclic topologies. Additionally, similar to Algorithms 5.1 and 5.2, Algorithm 5.3 is highly parallelizable, improving the scalability of *SuanMing*.

5.3 Summary

To summarize, *SuanMing* relies on two fundamental models influencing the prediction power of the algorithm: the propagation model D and the performance inference model F. In the first phase, predicted user requests are forwarded through the application. In the second phase, the performance of each service, starting with the backend services, is determined by backpropagating the performance through the application. These two steps are able to deliver an accurate prediction of an arbitrary set of r performance metrics of interest, provided these performance metrics are captured by the monitoring infrastructure. Combined with an accurate forecast of future user behavior, the models can predict the future state of the system. This answers **RQ II.1** (*"How can tracing data be utilized to predict the future performance of a system?"*).

Based on the predicted future state, an operator is then able to define target thresholds for every performance metric, either for all services or only for a subset, as a priority list of supervised endpoints. If any of these priority endpoints is expected to miss its target, *SuanMing* can alert the operator and deliver a list of responsible services for anticipated performance degradations. Therefore, the approach delivers the root cause for predicted degradations, as defined in **RQ II.2** (*"How can we pinpoint the root cause service of a performance problem?"*). By fixing these calculated responsible services (e.g., by adding resources), the operator can prevent the anticipated performance problems before they occur in the system. If required, *SuanMing* can still identify backend or intermediate services that experience performance degradation, even if they do not lead to any user-facing performance degradation.

SuanMing is designed to work scalable, lightweight, on top of online cloud measurement infrastructures, without prior application knowledge, and with arbitrary kinds of acyclic and regular application topologies. Furthermore, *SuanMing* is designed as a framework offering several extension points for the adaptation and improvement of all learning and prediction modules. Therefore, we achieve **Goal II** (*"Develop an approach for the prediction of performance degradation using application-level tracing."*). We evaluate the proposed contribution in Chapter 10 on page 171.

Chapter 6

Estimating Continuous Resource Demands

In this chapter, we introduce *SARDE*, our approach for the continuous estimation and improvement of performance models. Timely and precise resource demand estimates are a crucial input to autoscaling mechanisms [Bau+18] or performance modeling techniques [Hub+17; Kou+16; Reu+16; BKR09] used for elastic resource provisioning. A *resource demand* (or service demand [Spi+15; Spi17]) is the average time a unit of work (e.g., request or transaction) spends obtaining service from a resource (e.g., CPU, HDD, or SSD) in a system over all visits, excluding any waiting times [Laz+84; MDA04] (see Section 2.2.2 on page 22). However, the direct measurement of resource demands is not feasible during operation in most realistic systems [Spi+15] due to instrumentation overheads and possibly measurement interferences. Furthermore, Willnecker et al. [Wil+15b] show that statistical estimation approaches can provide comparable accuracy to direct measurements. Therefore, it has been shown that statistical estimation of resource demands is a valid and useful tool to realize precise elastic cloud resource management [Wil+15b; Bau+18], and several approaches for resource demand estimation have been proposed over the years [RV95; BKK09; Wan+12; ZWL08; Spi+15]. For a detailed overview, we refer to Section 3.2.2 on page 43.

When selecting an appropriate approach for a given scenario, a user has to consider different characteristics of the estimation approach, such as the expected input parameters, configuration settings, its accuracy, and its robustness to measurement anomalies. The accuracy of the different approaches is heavily dependent on factors including but not limited to system load, workload type, deployment structure, internal state, and monitoring granularity [Spi17]. Additionally, Spinner et al. [Spi+15] show that no single approach is optimal in all scenarios. This follows the no-free-lunch theorems for machine learning [Wol96] and optimization [WM97], stating that any two algorithms are equivalent when their performance is averaged across all possible problems.

In addition, many approaches offer configuration parameters to tune its behavior, which drastically influences the estimation accuracy [Spi17] of some approaches. However, it is not trivial to determine how the various parameters should be configured for optimal performance. Hence, correctly selecting and configuring the appropriate approach for a given scenario requires exhaustive testing or detailed expert knowledge.

The first steps focus on combining different estimation approaches into a single usable tool [Spi+14; WPC15] to enable the use and the seamless exchange of approaches. However, the only approach considering the automatic selection of resource demand estimators does not focus on continuous and online estimation [Spi17]. In addition, there exist no works on automatic and systematic evaluation of the best parameter settings for a given test set since previous approaches only do manual testing and develop rules of thumb for a chosen small set of parameters [KTZ09; CC07; CCT08; Zhe+05; ZWL08; Spi+15; Spi17].

The problem is furthermore aggravated as modern software paradigms, like DevOps and cloud deployment, become increasingly popular (see Section 2.1 on page 15). Therefore, timely and precise resource demand estimations get increasingly complex as more and more variables are subject to change, and estimates have to be continuously updated. For example, any autoscaler is constantly changing the deployment structure of the considered software system, and the applied workload is never truly constant in an online application. In consequence, the considered environment is both unknown at design time and constantly evolving during operation time [Cal+12]. Since the characteristics of the system and the properties of the measurements are the main influencing factors for the performance of resource demand estimators [Spi+15], the best-suited approach (together with its parameterization) for estimating resource demands is also continuously changing. It is therefore impossible for any human user to continuously select, parameterize and supervise resource demand estimators during system operation.

To address this problem, we introduce *SARDE*, a framework for continuous, **S**elf-**A**daptive **R**esource **D**emand **E**stimation. *SARDE* is able to operate, parameterize and select multiple different resource demand estimations in a continuous manner and adapts autonomously to changes in its environment or System under Study (SUS). This work focuses on combining and interlacing the different building blocks to create an adaptable and robust framework that can be applied in any continuous environment without requiring expert knowledge. To that end, *SARDE* continuously

- estimates resource demands,

- selects the best-suited estimation approach,
- learns and adapts the selection strategy in order to adapt to changing environments, and
- tunes the parameters of individual approaches based on online observations.

SARDE works as a fully autonomous, situation-aware, and self-adaptive ensemble resource demand estimation approach. It utilizes the above techniques to improve the performance of current state-of-the-art approaches without the need for human supervision or expert knowledge. Therefore, *SARDE* represents a significant step forward towards our published vision of self-aware performance models [GEK18; Spi+19], but also towards the vision of autonomic and self-aware computing [KC03; Kou+17] in general, as the techniques we introduce can also be transferred to other areas of research.

We published detailed descriptions of our works on optimization [Gro+17], clustering [Gro+19a], and recommendation [Gro+18], as well as a holistic combination [Gro+21b]. Furthermore, we presented the value of resource demand estimation for autoscalers [Bau+18]. The source code of *SARDE* is available on GitHub[1]. In addition, we published a replication package on CodeOcean [Gro+21a] for analyzing the experimentation in Chapter 11 on page 187. All contributions serve to answer **RQ III.1** (*"How can we combine different estimation approaches to efficiently produce continuous resource demand estimations?"*), in order to fulfill **Goal III** (*"Enable the continuous estimation and improvement of performance model parameters using production monitoring data."*) in conjunction with the *DepIC* approach introduced in Chapter 7 on page 109.

In the remainder of this chapter, we first give an example motivating the advantages of *SARDE* in Section 6.1. Following, we give a comprehensive overview of our approach in Section 6.2 and then add a more detailed description in Section 6.3. Finally, Section 6.4 closes with a short summary. We evaluate *SARDE* in Chapter 11 on page 187.

6.1 Motivating Example

In this section, we will first intuitively describe the idea behind resource demand estimation in general and then motivate our reasoning behind *SARDE* using a small example trace. Let us assume that we have a SUS for which continuous monitoring streams of throughput, response times, and resource utilization

[1]`https://github.com/jo102tz/LibReDE-SARDE`

are collected. The system serves three different request types, methods, or workload classes WC1, WC2, WC3, for which we want to estimate the respective resource demands, as these can be used to accurately model the performance of the SUS. For illustration purposes, during the first interval, a CPU utilization of 80% is measured, while on average, throughputs of 20 (WC1), 11 (WC2), and 5 (WC3) requests per second of the respective workload classes are measured. In the second interval, the utilization drops to 60%, as 30 (WC1), 4 (WC2), and 10 (WC3) requests per second are processed. The task of the resource demand estimators is now to calculate the resource demand of each workload class based on this set of coarse-grained measurements. One resulting estimation could be that WC1 and WC3 both take 10 ms per request, while WC2 is more intense and takes 50 ms as this would fit with the measurements on a one-core system. However, the actual truth is still unknown to us and, dependent on the chosen technique, different approaches arrive at different estimations.

Our current implementation of *SARDE* is based on the LibReDE [Spi+14]. The publicly available library LibReDE [SGK19] provides a set of implementations of different resource demand estimation approaches (see Section 3.2.2 on page 43). Furthermore, LibReDE is peer-reviewed and accepted by the Standard Performance Evaluation Corporation Research Group (SPEC RG) [SGK19]. Therefore, in the rest of this work, we will utilize the following base estimators from LibReDE:

- RTA: approximation with response times [BKK09],
- SDL: approximation using the Service Demand Law [BKK09],
- RR: LSQ regression based on queue lengths and response times [Kra+09],
- UR: LSQ based on utilization law [RV95],
- WKF: Kalman filtering based on utilization law [Wan+11; Wan+12], and
- KKF: Kalman filtering based on response times and utilization [ZWL08; KTZ09].

In order to illustrate and motivate the idea behind *SARDE*, Figure 6.1 shows the error (calculated as described in Section 11.1.3 on page 190) of the continuously updated estimation using the listed available estimation approaches over time. Details on the used system and workload are included in Section 11.1.1.2 on page 188.

We observe that over the course of three hours, the performance of each estimator is massively influenced by the type and amount of monitoring data

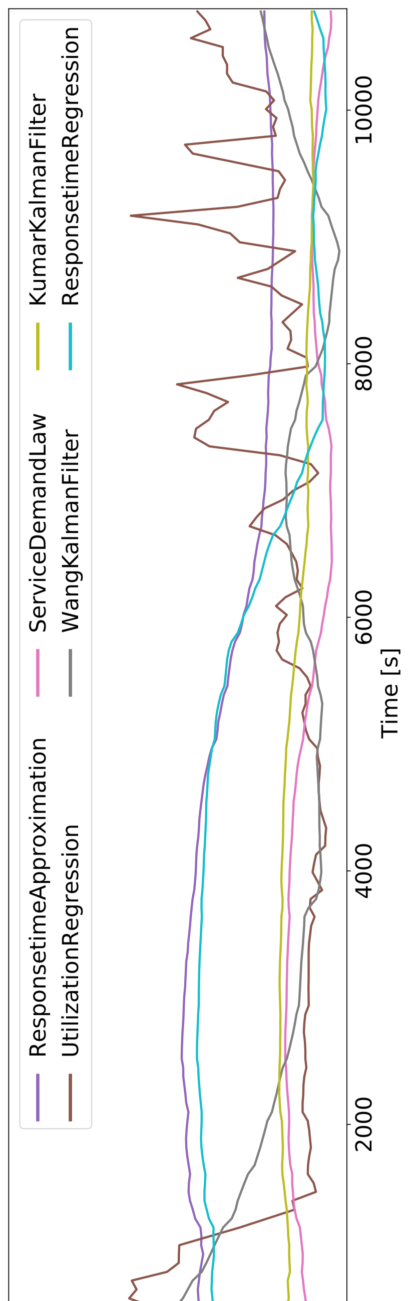

Figure 6.1: Example run showing the estimation error of different estimators over time.

available, as well as the underlying characteristics of the system. As a result, SDL (pink) starts as the best estimator, followed by UR (brown). However, the accuracy of UR starts to decline after a while, and in fact, continues to have the worst estimation performance of all available approaches. In total, four of six available estimators exhibit to be the best estimator at least once during our three-hour experiment. Additionally, it is not clear in advance which estimator will perform how well, especially as some estimators also tend to be very unstable.

Hence, *SARDE* acts as an ensemble estimator able to combine the best from all estimators and compensate for the weaknesses of some approaches using the strengths of others. In other words, the aim of *SARDE* is to successfully learn and adapt to the changing performance of the estimators in order to be able to always select the best approach for each scenario. In addition to that, we observe that some approaches are very susceptible to changes in their parameter settings [Gro+17]. Therefore, by adapting these parameters in the applied scenario, *SARDE* could even improve the performance beyond the current best method without the need for human supervision or expert knowledge.

6.2 Overview

This section gives a high-level overview of *SARDE*, as illustrated in Figure 6.2. First, *SARDE* comprises two running databases: One containing monitoring streams from the SUS, another storing the sequence of resource demand estimations made over time. Next to the databases, *SARDE* continuously runs the estimation engine, performing periodic resource demand estimations based on the continuously updated monitoring streams. The estimation engine offers different configuration interfaces, like the specific approach to use or the parameter settings of the individual approaches. The resulting estimations are then stored in the resource demand database. From there, external processes (e.g., an autoscaler [Bau+18; Bau+19] or a performance model extractor [Wal+17; Spi+19]) can retrieve the latest resource demand estimations. On top of that, *SARDE* consists of two interacting feedback loops: *Optimization* and *Selection*.

The optimization process deals with parameter tuning (e.g., the aggregation interval or the monitoring window) of the individual approaches. To that end, monitoring data from the system as well as the corresponding resulting estimations are utilized. The optimization then specifically tailors the parameters of each available estimation approach to the specific SUS to minimize the resource demand estimation error.

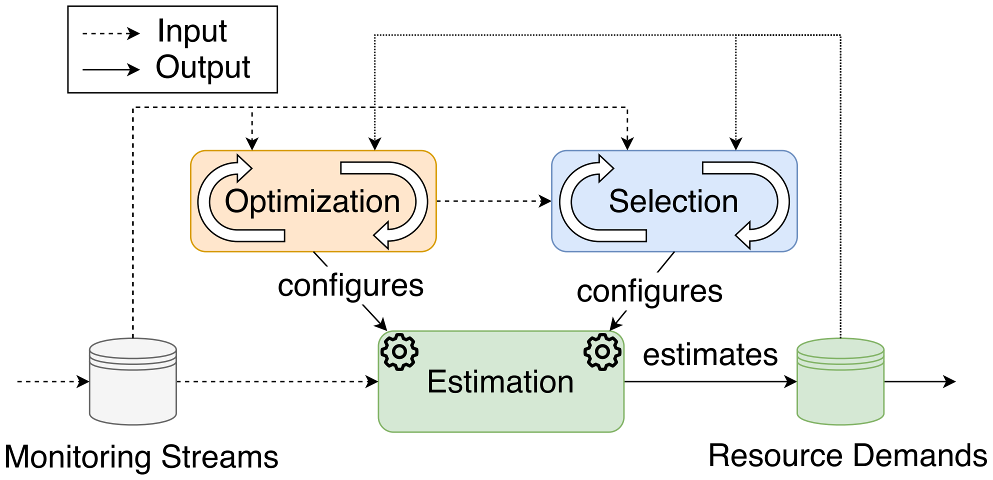

Figure 6.2: High-level overview of the *SARDE* approach.

The selection process utilizes the same data as the optimization process. Instead of optimizing the parameters, the selection process fits a machine learning model predicting which approach to select for a given situation. This is done based on specific features of the monitoring data, like the average CPU utilization, or based on properties of the SUS, for example, the number of servers or workload classes. Based on these features, the selection process can then select the best-suited estimation approach for the given situation.

As the optimized parameter settings influence the performance of the individual approaches, these settings have to be considered while training the machine learning model and are therefore directly fed into the selection process. The selection itself interacts only indirectly with the optimization, as the process has an impact on the resulting resource demand estimations in the resource demand database, which is, in turn, an input to the optimization loop. In addition to utilizing the historical data, both processes perform additional computations and resource demand estimations for exploring the space of all possible configurations.

6.3 Approach

In this section, we describe the two feedback loops presented in Section 6.2 and how communication between them is organized in more detail. As both the optimization process and the selection process interact with the estimation engine, as shown in Figure 6.2, synchronization and communication between these processes is required. To keep all sub-systems of *SARDE* up-to-date, we introduce a set of semaphore artifacts. These artifacts can only be written by one respective process but may be read by all other processes. This way, it can be ensured that the different feedback loops do not block each other during execution while using the most recent version.

Figure 6.3 depicts the five different activities running in parallel: monitoring, parameter optimization, selection model training, approach selection, and finally, resource demand estimation. In the following, we will discuss each of the individual processes in more detail.

6.3.1 Monitoring

As the different resource demand estimation approaches require both system-level and application-level monitoring, the monitoring engine has to monitor application-level metrics (like throughput and response time per workload class) and system-level metrics (e.g., average CPU utilization per instance) live from the running system. These monitoring streams are then stored in a database, and each entry is assigned a corresponding timestamp. The gathered data can then be fed into the remaining four processes, each of which requires the information as input.

6.3.2 Optimization

Different resource demand estimation approaches offer several parameters to be tuned. Additionally, some parameters like, for example, the aggregation interval of the monitoring data (step size) or the measurement window to consider (window size) can be tuned for all approaches. This is done by analyzing the estimation error of individual estimation approaches via cross-validation on the monitoring data gathered on the system. A configurable search algorithm then applies different parameter settings and searches for a (near-)optimal configuration of those parameters for each of the available approaches. Although this is a relatively straightforward task, the optimization still bears many challenges, as the number of different possible configurations rises exponentially with the number of parameters and the time available for optimization is limited. The goal is, therefore, to utilize an algorithm that is able to find a good parameter configuration using a small number of exploration runs.

The applied self-tuning algorithm is generally abstract and works for any generic parameter providing a minimum and a maximum value. We evaluated different approaches and then decided to rely on S3 in this work [Gro+17].

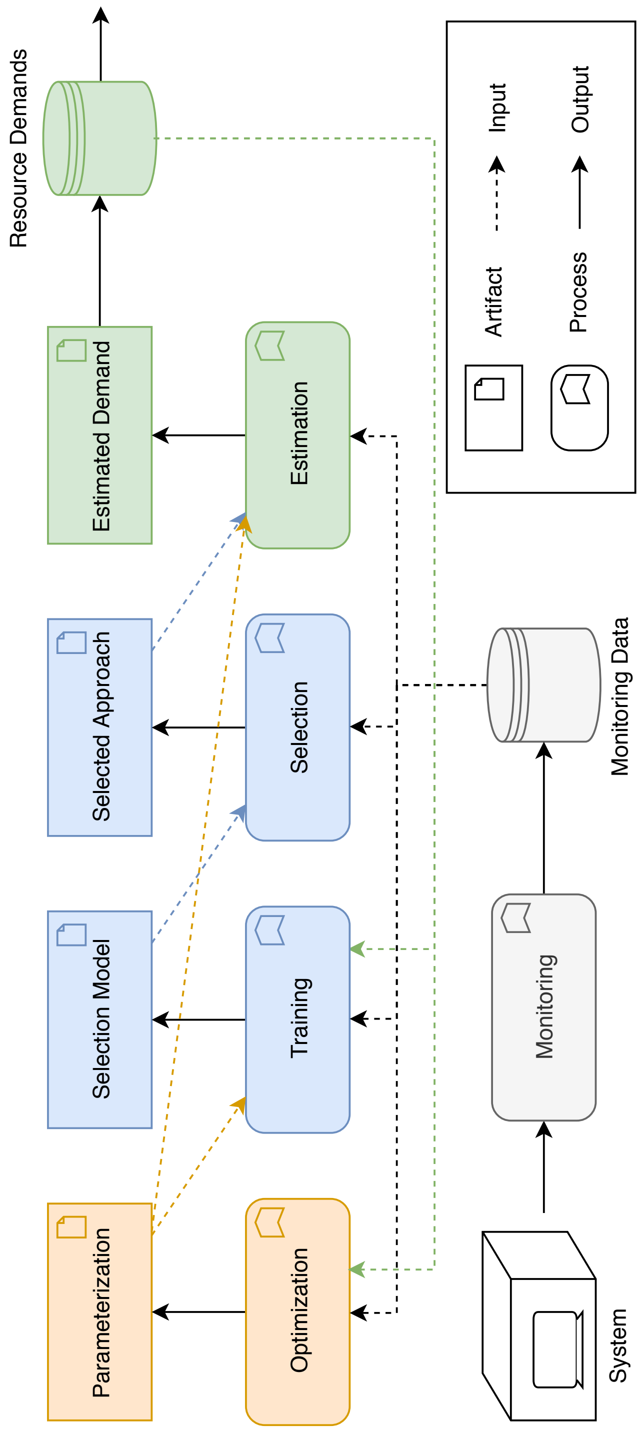

Figure 6.3: Conceptual flowchart of the different *SARDE* processes.

The S3 was developed by Noorshams [Noo15] in the context of regression model optimization. Here, we utilize this algorithm in order to optimize the parameters of our resource demand estimation techniques. The S3 algorithm can be configured by three hyperparameters: The number of splits per parameter k, the number of exploration points considered per iteration n, and the maximum number of iterations j_{max}. Noorshams et al. [Noo+13] show that the total complexity of the algorithm is given by $\mathcal{O}(j_{max} \cdot n \cdot (k+2)^l)$, where l is the number of parameters that are optimized simultaneously. Therefore, S3 offers good control over the trade-off between run time and solution quality by tuning its hyperparameters. Additionally, it is possible to optimize an arbitrary number of parameters simultaneously. This is important as inter-parameter dependencies, that is, one parameter influencing the other, can be taken into account. However, it has to be noted that the number of parameters to be simultaneously optimized heavily influences the computational complexity. Note that S3 is just one possible search algorithm; other algorithms focusing on modeling or optimizing configurable software systems [Zha+15; Sie+15; Guo+17; HZ19; Gro+20b] are applicable as well.

Although this step can be executed offline using a large trace database, the optimization is usually more effective when optimizing for a specific kind and type of system. Additionally, as the SUS evolves or the amount of available monitoring data increases, the parameters need to be adapted continuously. Therefore, the process is periodically triggered. However, depending on the chosen algorithm, this process can be very time-consuming, running for multiple hours or even days for huge systems. Therefore, the execution is triggered comparatively seldom.

6.3.3 Training

The third step is the process of training the estimation approach selector. The selection process in Figure 6.2 is split into two activities as the selection itself is executed far more frequently than the training of the selection model. During the training phase, a model is learned which is able to predict the best-suited approach for the given estimation problem. This model is then stored as the Selection Model, which is used by the actual selection process.

6.3.3.1 Problem Formalization

The problem of selecting the best algorithm for a specific problem instance was also formulated by Rice [Ric76] as the algorithm selection problem. Based on this work, Smith-Miles [Smi09] formalized the following four components for modeling a selection problem: (i) the problem space, (ii) the feature space, (iii) the algorithm space, and the (iv) performance space. In this work, we can translate this to the task of selecting the best-suited resource demand estimation approach as follows:

- The problem space P represents the measurement traces available for estimation,
- the feature space F contains the characteristics of each trace, as described in Section 6.3.3.3,
- the algorithm space A is the set of available resource demand estimators, and
- the performance space Y represents the mapping of each algorithm to the estimation error.

For a given measurement trace $p \in P$ with characteristics $f(p) \in F$, the objective is to find a selection mapping $S(f(p))$ into the algorithm space A, such that the selected algorithm $\alpha \in A$ minimizes the performance mapping $y(\alpha(p)) \in Y$. The task of the model learning is to find the function S, mapping each possible trace characteristic to the selected algorithm, while the actual selection process (see Section 6.3.4) is executing $S(f(p))$.

6.3.3.2 Dataset

One of the defining aspects during training is the available dataset. Note that the training procedure itself can be done either online or offline. This decision mainly influences what data is available during the training phase to extract knowledge from. *SARDE* utilizes a combination of offline and online training.

Offline training We refer to *offline training* as a training process that is performed once, using a variety of systems and configurations. Based on this set, one can apply all available approaches to the different training sets and use the feedback from those runs to determine which approach is suited best for the specific problem instance. This information, together with a set of descriptive features, is then given to a machine learning algorithm, which learns a model from all training sets, extrapolating the relationship between the different features and the best-suited approach. We call this resulting model the selection model. Naturally, the accuracy of this approach highly benefits from an increasing amount of training data and a high similarity of the training systems to the current problem instance.

Online training Offline training has the disadvantage of being trained before the actual application to the SUS. Therefore, in *online training*, we continuously monitor the current system and the performance of the different approaches, as these can also serve as training samples for our selection model [Ker+19]. Furthermore, the performance of the individual approaches changes if the optimization process described in Section 6.3.1 adapts the parameter settings of the respective approaches. If so, the training must be repeated for the newly found parameterization, which can be cost-intensive for the offline dataset. However, online learning has the disadvantage that the trained model is prone to over-fitting to a specific system and can not adapt very well to changes in the configuration or the structure of the SUS. This is due to the drastic reduction of training data in comparison to the larger dataset used in offline training.

Hybrid training As a consequence, we introduce *hybrid training*, a combination of both offline and online training, in this work. The idea of hybrid training is to utilize the training datasets as applied in offline training but iteratively adding online data from the SUS to the dataset and periodically triggering the training process. Therefore, the training process is able to adapt to the feedback of the running system while also maintaining robustness towards major changes of the respective system.

6.3.3.3 Features

Another central aspect of all machine learning approaches is the feature set used for training. This section contains the list of features we extract from each monitoring trace. These features capture certain characteristics of the input traces that we deem useful for judging which algorithm would be most suitable for estimating that respective trace.

The machine learning algorithms are heavily dependent on those features, and a careful selection, as well as the right amount, is crucial for a satisfactory outcome. Since machine learning algorithms try to distinguish between different classes of traces, too many features can actually be harmful. A *trace* refers to one training example of our dataset. A trace usually consists of a set of time series, for example, of the CPU utilization of each resource, the response time, and the arrival rate of each request of the respective workload classes. The CPU utilization measures the average utilization of the CPU for a certain interval, the response time contains the response time of each request, and the arrival rate holds the number of incoming requests for a certain interval. These traces are then given to the estimation approaches for their estimations. For each trace, we create a feature representation y that captures the characteristics of this trace.

Next to the time series itself, we have some general meta-information about the traces, including the number of resources (e.g., number of CPUs or CPU cores) and the number of different workload classes. For example, Spinner et al. [Spi+15] showed that the number of workload classes has a direct impact on the performance of the estimators. This meta-information is also added to the feature set. Another big impact on the performance of estimators is the utilization of the system [Spi+15]. Therefore, it is useful to include information about the average utilization of the available resources as well as the minimum and the maximum utilization. In addition, we also to extract statistical information about the time series of each trace.

However, it does not seem useful to average this information over all available resources. Especially since different workload classes are known for stressing each resource differently. We, therefore, define a set of statistical features to extract utilization information for each individual resource, together with information about the arrival rate and response times of each workload class and concatenate them to one feature vector y.

The extracted statistical features for a time series $T = (d_1, \dots, d_n)$ consisting of an ordered set of data points are as follows:

- The number of data points: $n = |T|$.
- The arithmetic mean: $\overline{T} = \frac{1}{n}\sum_{i=1}^{n} d_i$.
- The geometric mean: $\hat{T} = \left(\prod_{i=1}^{n} d_i\right)^{\frac{1}{n}}$.
- The standard deviation: $\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(d_i - \overline{T})^2}$.
- The quadratic mean: $x_{\text{rms}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} d_i^2}$.
- The minimum value: $T_{min} = \min(T)$.
- The maximum value : $T_{max} = \max(T)$.
- The kurtosis, measuring tailedness [Wes14]: $k = \frac{\frac{1}{n}\sum_{i=1}^{n}(d_i-\overline{T})^4}{\left(\frac{1}{n}\sum_{i=1}^{n}(d_i-\overline{T})^2\right)^2} - 3$.
- The skewness, measuring asymmetry [JG98]: $s = \frac{\frac{1}{n}\sum_{i=1}^{n}(d_i-\overline{T})^3}{\left[\frac{1}{n-1}\sum_{i=1}^{n}(d_i-\overline{T})^2\right]^{3/2}}$.

- The 10th percentile: $l = P_{10}(T)$.
- The 90th percentile: $u = P_{90}(T)$.

This results in a total of eleven statistical measures. Given that these are calculated for each resource and twice for each workload class (for arrival rates and response times), the total number of features amounts to $|y| = 2 + 11 \cdot r + 22 \cdot w$, with r being the number of resources, w being the number of workload classes in the training set, including the number of resources and workload classes available as two additional features.

One advantage of the selected features is that they are fairly easy and fast to compute. In addition, most of the features are standard statistical measures that are easy to comprehend as a user. Exceptions might be the kurtosis and the skewness metrics; however, kurtosis and the skewness are common metrics in time series analysis [JG98; Wes14]. We also examined other feature sets of diverse size and composition [Gro+18], including the correlations and the co-variances between the traces, the variance inflation factor, and information about the statistical distributions. While it might seem useful to include further features into the training, these features are costly to calculate and therefore greatly increased the required selection time [Gro+18]. As the respective features did not significantly impact the prediction accuracy, we decided to settle on the final feature list presented above. We also excluded any feature probing techniques [Hut+14; Kot+15] as we consider the performance impact too high. Additionally, removing any more features from the above list negatively influenced the selection results while offering only an insignificant runtime advantage.

6.3.3.4 Labels

After acquiring the feature vector per trace, one can execute all resource demand estimators on the given trace and then use the resulting estimation error as labels for training a machine learning algorithm. A selection engine can then be built by training different regression models, each predicting the error of individual estimators and then choose the one with the lowest expected error [Bis+16]. However, in the following, we work with a classifier-based approach. We compare the error values of each estimator in order to label each feature set with the value of the best algorithm. During the selection, the predicted label of the classifier can be viewed as the approach expected to perform best. This way, only one classifier model needs to be trained and executed, which saves computation time during online execution.

What remains is the determination of the estimation error of each approach during training. If available, the real estimation error can be used if the training set contains a set of artificial or specifically monitored traces. However, this will not be feasible for many traces, for example, during online training. As the real resource demand is per definition unknown to *SARDE*, we have to rely on the internal error calculation based on cross-validation. The validation error used for labeling is explained in more detail in Section 11.1.3 on page 190.

6.3.4 Selection

After the training process produced an accurate selection model, the selection process analyses the type and structure of the monitoring streams. It uses the provided selection model to make an informed decision about which approach to use for estimation. Simply put, the acquired machine learning model is utilized, and its prediction for the best-suited estimator is applied. This process was deliberately split from the training process, as this process can use the same selection model multiple times in order to update the selected approach based on changes in the system or the monitoring streams.

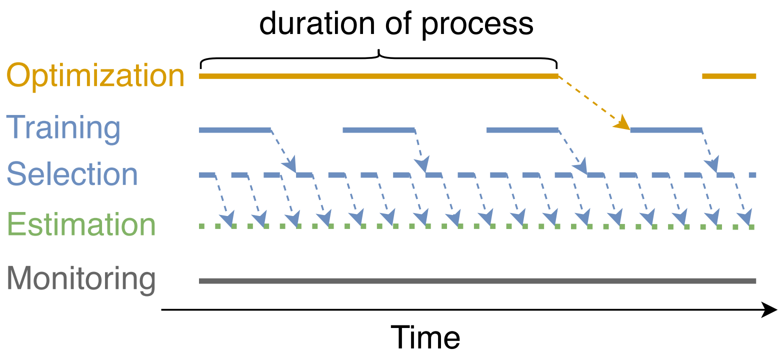

Figure 6.4: Exemplified timeline visualization.

Figure 6.4 illustrates an exemplified timeline, visualizing the five processes running in parallel. While monitoring is a continuous process, the estimation is executed quite frequently, with the more computationally expensive procedures running slower and fewer iterations. Note that this is just an exemplary configuration, the actual intervals of *SARDE* can be tuned by the user. Furthermore, the arrows of the respective colors show how the results of the particular process influence the other running processes. We observe that, for example, a finished training process updates the selection model used for the next selection process that has not started yet. This model is then used until it gets updated by a subsequent training iteration. Similarly, the output of the selection process, the selected approach to use for estimation, is applied for all subsequent estimation runs as long as the selection is not updated. It is furthermore shown how the optimization results influence the next training process. After a successful optimization, the optimization results take a while to come into effect at the actual estimation, as the estimation uses the old parameterization until the training with the new parameterization is finished and the newly parameterized approaches are selected for estimation. This has the advantage of protecting the continuous estimation from negative effects by a disadvantageous optimization run, as the training process is able to double-check and filter the respective approaches if necessary. However, the cost of this approach is the delay between a finished optimization and its parameterization coming into effect.

6.3.5 Estimation

The most frequent process is the actual estimation process. Its frequency mainly depends on the variability of the system and the monitored traces, as well as the quality of the estimated resource demands itself. Upon execution, the estimation process loads the approach selected by the selection process and updates it with the optimized parametrization by the optimization process, if available. Then, the estimation is executed on the newest monitoring data. Note that, as depicted in Figure 6.4, multiple subsequent estimation executions might be performed using the same approach. This is on purpose, as the monitoring data is updated between those executions, which impacts the estimation result. To that end, all process executions always utilize the most recent monitoring data available at the start of each process.

6.4 Summary

In this chapter, we presented *SARDE*, a framework for continuous self-adaptive resource demand estimation. *SARDE* continuously (i) estimates resource demands, (ii) selects the most suitable estimation approach from a set of available alternatives, and (iii) optimizes the parameterization of the estimation approaches in order to minimize the estimation error. This is achieved by continuously evaluating the performance of each estimator in the current and constantly changing scenario. Based on the characteristics of the current situations, *SARDE* is able to adapt each estimator itself, but also to select the most suitable approach as well as improving and hardening the overall estimation error. This enables *SARDE* to serve as an ensemble resource demand estimator, capable of delivering reliable estimations in unknown and constantly changing environments without expert knowledge or human intervention.

Therefore, the ideas presented in this chapter answer **RQ III.1** by presenting a technique to continuously combine different resource demand estimation techniques. Together with the contributions in Chapter 7 on page 109, *SARDE* achieves **Goal III** (*"Enable the continuous estimation and improvement of performance model parameters using production monitoring data."*). We evaluate *SARDE* in Chapter 11 on page 187.

Chapter 7
Learning Parametric Dependencies

A significant factor for the prediction accuracy of performance models is its parameterization, i.e., the values for model parameters such as loop frequencies, branching probabilities, or resource demands [Spi+15]. However, these model parameters often depend on the input parameters of a component (e.g., the size of a list impacts the time required to sort it). Therefore, many architectural performance models allow to explicitly model input parameters and their influence on model parameters in the form of so-called *parametric dependencies* [Koz08; Ham09; Bon+05; Eis+18; Sit+01a; Sit+01b]. These describe the resource demand of a function call in dependence on a given set of input parameters. The importance of including such influences has been discussed by a variety of authors, including Booth and Wiecek [BW80], Woodside et al. [Woo+95], Pozzetti et al. [Poz+95], Menascé [Men97], and Koziolek [Koz10].

Manually modeling parametric dependencies requires expert knowledge, is prone to errors, and causes significant manual effort. For example, in a case study by Krogmann et al. [KKR10], more than 24 hours were required to manually model the parametric dependencies in a small system. Therefore, manually modeling parametric dependencies for large systems is infeasible due to the effort required to model them. In another work, we furthermore found out that the required effort for creating the performance models hinders the adoption of performance modeling techniques in the industry [Bez+19].

As already mentioned during our discussion of related work on automatic extraction of performance models in Section 3.2 on page 40, most approaches are not able to extract parametric dependencies [Wal+17; Hri+99; Isr+05; MF11; BKK09; BVK13; Wil+15a; SWK16; Spi+19]. Notable exceptions include the works of Courtois and Woodside [CW00], Krogmann et al. [KKR10], and Brosig et al. [BHK11] (see Section 3.2.1.1 on page 41). Courtois and Woodside [CW00] propose to use regression splines combined with automated performance testing to derive functions that describe resource demands based on input parameters. Krogmann et al. [KKR10] use genetic search to find dependencies between a component's input parameters and the number of executed bytecode instructions. Brosig et al. [BHK11] model parametric dependencies in PCM models but require input information about parameter tuples for which dependencies exist. To summarize, approaches from literature either require running preliminary experiments in a testing environment [CW00; KKR10] or already detected parametric dependencies as input [BHK11].

Therefore, in this chapter, we address **RQ III.2** (*"How can the impact of parameters on resource demands be identified and characterized?"*). We do this by introducing *DepIC*, our approach to derive parametric dependencies solely from monitoring data available at run-time. This task can be split into two sub-tasks: (i) detecting the dependencies, that is, identifying which parameters influence a model variable, and (ii) characterizing the dependencies, that is, describing how the value of a parameter can be derived from the influencing parameters. In the following, we first focus on sub-task (i), i.e., the detection of parametric dependencies in Section 7.1. Second, we describe our solution to sub-task (ii), i.e., the characterization of already detected parametric dependencies in Section 7.2.

The proposed *DepIC* approach significantly reduces the required effort of modeling parametric dependencies, enables automated extraction of more detailed models, and therefore makes the modeling of dependencies feasible for large systems. *DepIC* can also be used as assistance for a performance modeling expert, who can use our system's suggestions as candidate dependencies in order to manually change or tune them. Furthermore, *DepIC* can work solely with production monitoring traces of the managed applications and does hence not depend on any kind of prior knowledge, source code information, or static analysis. As a consequence, all required structural information is obtained from the monitoring data itself. Therefore, this work fits well into our vision of self-learning and self-improving performance models [GEK18] and addresses **Goal III** (*"Enable the continuous estimation and improvement of performance model parameters using production monitoring data."*).

To that end, the presented approach can be used to enhance existing model extraction pipelines, as shown in Figure 7.1. Currently, a performance model without parametric dependencies can be extracted from monitoring data using existing model extraction approaches [Wal+17; Hri+99; Isr+05; MF11; BKK09; BVK13; Wil+15a; SWK16; Spi+19]. The approach presented in this work can be applied in parallel on the same monitoring data to automatically identify dependencies between model parameters in the first step (Dependency Identification) and then characterize these dependencies in the second step (Dependency Characterization). For the second characterization step, existing approaches for the characterization of dependencies, such as Brosig et al. [BHK11] and Courtois and Woodside [CW00], or our proposed incremental modeling techniques [Maz+20; Von+20], can similarly be applied using minor modifications.

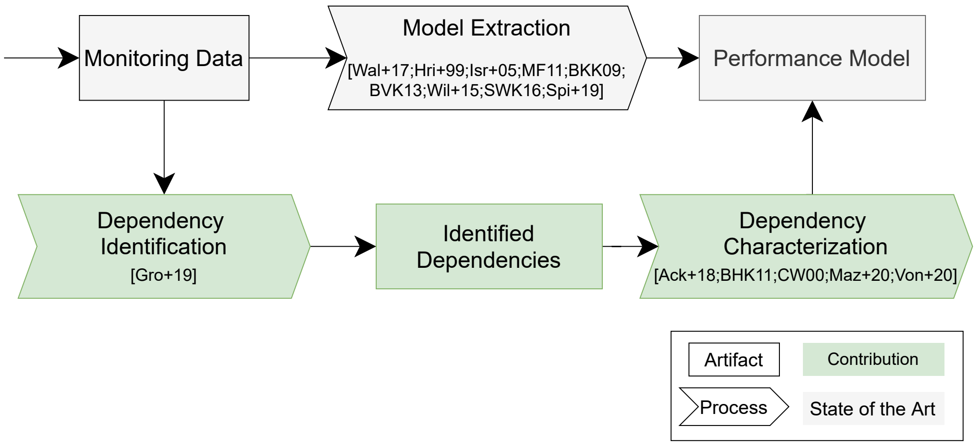

Figure 7.1: Model extraction workflow.

After the second step, we have a concrete function that describes the dependency between the target model parameter and the influencing parameters. The resulting functions can then be integrated into the previously extracted performance model to improve its expressiveness. The exact nature of this integration step depends on the modeling features offered by the specific modeling formalism. Possible supporting formalisms include the PCM [BKR09; Koz08] or the DML [Kou+16; Eis+18].

Our contributions detecting parametric dependencies [Gro+19b] and characterizing them [Ack+18] were published as individual publications and also referenced in a combining vision [GEK19]. Furthermore, we collaborated with other researchers on this topic and contributed to other publications in the area of model learning and especially the area of parametric dependencies. In particular, the work of Mazkatli et al. [Maz+20] presents an approach for the continuous integration of performance models. The idea [MK18] is to integrate the process of model extraction into modern CI/CD pipelines of DevOps processes, which incrementally extracts and calibrates the performance model, including parametric dependencies. The publication of Voneva et al. [Von+20] extends this by focusing in particular on optimizing the process of extracting parametric dependencies in this context by applying GP. In addition, we contributed to a similar approach that tries to speed up the analysis of architectural models using statistical response time models [Eis+19].

In the following, we concentrate on the detection of parametric dependencies in Section 7.1 and the characterization of already detected dependencies in Section 7.2. We conclude with a short summary in Section 7.3.

7.1 Detection of Parametric Dependencies

The detection of parametric dependencies can be framed as a classic application of feature selection: We define one model parameter as a target parameter and consider all other model parameters as potential features. The challenges when applying feature selection to this domain are obtaining suitable measurement streams, selecting the most promising dependencies, and discarding detected dependencies without modeling gain.

In this section, we propose a generic algorithm for the automated identification of parametric dependencies on monitoring streams. This includes the preprocessing of monitoring records, the creation of feature selection tasks, an interface to integrate a chosen feature selection technique, and three different heuristics to filter the identified dependencies. These heuristics utilize domain knowledge to drastically decrease the number of identified dependencies reducing them to only performance-relevant ones. Following, we apply and evaluate three different feature selection approaches [CS14a]: a filter method based on correlation-based feature selection [Hal99], a wrapper method based on M5 [Qui92], and an embedded method based on RF regression [Ho95; Bre01].

A high-level summary of our approach is shown in Algorithm 7.1. As the monitoring data is usually unstructured, we define the input data $in = \{r_1, \ldots, r_k\}$ as a set of n unordered monitoring records $r_1, \ldots, r_k$, with k entries in total. We describe a dependency $d = (p, p_e)$ as a tuple consisting of a dependent parameter p and an explanatory or independent parameter p_e used to describe p. For the remainder of this chapter, we assume a dependency to involve just one independent parameter p_e. Note that *DepIC* is still capable of identifying two separate dependencies for the same dependent parameter. Therefore, we can create multi-parameter dependencies by combining two or more separate dependencies.

First, Algorithm 7.1 initializes an empty set D containing all found dependencies. Then, we transform the unstructured monitoring entries into data streams in line 2. The resulting set of data streams S is a collection of different sub-streams. Different sub-streams are necessary as loop and recursion structures make it impossible to aggregate all data streams on the same call path. We elaborate on that problem in Section 7.1.2.

Algorithm 7.1: Detecting parametric dependencies.

```
Input: Monitoring data $in = \{r_1, \dots, r_k\}$.
Output: Set of found dependencies $D = \{(p_1, p_{l_1}), \dots, (p_n, p_{l_n})\}$.
1 $D = \emptyset$
2 $S$ = extractDataStreams$(in)$
3 foreach $s_i$ in $S$ do
4     $T_i$ = createFeatureSelectionTasks$(s_i)$
5     foreach $t_{i,j}$ in $T_i$ do
6         $scores_{i,j}$ = applyFeatureSelectionAlgorithm$(t_{i,j})$
7         $D = D \cup$ createDependencies$_\theta(t_{i,j}, scores_{i,j})$
8 $D$ = filterResult$(D)$
9 return $D$
```

We then iterate through all found data streams S to create a list of individual feature selection tasks T_i for each sub-stream s_i in line 4. Each task consists of one defined dependent parameter, together with all possible independent variables. Each task $t_{i,j}$ is then fed into a black-box feature selection algorithm in line 6. The algorithms are required to return a vector $scores_{i,j}$, assigning a weight to each independent parameter for the specific task. These scores are then used to create the set of dependencies in line 7. Currently, the method `createDependencies` includes dependencies if its score is higher than a given threshold θ. However, this threshold depends on the used feature selection algorithm and the corresponding scoring technique. Therefore, `createDependencies` has to be parameterized with a certain threshold θ. After all sub-problems and the resulting tasks have been investigated, the algorithm filters all found dependencies in line 8 and finally returns the set of found dependencies D.

In the following, we elaborate on the individual steps taken by Algorithm 7.1. Section 7.1.1 defines the monitoring requirements and some necessary preprocessing steps to receive a valid input for the described algorithm. We elaborate on the process of extracting data streams from the monitoring data (`extract-DataStreams`) in Section 7.1.2. Section 7.1.3 gives some details on how to create the individual feature selection tasks (`createFeatureSelectionTasks`). The application of the different feature selection techniques (`applyFeature-SelectionAlgorithm`) is explained in Section 7.1.4. Finally, the creation of the actual dependencies (`createDependencies`) and the filter procedures (`filter-Result`) are described in Sections 7.1.5 and 7.1.6, respectively.

7.1.1 Monitoring

In this section, we describe the monitoring streams used by *DepIC* to identify parametric dependencies.

7.1.1.1 Monitoring Requirements

Our approach completely relies on monitoring data without the additional consideration of system architecture or source code. The following data has to be contained in the monitoring stream of each relevant method invocation in order to ensure that the *DepIC* can extract parametric dependencies:

1. The Method identifier (e.g., method signature),
2. the call path trace information to reconstruct method invocations using recursion or loops,
3. the method parameter identifiers (e.g., the parameter name, type, as well as the concrete parameter values),
4. the method return identifiers, type, as well as the concrete parameter values, and
5. the resource demand of the specific method invocation.

The call path trace is required to extract structural information from the monitoring data, for example, which method is called by which other method. This is required for detecting dependencies between different method scopes. For example, if one parameter in method A influences the behavior of method B.

Monitoring frameworks that satisfy most of these requirements out-of-the-box are Kieker [Hoo+09; HWH12], along with other tracing tools like Pinpoint, Zipkin, or Dynatrace (see Section 2.2.1.3 on page 21). However, critical parameters are parameter and return values, as well as accurate resource demand measurements, which are not collected by most standard tools.

Therefore, we reside to use the open-source solution Kieker, as it offers the option to insert custom probes enabling customization of the collected metrics. We created a custom monitoring probe collecting the required information. Additionally, for complex parameters likes collections and arrays, we log the size of the collection or the length of the array. Binary objects can be captured using their id or their size in bytes. If required, Krogmann [Kro12] proposes heuristics for exporting even more parameter properties. Otherwise, there exists the possibility of using domain knowledge by an expert to manually model important parameter properties. As there already exist works toward that end, we do not consider the logging of parameters as a core contribution of our work and assume the values as given.

7.1.1.2 Resource Demand Estimation

As the goal of *DepIC* is to find dependencies describing the resource demand of a function call for a given set of input parameters, we require accurate resource demand estimates, as noted in the fifth requirement of Section 7.1.1.1. As the estimation of resource demand is an important topic for modeling, we cover the topic of estimating resource demands in extensive detail in Chapter 6 on page 91.

However, for the estimation of parametric dependencies, we require individual resource demands per method invocation, which is not supported by most statistical approaches. If the load is sufficiently low, we can assume the measured response time and resource demand to be similar, as there would be no queuing delays in that case [BKK09], otherwise known as RTA. As a result, we suggest circumventing the issue by selectively utilizing low-load scenarios for the input data. We investigate the impact of higher utilization levels in the evaluation in Section 12.3 on page 218. Future work can also adapt and utilize the techniques presented by *SARDE* in order to deliver more fine-granular resource demand estimates as required by *DepIC*.

7.1.1.3 Anomaly Detection

As our approach is based on measurement data and we aim to find relations and correlations in the measurements, the proposed technique is susceptible to outliers. Therefore, an additional step of anomaly detection and filtering is necessary while preprocessing the monitoring data. The concrete amount depends heavily on the algorithms used for feature selection. In this work, it was sufficient to apply a 99.9-percentile filter to any measured vector. Hence, after collecting the data, we take all values that are below the 99.9-percentile of the measurements to remove any outliers. This has proven to be sufficient in our experiments; however, it may, of course, depend on the used platform, measurement infrastructure, and application stack. Hence, we can not propose a general solution and acknowledge that other experiment setups might require more sophisticated solutions [HA04].

7.1.2 Creation of Data Streams

Monitoring streams generated by a system with different components are generally not structured in such a way that they can be directly inputted into standard machine learning algorithms. Consequently, preprocessing steps are necessary to adjust the data to fit into the required format. For this, structural information has to be extracted from the monitoring data, loops and recursions have to be resolved, and resource demands have to be extracted from the measured response times. These so-called data streams can then be fed into machine learning algorithms. We covered the topic of extracting the resource demands in Section 7.1.1.2. Therefore, this section focuses on resolving structural information.

In order to create the data streams from the monitoring streams, the initial data is transformed into a directed multi-graph representing the call path, where the vertices represent a method signature and edges the invocation of a method A (start node) calling method B (end node). This graph is constructed by evaluating the execution order index and execution stack size of each method invocation of the current trace. This can be done using the data listed under requirements (1) and (2) in Section 7.1.1.1. Each edge can now be associated with the respective measurements (parameter logs, together with response time measurements). Based on the constructed graph, the data can be transformed into data streams. This process is subdivided into two steps.

First, every call path of the graph is extracted. Each call path resembles a succession of method invocations. We define a call path as a sub-graph containing all direct and indirect successive calls of the specified root-vertex. Second, loops and recursions are resolved as they are problematic for the calculation of our parameter correlation.

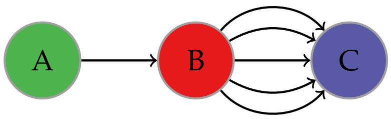

Figure 7.2: Example graph representation of a call path.

Consider, for example, the case of function A calling function B, which then calls function C in a loop. This is illustrated in Figure 7.2. Function A invokes B once, B invokes C 5 times in a loop. In this case, for every call of A, we get a set of parameter measurements of A and one measurement set for B, but multiple sets of measurements for C. Even more problematic is the fact that the number of measurements is variable and probably different for every invocation of A and B. This poses a challenge for our detection algorithms as most standard machine learning algorithms can not deal with a varying number of measurements per sample. Hence, the collected data can not be directly written into the data stream since each entry in one data stream has to correspond to the entry of another data stream with the same index.

Therefore, we have to aggregate all calls from B to C into one call in order to process them later on. However, if we ignore C and collapse all measurements into one aggregation, we lose a lot of information as C might itself consist of multiple vertices and call multiple other components (including further loops). Consequentially, we want to aggregate loop calls into one entry while preserving the ability to extract dependencies in the loop itself.

Resolving loops A loop can be identified in the graph data structure by analyzing the number of edges from one vertex to another. The number of calls from B to C can be directly assessed by counting the number of directed edges from B to C (see Figure 7.2). Since this is done for each call path trace, we can analyze the number of invocations from B to C. If the number of invocations varies on different call paths, we assume that we have a dynamic loop structure depending on the internal state or the input parameters.

This is solved by dividing the original data stream into two sub-problems (i.e., two sub-streams). That is, on the one hand, the scope of the function calling upon a looped function (A and B), and on the other hand, the scope of the loop itself (C). In order to do this, we create one data stream instance of our original problem with the loop collapsed to one entry, including the number of collapsed entries as a parameter. We call the number of entries invocation count or loop count, as it describes the number of calls from B to C. This addition is necessary in order to be able to extract dependencies containing the iteration count of the loop since this information is lost when collapsing the entries and splitting the data into sub-problems. The second sub-problem instance contains the parameters inside the loop. This is necessary to avoid losing information about dependencies inside vertex C (as C could consist of several vertices and edges itself). Our identification algorithms are then applied to each of the sub-problems individually.

Given the graph in Figure 7.2, two sub-problems are created. One with the data streams of methods A, B, and C with all execution entries of C collapsed to one entry; the other sub-problem is the loop itself with data streams of methods B and C, while the entries of B are replicated for each invocation of C.

Resolving recursions Another common programming concept applied in practice is recursion. Recursion can be identified by searching the constructed graph for cycles. We can distinguish between direct and indirect recursion as seen in Figures 7.3 and 7.4, respectively.

Recursions can be resolved similarly to loops. We represent each recursion as a black-box and only save the initial call parameters, the aggregated call parameters, the result, and the response time. Additionally, the recursion depth is saved as this information could be meaningful concerning the performance of the method (similar to the loop invocation count). The main difference of indirect recursions is that all vertices concerned in the recursion chain are contracted into a single vertex for further analysis of the data stream. We acknowledge that this is a sub-optimal solution for some applications since an indirect recursion might span over many methods, which will then be contracted to one single vertex. A more in-depth investigation of this issue can be the target of future work.

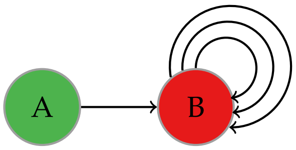

Figure 7.3: Example graph representation of a direct recursion.

In the example shown in Figure 7.3, function B calls itself after being invoked by A. Only the parameters of the initial invocation of the recursion of B is considered, while the other parameter values are aggregated to one set. However, in this example, the recursion depth parameter is added with a value of three as there are three calls observable in Figure 7.3. In Figure 7.4, functions C and B call each other after being invoked by A. Applying the approach on this graph leads to the vertices B and C being merged into a single node BC and the data of the invocations between both nodes being aggregated while adding the recursion depth parameter with a value of 2.

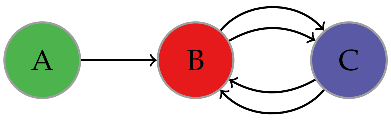

Figure 7.4: Example graph representation of an indirect recursion.

7.1.3 Creating Feature Selection Tasks

This section describes the creation of single feature selection tasks from existing data streams. Recall that a data stream $s_i = \{v_1, \ldots, v_k\}$ is a collection of k vectors v_i with $i \in 1, \ldots, k$. After our preprocessing, all vectors have the same length and can be seen as features. The feature selection techniques introduced in Section 7.1.4 require a distinction between a *dependent* variable, that is, the target variable, and a set of *independent* variables, also referred to as features. The dependent variable is predicted based on the values of the independent variables. In order to find all dependencies within a data stream, we construct a feature selection task for each vector v_i, which analyses the impact of the remaining vectors on this vector.

One way of achieving this is to simply brute-force the whole parameter space for each target variable. However, this is very inefficient as many parameter combinations are evaluated and would furthermore result in a set of unuseful dependencies. Therefore, our approach for creating feature selection tasks from data streams is presented in Algorithm 7.2. It requires a strict total order of all vectors as well as an additional label l_i for each vector v_i. The label $l \in \{MP, AVG, RET, NONE\}$ describes the data type of each vector v_i:

1. Model parameters (MP): The vector contains performance-relevant model variables, like resource demand measurements or loop invocation counts.
2. Averaged value (AVG): This vector contains averaged values as they are created when loops or recursions occur in the data stream (see Section 7.1.2).
3. Return value (RET): These are the logged return values of a specific function invocation.
4. Normal ($NONE$): Values of these vectors are non-averaged and usually describe function parameter values.

Note that any vector is assigned exactly one label depending on the first rule that applies, from top to bottom. Hence, averaged return values are always classified as AVG, instead of RET, as rule (2) applies first. Together with the labels, Algorithm 7.2 requires a strict total order of all vectors. This order describes the chronological order of the parameter occurrences. The vector v_i is monitored after v_j, if $v_i > v_j$ for $i, j \in 1, \ldots, k$ with $i \neq j$. For simplicity, we assume that we have no concurrent records and therefore $\forall i, j \in 1, \ldots, k : v_i = v_j \Leftrightarrow i = j$. Both the strict total ordering and the set of labels can be obtained directly during the creation of the data streams in Section 7.1.2 without additional overhead.

Algorithm 7.2: Creation of feature selection tasks.

```
Input: Data stream $S = \{(v_1, l_1), \dots, (v_k, l_k)\}$.
Output: Set of $l$ created tasks $T = \{t_1, \dots, t_l\}$.
1  $T = \emptyset$
2  Let $V_{MP}$ be the subset of $S$, with $\forall v_i \in V_{MP} : l_i = MP$
3  Define $V_{AVG}, V_{RET}, V_{NONE}$ analogously
4  Let $V_{IND} = S \setminus V_{MP}$                    // independent variables
5  foreach $v_i$ in $V_{MP}$ do
6      $T = T \cup (v_i, V_{IND})$
7  $V_{IND} = V_{IND} \setminus V_{RET}$
8  $V'_{RET} = \emptyset$
9  foreach $v_i$ in descending ordered $V_{IND} \cup V_{RET}$ do
10     $V_{IND} = V_{IND} \setminus v_i$
11     if $v_i \notin V_{AVG}$ then
12         $T = T \cup (v_i, V_{IND} \cup V'_{RET})$
13     if $v_i \in V_{RET}$ then
14         $V'_{RET} = V'_{RET} \cup v_i$
15 return $T$
```

Algorithm 7.2 describes how the four different classes influence the creation of feature tasks. First, we consider the performance-relevant model parameters V_{MP} as the most critical parameters, as we are aiming to find dependencies describing them. Hence, we use all available parameters to find such dependencies in line 5 and line 6 of Algorithm 7.2. However, we do not relate MPs with other MPs, as such dependencies are irrelevant (model variables are usually both known or unknown at the same time).

Second, we merge both return values V_{RET} and all remaining vectors V_{IND} and sort them in descending order in line 9 of Algorithm 7.2. Here, we implicitly use the defined ordering. However, return values can only influence other values of lower order, while standard parameters only influence parameters of higher order. Therefore, we first delete all V_{RET} from the set of independent variables in line 7 and then successively add them to the set of returned parameters V'_{RET} in line 14. These returned parameters V'_{RET} can then influence other parameters. On the contrary, all standard vectors V_{NONE} are left in the set of independent variables, but they get successively removed from V_{IND}

in line 10, once they can no longer serve as independent variables. We never use the averaged parameters V_{AVG} as dependent variables. Since they contain aggregated values over multiple invocations, they are considered in a different data stream in more detail. However, we use them as independent variables for all other tasks but successively remove them from the set of independent variables V_{IND} in line 10. This way, Algorithm 7.2 massively reduces the amount of created feature selection tasks while using our domain knowledge to ensure that no possibly relevant parametric dependencies are skipped.

7.1.4 Applying Feature Selection Techniques

After creating concrete sub-tasks for each data stream, we can use standard feature selection techniques from machine learning to select the most promising variables as dependencies. Recall from the high-level overview in Algorithm 7.1 that one task $t = (v_D, V_{IND})$ contains a dependent vector v_D and a set of independent vectors V_{IND} with the same length. The goal of the feature selection is to receive a $scores$ vector, that is, a ranking for each $v_l \in V_{IND}$ for the given v_D. This ranking expresses how useful v_l is to describe v_D, that is, how well v_D can be described using v_l. This task is a classic feature selection problem (see Section 2.3.3.2 on page 33), as machine learning engineers often face the problem of selecting the most promising features (independent variables) to predict a given target (dependent variable) [CS14a]. Therefore, all presented algorithms can return such a ranking.

However, usually, this ranking is not normalized. In classic feature engineering, this is not an issue as only the relative score of the different independent variables is of interest. Our problem statement is different in the sense that we not only need a relative rating but also a normalized, and hence comparable, score. This score is required during the filtering step in Section 7.1.6, where this score is the decision parameter, whether or not a feature selection task is modeled into a dependency. Therefore, we have to adapt some state-of-the-art techniques to make them applicable in our scenario. In the following, we discuss the three main classes of feature selection techniques [CS14a; GE03]: filter, wrapper, and embedded methods.

7.1.4.1 Filter

Filter techniques assign a value to the possible independent variables without any interaction with the target algorithm. The correlation-based feature selector ranks feature subsets according to a correlation with the dependent variable [Hal99]. Features are then either selected or removed based on a cutoff value for the scoring. The methods are often uni-variate and solely consider the independent variables; therefore, they are computationally cheap and commonly used as a preprocessing method.

In this work, we use absolute Pearson's r as a measure for correlation as it was shown to be effective in related work [Hal99]. Hence, for each independent variable, we compute the absolute correlation with the dependent variable and use it as the score. One advantage of Pearson's r is that it is already normalized between -1 and 1. We take the absolute value as negative correlations, that is, inverse relationships, should also be captured by *DepIC*.

7.1.4.2 Wrapper

Unlike filter approaches, wrapper methods evaluate subsets of independent variables proposed by different search algorithms (forward selection, backward elimination, etc.) and thus enable the detection of interactions between different independent variables. Scores are assigned based on the accuracy of the predictive model using the particular subset. Since multiple subsets have to be evaluated and a model has to be created for each evaluation step, the main disadvantages are the significant computation time and the increased risk of overfitting [GE03]. Additionally, the achieved filtering strongly depends on the chosen regression algorithm for evaluation.

For evaluating subsets of the variable space, M5 [Qui92] is used. This enables the ability to identify non-linear and complex dependencies. In order to find the optimal score, we evaluate the performance of all subsets and store the Root Mean Squared Error (RMSE) [HK06]. Then, each independent variable gets a score based on the weighted average of all scores of all subsets it was involved in. Each subset is weighted with the inverse of the number of parameters involved. Hence, the more parameters involved, the less influence on the score of the involved parameters it has.

However, as this score is based on the RMSE of the resulting regression, it is not normalized and therefore not comparable between different feature selection tasks. Therefore, we divide the score of each independent variable with the RMSE of a baseline classifier. The baseline classifier always returns the mean value of all training samples. Hence, we weigh the performance of each independent variable in dependence on the performance of this simple baseline. If the baseline performs well, the dependent variable does not show enough variance. This, therefore, justifies a low score, as the modeled dependency does not express a lot of information gain.

7.1.4.3 Embedded

Embedded techniques are a result of trying to combine the advantages of both filter and wrapper techniques. The selection of independent variables is accomplished during the execution of a specific learning algorithm, thus reducing computation time while still considering the interactions of variables. However, not all machine learning approaches support this technique. We chose the RF algorithm [Ho95] for this task, a specific type of bootstrap aggregated decision trees [Bre96].

The core idea of bagging is to use bootstrap samples using a standard training set for fitting the k ensemble models. A bootstrap sample of size n is generated by uniformly sampling n instances from the training set with replacement. RF uses a modified tree learning algorithm selecting a random subset of independent variables at each candidate split in the learning process instead of considering the entire variable space. The random forest algorithm assigns a rating based on the performance of the individual decision tree to each tree. Therefore, we use the selection of the independent variable in the respective decision tree together with the rating to obtain a ranking of the relevance of the independent variable for predicting the dependent variable.

There are different measures for evaluating the importance of a variable in tree-based models [KR10; HTF09]. In this work, we concentrate on the Gini importance [HTF09]. Additionally, we include an additional variable consisting of noise. We use it to compare the performance of independent variables to the noise as a criterion for normalization similar to the baseline regressor of the wrapper approach. We divide the score of each independent variable by the score of the noise variable. Therefore, we receive a ratio of how much more information a particular variable contains in comparison to a baseline variable.

7.1.5 Creating Dependencies

After obtaining the score vector computed in Section 7.1.4, we use this score to create dependencies for the discovered relations. Each of our adapted algorithms returns a vector $scores$ for each variable selection task. This vector assigns each independent variable an individual weight, reflecting its importance for describing the specific dependent variable. Furthermore, we ensure that the given score is normalized. Therefore, we can compare the scores of the different variable selection tasks with each other. Based on this score, we either accept a task and model the found relation as a dependency, or we reject the task as we do not see any valuable dependency in the considered relation.

We achieve this by applying a threshold θ for each selection task. As the methods introduced in Section 7.1.4 all support different scores, we define three different thresholds: θ_{Filter} for the filter approach, $\theta_{Wrapper}$ for the wrapper, and $\theta_{Embedded}$ for the embedded approach. Therefore, the method is parameterized with a threshold θ, as already discussed in Algorithm 7.1. For any given θ, if the score of a variable is higher than θ, we create a dependency from that particular variable to the dependent variable of the current feature selection task v. If none of the scores exceeds the defined threshold value θ, we do not create any dependencies and return an empty set. We analyze the impact of using different thresholds in Section 12.1. We describe the procedure for dependency creation more formally in Algorithm 7.3.

Algorithm 7.3: Creating the resulting dependencies.

```
  **Input:** Selection task $t = (v, V_{IND})$, score
          vector $scores = (s_1, \ldots, s_{|V_{IND}|})$.
  **Output:** Set of found dependencies $D$ or $\emptyset$.
1 Let $p, p_i$ be the corresponding parameters of $v, v_i$ for $i \in 1, \ldots, |V_{IND}|$.
2 $D = \emptyset$
3 **foreach** $v_i$ **in** $V_{IND}$ **do**
4     **if** $s_i \geq \theta$ **then**
5         $D = D \cup (p, p_i)$
6 **return** $D$
```

Note that in line 1 of Algorithm 7.3, we assume to have the corresponding parameter name for each vector v_i available. This is necessary as Algorithm 7.3 models the dependencies between the parameters, not their corresponding measurement vectors v_i. As the parameter names are usually given during implementation, we do not include it as a formal input but rather retrieve it during execution.

7.1.6 Result Filtering

Finally, in the last step of Algorithm 7.1, we filter all modeled dependencies from Section 7.1.5 to reduce the number of resulting dependencies. Since we are extracting dependencies for performance models, some parameters include more information than others. Hence, although some dependencies might be correct in terms of existing correlations in the monitoring data, the relation between the parameters might not be useful in our performance model. In the following, we present three post-processing steps reducing the number of detected irrelevant dependencies.

7.1.6.1 Filtering identical parameters

After observing the initial results, we discovered that many irrelevant dependencies are actually identified due to identical parameter values. This is a common practice in software engineering. For example, a parameter p (e.g., a list) is passed to one method m_1, which then forwards this parameter to the next method m_2, which processes it. Therefore, methods m_1 and m_2 share the same parameter p, which is correctly modeled and identified by *DepIC*. However, now the parameter values p_{m_1} and p_{m_2} of m_1 and m_2 both exhibit a dependency to the resource demand p_d of method m_2, as m_2 is concerned with the actual processing of p.

Therefore, we introduce the filtering of identical parameters. In our example, we delete the dependency from p_{m_1} to the resource demand of m_2. The relation between p_{m_1} and m_2 is captured by the dependency between p_{m_1} and p_{m_2} and the successive dependency of p_{m_2} to the resource demand of m_2. Note that this step is only applied if p_{m_1} and p_{m_2} are observed to be identical.

7.1.6.2 Filtering correlating dependencies

The second filtering step is correlation-based filtering. It is explicitly the desired behavior of our algorithm to extract two different dependencies d_1 and d_2 to a dependent parameter p_d. Both independent parameters p_1 and p_2 influence p_d, but are not identical (if they are, the dependency is filtered in the previous step.).

Now, we analyze the correlation between p_1 and p_2. If both p_1 and p_2 correlate, we have a redundant dependency and therefore want to select the stronger relation. Hence, we analyze the scores of the dependencies of p_1 and p_2 assigned by the feature selection technique in Section 7.1.4. We delete the dependency with the lower score and therefore keep the stronger relation. As both always appear in the same feature selection task, we can also simply consider the relative ranking of both variables to compare them to each other. This also implies that correlation-based filtering is independent of the normalization method applied during the scoring process described in Algorithm 7.2.

This reduces overfitting as we select fewer but more significant variables. Note that both relations are kept if there is no strong correlation between p_1 and p_2. In our experiments, an absolute Pearson's r correlation coefficient of at least 0.8 has shown to be sufficient. Therefore, independent variables are filtered if their Pearson's r correlation is equal to or higher than 0.8 or lower than -0.8.

7.1.6.3 Graph-based filtering

Finally, our last step is to eliminate all dependencies that do not influence any performance-relevant model parameters. For example, two input parameters might be influencing each other. However, if none of them influences any resource demand, they do not have any (modeled) performance impact. Therefore, capturing the relation between them is useless as it does not give us any performance-relevant information.

Hence, we construct a graph consisting of all model parameters as nodes and all dependencies as directed edges between them. The edges are the opposite direction of the dependency (i.e., if a dependency has a model parameter as a target, the model parameter will be the source of the directed edge). Now, we iterate through all performance-relevant parameters and perform a breadth-first search. All dependencies that were not discovered by any of the breadth-first searches are deleted. As they are not discovered, they have no path and therefore no (transitive) relation to any of the performance-relevant parameters. We consider resource demands as well as loop counts as performance-relevant parameters, i.e., the (transitive) targets of all detected dependencies. The impact of each of these filtering steps is evaluated in Section 12.2 on page 213.

7.2 Characterization of Parametric Dependencies

After we identified which parameters influence the relevant performance properties of the system, the second sub-task is characterizing the found dependencies. Therefore, this section focuses on the characterization of the previously identified dependencies, i.e., it aims at determining the exact relationship between the independent and the dependent target parameter.

This is a classic statistical regression task, which is why many regression techniques from machine learning could be applied. To do so, we create a representative dataset containing parametric dependencies and evaluate how well a set of different regression approaches can characterize the contained parametric dependencies. We find that no machine learning approach performs well for all parametric dependencies. This is in accordance with the no-free-lunch theorem for machine learning [Wol96], which states that there exists no single machine learning algorithm that is universally good for all problem instances. Based on these results, we propose a meta-selector selecting an appropriate machine learning technique for every dependency based on the characteristics of the available data.

Our approach takes potential dependencies (either labeled by a human or extracted by the algorithm presented in Section 7.1) as input and automatically characterizes them, i.e., automatically learns how to derive the model parameter such as loop frequencies, branching probabilities, or resource demands from input parameters. By applying our meta-selector, we significantly reduce the required effort compared to trying different available modeling approaches and therefore make the modeling of dependencies for large systems feasible. Similar to the proposed identification approach in Section 7.1, the approach works online, requires only run-time monitoring data, and otherwise treats the monitored application as a black-box.

We describe how we created our dataset in Section 7.2.1 and introduce the applied machine learning techniques in Section 7.2.2. Finally, we present why and how we created the meta-selector in Section 7.2.3.

7.2.1 Dataset Creation

We first create multiple datasets, each containing the measured values, in our case the response time, of a certain application (in the form of a short algorithm execution) in dependence on various observable input parameter values. As servers can run arbitrary applications and services, all kinds of applications might be relevant for our black-box learning approach. In consequence, we select popular algorithms from different domains (e.g., sorting, image processing, or cryptography) and with different characteristics (e.g., number of input parameters, noise intensity, or expressiveness of dependency) in an effort to meet the diversity of real-world application scenarios. Additionally, some datasets were added, where there is no correlation between the input parameters and the response time (e.g., `getRandomInt`).

Non-numeric input parameters are transformed into meaningful numeric values (e.g., binary values are mapped to 0 or 1, and enumerations to a range of integers between 1 and the number of different values). In order to obtain the response time dataset, we repeat each execution 100 000 times with different input parameter combinations.

For the selection of parameter inputs for each measurement point, we apply the following strategy:

- First, a realistic value range for each input parameter is defined.
- Next, the actual measurement points are created by dividing the parameter space into equidistant measurement points. The distance between

these points depends on the chosen size of the dataset, the number of parameters, and the range of each parameter.

- Next, the chosen tuples are randomly shuffled to prevent the datasets from being biased by underlying optimization processes (e.g., garbage collection or just-in-time compilation).
- Finally, each measurement point is executed and measured in a single-threaded and sequential fashion to avoid system overload and measurement interferences.

Since all tests are done in a single-threaded and sequential fashion, we assume that the measured response times equal the resource demand of the respective resource. In the following, we use the term response time, as all measurements are technically only response time measurements. However, in this context, we can interpret all response time measurements as resource demand estimates as well.

Table 7.1 lists the functions we used to create the dataset, together with the defined range for the respective input parameters. In the following, we will briefly describe the functionality of each used function.

ackermannFunction calculates the Ackermann function for the given parameters `n` and `m`. As the AckermannFunction is known to drastically increase its runtime for increasing values of `n` and `m`, we have to rely on a rather small set of parameter values.

fibonacci returns the i-th Fibonacci number. The parameter `i` defines the index of the requested number. The second parameter defines if an iterative (`iter.`), an optimized recursive (`recOp.`), or an unoptimized recursive (`rec.`) implementation should be chosen.

filterArray filters a given array for the given key. Hence, it returns a filtered array only containing elements with a specified key. The parameter `arraySize` defines the number of elements in the array, `filterKey` is the integer representation of the key.

gaussianFilter applies a Gaussian filter function to a given image. The parameters `imageWidth` and `imageHeight` define the width and the height of the processed image in pixels, `sigma` is the σ-parameter for the Gaussian filter.

getRandomInt returns a random integer in the range between the given parameters `minInt` and `maxInt`. Note that this function should NOT contain

Table 7.1: Measured functions and defined input parameter ranges.

Name	Input parameter name	Range
ackermannFunction	n	0 – 3
	m	0 – 3
fibonacci	i	1 – 40
	opMode (iter./recOp./rec.)	0 – 2
filterArray	arraySize	0 – 100 000
	filterKey	0 – 100 000
gaussianFilter	imageWidth (px)	100 – 6 500
	imageHeight (px)	100 – 4 010
	sigma	1 – 10
getRandomInt	minInt	1 – 100 000
	maxInt	1 – 200 000
histogramEqualization	imageWidth (px)	100 – 6 500
	imageHeight (px)	100 – 4 010
loadFile	fileSize (KB)	1 – 1 024
rgbFilter	imageWidth (px)	100 – 6 500
	imageHeight (px)	100 – 4 010
rsaEncryption	stringLength	0 – 30
	keySizeExponent (2^x)	9 – 11
rsaDecryption	stringLength	0 – 30
	keySizeExponent (2^x)	9 – 11
scaleImage	imageWidth (px)	100 – 6 500
	imageHeight (px)	100 – 4 010
	scaleFactor	0.1 – 3.0
searchArray	arraySize	0 – 100 000
	key	0 – 100 000
shaHashing	stringLength	0 – 10 000
	sha-Mode (-1/-256/-512)	0 – 2
sortArray	arraySize	1 – 10 000
subsetSum	arraySize	1 – 10 000
	sum	1 – 100 000

any dependencies from the input parameters to the response time of the function.

histogramEqualization enhances the contrast of a given image by adjusting the image intensities. As the procedure is calibrated by the histogram on the image itself, we only have the two parameters `imageWidth` and `imageHeight`, specifying the size of the image in pixels.

loadFile loads a file from disk into RAM. The only parameter specifying this process is the size of the file in KB as `fileSize`.

rgbFilter is another image operation for filtering with different color channels. The two parameters `imageWidth` and `imageHeight` specify the size of the image in pixel.

rsaEncryption encrypts a message according to the RSA algorithm [RSA78]. The parameter `stringLength` defines the length of the input message, while `keySizeExponent` (2^x) defines the length of the key. The range of the exponent is between nine and eleven, leading to the three possible key sizes 512, 1 024, and 2 048.

rsaDecryption decrypts an encrypted message according to the RSA algorithm [RSA78]. The parameter `stringLength` defines the length of the resulting non-encrypted message and `keySizeExponent` (2^x) defines the length of the key. The range of the exponent is between nine and eleven, leading to the three possible key sizes 512, 1024, and 2 048.

scaleImage resizes an image and scales its content to be either smaller or bigger than the original input. The parameters `imageWidth` (px) and `imageHeight` define the original image size in pixels, the parameter `scaleFactor` defines the desired output size. A factor smaller than one leads to a reduction of the image size, a factor greater than one leads to an increase.

searchArray performs a linear search through the given array and looks for the given key value. The length of the array is defined by `arraySize` and the desired item search key by `key`.

shaHashing computes the hash of a given string using the Secure Hash Algorithm (SHA) [Han05]. The length of the string to hash is defined by `stringLength`. There are three modes to operate: SHA-1, SHA-256, and SHA-512. This is configured by the parameter `shaMode` (-1/-256/-512).

sortArray reorders all elements of the given array. The parameter `arraySize` defines the number of elements to sort.

subsetSum calculates if any subset of the given list of elements can be found, such that the sum of all elements of the subset equals the given predefined value. The parameter `arraySize` defines the number of elements to choose from, and the parameter `sum` defines the required target sum.

The given parameters are either direct input parameters (e.g., `filterKey`) or derivated from the input parameters (e.g., `arraySize`). Note that not all parameters do have an impact on the measured response time. This is on purpose, as the regressors should be capable of filtering the important parameters. We analyze the corresponding set of measurements in more detail in the respective publication [Ack+18].

7.2.2 Applying Machine Learning Techniques

In order to characterize the dependencies from the measured dataset, we can apply standard regression techniques from the area of machine learning. For this, we define the dependent variable as the measured response time, i.e., the resource demand, and the independent variables as the given input parameter values.

The applied algorithms are chosen from various areas of machine learning, such as decision tree learning, instance-based learning, and ensemble learning. We apply the following algorithms:

- A mean predictor as a baseline to compare against,
- kNN [Alt92] dynamically choosing the number of neighbors between 1 and 5 using cross-validation and weighting neighbors by the inverse of their distance,
- LR [DS98] using batch gradient descent and the SSE loss function,
- HR [Hub64] as a robust regression approach using Stochastic Gradient Descent (SGD) [RM51; Bot99],
- Support Vector Regression (SVR) [CV95] using a polynomial kernel,
- NN [Gur97] configured as a fully connected feed-forward net using one node per input parameter on the input layer, followed by a single hidden layer with $(\#InputParameters+1)/2$ sigmoid nodes, and a linear output node,

- CART [Bre+84] using reduced-error pruning,
- M5 [Qui92] using squared error loss for fitting the LR models,
- Bagging [Bre96] using CART as the base model, and
- RF regression [Ho95; Bre01] also using CART as the base model.

The mean predictor serves as a dummy regressor and always returns the mean value of all training samples. All algorithms rely on the implementations from the Waikato Environment for Knowledge Analysis (WEKA) 3.8 [Hal+09; FHW16] library for Java. We use the default parameterization for all approaches, where not specified otherwise, to avoid bias and ensure comparability. For more details on the chosen parameters, we refer to the original publication [Ack+18]. Short explanations of the respective regression techniques can be found in Section 2.3.1 on page 24.

We systematically measure the prediction performance of different regression techniques on the collected datasets and thus evaluate their suitability to learn and represent parametric dependencies for performance models. We compare the different algorithms using the Mean Absolute Error (MAE) (see Section 2.3.2 on page 27). Therefore, the best regressor for a given dataset is the one observing the lowest MAE during the respective cross-validation.

In an effort to increase and diversify our training dataset, we use different sub-sets as training to determine how the prediction performance of each technique scales with the size of the training set. The evaluation process for each measurement set is as follows:

1. We randomly shuffle the dataset and isolate the first 1 000 measurement points as the validation set.
2. Next, we train each regressor on seven different training sets of increasing size and evaluate its performance on the isolated validation set. The training set sizes, i.e., evaluation steps, are 10, 100, 500, 1 000, 3 000, 6 000, and 9 000. Thus, we arrive at 105 evaluation runs (15 datasets, each using 7 training set sizes) per regressor.
3. Finally, we repeat steps (1) and (2) 10 times for each dataset with different seed values, resulting in a total of 1 050 (15 datasets, 7 training set sizes, and 10 repetitionseach) data points to compare.

Table 7.2 shows the distribution of the best regressor on our dataset. We observe that SVR performs best in most cases, being the best regressor in 40.8%

of all cases. However, kNN (18.5%), NN (15.3%), RF (9.7%), and M5 (9.1%) also each make up significant shares of performance. In fact, every applied approach performs best on at least one evaluation set. Surprisingly, even the mean predictor (Mean) observes the minimal MAE in some cases, hinting that the respective dataset is not reasonably learnable by any of the applied algorithms. This is in accordance with the no-free-lunch theorem [Wol96], which states that no machine learning algorithm can be the best in all given scenarios.

Table 7.2: Relative share of samples for which each predictor is the most accurate.

Algorithm	Relative share
Mean	0.4%
kNN	18.5%
LR	1.0%
HR	0.7%
SVR	40.8%
NN	15.3%
CART	1.3%
M5	9.1%
Bagging	3.3%
RF	9.7%

7.2.3 Meta-selector Construction

After reviewing the results of the previous Section 7.2.2, we can conclude that no single regression algorithm exists that is able to model all different types of parametric dependencies. Therefore, the need for a meta-algorithm that analyses the given monitoring data arises. The algorithm analyzes the respective dataset and selects the best algorithm based on a set of defined characteristics. A similar approach is also applied in Chapter 6 on page 91, when choosing the best approach for resource demand estimation. A key property of the meta-selector is the available characteristics that can be used as features for the training of the meta-selector. We select a small set of statistical properties that have proven to be easily collectible for all datasets and still be influential based on our domain knowledge:

- The number of training instances in the dataset (`size`),

- the number of input parameters to consider (`params`),
- the range of measured response time values, i.e., the maximum minus the minimum value (`range`),
- the Coefficient of Variation (CV) of the response time measurements (`CV`),
- the maximum linear correlation between any input parameter and the response time (`maxCorr`),
- the minimum linear correlation between any input parameter and the response time (`minCorr`), and
- the coefficient of determination [DS98] of the dataset ($\mathtt{R}^2$).

These features are extracted for the dataset described in Section 7.2.2. The label of each trace is the regressor that showed the lowest MAE on the given trace. Therefore, we can train a classification algorithm to classify and therefore recommend one of these available regressors based on the features of any dataset. In total, we have a set of 1 050 training samples, each with seven features and one target class (the desired algorithm).

As the task of selecting the best approach based on the feature values is a classic supervised machine learning problem, we use a standard classification algorithm to create a decision tree for the selection. The advantage of using decision trees is that their decisions are traceable and human-readable. In doing so, we hope to gain additional insights into the relations between the dataset characteristics and the algorithm performances. For the construction of the tree, we again utilize the CART [Bre+84] algorithm, implemented by the WEKA library [Hal+09; FHW16], version 3.8. For training, the minimum number of instances per leaf is set to two, and we limit the maximum tree depth to four. This should reduce the complexity of the tree and hence improve readability and interpretability.

The resulting decision tree is visualized in Figure 7.5. The nodes of the decision tree are depicted from left to right, which the decision feature in the rectangle and the respective thresholds on the outgoing edges of each decision node. The leaf nodes are depicted as rounded rectangles around the respectively chosen approach. In addition, each leaf node shows the number of correctly chosen approaches, together with the number of training samples that were falsely classified, i.e., originally belong to another class. As we configured a maximum depth of 4, there is a maximum of 4 decision nodes on each path before we reach a leaf node.

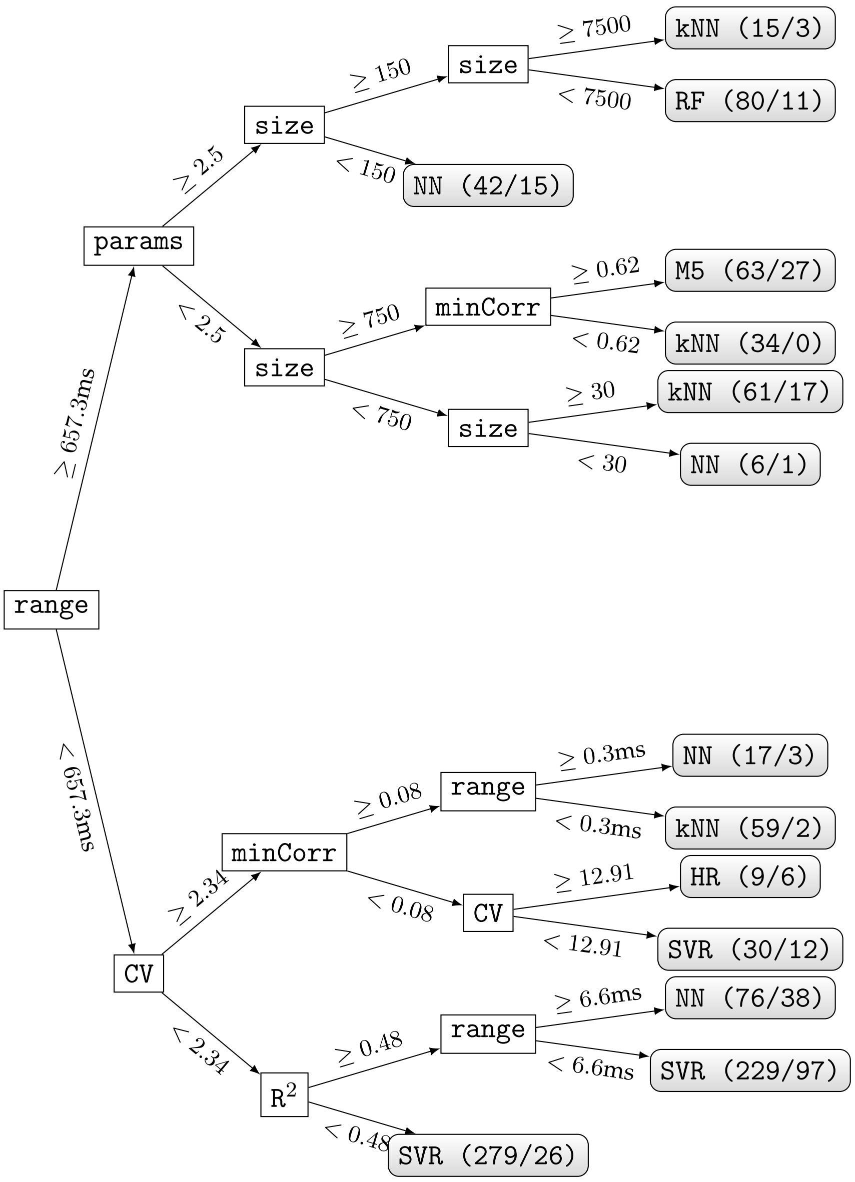

Figure 7.5: Resulting meta-selection decision tree.

We observe that a low difference between the minimum and the maximum of the measured response time values (`range`) together with a low Coefficient of Variation (`CV`) leads to the selection of SVR or NN as regression algorithms. Other important parameters that influence the selection are the number of parameters (`params`), the number of training instances (`size`), and the minimum linear correlation (`minCorr`). As expected, SVR is chosen for many of the given measurements, as it was the most prominent class in the dataset (see Table 7.2). Interestingly, kNN is also a popular choice, both for short and long measurement runs. The only approaches that are never selected by the meta-selector approach are the LR, CART, and the Bagging approach, together with the mean predictor baseline. This makes sense, as the sum of training samples of all of these algorithms combined just makes up around 6% of the total training set. However, although outperforming all approaches in only 0.7% of the cases, HR gets recommended by the meta-selector in one leaf node, albeit it being a relatively small one (15 samples).

7.3 Summary

In this chapter, we introduce *DepIC*, an approach to identify and characterize parametric dependencies for performance models. *DepIC* uses monitoring data from a running system without any further knowledge about the application, the deployment, or the component structure. This monitoring data is then analyzed, and correlations between different parameters are identified with the use of different feature selection approaches from the area of machine learning.

In addition, we investigate different techniques for the black-box characterization of these found parametric dependencies. The results show that no single approach performs well for all possible dependencies. Therefore, we construct a meta-selector that classifies a monitoring stream in order to select the best suitable regressor for it and trained it on a variety of different regressors and measurement sets.

DepIC represents a significant step towards the vision of self-aware performance models [GEK18] and introduces the respective algorithms to answer **RQ III.2** (*"How can the impact of parameters on resource demands be identified and characterized?"*). If integrated into a continuous and self-aware modeling workflow, *DepIC* enables a performance model to autonomously learn and improve itself during system operation in a production environment. Hence, it integrates with *SARDE* (see Chapter 6 on page 91) to achieve **Goal III** (*"Enable the continuous estimation and improvement of performance model parameters using production monitoring data."*).

Chapter 8

Modeling DBMS Configuration Spaces

In this chapter, we introduce *Baloo*, our approach for measuring and modeling the performance of distributed Database Management Systems (DBMSs) in cloud environments. Since microservice architectures require each service to take care of its own data [Rud19; LF14], modern software systems rely heavily on different DBMS systems for data storage [Aba+20]. Therefore, modeling, configuring, and optimizing the underlying DBMSs is still an important concern.

Similar to many other software systems, DBMSs offer a large set of parameters to adjust their internal configuration. These massively impact non-functional properties such as the performance, scalability, or availability in addition to the occupied resources and operating cost of the respective system [XYD19; Sey+19]. Distributed DBMSs are particularly challenging systems as they offer not only DBMS-specific configuration options such as consistency or availability settings but also configuration options from distributed systems, such as used cluster size or the applied replication factor [Sey+19]. Furthermore, cloud resources have become the preferred infrastructure of operating distributed DBMSs, as they provide scalability and elasticity on resource level [Sak14; Aba+20]. Unfortunately, this operational model further increases the configuration space by adding cloud-related dimensions such as resource type, storage backend, and others [Gal+18; Sey+19].

In addition to understanding the performance impact of each individual configuration, it is necessary to also understand the inter-dependencies between parameters in the overall configuration space [SD17; XYD19]. This is extremely challenging even for domain experts and therefore demands supportive methods covering the entire configuration space. Due to its size, we can not simply evaluate every configuration option [XYD19]. Instead, we need to improve decision-making by modeling and predicting the performance of the whole configuration space using only a subset of measurements.

This task is aggravated by the fact that we can not assume cloud measurements to be stable. Instead, cloud performance benchmarks and measurements are always subject to change [IYE11] and therefore require a sufficient amount of measurement repetitions in order to draw meaningful conclusions about the expected system performance.

We discuss the related work on the performance prediction of cloud-hosted DBMSs in detail in Section 3.3 on page 48. To summarize, the state of the art either focuses on single-node DBMSs ignoring distribution [Mah+17; Zhe+19], targets only specific DBMS technologies ignoring cloud resource characteristics [XYD19], or considers only cloud resources without DBMS characteristics [ZL19].

To summarize, a core challenge of modeling and optimizing the performance of distributed DBMSs is the time-intensive and expensive generation of the underlying dataset based on the following main reasons:

1. Measurements of single configuration points are costly, as they require a cluster of cloud resources that need to be reserved during the entire measurement period.

2. Performance measurements of distributed DBMS have high variability and therefore need to be repeated multiple times to achieve statistical significance [LC16; IYE11].

3. The thorough evaluation of a performance prediction approach for any configurable system requires measurements of every possible configuration, an exponentially growing space.

In this work, we, therefore, aim at solving these challenges and formulate **Goal IV** (*"Develop a workflow for modeling configurable, cloud-based, and distributed DBMSs."*). To fulfill **Goal IV**, we present *Baloo*, a novel framework for measuring and modeling arbitrarily complex configuration spaces of configurable software systems. The design of *Baloo* specifically targets distributed DBMSs, aiming to solve the specific challenges formulated above. Our approach (i) selects a suitable robust statistical measure for the given scenario, (ii) determines the minimal required number of measurement repetitions for a given measurement point, (iii) chooses the next required measurement point, and (iv) constructs a model of the configured parameter space using a machine learning model. Therefore, in this chapter, we strive to answer **RQ IV.1** (*"How can the influence of performance variabilities during benchmark measurements be mitigated?"*) and **RQ IV.2** (*"How can we analyze a configuration space that is too large to measure exhaustively?"*).

By modeling the whole configuration space, our approach can quickly extrapolate expected performance results for a given configuration without actually measuring it. This is a strong benefit over a naive black-box optimization search. Hence, the resulting DBMS performance configuration model provides the foundation for selecting a performance-optimal operation and deployment configuration of a configurable system. Besides finding the most performant configuration, it gives a better understanding of the entire configuration space, providing valuable insights for operators and architects when trading performance against other non-functional aspects such as security, reliability, and costs. Furthermore, the generic nature of the proposed framework enables researchers and practitioners to configure, adapt, and modify our approach as well as to transfer it to other domains. Although we adopt a white-box view, where the DBMS management is handled by the user, our approach can also be applied to serverless hosting scenarios in order to aid with the choice between the given options.

We worked on and published *Baloo* in collaboration with colleagues from Ulm University in Germany [Gro+20b]. Furthermore, we published the reference dataset [Sey+20] as well as a replication package [Gro+21c] which we use in the evaluation presented in Chapter 13 on page 227. The dataset consists of a total of roughly 450 measurement hours and 9 450 compute hours in a private cloud environment and supports other researchers analyzing the performance behavior of the investigated DBMS in detail and evaluate further approaches for performance prediction.

In the rest of this chapter, we first give an overview of the *Baloo* framework in the following Section 8.1, followed by a detailed description of the individual components in Section 8.2. Finally, we summarize our contributions in Section 8.3.

8.1 Overview

In this section, we present an overview of our approach. Figure 8.1 shows the workflow of *Baloo* with all involved steps designed to cope with the outlined challenges. We distinguish between an online phase and an offline phase. The offline phase is executed only once, whereas the blue steps of the online phase are executed whenever a new performance model is requested. In the following, we refer to a single configuration as a *configuration point* and to an execution of a benchmark as a *measurement run*. The overall goal of *Baloo* is to run as few measurements as possible for each configuration point and to use as few configuration points as possible to train an accurate performance model.

This performance model then enables the performance prediction of arbitrary configuration points without explicitly measuring them.

For creating the performance model, the workload runs as follows. First, the *Performance Model Construction* component step requests a training dataset from the *Configuration Point Selection* component. Based on this training dataset, the Performance Model Construction component builds a performance model and evaluates its accuracy via cross-validation. If the performance model is sufficiently accurate, it is returned to the performance engineer. Otherwise, the Performance Model Construction component iteratively increases the training set using the Configuration Point Selection component. For each requested training dataset sample, the Configuration Point Selection selects which specific configuration to measure and requests the respective measurement from the *Measurement Repetition Determination* component. This Measurement Repetition Determination module triggers the automatic performance measurements using the *Mowgli* measurement framework [Sey+19]. *Mowgli* automatically deploys a distributed DBMS to a predefined cloud infrastructure, injects the defined workload, performs performance measurements, and finally tears down the respective system. As performance measurements of distributed DBMS, especially in public clouds, experience high variability, the *Measurement Repetition Determination* triggers additional measurements until the resulting performance metrics are considered stable to be stable.

One remaining challenge of the *Baloo* framework is the aggregation of a *Mowgli* benchmark measurement run into a single value. In order to judge which configuration points are better, it is necessary to compare the outcomes of the measurement runs for different configuration points. Yet, this comparison is difficult, as the raw results of a measurement run are a time series of throughput and latency values. Instead, a higher-level metric is required that captures the quality of a measurement run for a configuration point. Therefore, Figure 8.1 shows an additional offline phase, which is concerned with the *Robust Metric Selection*. This phase first provides the means to summarize the results of individual measurement runs. The goal of this phase is to determine the robust metric that, based on a large dataset of existing performance measurements, best reduces the noise in the measurement data and is able to report a single value of interest. Using this robust metric, *Baloo* can aggregate and therefore evaluate different configuration points in order to guide the creation of a model for performance engineers. In the following, we will explain each component of Figure 8.1 in more detail.

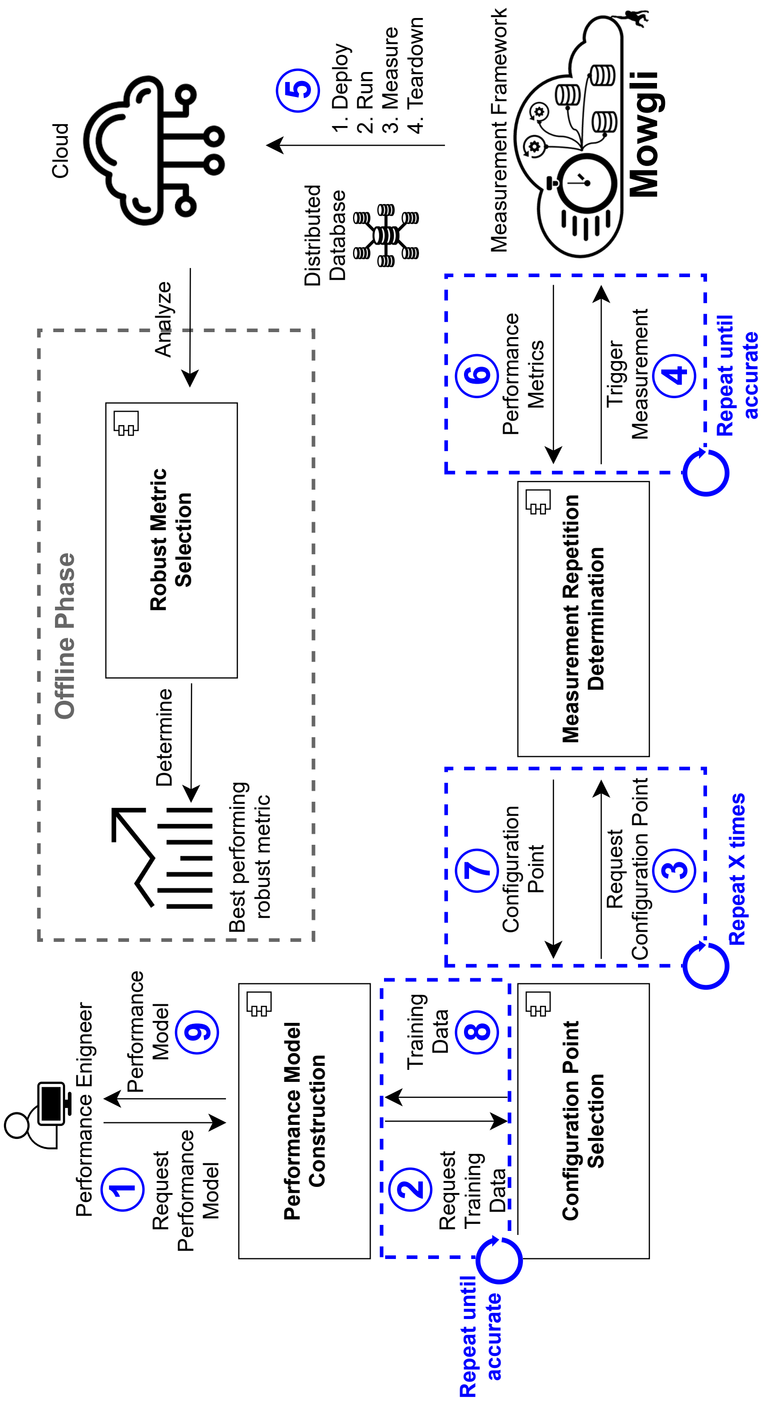

Figure 8.1: Overview of the *Baloo* workflow.

8.2 The *Baloo* Framework

In this section, we first describe the process of selecting a robust metric in Section 8.2.1, followed by the measurement of a distributed DBMS in Section 8.2.2 and the individual *Baloo* components Measurement Repetition Determination, Measurement Point Selection, and Performance Model Construction in Sections 8.2.3 to 8.2.5.

8.2.1 Robust Metric Selection

This section describes the offline phase of Figure 8.1. As already mentioned, this phase is executed only once for calibration. It is concerned with selecting a robust metric to aggregate the results of a measurement time series into a single value. By selecting a suitable metric for summarizing the performance time series of one performance measurement, we make the performance measurements we make the individual measurement runs comparable with each other. The perfect metric reports the same values for two repetitions of the same configuration point.

Therefore, our approach for finding the best robust metric tries to utilize this property. We compare different robust metrics candidates by analyzing their Coefficient of Variation (CV), a measure of the standard deviation in relation to the sample mean, over all measurement runs available. Based on the resulting list of CVs for each individual measurement, we rate each metric using the mean and the standard deviation of their respective CV scores. The metric observing the lowest mean CV is then defined to be the most suitable robust metric for the given scenario.

For potential robust metrics, we investigate common robust measures of central tendency from the literature [DD11]: The mean, the median, the 95th-, 90th-, 80th-, and 70th percentile, the trimmed mean trimming by 5%, 10%, 20%, or 30%, the winsorized mean using 5%, 10%, 20%, or 30%, the Trimean [Tuk77], and the Hodges-Lehman estimator [Leh06]. The results of the analysis of all robust metrics are presented in Section 13.2 on page 229.

8.2.2 Distributed DBMS Performance Measurement

This step lays the basis for the following steps as it generates the individual measurement runs (step 5 in Figure 8.1). For doing so, *Baloo* builds on the open-source and extensible DBMS evaluation framework *Mowgli*[1] that supports the design and execution of DBMS evaluations [Sey+19]. *Mowgli* allows to

[1]`https://omi-gitlab.e-technik.uni-ulm.de/mowgli`

define relevant domain-specific properties and allow their specification based on the supported technologies, such as the DBMS itself, cluster size, replication factor, cloud provider, resource capacity, and workload type. Furthermore, it fully automates allocation of cloud resources, deployment and configuration, workload generation, calculation of performance data, and processing of results. The resulting evaluation datasets (step 6) contain the benchmark metrics, monitoring data, resource metadata, and execution logs of the respective evaluation tasks.

8.2.3 Measurement Repetition Determination

Cloud-like environments are characterized by performance volatility [LC16; IYE11]. Therefore, multiple independent measurement runs are required to conclude the actual performance of a configuration point [Pap+19]. The more measurement runs per configuration point and the more configuration points are measured, the more precise the performance model will be. Yet, the more measurements and measurement repetitions are required, the higher are the resulting costs. Hence, *Baloo* needs to offer the capability to flexibly decide on the required quality and hence, costs.

This step addresses this problem, as it decides on the number of required measurements for a specific configuration point P. Here, the overall number of measurements depends on the desired confidence t_c and stability of the measurements, which depends on the volatility of the environment. Each individual measurement is then performed by `triggerMeasurement` as described in Section 8.2.2. *Baloo* requires at least two measurements for judging the stability of the results and uses a configurable upper limit n_{max} in order to control execution time and costs. The exact procedure is presented in Algorithm 8.1.

First, the set M of performance measurements is filled by two consecutive calls of `triggerMeasurement`(step 4 of Figure 8.1) as at least two values are required to judge the stability of the measurements. This function wraps invocations to *Mowgli* as described in Section 8.2.2. In lines 4 – 7, the obtained measurement values are analyzed and aggregated. If the calculated confidence c deceeds t_c or if n_{max} has been reached, the calculated aggregation m is returned (step 7 of Figure 8.1).

Baloo allows different implementations of outlier detection, aggregation, and confidence estimations in lines 4 – 7 of Algorithm 8.1. Our implementation used in Chapter 13 applies outlier detection based on isolation forests [LTZ08] through the Python version of Scikit-learn [Ped+11] with two isolation trees. We use the median as the aggregation function and quantify confidence through the CV of all measurements in M'. The confidence threshold t_c is set to 0.02.

Algorithm 8.1: Measurement repetition determination.

```
Input: Desired configuration point $P$, target confidence threshold $t_c$,
       maximum number of measurements $n_{max}$.
Output: Obtained measurement value $m$.
1 $M$ = triggerMeasurement($P$)
2 do
3     $M = M \cup$ triggerMeasurement($P$)
4     $M'$ = removeOutliers($M$)
5     $m$ = aggregate($M'$)
6     $c$ = confidence($M'$)
7 while $c > t_c$ and $|M| < n_{max}$
8 return $m$
```

Hence, the loop stops if $CV(M') < 0.02$. Note that both median and CV are not calculated on M but rather on M' which does not contain outliers.

8.2.4 Configuration Point Selection

This step selects the next configuration point (step 3 of Figure 8.1) that needs to be added to the training set in order to learn the behavior of the system. This process is commonly referred to as sampling [Per+19], while the underlying selection strategy used to choose the next configuration point is called the sampling method.

Pereira et al. [Per+19] conducted a survey analyzing different sampling methods in the area of learning the performance of configurable software systems. While a variety of approaches are available, random sampling is commonly used. Furthermore, a recent study could not identify a dominant sampling strategy. While our framework supports any strategy for the selection of the next configuration point, the implementation used in Chapter 13 is based on uniformly distributed random sampling. It was shown that uniformly distributed random sampling, in general, leads to the most accurate performance models [Per+20]. Although there are some exceptions, this finding is in line with a recent study by Kaltenecker et al. [Kal+20].

Our implementation of the random configuration point selection relies on enumerating the entire configuration space, which does not pose a problem in our scenario as the respective space is comparatively small. Otherwise, more sophisticated solutions are required, for example, based on binary decision diagrams [Oh+17] or satisfiability solvers [CMV13].

8.2.5 Performance Model Construction

Predicting the performance of unmeasured configuration points helps to reduce the required time and costs for creating the performance model. Therefore, this module is concerned with the construction of the actual performance model for a given configuration space S. Its iterative approach determines the minimal required number of measurements for achieving a configurable target accuracy t_S. A configurable maximum ratio r_{max} of S is used as the configurable upper limit in order to control execution time and costs. This way, Algorithm 8.2 also offers a trade-off between expected cost and solution quality.

Algorithm 8.2: Performance model construction.

```
  Input: Configuration space definition $S$, target score threshold $t_s$,
         maximum configurations ratio $r_{max}$.
  Output: Performance model $p$.
1 $C$ = getInitialMeasurements$_{init}(S)$
2 $p$ = constructPerformanceModel$(C)$
3 $s$ = scorePerformanceModel$(C, p)$
4 while $s < t_s$ and $|C| < r_{max} \cdot |S|$ do
5     $C = C \cup$ addMeasurements$_{ratio}(S, C)$
6     $p$ = constructPerformanceModel$(C)$
7     $s$ = scorePerformanceModel$(C, p)$
8 return $p$
```

Algorithm 8.2 describes the iterative process of constructing the performance model. Required inputs are a definition of the total configuration space S that is explorable, for example, the Cartesian product of all available feature values, the required target score t_s as a threshold, and the maximum ratio of configuration points r_{max} that can be explored in relation to the total configuration space.

First, Algorithm 8.2 conducts a set of initial measurements in line 1 based on the given configuration space. This is equivalent to step 2 of Figure 8.1. The result (step 8 of Figure 8.1) constitutes the set of available measurements C, from which a performance model p is built and scored using an internal scoring function. The function `getInitialMeasurements` is furthermore parameterized with the *init* parameter. This configuration parameter determines the ratio of S used for the first measurement set, that is, the number of initial measurements to conduct in relation to the size of the configuration space S.

The algorithm keeps adding measurement points in line 5 until $s \geq t_s$ or $\frac{|C|}{|S|} \geq r_{max}$. In each iteration, it recomputes p (line 6) and s (line 7). The parameter $ratio$ of `addMeasurements` determines the number of configuration points added in each iteration as a ratio of the size of S, similar to the $init$ configuration in line 1. After Algorithm 8.2 terminates, it returns the last trained performance model as step 9 of Figure 8.1.

The algorithm is highly parameterizable by adapting t_s and the scoring function. Note that the score function does not need to be bound between 0 and 1, as long as t_s is adapted accordingly. The restriction for the maximum number of measurement points via r_{max} is useful to bound the maximum costs. Otherwise, some scenarios would continue to measure the whole configuration space if the respective target threshold can not be achieved by the selected modeling type. For model construction, any regression algorithm (see Section 2.3.1 on page 24) or any other applicable performance modeling technique can be applied. The implementation evaluated in Chapter 13 on page 227 compares a set of different regression algorithms, applies a three-fold cross-validation score on C to determine model accuracy, and uses variance as a score function.

8.3 Summary

In this chapter, we presented *Baloo*, a framework for measuring, modeling, and predicting the performance of distributed DBMSs in cloud environments for different configurations. Therefore, *Baloo* meets the target of **Goal IV** (*"Develop a workflow for modeling configurable, cloud-based, and distributed DBMSs."*). Our approach builds upon the *Mowgli* framework and works by (i) measuring a performance configuration, (ii) determining the number of measurement repetitions, (iii) determining the next configuration point to be measured, and (iv) building a performance model using all available measurement points to predict the remaining unavailable measurement points of the configuration space. Especially the first two steps help to mitigate the performance fluctuations during cloud measurements and therefore answer **RQ IV.1** (*"How can the influence of performance variabilities during benchmark measurements be mitigated?"*), formulated in the introduction of this chapter.

By modeling the whole configuration space, our approach can quickly extrapolate expected performance results for a given configuration without actually measuring it. This answers **RQ IV.2** (*"How can we analyze a configuration space that is too large to measure exhaustively?"*). Therefore, *Baloo* can not automatically find the most performant configuration for a distributed DBMS but also provide a better understanding of the entire configuration space. As the presented algorithms are highly configurable, *Baloo* offers a trade-off between the maximum measurement effort (in terms of measurement time and costs) and the resulting model confidence. We evaluate *Baloo* in Chapter 13 on page 227.

Part III

Evaluation

Chapter 9

Evaluating the Detection of Resource Saturation

In this chapter, we evaluate the *Monitorless* approach introduced in Chapter 4 on page 53. We analyze three applications that are not included in the training phase. Namely, one three-tier web service [PSF16] and two microservice-based e-commerce applications. The microservice applications are (i) TeaStore [Kis+18], composed of five microservices, and (ii) Sockshop, composed of sixteen microservices. During the presented experiments, we focus on the following Evaluation Questions (EQs) in order to fulfill **Goal I** (*"Design an application-agnostic approach for the detection of resource saturation based on platform-level monitoring data."*):

- **EQ 9.1**: *Is the Monitorless model able to accurately detect resource saturation of unseen applications?*
- **EQ 9.2**: *Is the trained model able to accurately detect resource saturation on unseen hardware setups and interference patterns?*
- **EQ 9.3**: *Is the performance of the holistic model consistent on different target applications?*
- **EQ 9.4**: *Are latency and overhead feasible for online environments?*

If the answers to all the listed EQs are positive, we are able to show that it is possible to design the envisioned holistic and application-agnostic prediction model solely based on platform-level monitoring (**Goal I**). In the rest of this chapter, we introduce the TeaStore in Section 9.1 in more detail, as we also use this application in later evaluations. Next, we describe the experiment setup in Section 9.2. In Section 9.3 on page 156, we discuss the results of the three-tier web service. Following, we evaluate *Monitorless* using the two microservice applications in Section 9.4. We conclude this chapter in Section 9.5.

9.1 The TeaStore Microservice Benchmarking Application

Throughout this thesis, we utilize the TeaStore microservice benchmark application to evaluate our approaches were applicable. In particular, we evaluate the performance of *Monitorless* in Chapter 9, *SuanMing* in Chapter 10, and *DepIC* in Chapter 12 with the TeaStore. The TeaStore[1] is an open-source and distributed microservice benchmark application that represents a webshop for tea [Kis+18; Kis+19a]. Users can browse the available products by category and look at individual products. After logging in, the user can add items to the shopping cart, modify the content of the shopping cart, and checkout by entering shipping and payment information. Previous orders are tracked on the user's profile page. TeaStore displays advertisements for other products based on the user's previous orders, the current shopping cart, and the current item or category. We also contributed to creating and publishing the TeaStore application [Kis+18]. Furthermore, the TeaStore is peer-reviewed and accepted by the SPEC RG [Kis+19a].

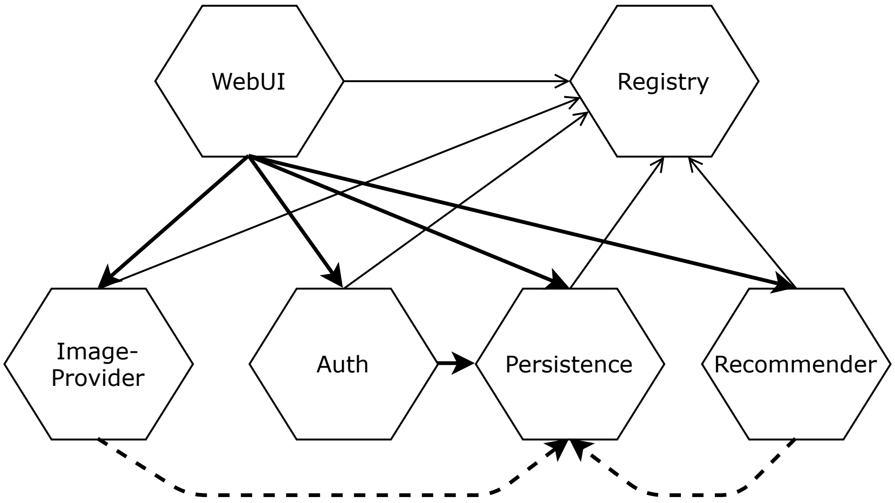

Figure 9.1: Overview of the TeaStore services and their communication patterns.

TeaStore consists of five services, a database, and a service registry, as shown in Figure 9.1. The `Database` service is not shown in Figure 9.1 as it just interacts

[1]`https://github.com/DescartesResearch/TeaStore`

with the persistence service and plays a minor role in our evaluations. The `WebUI` service delivers static web pages and fills them with dynamic information by querying the other four services. The `ImageProvider` delivers and caches the product images. Password hashing and session validation are managed by the `Auth` service. The `Persistence` service encapsulates access to the database and provides a caching mechanism. Finally, the `Recommender` service tailors the displayed advertisement to the current user by training a machine learning model on the previous orders. The different arrow types depicted in Figure 9.1 define the type of communication between the services. All services interact with the `Registry` but just send heartbeat and status updates. In addition, the `ImageProvider` and the `Recommender` services access the `Persistence` service only on start-up and do not send further requests during normal operation. All inter-service communication is based on REST APIs. To summarize, TeaStore uses a modern technology stack and is representative of current distributed microservice applications [Kis+18], which is why we utilize it in the following experiments.

9.2 Experiment Design

To measure the accuracy of *Monitorless*, it is necessary to determine whether or not a predicted label is correct. The threshold for determining saturation of the overall application is discovered by running a linearly increasing load test, as described in Section 4.1.2 on page 57, which yields a set of ground-truth labels $y_{\mathcal{A},t} = \tilde{\mathcal{P}}_{\mathcal{A}}(t)$ for each time t. Contrary to the training phase where only single-container services were considered, here we apply *Monitorless* to applications composed of multiple services. For scaling, our strategy to decide whether the application is experiencing resource saturation is to take the logical OR of the inferences over all instances of services comprising the application, where a predicted saturation is interpreted as `true`. In other words, our prediction vector is $\hat{y}_{\mathcal{A},t} = \bigvee_{\mathcal{I}\in\mathcal{S},\mathcal{S}\in\mathcal{A}} \hat{y}_{\mathcal{I},t}$. While other use cases might require different aggregations, OR should be sufficient for scaling instances. Although application performance may not show degradation when short saturation occurs in some components, scaling saturated instances is desirable, even if it does not directly influence the end-to-end latency.

9.2.1 Metrics

We compare the prediction $\hat{y}_{\mathcal{A},t}$ to the ground-truth label $y_{\mathcal{A},t}$ obtained by the threshold analysis $\tilde{\mathcal{P}}_{\mathcal{A}}(t)$, in order to evaluate the performance of our algorithm.

We use the notion of TPs, TNs, FPs, and FNs as defined in Section 2.3.2.2 on page 30. Therefore, the goal of *Monitorless* is to minimize FPs and FNs. We analyze the performance using the standard metrics of accuracy and F1 score (see Section 2.3.2 on page 27). Typically, the costs associated with each FP and FN are inherently asymmetric in practice because avoiding increasing latencies (FNs) could be more critical than avoiding unnecessary scaling decisions (FPs) or the other way around. Our solution supports adaptation towards this asymmetry by manipulating the prediction threshold of our classifier towards the direction of preference. In this work, we aim to minimize FNs by being more conservative with our predictions. Therefore, we set the prediction threshold of the RF classifier to 0.4.

A critical issue discovered during the preliminary evaluation is that many FPs/FNs and ground-truth saturated samples are close in time but not precisely aligned. For example, many FPs are followed by an FN within one or two samples. The reason for this delay is that saturated applications have increased response times. During peak periods, we observe response times of up to three seconds. After three seconds, a request is usually dropped by our load generators. Since requests take longer to arrive back to the load generator during these peak periods, a gap is introduced between the recorded platform metrics and the ground-truth labels, which are both monitored at a one-second interval. This is an issue since our ground-truth labeling considers that all responses arrived within the current one-second interval, and hence positive labels in the ground truth are frequently delayed.

To fix this, we introduce *lagged metrics*. For a given lagged metric FP_k (to be read as "false positives at distance k"), a false positive prediction is classified as such only if there are no ground-truth saturated occurrences within the next k samples. Analogously, for FN_k, a false negative for a ground-truth saturated sample occurs only if the previous k samples are predicted as non-saturated. Hence, if a false positive occurs at time t and a ground-truth saturated sample occurs in the time range $[t+1, t+k]$, then the sample at time t is classified as a true negative TN_k. If a false negative occurs at time t, and a positive prediction occurs in the time range $[t-k, t-1]$, then such samples are added to TP_k. Therefore, $FP_0 = FP$ and $FP_i \geq FP_{i+1}$.

This modification allows early saturated predictions to be transferred to later saturated ground-truth samples, thereby handling the above-mentioned application latency issues. Importantly, the symmetric case of a late prediction is still classified as incorrect by this metric. Therefore, after the saturation was already observed at the client, the predicted samples are still classified as FNs, if the saturated prediction a few samples later. Based on this definition and the fact that our peak response times were limited to three seconds, we perform our evaluation with $k = 2$ (note that k is specific to the applications we use in this evaluation). Thus, in all subsequent discussions, we use the $F1_2$ score and Acc_2, which are defined analogously to the F1 score and accuracy, but for the lagged metrics. This step is necessary for coping with monitoring delays in the real system that cause predictions to be misaligned with the ground truth. By doing so, we align the predicted data with the observed data in order to appropriately judge the behavior of the system.

9.2.2 Baselines

As approaches from related work (see Section 3.1.1 on page 36) follow a different paradigm, we compare *Monitorless* against four optimal baseline approaches based on static CPU and RAM thresholds. The first is CPU-threshold, based on the relative CPU usage of each service instance. The second is MEM-threshold, based on the relative RAM or memory usage of each service instance. We also look at a disjunctive CPU-OR-MEM and conjunctive CPU-AND-MEM combinations, where instances are predicted as saturated if CPU or/and MEM indicate saturation, respectively. We use CPU and RAM for demonstration as they are sufficient for the evaluated applications, but any other resource metric can be used as well.

We note that the considered baseline approaches have an unfair advantage over *Monitorless* in that they are configured with knowledge of the entire input data in advance, including ground-truth labels. These ground-truth labels are then used to choose the optimal threshold that maximizes the F1 score. The presented baselines, therefore, represent the best possible outcome for threshold-based approaches, given that the threshold would be optimally configured. In practice, these thresholds are unrealistic to find since they need to be configured before deployment based on expertise and understanding of the relationship between application performance and resource usage. This task is not required when using *Monitorless* as it works application-agnostic. In addition to the four platform-metric baselines, we use one application-specific baseline based on the actual KPI measurements in Table 9.3 serving as the upper bound.

9.3 Three-tier Web Application

The first validation application is based on the social networking engine Elgg[2] included in Cloudsuite [PSF16] and is designed to answer **EQ 9.1** (*"Is the Monitorless model able to accurately detect resource saturation of unseen applications?"*).

9.3.1 Setup

The hardware used was identical to the hardware used in Section 4.2.2.2 on page 62 to create the training datasets. The test application consists of three tiers, each deployed in their own service instance: (1) the front-end Elgg web server, (2) the InnoDB database, and (3) Memcached to speed up the database-driven application [Nis+13]. Given that Memcached and a similar database are already included in the training set (see Section 4.2.2 on page 60), we stressed the front-end tier by sending static requests to access the web server's index page. This ensures that the bottleneck is the front-end, a service on which the *Monitorless* model is not trained.

We deploy all service instances as containers using Docker on one physical machine. The Elgg container is assigned with one CPU core and 4 GB of memory; the other containers are unconstrained. The PCP monitoring agent is running on this machine and sends metrics to a second orchestrator machine. The workload is generated by a third machine running the HttpLoadGenerator[3] tool [KDK18]. The workload pattern is similar to `sinnoise1000` (see Table 4.1 on page 63) but scaled down to 1/10th of the intensity, as the web server could handle fewer requests than Solr.

Table 9.1: Comparing different baseline approaches to *Monitorless* using a three-tier web serving application.

Algorithm	TN_2	FP_2	FN_2	TP_2	$F1_2$	Acc_2
CPU (97%)	610	8	0	1838	0.999	0.997
MEM (43%)	544	74	14	1824	0.976	0.964
CPU-OR-MEM	538	80	0	1838	0.978	0.967
CPU-AND-MEM	616	2	14	1824	0.995	0.993
Monitorless	607	11	0	1838	0.997	0.995

[2]`https://elgg.org`

[3]`https://github.com/joakimkistowski/HTTP-Load-Generator`

9.3.2 Results

Table 9.1 shows the results of all baseline approaches compared to our model evaluated using the lagged metrics. We observe that the test samples have a ratio of saturated samples to non-saturated samples of about on to three, which is the inverse of the one in the training phase for *Monitorless*. This indicates that the model is not biased towards the majority training label. As the front-end component is heavily CPU-based, the CPU detector can effectively flag saturation. Observe that *Monitorless* is very accurate, achieving no false negatives and only a few false positives, even without being tuned for the application. We can, therefore, confidently answer **EQ 9.1** (*"Is the Monitorless model able to accurately detect resource saturation of unseen applications?"*). However, as both CPU and CPU-AND-MEM threshold-based approaches also perform well, we provide next an evaluation with more complex applications to showcase the advantage of *Monitorless*.

9.4 Multi-tenant Environment

In this section, we evaluate *Monitorless* in a more realistic multi-tenant scenario, where several microservice-based applications are running in a distributed environment. We aim to show the accuracy of our model, as well as its robustness to changes in the hardware configuration and underlying OS. Therefore, the services are running on a different OS than the one used during the training phase. Therefore, the following experiments now focus on **EQ 9.2** (*"Is the trained model able to accurately detect resource saturation on unseen hardware setups and interference patterns?"*) and **EQ 9.3** (*"Is the performance of the holistic model consistent on different target applications?"*).

9.4.1 Setup

In contrast to the previous experiments, the hardware for this test is composed of three different HP ProLiant DL360 Gen9 servers equipped with 32 GB of RAM and a 10-core Intel® Xeon® CPU E5-2650 v3 (Haswell) @ 2.30 GHz (M1), a 12-core Intel® Xeon® CPU E5-2650 v4 (Broadwell) @ 2.20 GHz (M2) and, finally, an 8-core Intel® Xeon® CPU E5-2640 v3 (Haswell) @ 2.60 GHz (M3). M1 and M2 run Debian 9, whereas M3 is running Ubuntu 16.04. We use two similar servers as workload drivers (M4, M5). These servers are connected via a 1 GB network. Recall that we used a 10 GB network for training. Using a different OS compared to the training is unproblematic, as PCP supports a wide range of OSs. This proves the general portability of the trained model between different OS distributions. We distribute the following two test applications among the three test machines M1, M2, and M3.

Our first evaluation application, the TeaStore [Kis+18], was already introduced in Section 9.1. It consists of a total of seven containers that need to be deployed (`WebUI`, `Auth`, `ImageProvider`, `Recommender`, `Persistence`, `Database`, and `Registry`). The second application is Sockshop[4]. Sockshop is a similar online storefront application but slightly larger than TeaStore. It consists of 14 different services: `Edge-Router`, `Front-end`, `Payment`, `Catalogue`, `Catalogue-DB`, `Carts`, `Carts-DB`, `User`, `User-DB`, `Order`, `Order-DB`, `Shipping`, `Queue`, and `Queue-Master`. In order to represent a multi-tenant environment, we deployed all TeaStore and Sockshop services on M1, M2, and M3 servers as follows. Entries marked with (T) are TeaStore services; all others belong to Sockshop.

- **M1**: `Recommender` (T), `Auth` (T), `Registry` (T), `Catalogue`, `Catalogue-DB`, `Front-end`, `Queue`.
- **M2**: `Database` (T), `Persistence` (T), `Edge-Router`, `Carts`, `Carts-DB`, `Order`, `Order-DB`, `Payment`, `Queue-Master`.
- **M3**: `WebUI` (T), and `ImageProvider` (T), `User`, `User-DB`, `Shipping`.

All containers have a memory limit of 4 GB. The `Auth` (T), `Catalogue-DB`, `Carts-DB`, `Order-DB`, and `User-DB` are assigned to two CPU cores. All other services are limited to one core. Additionally, we switched from remote logging with PCP to local logging, as otherwise, the limited bandwidth network bandwidth creates a bottleneck.

TeaStore Load Generation We stressed the TeaStore application with the HttpLoadGenerator [KDK18]. The request contents are defined using stateful user profiles. Each time a request is sent, an idle user from a pool is selected to execute a single action on the store. The actions available to the user are: 1) log in, 2) browse the store for products, 3) add these products to the shopping cart, and 4) log out. The user's distinct actions stress the services in different ways. The number of users is chosen depending on the maximum load intensity, such that it guarantees that an idle user is available each time a request is sent. The arrival rate profile represents a realistic but worst-case workload for clouds [SBI15] with more variance and multiple daily patterns within the experiment. It is depicted in Figure 9.2. Note that the training is done on mostly

[4]`https://microservices-demo.github.io`

smooth workloads. We purposely choose this challenging workload to evaluate the robustness of *Monitorless*.

Sockshop Load Generation The load is generated via Locust[5] using the standard user profile delivered by Sockshop. Users log in, browse the catalog, add items to their cart, and then place orders. The load intensity is controlled via the number of concurrent users issuing requests to the system. Locust usually applies a constant load once all clients are hatched. In order to apply varying load patterns, we start three different Locust runs in parallel. Each run takes 1 000 seconds and slowly hatches all clients until a maximum load of 700 concurrent clients is reached after 700 seconds. A constant load pattern is then applied for the remaining 300 seconds before one run finishes. We start such runs after 1 000, 3 000, and 5 000 seconds in parallel to the continuously running TeaStore workload.

9.4.2 TeaStore Results

Applications containers are dimensioned such that most of the target load can be handled; that is, only large load peaks cause the application to saturate. This leads to a saturation-to-non-saturation ratio of 2.9%, far lower than the training data and the previous evaluation in Section 9.3. By analyzing the predictions of *Monitorless*, we note that most TP_2 are for the `Auth` service, the `WebUI`, and the `Recommender` service. Figure 9.2 displays a detailed breakdown of the predictions made by *Monitorless* over time, also showing the injected workload and the measured response times. Green dots mark TP_2, yellow dots mark FP_2, and red dots are FN_2. TNs are not shown since they are the most common and easily predicted. The gray curve displays the workload intensity (in requests per second) over time. The purple curve displays the measured average response time per second.

We observe that *Monitorless* is able to detect saturation occurring in different types of services (`WebUI`, `Recommender`, `Auth`, `Registry`, and `Persistence`), each with its own resource bottleneck. Note that we can not determine the distribution of FN_2 among the individual services as the KPI function $\mathcal{P}_{\mathcal{A}}$ is only observable at the application level and not at the service level. However, we verify that the services that *Monitorless* predicted to be saturated were indeed saturated by subsequently scaling the saturated components and re-injecting the same load pattern. Thus, we repeat the same test with additional instances (scaling) of the `Recommender`, and the `Auth` service on M2 and `WebUI` on M3,

[5]`https://locust.io`

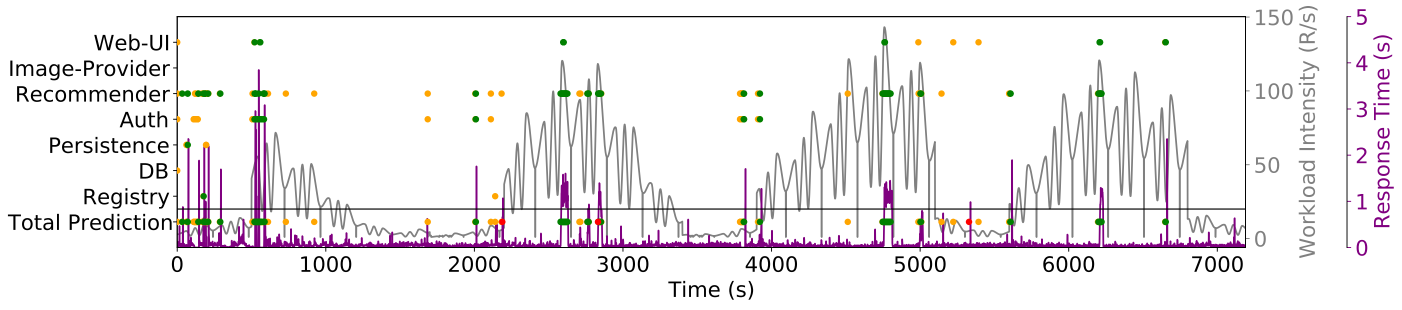

Figure 9.2: Predictions and measurements of the TeaStore services over time.

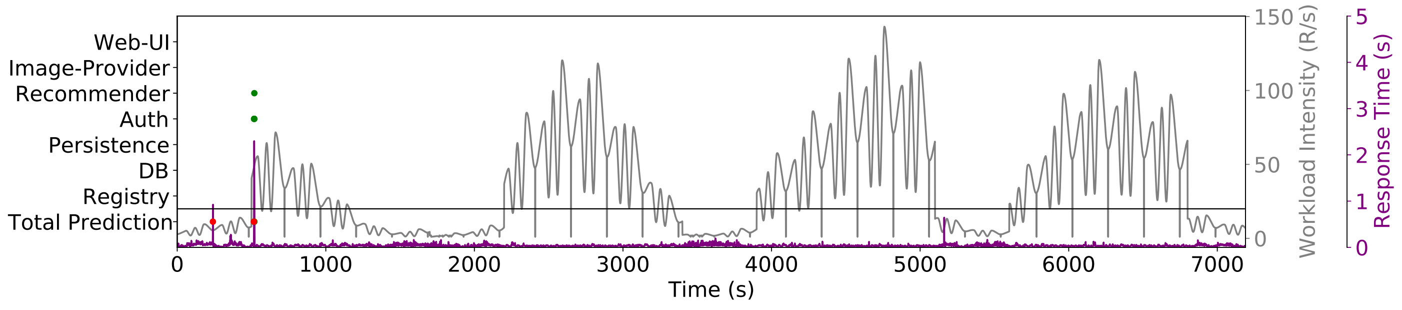

Figure 9.3: Predictions and measurements of the TeaStore services over time, after the proposed scaling is applied.

since those were the three services causing the most saturation predictions. The result of this repeated run is shown in Figure 9.3.

In addition, we analyze the performance of *Monitorless* in more detail by comparing it to the baseline approaches in Tables 9.2 and 9.3. Table 9.2 shows the comparison between *Monitorless* and the baseline approaches for the run depicted in Figure 9.2 in terms of sample-based prediction accuracy. Note that *Monitorless* achieves an $F1_2$ score of 0.712 compared to the best baseline approach with 0.738. Although the $F1_2$ score is lower than in the previous examples, it still amounts to an accuracy of 0.977 with only three false negatives. In contrast to the previous experiment, the CPU-AND-MEM has the best score but at the expense of a higher FN_2 count, which implies that it fails to detect crucial saturation samples. Recall that we deliberately lowered the prediction threshold in order to minimize FNs. Therefore, the comparatively high amount of FPs could also be reduced at the expense of more FNs.

Table 9.2: Comparing different baseline approaches to *Monitorless* on the TeaStore dataset.

Algorithm	TN_2	FP_2	FN_2	TP_2	$F1_2$	Acc$_2$
CPU (95%)	6805	179	7	202	0.685	0.974
MEM (90%)	228	6756	4	205	0.057	0.060
CPU-OR-MEM	180	6804	1	208	0.057	0.054
CPU-AND-MEM	6853	131	10	199	0.738	0.983
Monitorless	6820	164	3	206	0.712	0.977

To show this effect in more detail, we analyze the end-to-end application performance for an autoscaling scenario. We start with the baseline deployment of one repetition of each service and scale up when saturation is predicted (see Figure 9.2). All replicated services have a lifespan of 120 seconds, after which the service is scaled scaled down in order to simulate an autoscaling environment. This is the same for all analyzed approaches. Results are shown in Table 9.3, whereas the resulting performances are depicted in Figure 9.3. Table 9.3 reports the additional container provisioning related to the baseline non-scaled application (see the second column) and the number of SLO violations (see the third column) incurred by each technique (see the first column). The average provisioning is calculated as the percentage of containers elastically added to the baseline case (i.e., the non-scaled initial deployment). Using the approach in Section 4.1.2 on page 57, we indicate SLO violations when the average response time of all requests is higher than 750 ms, if any request is dropped due to overload, or if more than 10% of requests fail during each one-second interval.

Note that we add the baseline worst-case *No Scaling* for reference. This baseline has static resources over the execution and allows us to understand how the different techniques provision containers upon scaling out events. Intuitively, the number of SLO violations should decrease with larger provisioning (i.e., the system is more elastic). However, larger provisioning incurs a higher cost. Similarly, we also include the optimal response time autoscaler (RT-based), which is based on a-posteriori end-to-end latency measurements. As the optimal autoscaler does not know which instance causes a latency increase, we use our application knowledge to scale the `WebUI`, the `Auth`, and the `Recommender` service when the latency increases. (Note that this again implies application knowledge that *Monitorless* does not have.) For a fair comparison, all approaches are tied to scaling the `WebUI`, the `Auth`, and `Recommender` at the same time. Therefore, all three services are scaled if one of them is predicted as saturated.

Results are in line with our previous observations from Table 9.2. The optimal approach is able to reduce the SLO violations from 183 to 1 by using 7% more resources. This is expected as the approach based on response time directly observes the response time that is used as SLO. The remaining violations are due to the natural reaction time of the approaches and are therefore unavoidable.

Table 9.3: Comparing different scaling approaches on the TeaStore dataset.

Algorithm	Additional provisioning	SLO violations
A-posteriori CPU (95%)	+12%	12
A-posteriori MEM (90%)	+33%	9
CPU-OR-MEM	+39%	4
CPU-AND-MEM	+9%	17
Monitorless	+10%	7
No Scaling (baseline)	0%	183
RT-based (optimal)	+7%	1

We observe that *Monitorless* manages to effectively reduce violations while provisioning only 10% additional resources. Consider that the optimal approach provisions 7% in excess. The only approach that is cheaper in terms of resource provisioning is the CPU-AND-MEM approach. However, CPU-AND-MEM allows more than twice as many SLO violations while saving only 1% in comparison to *Monitorless*, which is not a favorable trade-off. Both CPU-OR-

MEM scaler and the MEM scaler provision 3 to 4 times more containers than *Monitorless* due to the higher number of FPs_2, as shown in Table 9.2. The CPU scaler generally makes a reasonable trade-off between cost and violations but performs considerably worse than *Monitorless* in both dimensions.

Although there is room for improvement with *Monitorless*, we believe its performance on the TeaStore application is very impressive considering that: 1) it has never been trained with any of the TeaStore services; 2) it is running on slightly different hardware and OSs than what it was trained on; 3) it is not using application metrics to make predictions; 4) it is operating in the presence of interference from another application, and, finally; 5) it achieves this performance with no knowledge of how the TeaStore behaves for this workload, unlike the threshold-based baseline approaches. Therefore, we can answer **EQ 9.2** (*"Is the trained model able to accurately detect resource saturation on unseen hardware setups and interference patterns?"*) with "Yes", as the setup was consciously chosen to be different.

Furthermore, the experiment shows that *Monitorless* can detect bottlenecks on co-located services stressing different resources. Scaling these services results in a significant reduction of response time, where *Monitorless* achieves results comparable to a simple autoscaler but without using any application knowledge. Note that even though the baseline approaches use perfect information, that is, application-level metrics as ground-truth with a-posteriori knowledge of the evaluation set, there is no approach in the evaluation that outperforms *Monitorless* in all scenarios. Moreover, the specifically tuned thresholds differ for each alternative approach, whereas *Monitorless* performs consistently across all experiments without tuning.

9.4.3 Sockshop Results

Compared to TeaStore, the performance of the larger and more complex Sockshop application is more challenging to predict, especially considering the interference from other services. (Recall that during Sockshop operations, 21 different services are running on the three assigned hosts.) The Sockshop runs are similar to those of TeaStore but contain only 999 samples each, resulting in a total of 2 997 samples. The results are shown in Table 9.4. The percentage of saturated samples is 10.1%.

We observe that both the FP_2 and the FN_2 rates are higher than in the TeaStore experiment for *Monitorless*. Overall, *Monitorless* achieves an $F1_2$ score of 0.598 together with an accuracy of 89%, only surpassed by the CPU-AND-MEM baseline, with an $F1_2$ score of 0.699. As our aggregation function over services is the logical OR of all predictions, it naturally creates more FPs as we increase

Table 9.4: Comparing different baseline approaches to *Monitorless* using the Sockshop dataset.

Algorithm	TN_2	FP_2	FN_2	TP_2	$F1_2$	Acc_2
CPU (99%)	2036	657	3	301	0.447	0.780
MEM (10%)	683	2010	8	296	0.227	0.327
CPU-OR-MEM	595	2098	2	302	0.223	0.299
CPU-AND-MEM	2604	89	93	211	0.699	0.939
Monitorless	2418	275	57	247	0.598	0.889

the number of services. Also, note that due to the higher amount of individual services, more individual service predictions need to be done, which increases the chance of FP predictions. Hence, in order to tackle larger applications, this experiment motivates a more sophisticated approach for aggregation. However, note that the baseline approaches apply different thresholds than in the other experiments (Web serving: 97%/43%, TeaStore: 95%/90%, Sockshop: 99%/10%). In contrast, the *Monitorless* model was unmodified throughout this whole evaluation. This illustrates that *Monitorless* continues to perform well, even on significantly more complex applications, without having to manually adapt it to different application-level metrics. In summary, we argue that **EQ 9.3** (*"Is the performance of the holistic model consistent on different target applications?"*) can be answered with "Yes". Although the accuracy of *Monitorless* is noticeably changing between different applications, the holistic modeling enables it to function without adaptation, while all baseline approaches need to be reconfigured with a-posteriori knowledge.

9.4.4 Experimental Repeatability

In this section, we aim at verifying whether the experimental results presented in Table 9.2 were statistically significant by repeating the same experiment three times. Table 9.5 compares the $F1_2$ of *Monitorless* with our four different baseline approaches over three different experiment repetitions. We excluded the accuracy, as it observes less variation due to the high number of high TNs in the datasets and is, therefore, less insightful than $F1_2$.

We observe that all approaches are subject to strong fluctuations between the different experiment repetitions. The $F1_2$ of the *Monitorless* model has a variation of almost 0.1 between repetition one (R1) and repetition two (R2). However, we observe that the performance of the baseline approaches drops even further for R2, with over 0.15 for MEM and CPU-AND-MEM. In addi-

Table 9.5: Comparison of repeatability of the TeaStore dataset over three repetitions.

	Monitorless	CPU	MEM	CPU-OR-MEM	CPU-AND-MEM
R1	0.772	0.067	0.346	0.067	0.346
R2	0.689	0.050	0.186	0.050	0.187
R3	0.710	0.059	0.270	0.059	0.270
Tab. 9.2	0.712	0.685	0.057	0.057	0.738

tion, *Monitorless* outperforms all baseline approaches by a large margin on all three repetitions. This again shows the fragility of the configured optimized a-posteriori thresholds of the baseline approaches. Although the thresholds perform equally well to *Monitorless* in Table 9.2, the performance of these approaches immediately drops when repeating the experiment in Table 9.5, as subtle changes in the measurement (e.g., a 1 – 2% average increase in CPU measurement) can strongly influence their prediction behavior. Although *Monitorless* is also affected by these variations, its performance has shown to be much more stable during the different runs than the statically configured threshold approaches.

9.5 Summary

In this chapter, we evaluate a preliminary *Monitorless* model trained on workloads produced by four different benchmark services by analyzing its performance on three unknown applications composed of three, five, and fourteen different microservices, respectively. Our evaluation provides evidence that it is feasible to create a single holistic model that can accurately predict resource saturation while being application-agnostic.

Even for complex microservice applications operating in the presence of interference, the accuracy achieved by our approach can be as high as 97%. This motivates the inclusion of even more test services with diverse resource boundaries to improve predictions and detect performance degradation across multiple resources.

In summary, we are able to answer four different evaluation questions in order to evaluate whether we can fulfill **Goal I** (*"Design an application-agnostic approach for the detection of resource saturation based on platform-level monitoring data."*). We show that **EQ 9.1** (*"Is the Monitorless model able to accurately detect resource saturation of unseen applications?"*) can be answered by analyzing three different microservice applications during our experiments. Additionally, **EQ 9.2** (*"Is the trained model able to accurately detect resource saturation on unseen hardware setups and interference patterns?"*) can be addressed as we include different hardware and interference patterns in Section 9.4. By comparing the accuracy of *Monitorless* on different applications, we are also able to answer **EQ 9.3** (*"Is the performance of the holistic model consistent on different target applications?"*). Finally, we already discuss the training overhead and prediction time for the different model types in Section 4.2.4 on page 68. Therefore, we can also answer **EQ 9.4** (*"Are latency and overhead feasible for online environments?"*), as the overhead of all modeling approaches was within the feasible range for employment in a real cloud environment due to classification times way below one second. In the following, we discuss some remaining threats to validity, together with limitations and assumptions in Section 9.5.2

9.5.1 Threats to Validity

We group the following threats into concerns regarding the internal or the external validity of our evaluation.

9.5.1.1 Internal Validity

We already discussed the implication of the applied metrics in-depth in Section 9.2.1. Generally, the lag parameter k strongly influences the results of all approaches. However, we argue that $k = 2$ is a reasonable setting for all our experiments, as our load generator starts dropping requests after 3 seconds of delay. In addition, the lagged metrics do not invalidate our conclusions as they impact the performance of all approaches at the same time.

An additional threat to validity is the presented scaling analysis depicted in Figure 9.3 and analyzed in Table 9.3, as it applies a set of strong assumptions. For example, we do not conduct the scaling in an online environment but record different independent runs, where one was applying the baseline and another used the scaled deployment. During the prediction steps, rather than changing the deployment, we switch between the different monitoring streams that the individual approaches are given. This is done as the cloud experiments are inherently noisy, and we need to ensure that all compared approaches see the same monitoring stream and receive the same reaction after a scaling decision. Otherwise, a delay in, for example, the start-time of a container could influence the results.

In addition, the optimal application-level autoscaler based on response time has no information about the individual service metrics, as it is just based on comparing the end-to-end response time with a pre-defined threshold. We, therefore, had to tie its scaling decisions to the `WebUI`, `Recommender`, and `Auth` services. Consequentially, all other approaches are needed to tie these services together as well. This step was necessary to ensure fairness. Otherwise, the other autoscaling could theoretically scale cheaper than the optimal approach, as they would only scale one of the two services at a time.

Finally, as neither *Monitorless* nor the compared approaches have a notion for under-saturation (i.e., a label for scaling down), we applied a relatively simple time-based down-scaling rule. A pre-defined time interval after each up-scaling, the system is scaled down. This time interval is independent of the approach and was set to 120 seconds in this experiment. However, all approaches could react with an immediate up-scaling after a forceful down-scaling decision in order to avoid performance losses.

We are aware that the presented assumptions limit the interpretability and generalizability of the results. However, we do not see *Monitorless* as an autoscaling solution but rather an approach for detecting services that experience resource saturation without utilizing application-level knowledge. Therefore, the presented autoscaling scenario simply aims at demonstrating one of many use cases (including, for example, resource consolidation, KPI degradation prediction, service type classification, bottleneck identification, or detection of resource leaks) in that *Monitorless* can be applied. Its main purpose is to validate whether the results and the conclusion drawn from Table 9.2 and Figure 9.2 can be supported and utilized in such a use case.

9.5.1.2 External Validity

We are aware of the diversity of microservices and cloud platforms that can generate mixed bottlenecks and more complex resource utilization patterns. Although we actively tried to employ different microservice applications, the expressiveness of the evaluation is limited to the evaluated service types and application categories. Therefore, to further generalize *Monitorless* while improving its accuracy and robustness, we propose to incorporate datasets with different resource saturation samples, together with different and noisy workload patterns, applications with different resiliency, scaling, and load balancing setups.

Additionally, our generated feature set was specifically tailored to the obtained dataset during training. Therefore, the application of the feature set on other applications in practice or its general applicability to other training datasets remains to be shown. Our methodology intentionally removes features with less information gain. Therefore, other training applications that stress prior unseen features naturally also require the addition of different features. Future work could validate whether our methodology for engineering the features proves to be valid for other datasets and application ranges.

As our goal was to minimize FNs during this evaluation, the prediction probability threshold of the applied RF algorithm was set to 0.4. Therefore, the achieved results may require additional calibration to infer the performance of applications with higher or lower confidence, as we could not validate whether or not this threshold was specific to our tested scenarios or generally transferable to other usage patterns as well.

We ran the experiment presented in Section 9.4 a total of three times in order to test the reproducibility of our evaluation. The results of *Monitorless* were consistent within variations of 10%, as Table 9.5 shows. Unfortunately, the optimal thresholds of the baseline approaches were not consistent between the different repetitions, which made the comparison with the *Monitorless* model infeasible. Although we could show that *Monitorless* generally produces more stable results than the baseline approaches, no statistically significant difference could be observed due to the strong variations between the individual runs. This, therefore, remains a threat to the validity of this evaluation.

9.5.2 Limitations and Assumptions

We have shown that *Monitorless* is generic and feasible in practice for predicting performance degradation. However, there are still some limitations requiring further research. In this section, we elaborate on some limitations and discuss how some could be addressed.

One central limitation of our work was that we excluded the network overhead in our experiment by relying on local logging in Section 9.4. This was necessary as the network overhead caused by remote logging was significantly impacting the application performance using the 1 GB connection. Note that this problem did not arise during the training and the evaluation in Section 9.3 as they were done using a 10 GB connection. A possible solution is to offload orchestrator functionality to the agents, for example, the saturation prediction in step 1 (see Section 4.1 on page 55). This allows network traffic to be reduced at the expense of higher CPU overhead in the agents and fewer data available at the orchestrator, possibly preventing more elaborated decision-making at the cluster-scale level.

In the evaluation, we applied the predictions of *Monitorless* to autoscaling policies for demonstration purposes but excluded the topic of down-scaling. Although scaling down decisions are less critical from a system perspective, it is possible to extend our approach by either utilizing a three-class prediction system or by training an additional classifier for detecting over-provisioned services and conservatively scale down to reduce costs. This makes it possible to extend *Monitorless* in order to recommend the exact amount of service instances required for any particular scenario in future work. In addition, we currently only concentrate on horizontal scaling, although *Monitorless* could be applied for both horizontal and vertical scaling if a certain classification labeling is provided.

Our experimentation was focused on cloud-based commodity server hardware. Therefore, we excluded the influence of specialized graphics processors or other accelerators. One important issue with hardware accelerators is the monitoring stack, as it has to be consistent among different deployed models and types. However, this represents only a technical limitation and does not impact our conceptual approach.

Generally, *Monitorless* may require additional calibration to infer the performance of applications with resource usage patterns significantly different from those in the training set. Therefore, it may be required to add further training applications in the future. However, we also plan to experiment with transfer learning techniques [PY10] in order to calibrate the model for other target applications as well.

The current version of *Monitorless* does not provide any explanation for its predictions. However, one benefit of decision tree-based models is the possibility of generating user-interpretable predictions. These might be utilized by the application developers to identify bottlenecks in their software and make design decisions. One approach could be to work with depth-restricted decision trees or try to add explainability to the machine learning techniques via Local Interpretable Model-agnostic Explanations (LIME) [RSG16] or SHapley Additive exPlanations (SHAP) [LL17] in order to achieve model explainability.

In addition, the presented version of *Monitorless* focuses on reactive predictions rather than on forecasting the future, as we assume the time frame of creating new services to be sufficiently small. However, in a proactive scenario, *Monitorless* could include a forecasting component that predicts the future value of system metrics, similar to the work of Nguyen et al. [Ngu+13] or Fernandez et al. [FPK14]. For that reason, we also contributed towards a specialized forecasting engine [Bau+20a] and a benchmarking framework [Bau+21] during this thesis. However, as the last two aspects of delivering explainable and proactive predictions are quite challenging based on the limited information available to the *Monitorless* approach, we present an orthogonal approach that specifically focuses on these issues in Chapter 5 on page 73.

Chapter 10

Evaluating the Prediction of Performance Degradation

In this chapter, we evaluate the performance of our *SuanMing* framework, as introduced in Chapter 5 on page 73. For evaluating the effectiveness and the performance of *SuanMing* and the fulfillment of **Goal II** (*"Develop an approach for the prediction of performance degradation using application-level tracing."*), we pose ourselves the following EQs:

- **EQ 10.1**: *Can SuanMing accurately predict the propagation of performance degradations between different services?*
- **EQ 10.2**: *How do different regression modeling approaches compare for modeling the performance of an individual service?*
- **EQ 10.3**: *Are the overheads of model training and performance inference feasible for online environments?*
- **EQ 10.4**: *Is SuanMing effortlessly portable to different applications and monitoring environments?*
- **EQ 10.5**: *How do different forecasting horizons affect the accuracy of SuanMing?*

We analyze the predictions of *SuanMing* using the TeaStore [Kis+18] (see Section 9.1) and the TrainTicket [Zho+18] microservice applications. Section 10.1 presents our results for the TrainTicket application, while Section 10.2 focuses on the TeaStore. All presented results are open-sourced and can be replicated using a CodeOcean capsule [Gro+21e]. Finally, we summarize our results in Section 10.3.

10.1 TrainTicket

Our first experiment series focuses on the TrainTicket application [Zho+18]. TrainTicket is a representative microservice application consisting of 42 different services concerned with administering, searching, and booking train tickets. Due to the number of services in the application and the depth of the call chains, it is suitable for demonstrating the performance propagation and pinpointing capabilities of *SuanMing*.

10.1.1 Experiment Setup

We deploy each TrainTicket service in a single Docker container deployed on an HPE ProLiant DL360 Gen9 cloud server equipped with a 12-core Intel® Xeon® E5-2640 v3 @ 2.50 GHz CPU, 32 GB of RAM, running Ubuntu 18.04 with Docker 18.09.7. All containers are resource-limited to minimize performance interference between the services. Our implementation of the *SuanMing* framework runs on an HPE ProLiant DL20 Gen9 with an 8-core Intel® Xeon® E3-1230 v5 @ 3.40 GHz CPU, 16 GB of RAM, with the same Ubuntu and Docker versions. The TrainTicket services are monitored using the Pinpoint monitoring framework deployed on a different VM equipped with 2 CPU cores and 16 GB of RAM, running on a third host machine.

Lastly, we used an additional machine for emulating users visiting the TrainTicket website. We use the HTTPLoadGenerator [KDK18] to generate a periodically increasing and decreasing amount of users on the system. Upon visiting the site, users log in, solve a captcha, search for a set of possible trains on a route, reserve and buy tickets, as well as collect and check-in the booked tickets. Our users randomly send incomplete or faulty data when searching or booking in order to make the overall user behavior less predictable. In total, 26 services and 58 service endpoints are involved in processing the user requests. The baseload varies between 3 and 22 requests per second. We set the prediction period to five seconds. Hence, the experiment time is divided into periods of five seconds. Therefore, our models generate a new prediction about the next period every five seconds, resulting in a total of 676 analysis periods with corresponding predictions and 700 total training periods. As we have 58 endpoints in the application, *SuanMing* trains 58 machine learning models, using at least one and at most 12 features, depending on the position of the endpoint in the graph. In addition to this relatively low baseload, we now specifically overload one of the backend services (`train`) with 300 requests per second to evaluate how the created performance degradation propagates through the application and whether *SuanMing* is able to predict and pinpoint the anticipated performance degradation correctly. As we want to evaluate how this impacts the prediction performance on the `travel` service, we furthermore ensured that the load on `travel` was effectively constant during the whole experiment (compare Figure 10.2 on page 175).

Concerning performance metrics to measure and predict, we focus on one metric in this evaluation, namely the service response time. For the frontend services `preserve`, `preserveOther`, and `travel`, we define thresholds of 1 500, 1 500, and 600 ms as performance thresholds, respectively. Hence, if the average response time of `preserve` rises above 1.5 seconds in any given time period, `preserve` is considered to experience performance degradation. The goal of *SuanMing* is now to predict this degradation at least one period before it can be measured in the system. The thresholds of all endpoints are set to at least two times their normal response time during low load. For fast services with response times smaller than 30 ms (except `train`), we set the threshold to 110 ms. As the propagation matrix of TrainTicket is linear, we can apply the linear propagation algorithm discussed in Section 5.2.5 on page 82.

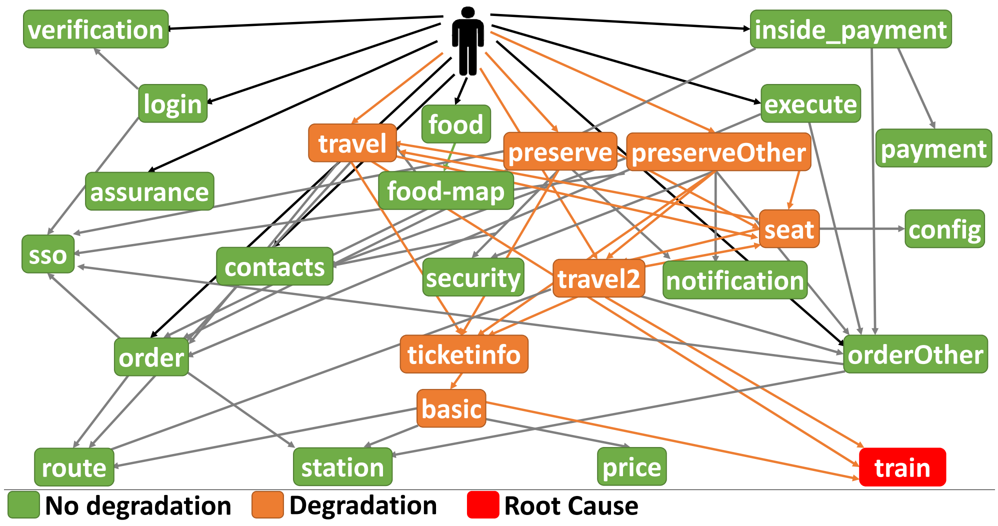

Figure 10.1: Schematic state of the TrainTicket application at measurement period 680.

10.1.2 Performance Propagation

Figure 10.1 shows a schematic overview of all 26 involved service instances and their connections at period 680, hence, after 3 400 seconds. Each box represents a service, each arrow depicts a user or inter-service call. Figure 10.1 shows the system status during a performance degradation at the `train` service (red). We observe that the performance degradation propagates as expected through the application. Although the frontend services `travel`, `preserve`, and preserveOther only receive a minimal number of requests (8.8 requests per second), the average response time of these services increases massively after the backend service train decreases its performance. We observe performance degradations at several services (orange), all of which can be avoided by addressing the problem at the train root cause service (red). In the following, we analyze if *SuanMing* is able to correctly predict, detect, and propagate these performance degradations, as well as pinpoint the respective root cause service.

10.1.3 Regression Analysis

For answering **EQ 10.2**, we first analyze the performance of different regression models using the /travel/query endpoint, an endpoint of the travel service. We focus on /travel/query, as travel is a frontend service and therefore relevant for the user experience. The goal of all modeling approaches is to predict the performance degradation at the train service and to propagate the performance problems up to the travel service. As travel itself is not experiencing a high baseload, this assesses the model capabilities to propagate the performance degradation to the system correctly.

The black line of Figure 10.2 shows the average response time for a request at the /travel/query service over time, that is, the time one user has to wait for the response of a search for possible trains. At the same time, the gray background curve depicts the number of requests arriving at travel. We observe that the observed response time spikes are not correlated to the number of incoming requests at travel but are due to the poor performance of other services.

We compare four different regression models using (1) Random Forest (RF), (2) k-Nearest Neighbors (kNN), (3) Bayesian Ridge Regression (BRR), and (4) Support Vector Regression (SVR). All implementations are provided by the Scikit-learn library [Ped+11]. The models are trained using six-fold cross-validation using out-of-sample forecast evaluation [BCB14] in order to optimize their hyperparameter settings by performing a grid search on a set of three to five hyperparameters to minimize the classification error measured using F1 score.

We observe that almost all modeling approaches depicted in Figure 10.2 closely resemble the anticipated load spikes. From a visual inspection, RF seems to perform best, as SVR tends to underestimate the load spikes, BRR tends to overestimate the low load phases, and k-nearest neighbor occasionally massively overshoots the predictions. However, generally, all modeling approaches are able to predict the performance degradation at /travel/query, although the incoming load intensity is effectively stable. Note that the measurement and the prediction curves were aligned for better visibility. In a live

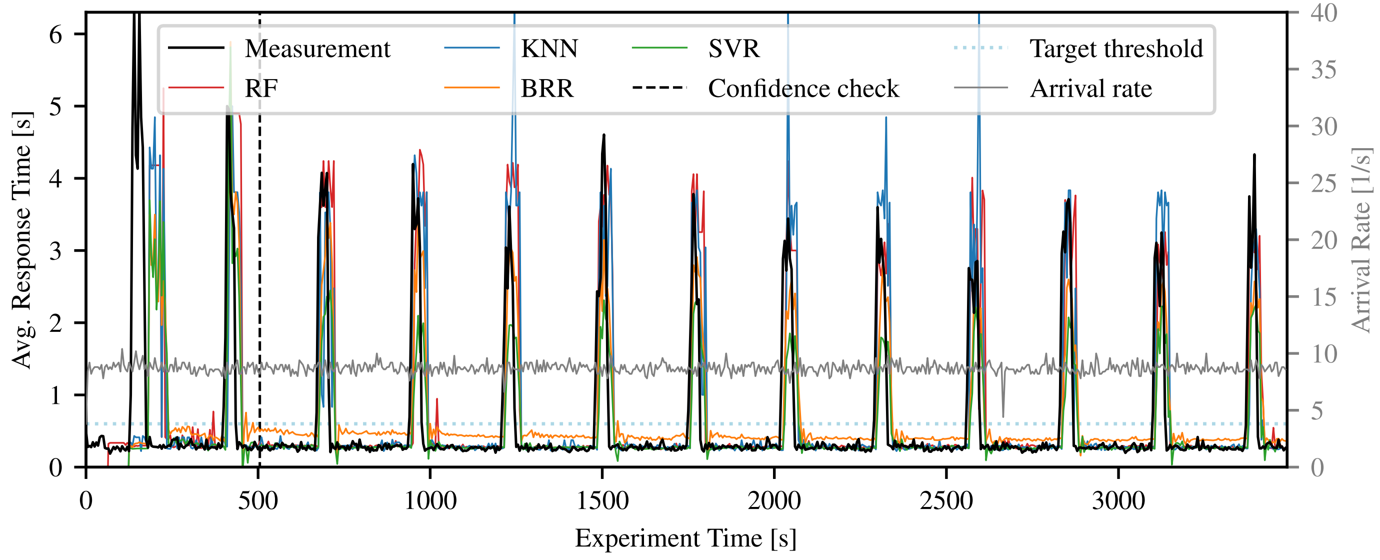

Figure 10.2: Measured and predicted response times at `/travel/query`, together with the incoming request rates.

system, the prediction curves would rise before the measured response time increases, as it is necessary to enable the mitigation of the degradation.

Figure 10.2 additionally depicts the evolution of the regression model over time. The dashed line at 505 seconds resembles the confidence threshold to be passed for the first time. Hence, *SuanMing* did not output any degradation predictions before that time, as the model confidence was too low. This makes sense, as all *SuanMing* models start without any information or prior knowledge. Hence, they need time to learn the application structure as well as the performance behavior of the TrainTicket application. The chosen threshold itself is also reasonable, as the regression models are fairly inaccurate during the first phase of the experiment. As the confidence is calculated based on the forecast, it is model-agnostic and can be applied to all model types at the same time. For the presented experiment, the threshold was passed after 101 periods or 505 seconds.

After the confidence threshold is passed, the RF regression curve closely fits the measured performance, although the response time fluctuates heavily between 200 ms and over 3000 ms. However, we observe that the models iteratively learn and adapt to the incoming measurements. For example, the load peak right after 2500s is lower than anticipated; therefore, the RF and kNN curves overshoot the expected response time during that degradation. Following, the models adapt to these changes and lower their predictions for the succeeding periods in order to better resemble the measurements. Generally, these small prediction errors lead to a relatively high regression error (compare Table 10.1) but have only minor effects on the classification accuracy.

10.1.4 Classification Analysis

We now evaluate the performance of *SuanMing* in terms of the overall classification accuracy to gain more details for answering **EQ 10.2** and **EQ 10.1**. Table 10.1 shows different regression and classification scores of different model types summarized over the course of the whole experiment.

We report the Mean Absolute Error (MAE) as a regression metric for each approach, quantifying how closely the perceived load curve resembles the model prediction. As the predicted response time of each service is then translated into a binary classification (healthy or not healthy) using the defined thresholds, we further study the classification performance of the individual approaches. We analyze True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN) scores, as well as the resulting accuracy and F1 scores. Additionally, we report the *global* classification metrics, that is, the accuracy and the F1 score for all 58 service endpoints combined. The values reported

Table 10.1: Model accuracy comparison of the TrainTicket experiment.

Approach	Example Endpoint `/travel/query`							Application-Wide	
	MAE	TP	FP	TN	FN	Accuracy	F1 Score	Accuracy	F1 Score
RF	331 ms	63	33	481	22	0.908	0.696	0.977	**0.626**
kNN	337 ms	64	31	483	21	**0.913**	**0.711**	0.975	0.594
SVR	**288** ms	64	31	483	21	**0.913**	**0.711**	**0.978**	0.610
BRR	357 ms	64	38	476	21	0.902	0.684	0.955	0.484
Mean Regressor	558 ms	37	287	227	48	0.441	0.181	0.931	0.203
All-green	–	0	0	514	85	0.858	0.000	0.973	0.000
All-red	–	85	514	0	0	0.142	0.249	0.027	0.052
Random	–	46	265	249	39	0.492	0.232	0.497	0.050
Optimal RF	221 ms	75	13	501	10	0.962	0.867	0.987	0.772

in Table 10.1 all refer to the periods after the model confidence threshold has passed, hence, excluding the first 505 seconds.

For comparison, we added a set of different baseline approaches. First, we add *Mean Regressor*, a regression approach that always predicts the mean of all previously observed values. The *All-green*, *All-red*, and *Random* approaches are classifiers that always predict green, red, or randomly for each of the experiment periods. As they do not work with regression, they do not have any regression metrics available. Finally, we also add a variant of the RF predictor that receives perfect load forecasts (Optimal RF). This approach assumes that the given forecast is perfect (i.e., it works a-posteriori and not online) and therefore helps at determining the impact of the forecasting error.

In total, the used experiment contains 85 periods experiencing performance degradation at the `/travel/query` endpoint. Adding the other endpoints, a total of 933 performance degradations were recorded. In comparison to the total amount of periods, this is a relative share of 14.2% for `/travel/query` or 2.7% for the total application. This ratio is representative, as performance degradations are expected rather infrequently in practice.

Generally, we observe that all regression algorithms depicted in Table 10.1 are generally capable of capturing the performance behavior of `/travel/query`. The most notable difference is with the BRR, which performs poorly on the MAE and has a higher FP-rate than the other approaches. SVR shows the lowest MAE and ties with kNN for accuracy and F1 score on the `/travel/query` endpoint. However, when we analyze the global score, RF performs slightly better than SVR regarding the F1 score, despite its higher regression error. We furthermore note that the MAEs of all approaches are relatively high in comparison to the configured response time threshold of 600 ms. However, by analyzing the actual prediction in Figure 10.2, we observe that the high average errors are most likely due to the phases with high response times.

As the global F1 is our main metric of interest, and as the increased training time of SVR speaks against it (compare Table 10.2 on page 180), we will restrict to RF regression for the rest of this analysis. The results obtained from Table 10.1 are in line with the impression of the regression curves in Figure 10.2 and can now be utilized to answer **EQ 10.2** (*"How do different regression modeling approaches compare for modeling the performance of an individual service?"*) and **EQ 10.1** (*"Can SuanMing accurately predict the propagation of performance degradations between different services?"*).

When comparing with the baseline approaches, we observe that all modeling techniques consistently outperform the given baseline techniques. While the poor performance of All-green, All-red, and Random is expected, the Mean

Regressor baseline also achieves significantly lower scores. Note that the high accuracy of the All-green and the Mean Regressor approaches is due to the large share of TN periods in the experiment.

Finally, we observe that the performance of the RF regressor can be significantly improved by eliminating the forecasting error, as seen with the optimal RF model. Although this approach is not realistic to apply in practice, we can conclude that RF can correctly model the performance behavior of the system, given the correct number of incoming requests. This shows the benefit of the modular architecture of *SuanMing*, as the forecasting engine can be easily exchanged if more accurate forecasts are required.

10.1.5 Root Cause Detection

As the RF approach is able to predict the performance degradation at the `travel` service, we now analyze the list of root cause services returned by Algorithm 5.3 on page 87. For the scenario shown in Figure 10.1, *SuanMing* returns a list of `train`, `travel`, `travel2`, `basic`, and `ticketinfo` as problematic or root cause services that need to be addressed in order to solve the performance degradations in the system. However, by halving the inserted performance thresholds $\tilde{P}_i'$ the list is reduced to contain only `train`. This shows that *SuanMing* pinpoints `train` as the respective root cause service but also identifies additional services that require attention. We hypothesize that these inaccuracies are due to the lack of training samples during high load phases, as performance degradations at `train` are always accompanied by performance degradations in the shown scenario.

10.1.6 Overhead Analysis

In this section, we assess the overhead of model training and prediction in order to demonstrate the feasibility of using *SuanMing* in an online environment. To that end, Table 10.2 depicts the maximum time required for training the regression models together with the mean prediction time per period for the TrainTicket experiment shown in Table 10.1. We analyze the maximum training time as the time required for training the regression models increases with the amount of monitoring data available. The prediction time is averaged over all predictions and includes both the execution of the request propagation and the performance inference for all endpoints of the service. For the prediction time, the average gives a better picture of the expected prediction latency than the maximum.

Table 10.2: Overhead analysis of different model types for the TrainTicket experiment.

Approach	Max. Train Time	Avg. Prediction Time
RF	6.243 s	0.226 s
kNN	3.102 s	0.135 s
SVR	17.405 s	0.092 s
BRR	**2.915** s	**0.067** s

We observe that all models are able to train all regression models in a matter of seconds and deliver predictions in less than a second. Both timescales are assumed to be sufficient for an online environment. In particular, prediction times of less than one second are sufficiently fast for prediction periods of 5 seconds. We assess that BRR is the fastest of all approaches. However, this is offset by its relatively poor prediction accuracy. Therefore, while the choice of the best-suited modeling type is up to the user, we still recommend using the RF approach. This answers **EQ 10.3** (*"Are the overheads of model training and performance inference feasible for online environments?"*), as all assessed modeling types qualify for execution in online environments.

10.2 TeaStore

After we verified the capabilities of *SuanMing* on the TrainTicket application, we now use *SuanMing* to predict performance degradations of a different application in a more realistic testing environment. As a second application, we use the TeaStore [Kis+18], a microservice benchmarking application for buying tea, consisting of seven services. The users of TeaStore can log in, browse different categories and products, add and modify items in their shopping cart, and checkout by entering shipping and payment information. For more details on the TeaStore application, we refer to Section 9.1 on page 152.

In contrast to the previous experiment, we deploy TeaStore in a realistic and commercial cloud environment (Huawei Cloud[1]) and feed the available cloud monitoring and tracing into the *SuanMing* framework. Additionally, we increase the applied load pattern in order to represent realistic daily or weekly fluctuating workload intensities and to regularly overload the TeaStore application at specific services. Therefore, over the experiment duration of 6

[1] `https://www.huaweicloud.com`

hours, the workload intensity irregularly fluctuates between 1 and 140 user requests per second. The target thresholds of the frontend service `WebUI` are set to 200 ms and for all other services to 100 ms. This aims at verifying that *SuanMing* is not only able to deliver root cause predictions under lab conditions but also deliver accurate performance degradation predictions in a realistic cloud setting.

10.2.1 Classification Performance

We evaluate *SuanMing* by analyzing the performance of the RF modeling technique with different baseline approaches, similar to the previous experiment. Table 10.3 compares the classification metrics of *SuanMing* with other baseline approaches, focusing on the frontend service `/WebUI/main`. The shown values refer to after the confidence threshold was passed, in particular, after 13 periods or 65 seconds.

Table 10.3: Accuracy comparison of the TeaStore experiment.

Approach	`/WebUI/main`		Application-wide	
	Accuracy	F1 Score	Accuracy	F1 Score
SuanMing	0.826	0.816	0.913	0.693
Mean Regressor	0.520	0.681	0.802	0.574
All-green	0.486	0.000	0.857	0.000
All-red	0.514	0.679	0.143	0.251
Random	0.504	0.495	0.503	0.224

Similar to our results on the TrainTicket application, *SuanMing* is able to outperform all baseline approaches in terms of accuracy and F1. However, we notice that the accuracy is lower, while the F1 score has increased in comparison to the TrainTicket environment. This is due to the increased number of performance degradations in the dataset and the correspondingly rising number of true positives. Nevertheless, *SuanMing* is able to achieve a global accuracy of over 91% with an F1 score of almost 0.7, which shows that the results of *SuanMing* are transferable to different applications and monitoring environments. Hence, this answers **EQ 10.4** (*"Is SuanMing effortlessly portable to different applications and monitoring environments?"*).

10.2.2 Prediction Horizon

One advantage of the *SuanMing* design is that the performance prediction models are time-agnostic and are sub-sequentially able to calculate performance predictions for any arbitrary application state. After analyzing the performance of single-horizon predictions, that is, the prediction for one period or five seconds into the future, we now want to focus on increasing this prediction period. The advantage of a higher prediction horizon is that it allows for an earlier notification for anticipated performance degradations and hence gives more time to address and mitigate the predicted problems.

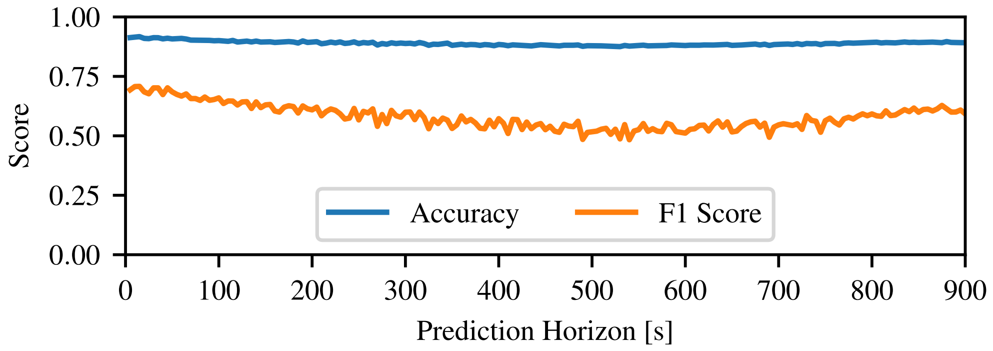

Figure 10.3: Global classification accuracy with increasing prediction horizons.

Figure 10.3 shows the global classification accuracies averaged over the whole experiment duration for increasing prediction horizons. We observe that the F1 score is significantly impacted by an increasing horizon, while the reported accuracy stays almost constant. This is due to the fact that most services do not experience performance degradations, leading to a high amount of true negatives that can be accurately predicted by *SuanMing*. The F1 is mainly influenced by the predictions on the `WebUI` service, as it was shown to be the bottleneck for the TeaStore application. Overall, the curve drops almost linearly until a score of around 0.6 is reached at a total prediction horizon of around 200 seconds. Increasing the prediction horizon to over 500 seconds does not decrease the F1 anymore.

While a performance prediction of 600 seconds in advance is theoretically possible, the F1 regularly drops below 0.6 for large horizons. It is then up to the user to decide if the accuracy drop is acceptable. However, if a small drop in prediction accuracy is acceptable, Figure 10.3 shows that *SuanMing* predicts performance degradations up to 200 seconds in advance. Generally, the prediction accuracy of large horizons is strongly dependent on the applied workload and whether the forecasting engine is able to correctly predict the future load pattern. Unforeseen noise or load spikes make this task increasingly difficult. Therefore, we conclude that *SuanMing* can arbitrarily increase the prediction horizon but increasingly depends on accurate forecasts in order to perform its predictions. Finally, this answers our last EQ, namely **EQ 10.5** (*"How do different forecasting horizons affect the accuracy of SuanMing?"*).

10.3 Summary

To summarize, in this chapter, we evaluated the capabilities of the *SuanMing* framework to anticipate performance degradations of microservice applications. Our evaluation shows that *SuanMing* predicts and explains performance degradations with an accuracy of over 90% on both the TrainTicket and the TeaStore microservice applications. These results were the focus of **EQ 10.1** (*"Can SuanMing accurately predict the propagation of performance degradations between different services?"*) and **EQ 10.4** (*"Is SuanMing effortlessly portable to different applications and monitoring environments?"*). We also compared different regression modeling approaches with each other in order to answer **EQ 10.2** (*"How do different regression modeling approaches compare for modeling the performance of an individual service?"*). Our analysis indicates that while multiple techniques can be utilized, RFs regression performs best in our scenarios.

We furthermore assess whether or not *SuanMing* is capable of delivering the required predictions within a reasonable time frame in a realistic cloud environment. Therefore, **EQ 10.3** (*"Are the overheads of model training and performance inference feasible for online environments?"*) concentrates on the overhead of each used modeling technique. We see that all utilized modeling techniques perform sufficiently fast for deployment in an online cloud environment. Finally, we analyze the forecasting horizon of our framework and the resulting implications on the prediction performances by answering **EQ 10.5** (*"How do different forecasting horizons affect the accuracy of SuanMing?"*). We observe that it is possible to receive performance degradation predictions several minutes in advance by increasing the prediction horizon if accompanying accuracy decreases are acceptable. Based on these results, we conclude that *SuanMing* presents a first step towards autonomically supervising microservice applications in order to avoid performance degradations before they occur in the system. In the following, we discuss the threats to the validity of the presented evaluation as well as the limitations and the assumption of the presented *SuanMing* approach.

10.3.1 Threats to Validity

In the following, we split the relevant points into threats for the internal and the external validity of our experimentation and discuss each of them in the respective sections.

10.3.1.1 Internal Validity

From our analysis of Section 10.1.4, we conclude that *SuanMing* and the accuracy of its performance predictions strongly rely on the applied forecasting technique. Therefore, many of the reported accuracy metrics might be influenced by our choice of GluonTS as a forecasting engine. We mitigate this issue by explicitly including an analysis including the optimal forecaster in Table 10.1 on page 177. Due to the modular design of the *SuanMing* framework, it is possible to exchange or optimize forecasting engines, depending on the specific scenario or workload. We also experimented with Telescope [Bau+20b] as an alternative forecasting engine but then decided to use GluonTS for the analysis presented in this work based on the preliminary results.

We quickly discussed the choice of metrics in Section 10.1.4. Table 10.1 includes the MAE as our main regression metric. As our target is to classify response time increases rather than to perform exact predictions, we decide against squared errors like MSE or RMSE as those put a stronger focus on outlier predictions. The presented analysis generally focuses more on the actual classification metrics and less on the regression error of each individual approach. Hence, we view any TP as a correct prediction, although the predicted metrics (and the resulting regression error) might massively overshoot the measurements. Therefore, all presented optimizations and perceptions are biased towards classification. When focusing on minimizing the actual regression error, some of the presented insights may not transfer.

10.3.1.2 External Validity

Similar to our previous evaluation in Chapter 9 on page 151, we concentrated on a limited set of microservice applications in this work. We are aware of the diversity of different available applications and their performance behavior, as well as the varying amounts of information provided by different tracing tools. Hence, although we purposely vary between two different applications and monitoring stacks, the expressiveness of the evaluation is limited to the presented applications. Especially, **EQ 10.4** (*"Is SuanMing effortlessly portable to different applications and monitoring environments?"*) usually requires a more general analysis of a wide range of different monitoring tools and applications.

However, based on the initial results and the modular architecture of *SuanMing*, we are convinced that other monitoring stacks will only require minimal amounts of implementation effort for the Provider component, as all other components should work unchanged.

The focus of *SuanMing* is generally on synchronous request-response patterns. However, microservice applications might also utilize asynchronous communication, as well as relying on message-oriented middleware stacks. As neither of our evaluated applications utilized such stacks, the applicability of other communication patterns represents a threat to the generalizability of *SuanMing*. However, in our view, asynchronous communication patterns are less important for the type of performance modeling attempted by *SuanMing*, as they do not directly influence the response time of the waiting service. Nonetheless, the request propagation model is generally capable of tracking all incoming requests, including asynchronous and middleware requests, as it is only focusing on the statistical properties. Therefore, although excluded in this evaluation, such communication patterns are also covered by *SuanMing* if the monitoring tool includes the respective tracing data.

Finally, we did not consider replicated service instances in our experiments. If the number of replications per service is expected to be constant, we can abstract all service instances into the same service model. However, we could also create individual performance models for each service instance and treat them as technically independent services. If the performance of each replication is expected to be equal, we could also dynamically model the number of current replicas using the additional performance parameters α (see Section 5.2.4 on page 80).

10.3.2 Limitations and Assumptions

The design of *SuanMing* is influenced by a set of limitations and assumptions. First, we assume a static performance behavior by the underlying cloud platform stack. Therefore, we assume no autoscaling, redeployments, service migrations, or other performance fluctuations to happen during the operation of *SuanMing*. Although this is not a conceptual limitation, the trained performance models require repeatable performance behavior in order to deliver accurate predictions. If required, major platform changes can also trigger retraining of the respective performance models. Otherwise, we could include the state of the cloud platform using the additional performance parameters α (see Section 5.2.4 on page 80).

The current version of *SuanMing* focuses on pinpointing the respective service as the root cause of expected performance degradations. In future work, one can extend this error search by correlating predicted performance degradation with measurement from resource usage metrics. This can be done using the auxiliary metrics to improve the performance model accuracy. That way, we are able to deliver root cause predictions for not only which service is responsible but also what is causing the problem at the respective service itself.

SuanMing only considers the call hierarchy when calculating the performance dependency map. However, if deployment information of the individual services are given by the APM, the dependency map could be extended by taking co-location dependencies into account. After augmenting the list of dependent services with all co-located services, a performance degradation could also be predicted by too many requests to a busy or a faulty service on the same host or simply by other services consuming all computing resources. An application of this was successfully demonstrated by Lin et al. [LCZ18].

Similarly, we did not consider the performance of external services during the presented evaluation. If the performance of the external service is assumed to be static, then its behavior can be included in the performance model of the calling service. Otherwise, external services can be considered as additional services, which are modeled by their own performance functions. Unfortunately, if an external service is pinpointed as a root cause, no automatic problem mitigation can be triggered.

Furthermore, *SuanMing* currently relies on threshold definitions for all services to define the expected performance behavior. Future work can extend the capabilities of *SuanMing* by applying anomaly detection approaches [CBK09] for automatically defining the thresholds based on historical data. The advantage of these approaches is that they do not require SLA or threshold definitions by the user but instead derive the thresholds from the actual system monitoring. This is especially advantageous for long-running systems and when the defined thresholds are possibly subject to change.

Finally, *SuanMing* is technically limited to predicting the performance of services that are included in the monitoring. In addition, the framework requires performance measurements for the SLA-relevant performance metrics to label its training data. If more than one performance metric is desirable, one could add indirect measurement strategies by inferring one unobservable performance metric based on other observable quantities. This is currently not in the scope of this framework but could be an interesting topic for future work.

Chapter 11

Evaluating Continuous Resource Demand Estimation

In this section, we evaluate and analyze the performance of *SARDE* concerning various aspects. As *SARDE* aims to provide continuous resource demand estimations in changing environments, *SARDE* is a part of fulfilling **Goal III** (*"Enable the continuous estimation and improvement of performance model parameters using production monitoring data."*). To evaluate the continuous estimations, we pose ourselves the following EQs:

- **EQ 11.1**: *What is the gain of continuously repeating the estimation?*
- **EQ 11.2**: *What is the impact of applying algorithm selection, optimization, and both combined to the repeated estimation?*
- **EQ 11.3**: *What is the overhead of applying these techniques?*

In the following, we will describe and analyze the experiment series we conducted in order to answer these questions. The precise experiment setup is described in Section 11.1. Following, Section 11.2 focuses on the analysis of the selection process, while Section 11.3 analyses the performance of the optimization algorithm. We put both aspects together and analyze the performance to answer **EQ 11.2** in Section 11.4. Finally, we analyze the workload properties in Section 11.5 and the overhead in Section 11.6. We conclude our analysis in Section 11.7.

11.1 Experiment Setup

We designed two different experiments to validate the accuracy of our approach. Section 11.1.1 describes the two datasets. In Section 11.1.2, we describe the metrics used for evaluation, while Section 11.1.3 lists the configuration of *SARDE* used in this evaluation.

11.1.1 Datasets

First, we applied a common dataset consisting of a set of micro-benchmarks executed on a system and already applied in a previous study of Spinner et al. [Spi+15]. Second, we extend this analysis by adding a long-term measurement trace from a realistic application.

11.1.1.1 Micro-benchmark Datasets

This dataset consists of a set of measurements obtained by executing micro-benchmarks on a real system. It contains 210 traces, each with an approximate runtime of one hour. The micro-benchmarks generate a closed workload [SWH06] with exponentially distributed think times and resource demands. The think times themselves were set to fit the targeted load level of each specific experiment. As mean values for the resource demands, 14 different subsets of the base set [0.02s; 0.25s; 0.5s; 0.125s; 0.13s] were selected, with a varying number of workload classes $C = \{1; 2; 3\}$ and target load levels $U = \{20\%; 50\%; 80\%\}$. The subsets were arbitrarily chosen from the base set. This way, it can be ensured that the resource demands are not linearly growing across workload classes. Additionally, the subsets intentionally contained cases where two or three workload classes had the same mean resource demand.

11.1.1.2 Realistic Application

In addition to the micro-benchmark datasets, we conducted a long-term study of a realistic microservice application measured on a real system. However, in order to evaluate the accuracy of the approach, it is necessary that we know the exact resource demands to be estimated. Therefore, we developed a synthetic application that offers three different interfaces via a REST API that perform a prior defined load for each service call. For the following of this section, the first workload class (WC1) performs an exponentially distributed load with a mean of 0.01s, the second workload class (WC2) performs an exponentially distributed load with a mean of 0.03s. The third workload class (WC3) performs a normally distributed load with a mean of 0.005s and a standard deviation of 0.001. Therefore, the presented application, although comprising only one microservice, serves as an example of a possible microservice application.

To evaluate the adaptability of the individual approaches in comparison to *SARDE* with respect to different influence factors, we varied both the load intensity and the distributions of the individual workload classes. Figure 11.1 depicts the load intensity, that is, the number of requests per second of each workload class as a stacked line chart. The load is intentionally noisy and strongly varies over time. Additionally, the relative share of the different workload classes changes. As the different workload classes each have different resource demands, the resulting utilization curve is not obvious.

In order to reflect a realistic cloud setup, we deployed the application inside an Ubuntu 18.04 VM associated with one pinned CPU core and 4 GB RAM running on an HPE ProLiant DL160 Gen9 server equipped with an Intel® Xeon® CPU E5-2640 v3 @ 2.60 GHz and 32 GB total RAM, using a KVM hypervisor. The load driver generating the REST requests was situated on another host in the same cloud to isolate the performance behavior and also include the network overhead per request.

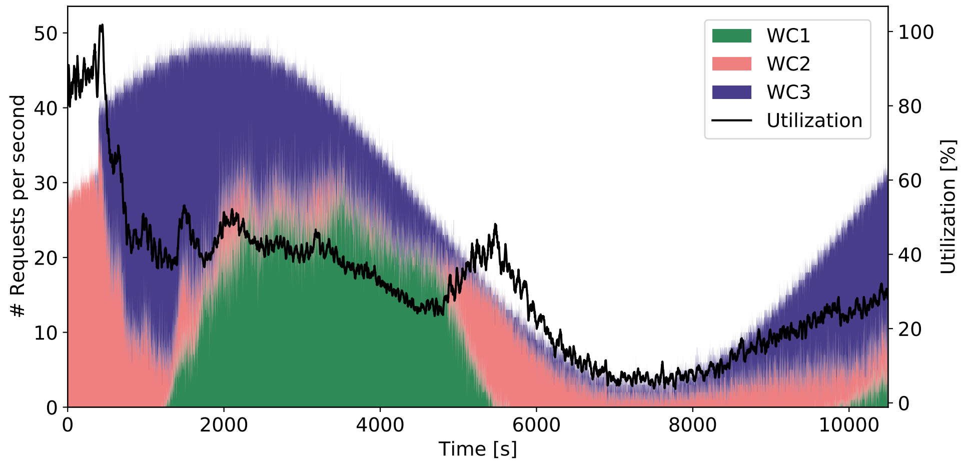

Figure 11.1: Server utilization and throughput of the different workload classes of the monitored application.

11.1.2 Evaluation Metrics

In this section, we describe the metrics we use during our evaluation. We focus mainly on execution time and estimation accuracy. All execution times were measured using version 1.0 of the publicly available Java implementation of *SARDE*[1] and version 1.1 of the underlying LibReDE engine[2] by relying on the internal time measurement. All reported experiment times were conducted on a Windows 10 machine using an Intel® Core® i7-6600U CPU @ 2.60 GHz and 16 GB RAM.

[1]`https://github.com/jo102tz/LibReDE-SARDE/releases/tag/v1.0.0`
[2]This is also the version endorsed by the SPEC RG [SGK19].

For accuracy, we evaluate the estimation error ϵ_E per approach by averaging the relative estimation error of each workload class:

$$\epsilon_E = \frac{1}{C}\sum_{c=1}^{C}\left|\frac{\tilde{D}_c - D_c}{D_c}\right|,$$

where C is the number of workload classes, $\tilde{D}_c$ is the resource demand estimate for workload class c, and D_c is the real resource demand of class c.

11.1.3 Configuration

There are several generic and configurable parts of the *SARDE* approach described in Chapter 6 on page 91. In this section, we describe the specific configurations that we applied for the presented evaluation.

First, we concentrate on the estimation of the resource demand error. As all evaluations and optimizations performed by *SARDE* rely on the internal estimated error, it is crucial that the applied error validation closely resembles the actual resource demand error. Recall that *SARDE* does not have the real resource demands available for validation as they are naturally unknown to *SARDE* during operation. Therefore, *SARDE* calculates the estimated validation error ϵ_V using the estimated relative response time error ϵ_R and the estimated absolute utilization error ϵ_U. This error is then used for all internal validation processes. The two error functions are defined as follows:

$$\epsilon_R = \frac{1}{C}\sum_{c=1}^{C}\left|\frac{\tilde{R}_c - R_c}{R_c}\right|,$$

$$\epsilon_U = \left|\sum_{c=1}^{C}(X_c \cdot \tilde{D}_c) - U\right|,$$

with C being the number of workload classes, R_c the average measured response time of workload class c over all resources, $\tilde{R}_c$ the predicted average response time using Mean Value Analysis [Bol98] based on the estimated resource demands, X_c the measured throughput of workload class c, $\tilde{D}_c$ the estimated resource demand of workload class c, and U the average measured utilization over all resources.

Using both errors, we can compute the compound validation error ϵ_V as a weighted sum of ϵ_R and ϵ_U:

$$\epsilon_V = \frac{1}{2}\min(1, \epsilon_U) + \frac{1}{2}\min(3, \epsilon_R).$$

Note that we bound the utilization error at one and the response time error at three. This is necessary since both errors are effectively unbounded and, therefore, might dominate the other error during the validation. The values are chosen, as during capacity planning, response time errors are usually acceptable to be higher than utilization errors [Men02a; MG00]. Apart from that, both ϵ_U and ϵ_R are currently weighted at an equal ratio. However, this configuration could be adapted if a user is more interested in minimizing the respective error value.

For the online analysis of the realistic application, we use an estimation interval of 70 seconds, a selection interval of 170 seconds, a training interval of 700 seconds, and an optimization interval of 1 000 seconds in order to keep a reasonable amount of repetitions for each activity during the experiment. Based on our results in Section 11.2.1, we applied a random forest classifier as the selection algorithm. Concerning the S3 optimization algorithm, we use five splits, four exploration points, and five iterations for single parameter optimizations. For multi-parameter optimizations, we reduce to one split, with two exploration points and two iterations in order to reduce the algorithmic complexity.

11.2 Selection

This section presents results concerning the selection of the best-suited estimation approach. The first section compares different selection algorithms with each other using our set of micro-benchmark experiments. Then, we analyze the performance of continuous training and selection over time in our realistic application.

11.2.1 Micro-benchmarks

To compare the different selection algorithms with each other, we utilize the set of micro-benchmarks as they represent a wide variety of different scenarios in their characteristics. Therefore, we can get a holistic analysis of the performance of each selection algorithm.

We include a CART [Bre+84], an AdaBoost [Has+09], a RF [Bre01], a Logit [Cox58], an SVM [CV95], and a NN [Gur97] algorithm. The NN is a sigmoid perceptron consisting of two fully connected inner layers, an input layer, as well as an output layer for the selection. We used 100 neurons in total and applied the back-propagation algorithm based on the least-squares error for learning. For all algorithms, we relied on the implementations provided by the Smile [Li14] library. For a fair comparison, all algorithms were used in their default parameterization. Furthermore, we add a random classifier (Random), always choosing a random approach as a baseline. We split the 210 available scenarios into 168 training and 42 validation traces. The machine learning algorithms were trained with the 168 training sets, and Table 11.1 shows their performance on the 42 remaining validation sets.

Table 11.1: Comparison of different selection approaches using the micro-benchmark set.

Algorithm	Avg. error	Hit rate	Train time	Estimation time
Random	43.5%	16.7%	–	1.4 s
CART	22.5%	52.4%	211.1 s	1.1 s
AdaBoost	19.8%	66.7%	241.1 s	2.0 s
RF	17.9%	71.4%	533.0 s	2.1 s
Logit	25.0%	42.9%	305.6 s	1.5 s
SVM	18.0%	59.5%	262.3 s	1.5 s
NN	18.0%	59.5%	243.2 s	13.4 s

The first column of Table 11.1 shows the average resource demand estimation error on the 42 remaining traces when applying the respective selected approach. We observe that, as expected, the random classifier has the worst performance; the CART and the Logit models also fall behind. However, AdaBoost, RF, SVM, and NN all perform comparatively well. RF has the best accuracy, with an average estimation error of 17.9%. This is impressive if you consider that the average minimum error of all approaches (and therefore the de-facto perfect result) is 17.6%. Therefore, the performance of the approaches chosen by RF is just 0.3% worse than the theoretical optimum. These results are in line with the hit rate, that is, the relative share of scenarios in which the algorithm selects the best approach. Again, Random Forest outperforms all other approaches with a hit rate of almost 72%, while a random classifier baseline achieves only 16.7%.

When analyzing the training time, we observe that all approaches take between 4 and 10 minutes for completing the training with a training corpus of 168 traces. Here, RF takes the longest time for training (almost ten minutes), while all other approaches terminate within four to five minutes. However, considering the large size of the training set (168 measurement hours), we argue a training time of ten minutes is more than acceptable for online use. Similarly, the average estimation time (including feature extraction, selection, and the estimation process itself) is sufficiently fast. Most approaches terminate in under three seconds; only the NN approach requires up to 15 seconds of estimation time. As typical estimation windows are usually in the range of several minutes, these time scales are more than sufficient. One interesting observation is that the random baseline, despite the lack of an actual selection process, is not the fastest of the approaches. This undermines our observation that the most dominant time factor for the average estimation time is, in fact, not the selection algorithm itself (excluding NN) but the estimation time of the selected approach.

Based on our results, for the remainder of this chapter, we concentrate on the RF algorithm with a parameterization of five trees (`ntrees`), two features per node decision (`mtry`), a maximum leaf node size of one (`nodeSize`), applying the Gini splitting criterion (`rule`), and using feature sampling with replacement (`subsample`).

11.2.2 Realistic Application

Following the broad analysis of multiple validation scenarios, we now analyze the performance of the random forest selection for our realistic application. For this, we look at the continuous training and selection of the algorithm over time. Figure 11.2 shows the estimation error for every approach over time. The activities are depicted in the timing diagram at the top of Figure 11.2. The red bars indicate timing and duration of training phases, the orange bars indicate selections accompanied by an abbreviation of the chosen approach, and the blue bars indicate the regularly repeated estimations of all approaches.

In each training phase, the chosen selector algorithm (RF in this case) was trained on all available offline traces from the previous section, plus the additional experience from the currently running trace (hybrid training). Therefore, the first trained model only has the micro-benchmark dataset available as a training dataset. The second one uses the micro-benchmark set, plus the first 700 seconds of experiment time, and so on. As we had a maximum of three different workload classes ($r = 3$) and one resource ($w = 1$) in the training set, the feature vector y had a length $|y|$ of 57 for training (see Section 6.3.3.3 on page 102).

We observe that the estimates, as well as the corresponding accuracy of each individual approach, are massively changing during the experiment. There is, therefore, a good rationale for continuously repeating the resource demand estimations and simultaneously for changing the applied approach (see Section 6.1 on page 93). This already answers **EQ 11.1** (*"What is the gain of continuously repeating the estimation?"*).

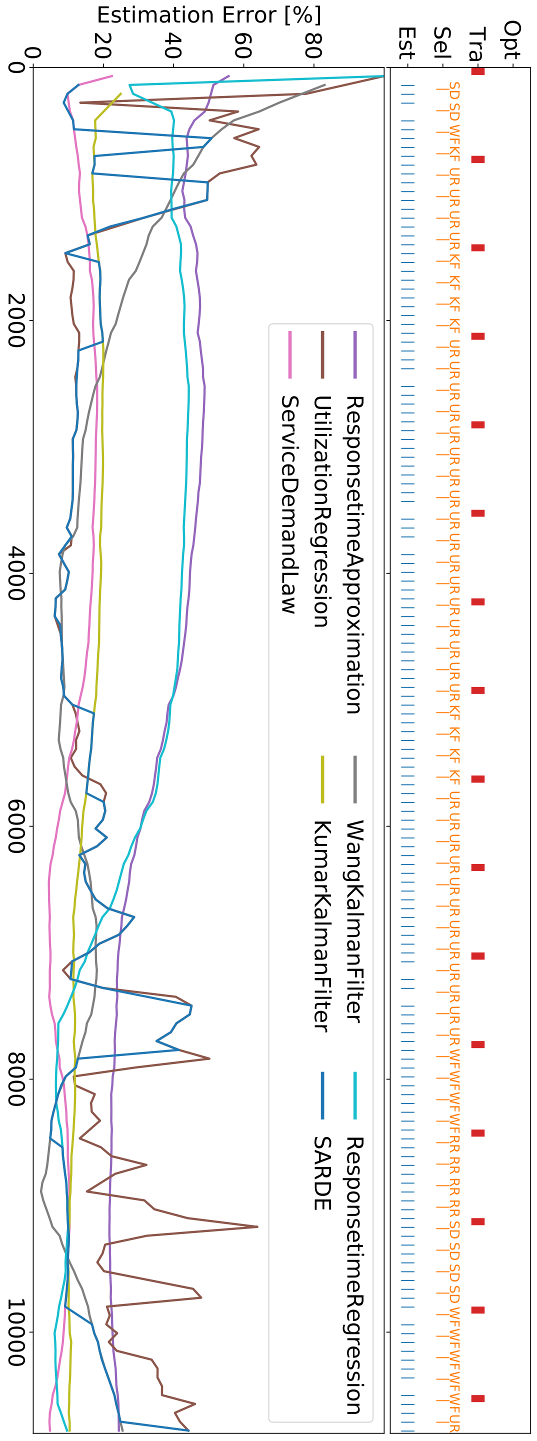

Figure 11.2: Estimation error of different approaches compared with *SARDE*'s selection.

Additionally, we observe that the *SARDE* approach (blue) jumps between different respective approaches. While *SARDE* needs a while to learn and adapt to the current trace (before 2 000 seconds), it then is able to predict and select among the best-performing approaches until the environment changes and the accuracy decreases (starting at 6 000 seconds). In reaction to this development, another approach is chosen at around 8 000 seconds until its performance decreases as well.

Table 11.2: Overview of the quality of selected approaches using the realistic application.

Approach	Average Rank	Accuracy	Accuracy Loss
SDL	2.02	11.52%	3.11%
RTA	5.47	35.04%	26.63%
RR	3.69	27.94%	19.53%
UR	3.64	23.84%	15.43%
WKF	2.94	18.74%	10.33%
KKF	3.21	15.17%	6.91%
SARDE	2.82	16.88%	8.64%
Random	3.08	18.49%	10.15%

In the following, we analyze Table 11.2 for more details on the selection results. Table 11.2 shows the average rank of each selection approach, together with its average total accuracy loss, that is, the average difference of the relative estimation error of the given approach in comparison with the current best approach. We observe that Kumar Kalman Filter (KKF) and SDL both have relatively low ranks and a small accuracy loss in comparison to other approaches. The RTA approach has a particularly high accuracy loss, as its performance is consistently worse than any of the other approaches.

SARDE is able to achieve an average rank of 2.82 with only 8.6% of accuracy loss towards the theoretical optimum. Compare this with a baseline approach of the random classifier, which achieves an average rank of 3.08 together with an accuracy loss of 10.2%. Note that it is not possible to simply choose SDL as the best approach, for example, as the knowledge about the performance of the individual approaches is not known prior to execution. Instead, the self-adaptive features of the selection approach of *SARDE* enable it to constantly monitor the performance of the individual approaches and switch between the most promising approaches. Therefore, *SARDE* is able to learn from and adapt to a scenario without any prior knowledge or training for that environment.

This partly answers **EQ 11.2** (*"What is the impact of applying algorithm selection, optimization, and both combined to the repeated estimation?"*). Next, we analyze the impact of the optimization process.

11.3 Optimization

After analyzing the selection process in detail, this section now focuses on the optimization process. Similar to the previous section, we first analyze the set of different micro-benchmarks representing a wide variety of test applications and then concentrate on a more in-depth analysis of our realistic application.

11.3.1 Micro-benchmarks

The focus of this section is to show the potential benefit of parameter optimization on our trace dataset. Naturally, not all estimation approaches have the same set of parameters available. For example, the two KF-based approaches, KKF and Wang Kalman Filter (WKF), have five approach-specific parameters that can be tuned. On the other side, other approaches, like SDL or RTA, do not have any parameters to fine-tune the respective approach. Table 11.3 shows the available optimization parameters for *SARDE* as well as the respective lower and upper bounds.

Table 11.3: Overview of available optimization parameters and the approaches supporting the respective parameters.

Parameter name	Lower bound	Upper bound	Approaches
Step size	10 s	360 s	SDL, RTA, UR, RR, WKF, KKF
Window size	1	60	SDL, RTA, UR, RR, WKF, KKF
Initial bounds distance	0.0	0.1	WKF, KKF
Bounds factor	0.0	1.0	WKF, KKF
State noise covariance	0.0	2.0	WKF, KKF
Observe noise covariance	0.0	0.1	WKF, KKF
State noise coupling	0.0	2.0	WKF, KKF

The only two parameters that are common to all approaches are concerned with the input processing of monitoring data. The *step size* describes the aggregation interval, that is, the interval for which all monitoring measurements are aggregated, and serves as the minimal time unit for each estimation approach. Additionally, the *window size* defines the memory of each approach, that is, the number of steps that are considered for each estimation approach. For example, if the step size is 60 seconds, and the window size is 60, then only the last $60s \cdot 60 = 3\,600s$ of measurements are considered for the estimation. Hence, the specific tuning of both parameters is more dependent on the individual trace than on the specific approaches, as it is more a configuration parameter (i.e., a parameter that needs to be set based on external requirements) than an optimization parameter (i.e., a parameter that can be freely chosen to optimize performance). This effect can also be observed later in Figure 11.3 on page 199.

Therefore, Table 11.4 focuses on the parameters of the two KF-based approaches, KKF and WKF. Table 11.4 shows the performance of our optimization tuning the five tunable parameters *initial bounds distance*, *bounds factor*, *state noise covariance*, *observe noise covariance*, and *state noise coupling* using the bounds defined in Table 11.3. In order to evaluate the results on the micro-benchmarking training sets, we split the 210 traces into 168 training traces and 42 validation traces. The training algorithm optimized the parameter of the training traces, while Table 11.4 shows the performance of the remaining 42 validation traces.

Table 11.4: Estimation error and chosen configuration parameters of our validation benchmarks before and after optimization.

Algorithm	KKF		WKF	
	Default	Optimized	Default	Optimized
Optimization time	–	6 456 s	–	8 878 s
Average estimated error	0.273	0.227	0.823	0.752
Relative improvement	–	16.7%	–	8.6%
Parameter values:				
Initial bounds distance	0.0001	0.0	0.0001	0.1
Bounds factor	0.9	0.75	0.9	1.0
State noise covariance	1.0	0.0	1.0	1.0
Observe noise covariance	0.0001	0.1	0.0001	0.0
State noise coupling	1.0	2.0	1.0	1.0

We observe that the default parameterizations (as proposed by the default configuration of the implementations) are sub-optimal for both KF variants. Both estimators could significantly improve the estimated error on the validation set. However, it is interesting that the KKF which performs already significantly better than WKF in its default configuration, also profits more from the optimization. Although the absolute error reduction is greater for the WKF, the relative improvement for the KKF (16%) is almost double the relative improvement for the WKF. In addition, we note that although KKF is slightly faster than WKF, both optimizations take comparatively long to optimize as they need to take all 168 training traces into account. To summarize, we can say that the optimization finds effective parameter optimizations, even if the validation traces are unknown to the algorithm.

11.3.2 Realistic Application

After analyzing the performance of our optimization procedure on the different micro-benchmarks, we now continue on our realistic application dataset. As already discussed in the previous section, most approaches are limited to only two configurable parameters: the step size and the window size. Therefore, we configure the optimization used in the previous section to optimize the KF parameters for the two KF variants while focusing on step size and window size for all other approaches (see Table 11.3). As these two parameters heavily influence each other, the optimization combines both into one parameter that only changes the window size relative to the respective step size.

Figure 11.3 depicts the estimation accuracy of the different approaches over time. Drawn lines represent the original error, dotted lines are the optimized versions. In addition, the dashed lines of each color represent the accuracy of the optimized approach. A new parameterization comes into effect at the first estimation interval (blue) after the end of each optimization interval (green). Every optimization run is able to utilize more data, as all collected data from the previous trace is used.

First, we observe that not all approaches (purple, turquoise) profit from the parameter optimization. This is due to the limitations of the optimizable parameter set as discussed above. On the other hand, there are other approaches (green, pink) that can profit greatly from changing the parameters. However, in summary, Figure 11.3 does unfortunately not conclusively prove or disprove the applicability and the effect of the optimization process. It can certainly affect the performance of the algorithms in both ways; hence, it is important to analyze the interplay between the optimization and the selection component. If the correct approaches are chosen, the optimization can help to improve the current approaches, while its negative effects are mitigated by the selection process. With this, we analyzed the impact of algorithm selection and optimization in order to address **EQ 11.2**. Following, we analyze the interplay of both processes.

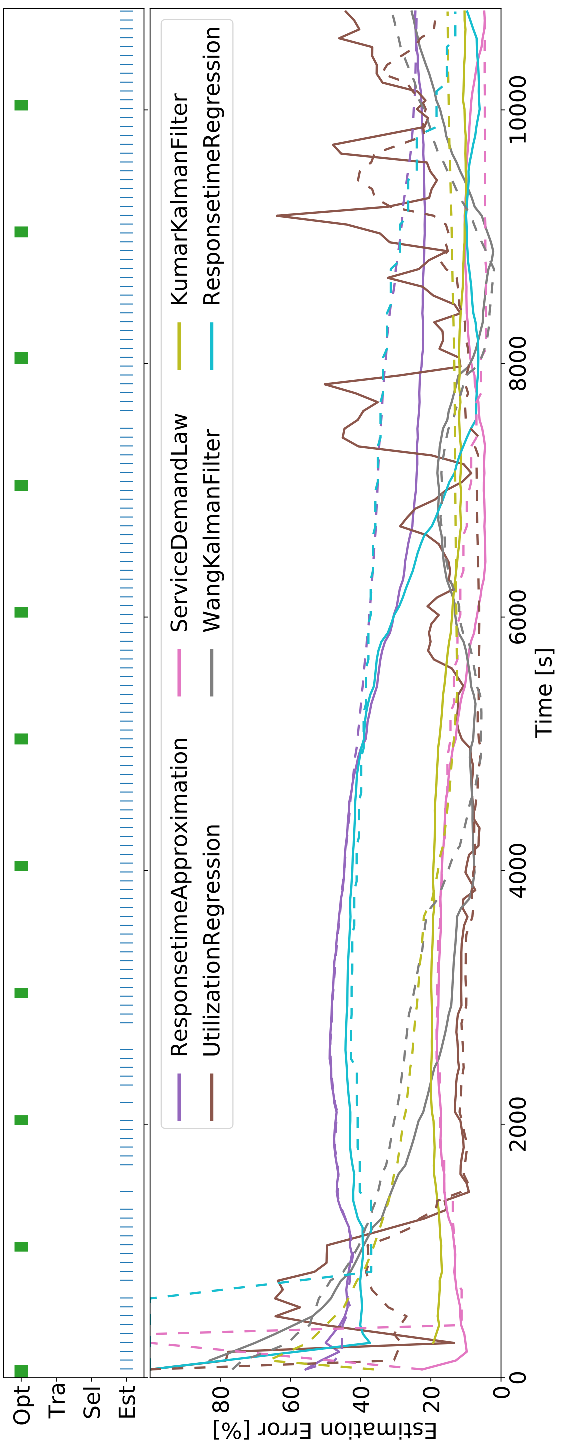

Figure 11.3: Estimation error while running the optimization.

11.4 Combination

Finally, we now combine the two processes of optimization and selection in order to evaluate their interplay as intended by the *SARDE* approach. For this, we focus solely on the realistic application dataset, as the optimization procedure and the selection interplay can only be analyzed over time which is infeasible for the 210 available micro-benchmark traces.

Analogously to the previous sections, Figure 11.4 depicts the estimation errors of the individual approaches over time. The individual approaches remain unchanged in comparison to the previous experiments. However, we include the blue estimation line that represents the *SARDE* estimation. We observe that *SARDE* is again efficiently able to choose between the different available selection approaches, as already seen in the analysis of Section 11.2. In addition to that, however, the blue estimation line now deviates from the standard approach estimations as the parameter optimizations change the performance of the estimations.

In the first half, *SARDE* shows some degrees of instability observable from frequent changes in the selected approaches as well as sudden spikes in estimation error. However, as soon as a spike occurs, the self-adaptation mechanisms counteract that behavior by changing the chosen approach or the applied parameters. Therefore, towards the end of the trace, the stability gradually increases. Additionally, we observe that at different points in time, the blue estimation line exhibits a lower estimation error than any of the other approaches. This is possible as the parameter optimization process gradually adapts to the specific properties of the trace and learns to fine-tune the estimation approaches towards that.

Table 11.5: Summary of selected approaches during the execution.

Approach	Number of selections
Service Demand Law	23
Response Time Approximation	0
Utilization Regression	7
Response time Regression	1
Kumar Kalman Filter	20
Wang Kalman Filter	12

Table 11.5 summarizes the different selections also observable at the top of Figure 11.4. Similar to our analysis in Section 11.2, we can confirm that the

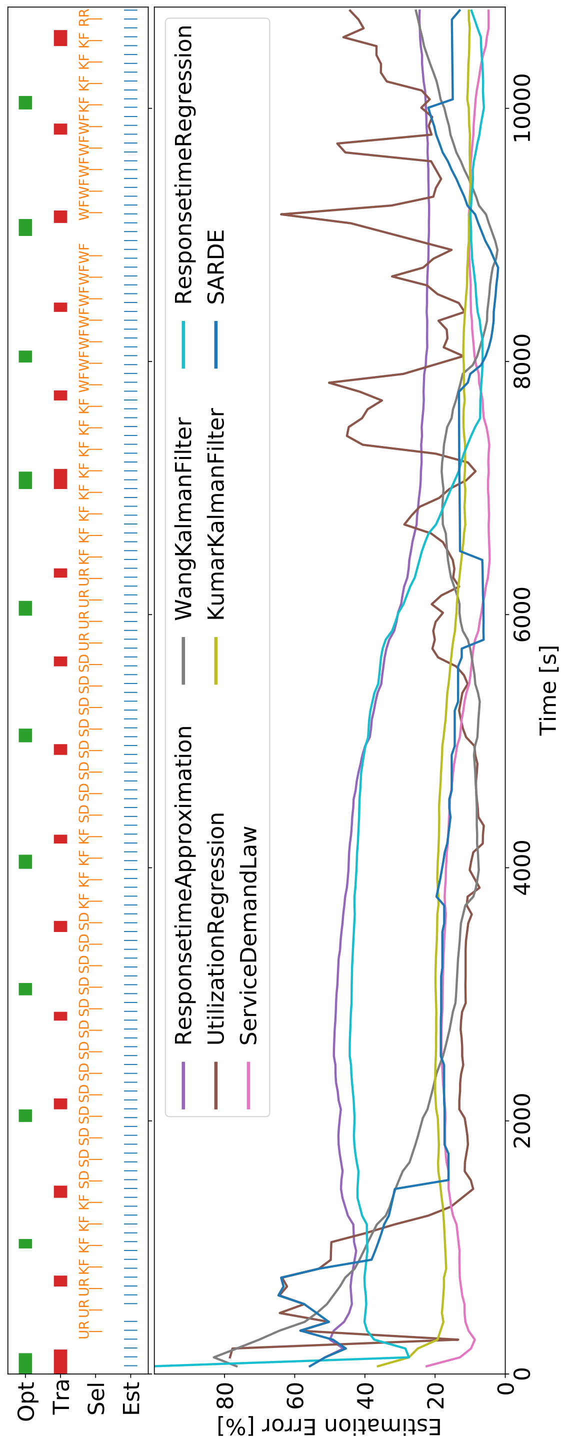

Figure 11.4: Estimation error while running all *SARDE* processes.

selection algorithm still chooses from almost all estimation algorithms (except the poorly performing RTA) in order to adapt to the respective situations. A qualitative comparison, as presented in Table 11.2, is not feasible as the respective selection always relates to the optimized estimation approaches.

Table 11.6: Estimation quality of *SARDE* using the realistic application.

Approach	Average Rank	Accuracy	Accuracy Loss
SARDE	2.79	15.96%	7.69%

In general, taking the results of the algorithm selection presented in Section 11.2, the optimization presented in Section 11.3, and the combination of both into account, it can be said that *SARDE* effectively combines the accuracy gain of both processes. We validate this by repeating the analysis from Table 11.2 on page 195 in Table 11.6. We exclude the optimization gain of the running optimization process and still use the unoptimized base estimators as a comparison. Therefore, Table 11.6 only updates the performance of *SARDE*, as the performance of all other approaches stays the same. Table 11.6 shows that the combination of optimization and recommendation slightly improves all three measures in comparison to only applying the recommendation in Table 11.2 on page 195. However, with an average rank estimation accuracy of 15.96%, in comparison to 16.88% without optimization, the gain is unfortunately not as significant as one might hope. Therefore, we can now fully answer **EQ 11.2** (*"What is the impact of applying algorithm selection, optimization, and both combined to the repeated estimation?"*). With all processes enabled, *SARDE* achieves an average estimation error of 15.96%.

11.5 Workload Analysis

The analysis in Section 11.4 helps us to understand the performance of *SARDE* during a continuous estimation. However, another angle at analyzing the given workload is to section it into different intervals. This enables us not only to analyze the performance of *SARDE* but also to relate it to the workload properties of the respective interval.

Therefore, Table 11.7 presents the main arrival rate properties of the three workload classes described in Section 11.1.1.2, together with the performance of *SARDE*, split into ten different intervals. Recall that workload class 1 (WC1) and workload class 2 (WC2) perform an exponentially distributed load with a mean of 0.01s and 0.03s, respectively. In contrast, the third workload class (WC3) performs a normally distributed load with a mean of 0.005s and a standard deviation of 0.001. Therefore, WC3 follows another intensity distribution and is comparatively light.

Table 11.7: Workload properties of different experiment intervals.

#	Mean			Standard Deviation			Index of Dispersion		
	WC1	WC2	WC3	WC1	WC2	WC3	WC1	WC2	WC3
1	0.00	21.26	16.59	0.00	9.11	14.67	–	3.90	12.98
2	10.21	6.88	30.22	7.76	3.08	7.32	5.90	1.38	1.77
3	24.25	4.42	17.02	3.58	2.40	3.75	0.53	1.30	0.83
4	24.41	2.36	9.57	3.61	1.88	2.84	0.53	1.50	0.84
5	13.76	5.35	3.74	6.64	4.69	2.80	3.21	4.11	2.10
6	0.13	8.66	1.55	0.48	3.55	1.21	1.77	1.46	0.94
7	0.00	2.17	1.46	0.00	1.32	1.04	–	0.81	0.74
8	0.00	2.34	2.75	0.00	1.38	1.69	–	0.82	1.04
9	0.00	4.49	9.87	0.04	1.86	3.58	1.00	0.77	1.30
10	2.73	5.23	19.87	2.92	2.09	3.06	3.13	0.84	0.47

Table 11.7 shows the mean, the standard deviation, and the index of dispersion [Cox14] of each workload class arrival rate in requests per second during the respective interval. The index of dispersion is calculated by dividing the variance, that is, the squared standard deviation, by the mean [Cox14]. We observe that all ten intervals show vastly different workload characteristics. For WC1, the intervals vary between 0 and 25 requests per second, together with the standard deviation between 0 and over almost 8. The respective index of dispersion is not defined for mean values of zero. In other cases, the index rises to over 5 in interval 2. The other workload classes show similar behavior, with mean arrival rates varying by a factor of 10 and index of dispersion values ranging from as low as 0.8 up to a maximum of almost 13 in interval 1.

In addition, we note that the variations of the three workload classes are independent and widespread in the different analyzed intervals. For example, in interval 1, WC3 has the highest Index of Dispersion of almost 13, while WC2 also has a significant amount of dispersion, and WC1 is absent. In the following interval, the measured dispersion drops for WC2 and WC3, while it increases to the trace maximum of 5.9 for WC1. Hence, we conclude that all intervals contain vastly different workload patterns and intensity variations.

Therefore, we can now analyze the performance on *SARDE* at different intervals to see how the estimator performs. Table 11.8 shows the average error

Table 11.8: Estimation error of *SARDE* for the different experiment intervals.

Interval	1	2	3	4	5	6	7	8	9	10
SARDE	0.52	0.23	0.18	0.18	0.15	0.09	0.13	0.07	0.09	0.17

of *SARDE* for each of the intervals. We observe relatively high errors in the first two intervals, while the performance stabilizes starting in interval 3. This could be due to the massive dispersions shown by WC3 and WC1 in the first two intervals or to the fact that *SARDE* has not yet collected a sufficient amount of knowledge over the system. However, after these two critical intervals, we observe that *SARDE* delivers relatively stable estimations, which are not influenced by the distributions of the data. One possible conclusion is that the task becomes significantly easier if one workload class is removed from the trace, as the accuracy improves for intervals 6, 7, 8, and 9, where WC1 is mostly absent. In summary, Table 11.8 shows that *SARDE* has a reliable and stable performance in our test evaluation, although the workload characteristics of the individual intervals listed in Table 11.7 are vastly different.

11.6 Overhead Analysis

Lastly, we evaluate the overhead introduced by applying the *SARDE* approach. Naturally, all self-adaptation and self-optimization processes we introduced in this work increase the computation effort for estimating the resource demands. Therefore, the question arises whether or not the additional effort is worth spending and to weigh the achieved benefit with the required additional costs. The additional computation effort can already be seen by analyzing the top part of Figure 11.4. However, for a more quantitative approach, we additionally summarize the different execution times in Table 11.9.

Table 11.9: Overhead analysis of the individual activities.

Activity	Executions	Avg. time	Std. deviation	Total time
Estimation	154	0.2 s	0.7 s	38.5 s
Optimization	11	113.1 s	23.4 s	1244.3 s
Selection	63	0.2 s	0.1 s	11.5 s
Training	16	96.8 s	33.4 s	1548.1 s

First, we notice that in total, 154 resource demand estimations are conducted. On average, each estimation takes around 200 ms to compute, resulting in roughly 39 seconds of computation time spent for the continuous estimation. The second most executed process is the selection of an estimation approach based on an already trained machine learning model. This selection process is similarly cheap compared to the actual estimation process, resulting in additional 12 seconds of computation effort spent on recommending.

In contrast to executing the selection model, which is comparatively fast, each machine learning training run takes about 97 seconds to complete. Therefore, the training is executed much more sparsely, resulting in a total training time of just under 26 minutes. Finally, the optimization process is, as expected, the most expensive technique of all self-adaptation processes. However, due to the relatively low amount of 11 executions, just 21 minutes of computation power is spent, as each optimization procedure takes slightly less than over 2 minutes on average.

In total, *SARDE* consumes 2 844 seconds or 48 minutes of computation time over the full duration of our three-hour experiment. Given that one is able to efficiently scale the required computation power (as standard in modern-day cloud computing environments), one is expected to utilize well under one CPU-core while running *SARDE* (27% in this experiment). Note that this number is strongly dependent on the applied configurations, mainly on the two most expensive processes of optimization and training. Fewer executions or different parameterizations greatly influence the perceived overhead. Hence, this answers **EQ 11.3** (*"What is the overhead of applying these techniques?"*).

11.7 Summary

In this chapter, we evaluate the selection and the optimization of resource demand estimation approaches using two different datasets. The first is a collection of many different short-lived micro-benchmark scenarios, and the second one is a realistic web application. Additionally, we analyze how the combination of both approaches inter-operates on the web application and also analyze the overhead of each individual activity performed by *SARDE*. In general, the web application shows that resource demand estimates are continuously changing. This insight demands the continuous repetition of estimations, and, therefore, answers **EQ 11.1** (*"What is the gain of continuously repeating the estimation?"*). We see that selection and optimization are able to continuously provide benefits for the ongoing estimations. In total, *SARDE* achieves an average estimation error of 15.96% over the whole three-hour measurement period. These results help to address **EQ 11.2** (*"What is the impact of applying algorithm selection, optimization, and both combined to the repeated estimation?"*). Finally, we conclude that on our evaluated datasets, the overhead is minimal in comparison to the achieved self-adaptive properties *SARDE* offers, as *SARDE* requires 48 minutes of computation time during the three-hour experiment. This addresses **EQ 11.3** (*"What is the overhead of applying these techniques?"*).

In addition to the source code of *SARDE* (see Chapter 6 on page 91), we published the code for constructing and analyzing the experimentation dataset[3] and a replication package for the evaluation results on CodeOcean [Gro+21a]. In the following, we discuss the remaining threats to the validity of our evaluation in Section 11.7.1 and the limitations of *SARDE* in Section 11.7.2.

11.7.1 Threats to Validity

Although we conducted the presented evaluations with great care, there are some remaining threats to validity to discuss.

11.7.1.1 Internal Validity

Our evaluation of the online application is based on a synthetic application, written especially for this analysis. This way, it is possible for us to exactly define and program the specific resource demands into the application, which is crucial in order to calculate the respective estimation errors. However, although the resource demand of the application was precisely programmed, the experienced resource demand is still influenced by other factors, like network and middleware overhead or hardware variabilities. Therefore, the error calculation itself has to be viewed as an estimation. However, the presented synthetic application still provides reasonable results compared to the completely unknown resource demands of any real-world application. Therefore, the presented analysis still provides the best accuracy for our analysis.

Additionally, we note that all self-adaptation and optimization processes of *SARDE* are dependent on the internal validation error. The internal error estimates the error of the respective estimation based on the incoming measurements (as the gold standard is unknown). Therefore, this internal error function is of paramount importance for the performance of all self-adaptation techniques of *SARDE*. Therefore, all presented benefits and overheads are strongly dependent on the chosen error configuration, next to the other given configuration parameters presented in Section 11.1.3.

[3]`https://github.com/jo102tz/LibReDE-SARDE-data`

11.7.1.2 External Validity

Concerning external validity, all presented error measures and especially the measured computation time of the realistic application reflect just the one repeatable estimation run. Different input data streams from different applications or measured on different systems could lead to different results. Especially the overhead analysis must be viewed as an exemplary analysis, as its values are heavily dependent on the chosen parameterization as well as the respective machine learning algorithms or optimization techniques. As already discussed in the previous section, the repetition intervals can be arbitrarily changed as well; therefore, the results of the overhead analysis can not be directly transferred to any arbitrary system.

In addition, our experiment results are limited to the evaluated workload patterns and resource demands presented in Section 11.1. We did our best to spread and diversify the analyzed scenarios, which is observable in our analysis in Section 11.5. Nevertheless, future work could aim at extending our analyses in order to verify whether the results transfer other scenarios as well.

11.7.2 Limitations and Assumptions

In this section, we discuss the limitations that *SARDE* currently faces. The presented results only focus on six of the eight available approaches within LibReDE, as the two techniques based on recursive optimization [Men08; Liu+06] are based on an incompatible optimization library and are therefore not usable for the presented study. However, the results using the presented six methods already show the benefits of *SARDE*. Note that this represents a strict technical limitation that does not affect the conceptual contribution of this work and could be therefore addressed in future work to further improve the presented results.

Similarly, LibReDE currently does not support the notion of uncertainty in the monitoring streams, being it due to missing values, low accuracy, or lack of precision. Therefore, *SARDE* is also not able to support uncertain monitoring streams. However, future versions might enable confidence values or multiple measurement repetitions in order to remedy that problem.

While all activities are designed for continuous and online applications, the current implementations are based on repeated batch learning. Therefore, while the data patterns offer the possibility for online learning and online algorithm selection capabilities, this is currently not implemented. However, it is expected that such techniques would mainly improve the computation times and therefore further simplify the use of the *SARDE*.

Finally, our experiment explicitly does not focus on extrinsic changes in resource demands. Such a resource demand change would, for example, occur if the running application is redeployed or changed using CI/CD pipelines following a new commit. This would invalidate all previous resource demand estimations and require a reset of the monitoring traces of the affected parts of the system. We focus specifically on such incremental extraction approaches in another line of our research [Maz+20; Von+20] in collaboration with researchers from the Karlsruhe Institute of Technology. However, as *SARDE* is designed for continuous changes in the environment, we are confident that the approach is able to work in such scenarios. This could be done by adding a trigger interface to the framework, signaling invalidation of parts or the total current application model.

Chapter 12

Evaluating the Learning of Parametric Dependencies

In this section, we will concentrate on evaluating our contributions in the area of learning parametric dependencies, as introduced in Chapter 7 on page 109. Therefore, our goal is to evaluate *DepIC*, our approach towards dependency extraction addressing **RQ III.2** (*"How can the impact of parameters on resource demands be identified and characterized?"*). Based on this goal, we devised the following EQs:

- **EQ 12.1**: *Which feature selection technique is suited best for the identification of parametric dependencies?*
- **EQ 12.2**: *What is the effect of the proposed result filtering approach on the quality of the identified dependencies?*
- **EQ 12.3**: *What is the impact of increased system utilization on the quality of the identified dependencies?*
- **EQ 12.4**: *How does the created meta-selector influence the characterization accuracy?*

In the following section, we designed experiments for specifically answering the posed EQs. We will answer **EQ 12.1** in Section 12.1, focus on **EQ 12.2** in Section 12.2, and discuss **EQ 12.3** in Section 12.3. Finally, Section 12.4 concentrates on the characterization and **EQ 12.4**. We conclude this chapter with a summarizing reflection in Section 12.5. The TeaStore [Kis+18], a microservice reference application that we utilize in this experiment, is already introduced in Section 9.1 on page 152.

12.1 Comparison of Feature Selection Techniques

Our approach is modular and can include any existing feature selection technique. As described in Section 7.1.4 on page 121, we integrate one technique from each of the three categories of feature selection methods: a filter method, an embedded method, and a wrapper method. We evaluate which of these techniques produces the best results and what the advantages and disadvantages of each technique are in order to answer **EQ 12.1** (*"Which feature selection technique is suited best for the identification of parametric dependencies?"*).

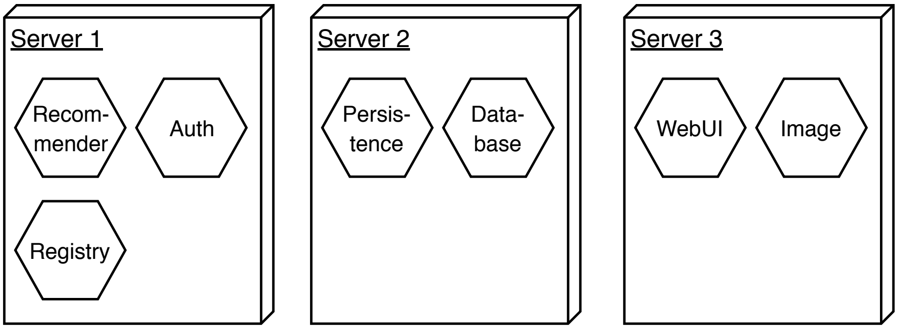

Figure 12.1: Deployment of the TeaStore services in the presented experiment.

12.1.1 Experiment Setup

For this experiment, we distribute TeaStore across three servers as shown in Figure 12.1: The `Recommender`, the `Auth`, and the `Registry` are deployed on Server 1 (10-core Intel® Xeon® E5-2650 v3 @ 2.30 GHz). The `Persistence` and the `Database` are co-located on Server 2 (12-core Intel® Xeon® E5-2650 v4 @ 2.20 GHz). Lastly, the `WebUI` and the `ImageProvider` are deployed on Server 3 (8-core Intel® Xeon® E5-2640 v3 @ 2.60 GHz). Apart from the CPU, all hosts are identical HP ProLiant DL360 Gen9 servers, equipped with 32 GB of RAM, running Debian 9 and Docker 17.03.2-ce. All services run inside Docker containers, each assigned one core and 4 GB of memory limits to ensure resource isolation. We apply a closed workload [SWH06; Kis19] of one user with a think time of zero. The workload profile is similar to TeaStore's browse profile covering a normal user interaction with TeaStore. We use the closed workload so we can ensure that the monitored response time is equal to the resource demand. We run the experiment for two hours, resulting in a total of 2.4 million monitoring records. We implement the framework using Java and rely on the WEKA library [Hal+09; FHW16] for the implementations of the machine learning algorithms.

In order to evaluate the feature selection techniques in isolation, we firstly apply the proposed approach without filtering the results as presented in Section 7.1.6 on page 124. This enables us to compare the unprocessed results of the different selection techniques. As a gold standard, an expert creates a set of expected dependencies. For this experiment, the expert labels any dependencies that exist and not only the dependencies he would include in a performance model. This makes sense, as we are looking to evaluate the ability of the feature selection techniques to detect any existing dependencies from the monitoring data. As the process of manual labeling is time-intensive and requires an in-depth analysis of the source code, we limit the scope of this evaluation to the `Recommender` service.

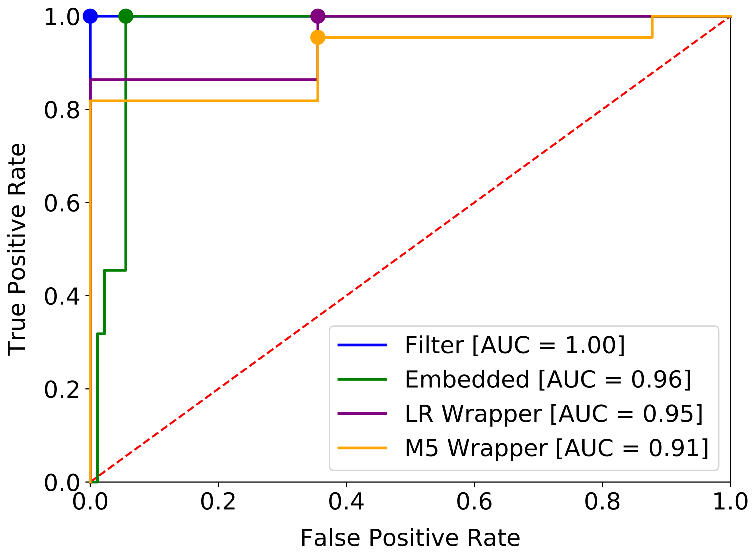

Figure 12.2: ROC curve and AUC score for the four analyzed feature selection techniques.

12.1.2 Threshold Analysis

Next, recall that all feature selection approaches can be parameterized with a threshold parameter θ, as introduced in Section 7.1.5 on page 123. We study the impact of the threshold parameter θ by analyzing the ROC curve of each proposed feature selection technique. Figure 12.2 depicts the Recall (True Positive Rate) against the FPR. The circles indicate the selected threshold. We describe these metrics in more detail in Section 2.3.2 on page 27. Simply put, the ROC curve displays the impact of decreasing the threshold θ, as it shows what Recall can be achieved for a given FPR. The AUC score is defined as the area under the ROC curve. Therefore, the AUC is the integral over the ROC function and has an upper bound of one. Higher values are preferable, as they express the probability of ranking a positive example over a negative one.

Figure 12.2 includes the ROC curves for the four approaches together with the corresponding AUC score. The dashed red line shows the theoretical performance of a random classifier. All approaches clearly outperform a random classifier, each offering a certain trade-off. We note that all approaches achieve an AUC score > 0.9, with the LR wrapper and the embedded approach slightly outperforming the M5 wrapper. Note that we included two variants of the wrapper approach, as explained in the following section. However, the filter approach even exceeds their performance by offering a perfect ranking and an AUC score of 1.0. This means that there exists a threshold resulting in a perfect classification of all possible dependencies.

As our focus is on the detection of dependencies, a high number of true positives is desirable. Therefore, based on the threshold analysis, we choose the thresholds depicted in Figure 12.2, with a threshold $\theta_{Filter} = 0.8$ for the filter approach, a threshold $\theta_{Wrapper} = 200$ for the M5 and the LR wrapper, and a threshold $\theta_{Embedded} = 30$ for the embedded approach.

12.1.3 Results

The first thing to note is that during the execution using this comparatively small dataset, the wrapper algorithm using M5 took over two hours to complete. Therefore, we also include a wrapper variant using the significantly faster LR base algorithm. This variant terminates after ten minutes. In comparison, the Embedded approach terminates after 14 minutes, while Filter runs for only five minutes. We conclude that all but the M5 wrapper terminate after acceptable time, as monitoring data from a two-hour period and containing 2.4 million records is analyzed. Therefore, we focus on the accuracy of the individual approaches in the following.

The results of the respective approaches are presented in Table 12.1. All approaches use the optimized thresholds from Section 12.1.2. It shows the absolute number of TPs, FNs, and FPs of each approach based on the gold standard labeled by our expert. We furthermore add the TNs, that is, the number of correctly not-identified dependencies as well as the resulting precision, recall, and F1 scores.

We observe that the filter approach seems to be suited best for the automated identification of parametric dependencies (**EQ 12.1**). It manages to identify all 22 dependencies without identifying any incorrect dependencies and therefore achieves an F1 score of 1.0. The embedded approach performs reasonably well, as it also identifies all 22 correct dependencies with only five false positives, achieving an overall F1 score of 0.9. In contrast, both wrapper approaches identify 32 additional incorrect dependencies. However, the wrapper approach based on LR also detects all 22 dependencies, while the M5 wrapper misses one of the correct dependencies, resulting in the lowest F1 score of 0.56.

Table 12.1: Accuracy of different feature selection techniques.

Approach	TP	FN	FP	TN	Precision	Recall	F1
Filter	22	0	0	90	1.00	1.00	1.00
Embedded	22	0	5	85	0.81	1.00	0.90
LR wrapper	22	0	32	58	0.41	1.00	0.58
M5 wrapper	21	1	32	58	0.40	0.95	0.56

To summarize, all approaches manage to identify almost all of the required dependencies and achieve a high recall. However, the filter approach outperforms the comparable approaches by achieving a perfect F1 score of 1.0 by not including any false positives. We attribute the large number of false positives of the wrapper and the embedded approach to the fact that their respective underlying machine learning techniques are prone to over-fitting. Additionally, the tuning of the hyperparameters of the underlying machine learning algorithms for both the embedded and wrapper approaches has a significant influence on the classification performance. Overall, we conclude that the wrapper approach is not well suited for the identification of parametric dependencies, as it results in a large number of false positives. Furthermore, the long run time of the M5 wrapper approach (over two hours for a single service) makes it infeasible to execute for larger systems. The embedded approach can be used in scenarios where a human validates the proposed dependencies, as in such cases, one may prefer a slight tendency towards false positives in order not to miss any relevant dependencies. These results can be used to answer our first evaluation question, **EQ 12.1** (*"Which feature selection technique is suited best for the identification of parametric dependencies?"*), as we conclude that the filter approach is suited best for our scenarios.

12.2 Result Filtering

In Section 7.1.6 on page 124, we introduced three steps of filtering the identified dependencies in order to retain only performance-relevant dependencies that should be considered for inclusion in a performance model. We now evaluate if

DepIC accidentally removes any performance-relevant dependencies or, if it fails to remove any dependencies that are not performance-relevant by answering **EQ 12.2** (*"What is the effect of the proposed result filtering approach on the quality of the identified dependencies?"*).

12.2.1 Experiment Setup

We apply the same test setup as before but with traces from the entire TeaStore application as input for this section of our analysis. This makes it infeasible to have a predefined gold standard by a human expert, as the evaluated system is too complex for detailed manual inspection. Instead, we go through all identified dependencies and label them with respect to the expert's knowledge of the system.

This setup is appropriate for this evaluation step, as here we are only interested in analyzing the impact of the result filtering (**EQ 12.2**). The results of Section 12.1 suggest that the filter approach is the most effective technique to identify dependencies. Hence, in the following experiments, we focus our evaluation on the filter approach.

12.2.2 Results

We distinguish between three types of dependencies: (1) relevant dependencies, (2) irrelevant dependencies, and (3) invalid dependencies. *Relevant* dependencies are the ones we are looking for and want to include in our performance model. *Irrelevant* dependencies are dependencies that are semantically correct but do not influence the accuracy of the performance model (neither positively nor negatively). *Invalid* dependencies are dependencies that are either semantically incorrect or negatively influence the accuracy of the performance model. Hence, we want to focus on relevant dependencies to keep the performance model reasonably sized and understandable for a human. As an automated approach can not decide whether or not a modeled dependency is semantically correct based on the monitoring data, the goal of the filtering step goal is not primarily targeted at filtering invalid dependencies. Instead, we focus on reducing the number of irrelevant dependencies, as already explained in Section 7.1.6 on page 124.

Table 12.2 presents the impact of the individual steps of the result filtering. Initially, our approach identifies 110 dependencies on the dataset, of which 94 are irrelevant and five invalid. After filtering dependencies that involve identical parameters, we are left with 61 dependencies. Hence, 49 dependencies are filtered, all of which are irrelevant.

Table 12.2: Impact of each result filtering step.

Result filtering	Identified dependencies			
	Relevant	Irrelevant	Invalid	Total
None	11	94	5	110
Identical	11	45	5	61
Ident. + Correlating	11	35	1	47
Ident. + Correl. + Graph-based	11	8	1	20

Figure 12.3 shows some example dependencies that are deleted during step 1 of the filtering process. The class and call names are abbreviated. The `list` parameter refers to the size of the list. The `WebUI` issues a query, requesting a list of product recommendations based on the list of items in the cart of a user. This query is received by the REST endpoint of the `Recommender` component, which then forwards the request to a strategy selector. The strategy-selector chooses the appropriate algorithm (in this case, the `Orderbased` recommender), which then processes the actual request.

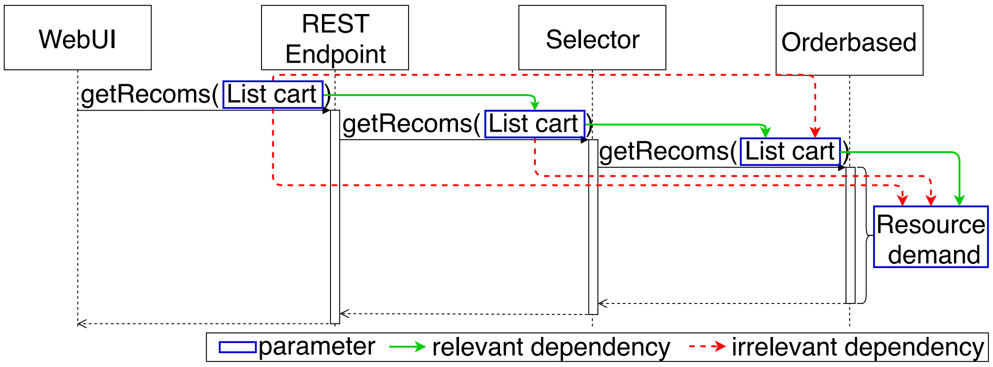

Figure 12.3: Call path for `getRecommendations` annotated with identified dependencies.

The resource demand of the `Orderbased` recommender is dependent on the number of items in the cart (i.e., the length of the `cart` list parameter), which is correctly identified. However, as shown in Figure 12.3, the approach actually detects a dependency from all list sizes to the respective resource demand. While this is technically correct, we drastically improve readability by removing all red dashed dependencies in Figure 12.3. We end up with the description of the list being forwarded through several components and its impact on the resource demand of the `Orderbased` recommender, where it finally gets processed. As these types of indirection are a common practice in software engineering, 46 additional and similar cases of parameter pass-throughs can be filtered in step 1.

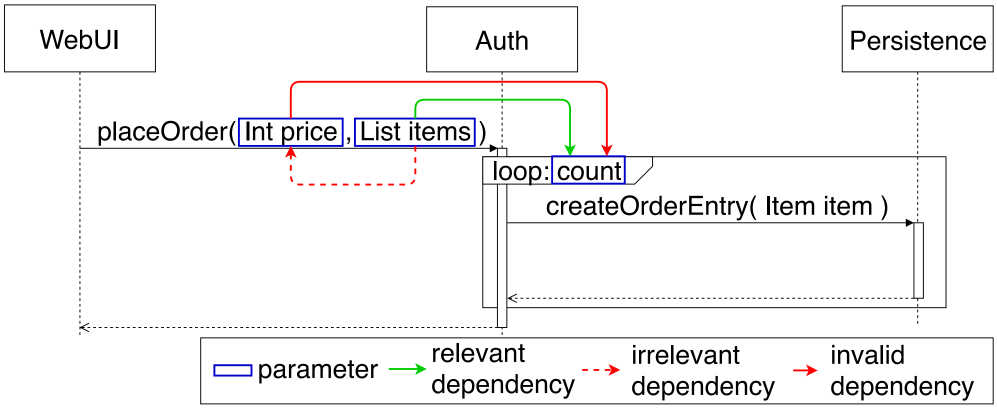

Figure 12.4: Call path for `placeOrder` annotated with identified dependencies.

The second filtering step deals with correlating dependencies. An example is shown in Figure 12.4, again with class and call names abbreviated, while the `list` parameter refers to the list size. Here, the `WebUI` places an order which is sent to the `Auth` service to verify that the user is logged in. If so, for each item in the cart, a new `OrderEntry` is created at the `Persistence` service. This represents a loop that is called once for each element in the `items` list. However, we can see that two dependencies influencing the loop count are identified: (1) the size of the list, which does make sense, and (2) the price of the order, which seems strange. Indeed, there is a correlation between the size of the list and the price of the order. The probability of a higher price of the order increases if more elements are in the cart. Therefore, both the cart size and the total price of the order correlate with the loop count. However, the price of the order directly influencing the loop count is an invalid dependency since it would imply that an increase in item prices would lead to a higher loop count, which is invalid. As there exists a dependency and, therefore, a correlation between the size of the cart and the total cart price, step 2 deletes the dependency of `price` to the loop count since the relation involving the size of the cart is stronger. In addition to ten irrelevant dependencies, the second step filters four invalid dependencies, resulting in a total of 14 filtered dependencies.

Finally, we look at the graph-based filtering introduced in Section 7.1.6 on page 124. Note that the irrelevant dependency mapping the size of the cart to its price, as depicted in Figure 12.4, is not deleted in step 2. Instead, the following graph-based filtering filters this dependency. The graph-based filtering will not mark this dependency during the breadth-first step, as it is not related to any performance-relevant parameter. The irrelevant dependency depicted in Figure 12.4 will therefore be deleted by the third filter step, together with 27 other irrelevant dependencies.

To summarize, we can say that all three filtering steps combined delete 86 irrelevant and four invalid dependencies. Therefore, over 90% of undesired dependencies are filtered. After analyzing the remaining one invalid dependency, we conclude that its existence is rooted in the applied workload. Our approach finds a dependency between the number of requested images and their requested size at the `ImageProvider` component. This is due to the implementation of the `WebUI`, requesting either a list of small product review pictures or one single high-resolution product detail image. Therefore, this relation is contained and can be observed from the monitoring data. However, we classify it as invalid as this relationship is not based on the software code but rather on the workload profile of the specific component. Nevertheless, based on the monitoring data, this observation is actually correct. Without in-depth knowledge about the application, it is not possible for *DepIC* to filter these rare edge cases.

Additionally, seven of the remaining eight irrelevant dependencies are added due to the addition of the discussed invalid dependency. These are technically correct (e.g., modeling the parameter passing of the requested image size) but finally relate to the discussed invalid dependency and are therefore marked as irrelevant as they do not model a performance-relevant property. If we were to remove the discussed invalid dependency by hand, these seven irrelevant dependencies would subsequently be filtered by the described filtering mechanisms, as they describe a relationship to this invalid dependency. After doing so, we are left with a single irrelevant dependency. In addition, we note that no relevant dependencies were filtered, which answers **EQ 12.2** (*"What is the effect of the proposed result filtering approach on the quality of the identified dependencies?"*).

We conclude that *DepIC* is able to enrich the base performance model of TeaStore with 11 relevant dependencies learned from monitoring data. We can furthermore calculate the precision using relevant dependencies as true positives and invalid ones as false positives (given that irrelevant dependencies can neither be counted as true positives nor false positives). With 11 relevant and a single invalid dependency, *DepIC* achieves a precision of 91.7% after all filtering steps.

12.3 Impact of System Utilization

In the current version, *DepIC* approximates the resource demands based on the observed response time, as detailed in Section 7.1.1.2 on page 115. This approximation works well under low resource utilization but becomes inaccurate with increasing load [Spi+15]. We now evaluate the impact of different utilization levels on the accuracy of our approach. To this end, we conduct our next measurement series.

12.3.1 Experiment setup

We start with the identical setup as already used in Section 12.2. However, we now increase the number of users concurrently using TeaStore to cause higher system utilization. We first determine that the `WebUI` is the bottleneck service by analyzing the utilization of each container. Next, we empirically analyze the workload intensity of a closed workload that leads to a certain CPU utilization of the bottleneck service. For each utilization level, we collect monitoring data for two hours. We feed this data into the filter-based dependency detection algorithm with all filtering steps activated, similarly to the setup used in Section 12.2, and label the resulting dependencies according to the same scheme. The applied setup used in Section 12.2 created roughly a utilization of 15% at the `WebUI`. In this experiment, we now add measurements for utilization levels of approximately 20%, 30%, 40%, 50%, 60%, 70%, 80%, and 90% CPU utilization at the bottleneck resource.

12.3.2 Results

The results of our analysis are shown in the stacked chart of Figure 12.5. We can observe that the number of irrelevant and invalid dependencies increases with higher system utilization. This is expected as the accuracy of the response time estimation for resource demands decreases with increasing utilization [Spi+15]. However, our approach still identifies all correct dependencies even with increasing resource utilization at the bottleneck service. Furthermore, we observe that there exist stronger factors influencing the accuracy of the approach than the load level alone, as the number of irrelevant and invalid dependencies is not increasing monotonously and is therefore influenced by other factors as well.

This is contrary to our initial expectation, as inaccuracies of the resource demand estimation should lead to inaccuracies in the detected dependencies. We conclude that the inaccuracies of resource demand estimates do not affect

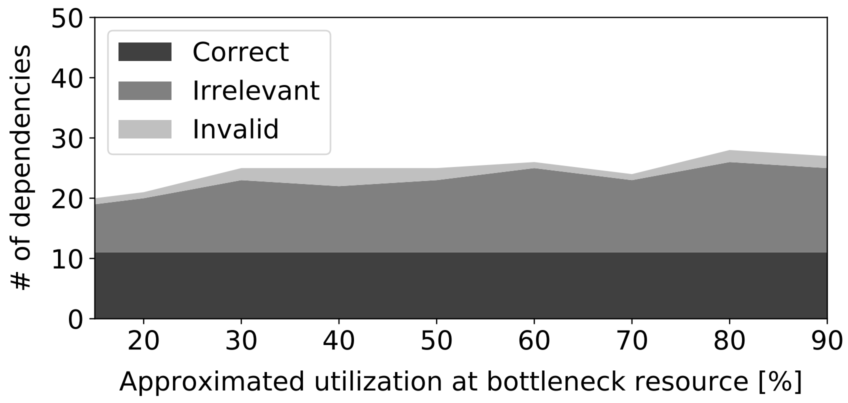

Figure 12.5: Classes of identified dependencies from monitoring data for increasing utilization levels.

the approach as strongly as expected since the relative differences between the resource demand estimates remain the same. Even though the increased load has a negative effect on the accuracy of the estimated resource demands, all demands get affected to a similar degree. Therefore, the relative proportions between the estimates stay intact and are still observable by *DepIC*. This answers **EQ 12.3** (*"What is the impact of increased system utilization on the quality of the identified dependencies?"*). We conclude that *DepIC* is still usable with monitoring data from higher utilization levels; however, manual review of the identified dependencies becomes increasingly necessary as the precision slightly deteriorates.

12.4 Dependency Characterization

Finally, this section aims at evaluating the impact and the performance of the meta-selector trained and created in Section 7.2.3 on page 133. Therefore, the following experiment focuses on evaluating the selections of the trained selector on previously unseen dependency characterization tasks. The gained insights can be used to answer **EQ 12.4** (*"How does the created meta-selector influence the characterization accuracy?"*).

12.4.1 Experiment Setup

Using the dependent Student's t-test [Stu08] for paired samples, we can test whether using the selected technique A significantly reduces the average prediction error μ (i.e., the MAE) compared to using a fixed machine learning approach B that generally performs well. Recall that SVR was by far the best approach in our experiment data, which is why we choose SVR as a baseline algorithm for this experiment (see Section 7.2.2 on page 131). The paired t-test is appropriate, as the samples are randomly selected, the data is paired, and approximately normal distributed. We furthermore assume that the variance of the two approaches is equal. However, this assumption might not necessarily be accurate as the two approaches might have a different prediction behavior. We choose the null hypothesis $H_0 = \mu_A \geq \mu_B$ and alternative hypothesis $H_1 = \mu_A < \mu_B$.

Table 12.3: Dataset for meta-selector evaluation.

Name	Input parameter name	Range
ArrayListSerialization	arrayListSize	1 – 1 000
BinarySearchArray	arraySize	1 – 10 000
	key	0 – 100 000
	isSorted	0 – 1
MatrixMultiplication	numRowsA	1 – 50
	numColumnsA	1 – 50
	numColumnsB	1 – 50
SolveTowersOfHanoi	numDisks	1 – 20
TrainMLP	numInstances	1 – 2 000
	numEpochs	1 – 2 000

The paired samples are obtained on five new runtime datasets, listed in Table 12.3. In the following, we give a brief description of each dataset:

ArrayListSerialization serializes and stores a numerical array to the disk. The parameter `arrayListSize` describes the number of elements contained in the array.

BinarySearchArray performs a binary search in a given integer array and a given integer key. The parameter `arraySize` describes the number of numerical elements contained in the array, `key` is the element to search for, and `isSorted` is a boolean value. The value is true if all values in the array are ascending in value and false otherwise.

MatrixMultiplication executes a multiplication of two matrices. Their size is specified with `numColumnsA` (columns of the first matrix), `numRowsA` (row of the first matrix) and `numcolumnsB` (columns of the second matrix). The number of rows of the second matrix is implicitly defined by the size of A.

SolveTowersOfHanoi is an algorithmic solver of the Towers of Hanoi problem [HKP18]. The parameter `numDisks` describes the number of disks that need to be moved.

TrainMLP trains a multi-layer perception. The two parameters `numInstances` and `numEpochs` describe the number of instances used for training and the number of training epochs, respectively.

These datasets were obtained using the same methodology as presented in Section 7.2.1 on page 127. Per set, we use 10 different training set sizes ($n =$ 20, 50, 200, 400, 700, 2 000, 4 000, 6 000, 8 000, 10 000), resulting in a total of 50 paired samples. For each sample i, we calculate the difference d_i between the MAE of A and B on the dataset's respective test set:

$$d_i = \mathrm{MAE}_i(A) - \mathrm{MAE}_i(B).$$

Next, we calculate the mean $\bar{d}$ and the standard deviation $\hat{\sigma}_d$ of the set of differences d. With this, we can calculate the t-value of our dependent samples:

$$t = \sqrt{n}\frac{\bar{d}}{\hat{\sigma}_d}.$$

The following section presents the results for our specific datasets.

12.4.2 Results

After discussing the experiment methodology, we now analyze the results achieved by the trained meta-selector. The mean MAE difference $\bar{d}$ over all samples is 20 884 576 ns and the standard deviation $\hat{\sigma}_d$ of the differences is 6 166 633 ns, resulting in the following t-value:

$$t = \sqrt{50} \cdot \frac{20884576\mathrm{ns}}{6166633\mathrm{ns}} = 23.9476.$$

The critical t-value for 49 degrees of freedom and a probability level of 1% is $t(0.99; 49) = 2.404892$. As $t > t(0.99; 49)$, we can reject the null hypothesis and conclude that the proposed ensemble technique is superior to SVR.

Compared to always choosing SVR for all test sets, the selector improves the overall MAE by 30%. Additionally, it is interesting to note that the best-suited approach changes with the number of available measurement points. For two of the datasets, even four different approaches performed best depending on the size of the training set. This underlines the importance of using a meta-selector.

Our evaluation shows that for the given samples, the average prediction error is significantly lower when using the meta-selector instead of always applying SVR, the best individual machine learning approach. We infer that our meta-selector is an appropriate approach for prediction technique selection. Overall, the experiment shows that it is feasible and beneficial to adapt prediction techniques to observable dataset characteristics without applying domain knowledge or manual effort. This answers **EQ 12.4** (*"How does the created meta-selector influence the characterization accuracy?"*).

12.5 Summary

In this chapter, we evaluate the performance of *DepIC*, our approach for (i) the automated detection of dependencies from monitoring data, and (ii) the selection of characterization techniques for identified dependencies presented in Chapter 7 on page 109. In order to do so, we pose ourselves four Evaluation Questions (EQs).

We apply three different approaches for feature selection and show that a filter-based approach outperforms the competing techniques in terms of solution quality and run time to answer **EQ 12.1** (*"Which feature selection technique is suited best for the identification of parametric dependencies?"*). Furthermore, we analyze different post-processing steps intended to reduce the number of irrelevant dependencies on a microservice reference application. The post-processing steps eliminate over 90% of the unwanted dependencies and increase the precision for 11 correctly identified dependencies to 91.7%. This answers our second question: **EQ 12.2** (*"What is the effect of the proposed result filtering approach on the quality of the identified dependencies?"*). **EQ 12.3** (*"What is the impact of increased system utilization on the quality of the identified dependencies?"*) aims at uncovering the limitations of *DepIC* by conducting additional experiments with different load levels. We observe that although the precision of *DepIC* suffers from increasing load levels, it still correctly identifies all dependencies under high utilization. Finally, we investigate the performance of the created meta-selector for dependency characterization. It reduces the prediction error by 30% compared to using the best individual approach, answering **EQ 12.4** (*"How does the created meta-selector influence the characterization accuracy?"*). We now discuss the threats to the validity of the presented evaluation as well as the limitations and the assumption of the presented approach.

12.5.1 Threats to Validity

Although the presented evaluations were conducted with great care, there are still some remaining validity threats to be discussed.

12.5.1.1 Internal Validity

We evaluate the capability of *DepIC* to accurately identify the existence of parametric dependencies. However, we do not investigate how much the prediction accuracy of a performance model improves after including the identified parametric dependencies. The reasoning behind this is that the prediction accuracy of a performance model depends on many different factors, including the applied modeling formalism, model solver, the system under consideration, and the granularity of the applied model.As *DepIC* is generally applicable, we did not restrict ourselves to any existing modeling formalism. We, therefore, minimized the impact of any side effects in our evaluation and decided to evaluate our approach in isolation. However, this prevents an evaluation of the expressiveness of the detected dependencies since this would require applying *DepIC* to a specific formalism, together with a technique for characterizing the found dependencies.

Additionally, we focused on executing the t-test when analyzing the effectiveness of the meta-selector in Section 12.4. Although other metrics like the hit rate (share of samples, where the optimal approach was selected) or the overall accuracy achievement might give us more information about the actual performance, these values are very dependent on the chosen evaluation datasets themselves. As the datasets can not be viewed as representative of the real-world measurements (see Section 12.5.1.2), we intentionally avoided the use of those metrics as they might suggest false confidence. Instead, our goal was to verify the superiority of the meta-selector over a single approach application, which could be convincingly shown using the applied t-test experiment in Section 12.4.

12.5.1.2 External Validity

From our experiments, we can not conclusively show that the thresholds determined in Section 12.1.2 are transferable to other systems or if the threshold tuning is an elemental step of the algorithm calibration for each system. Exploring this would require a case study spanning a large number of representative systems and workloads, including a variety of approaches for hyperparameter tuning.

In Section 12.3, we explore the impact of system utilization on the accuracy of our dependency identification approach. We showed that the results of *DepIC* are usually stable, as long as the relative proportions between the measured or estimated resource demands stay the same. As our experiments included constant load scenarios, utilization levels did not have a strong impact on the approach. However, as varying load levels might impact the measured relative proportions, future research will investigate these effects and propose possible countermeasures. Therefore, the results from this experiment are not necessarily transferable to all other software systems.

Although we put in our best efforts to select and create a variety of representative datasets (see Section 7.2.1 on page 127), the number of used datasets is still limited. Our collection was sufficient to support our assumptions and evaluate that the no-free-lunch theorem [Wol96; WM97] holds in the domain of dependency characterization. However, due to the sheer number of possible measurement experiments and resulting datasets, the constructed meta-classifier must not be seen as a final rule set but rather as a proof-of-concept implementation of the respective contributions.

12.5.2 Limitations and Assumptions

In this work, we compare three different feature selection algorithms, one from each major group of feature selection algorithms [GE03]. This assumes that the performance of the chosen approach is representative of the whole group. However, this comparison can be extended with further feature selection techniques as surveyed by Guyon and Elisseeff [GE03].

When creating the data streams in Section 7.1.2 on page 116, we settled on strong simplifications for resolving indirect recursions. As our test application did not contain many indirect recursions of considerable size, this did not cause any problems in the presented study. However, we acknowledge that this is a sub-optimal solution for some applications since an indirect recursion might span over many methods, which lead to the contraction of them into one single vertex.

We addressed the problem of monitoring resource demands shortly in Section 7.1.1.2 on page 115 and evaluated the impact of increasing loads in Section 12.3 of the evaluation. However, the chosen resource demand estimation is still a strong simplification and could be improved by integrating it with more elaborated techniques as presented in Chapter 6 on page 91.

The current approach relies on a certain amount of monitoring data to be available. Next to the requirements discussed in Section 7.1.1.1 on page 114, the accuracy will likely be sub-optimal until a certain amount of monitoring data has been gathered. This is due to the fact that the accuracy of machine learning algorithms is usually strongly dependent on the available training set size. However, recall that the meta-selector is able to partially cope with this issue by selecting the respective approach.

Lastly, we acknowledge that *DepIC* only covers dependencies between model parameters on the same call path. Parameters describing the current state of a system, for example, the number of entries in a database, can also influence the performance of a software component [HBR13] while not lying on the same call path as the resource demand they influence. Currently, our approach is not capable of detecting such stateful dependencies, as proposed by Happe et al. [HBR13]. However, the presented technique could be extended to support such dependencies by adding state variables into every feature selection task. We did not cover this in the current iteration, as this drastically increases the run time and requires manual identification of state variables by an expert.

Chapter 13

Evaluating DBMS Configuration Models

In this section, we evaluate *Baloo*, our approach for measuring and modeling the configuration space of a distributed DBMS as introduced in Chapter 8 on page 137. In the first step, we use an existing dataset to determine the robust metric. We then use a newly generated dataset to validate the chosen metric in Section 13.2). The new dataset is then used to evaluate the quality of the measurement repetition determination in Section 13.3 and the performance model construction in Section 13.4. The resulting analysis aims to answer the following EQs for the respective **Goal IV** (*"Develop a workflow for modeling configurable, cloud-based, and distributed DBMSs."*).

- **EQ 13.1**: *What metrics are suited for summarizing the measurement of one configuration?*
- **EQ 13.2**: *What is the impact of the proposed dynamic repetition determination approach on measurement cost and accuracy?*
- **EQ 13.3**: *What is the most appropriate regression modeling approach?*
- **EQ 13.4**: *What are the cost and accuracy implications of different target thresholds?*

In the following, we introduce both datasets used in this evaluation in Section 13.1. Our implementation of the *Baloo* framework, as well as all evaluation scripts and the respective data, is publicly available for repetition as a Code-Ocean capsule [Gro+21c]. As discussed, Sections 13.2 to 13.4 then analyze the presented EQs. Finally, we present an evaluation summary, as well as a discussion of the threats to validity and the limitations of the approach in Section 13.5.

13.1 Validation Datasets

For evaluating *Baloo*, we make use of two different datasets. The configuration space of the respective datasets is depicted in Table 13.1 and comprises eleven dimensions. The first dataset is an existing and openly available dataset [SD19], comprising 102 configuration points for Apache Cassandra and Couchbase. The Apache Cassandra configuration points serve as a reference dataset for determining the robust metric.

Table 13.1: Configuration space of the two evaluation datasets.

Parameter	Seybold et al. [SD19]	*Baloo* dataset [Sey+20]
Infrastructure	public Amazon EC2, private OpenStack	private OpenStack
VM type	small – t2.medium	tiny – small – large
DBMS	Apache Cassandra	Apache Cassandra
Cluster size	3 – 5 – 7 – 9	3 – 5 – 7 – 9 – 11
Client consistency	any – one – two	one – two – three
Replication factor	3	1 – 2 – 3
Benchmark	YCSB	YCSB
Workload	write-heavy	write-heavy
Records	4 000 000	4 000 000
Record size	5 KB	5 KB
Storage backend	SSD, HDD, remote	SSD

The second dataset has been created for this work and is also published on Zenodo [Sey+20]. For this evaluation, we defined seven configuration dimensions as static. We select a private OpenStack-based cloud infrastructure as this gives us control over OpenStack-specific configurations such as the overcommitting factor and VM placement. We select Apache Cassandra as representative cloud-hosted NoSQL DBMS. Finally, we use a write-heavy workload issued from the YCSB [Coo+10] and four million records with a record size of 5 KB for comparability to other work [Sey+19]. We use SSDs for storage as it is recommended for most DBMSs.

For the remaining four dimensions (VM type, client consistency, replication factor, and cluster size), we use three different configuration options for VM type, client consistency, and replication factor. For the cluster size, we use five different options. Due to the fact that the client consistency and the replication factor can not be chosen fully independently, this yields a total of 90 different configuration points. The dataset contains ten measurement repetitions for each configuration point and a time series of performance metrics, system metrics, and additional metadata for each measurement run, summarized by Table 13.2. More details on the datasets can be found in the respective publications [SD19; Sey+20].

Table 13.2: Summary of the gathered dataset and the contained monitoring.

Data	Amount/Description
Configuration points	90
Repetition per configuration	10
Total data points	900
Performance metrics	Throughput, latency
System metrics	CPU, RAM, disk, and network usage
Metadata	Execution, workload, and deployment logs

13.2 Robust Metric Selection

In this section, we evaluate the different statistical measures that are robust metric candidates. We start with a publicly available dataset [SD19] and then evaluate the transferability by comparing the results with our dataset. Table 13.3 compares the average CV score of the different robust metrics. The lower the score, the less variation and hence the better the metric fits our needs. As listed in Section 8.2.1 on page 142, we include the mean and the median, as well as different variants of percentiles, the trimmed mean, and the winsorized mean, the Trimean [Tuk77], and the Hodges-Lehman estimator [Leh06] in this evaluation [DD11].

Although the two datasets are different with regards to the used cloud infrastructure, the VM-sizes, and the number of measurement repetitions, we observe that the performance of the different metrics is very comparable for both throughput and latency. For throughput, the 95$^{\text{th}}$ percentile achieves the best score across both datasets. When analyzing latency, we observe that both datasets have one minimum for the trimmed mean of 30%. The first dataset has an additional minimum using the 30%-winsorized metric, while the second dataset performs slightly better using the median. However, both the 30%-winsorized and the median are in the third place for the respective other dataset.

To summarize, we can say that the results from the publicly available dataset transfer very well to our own dataset. Hence, we conclude that the 95th percentile for throughput, as well as the trimmed mean or the winsorized mean for latency, are viable, robust metrics that can be applied for comparing DBMS cloud performance. As DBMS are usually optimized for throughput, we concentrate on throughput for the remainder of this work. Based on our insights, we use the 95th percentile as a robust metric. With this, we can now confidently answer **EQ 13.1** (*"What metrics are suited for summarizing the measurement of one configuration?"*).

Table 13.3: Comparing the average CV for each robustness metric for the external dataset with our own dataset.

Metric	Throughput CV		Latency CV	
	External	Own	External	Own
Mean	0.039	0.062	0.164	0.201
Median	0.044	0.063	0.036	**0.051**
95th percentile	**0.028**	**0.039**	0.138	0.174
90th percentile	0.029	0.042	0.079	0.107
80th percentile	0.032	0.047	0.052	0.076
70th percentile	0.036	0.052	0.042	0.062
Trimmed (5%) mean	0.039	0.062	0.046	0.076
Trimmed (10%) mean	0.039	0.062	0.038	0.056
Trimmed (20%) mean	0.041	0.062	**0.035**	0.052
Trimmed (30%) mean	0.042	0.062	**0.035**	**0.051**
Winsorized (5%) mean	0.039	0.062	0.057	0.090
Winsorized (10%) mean	0.039	0.062	0.043	0.059
Winsorized (20%) mean	0.039	0.063	0.036	0.055
Winsorized (30%) mean	0.041	0.063	**0.035**	0.052
Trimean	0.042	0.062	0.036	0.053
Hodges-Lehmann	0.039	0.063	0.036	0.054

13.3 Measurement Repetition Determination

In this section, we evaluate the measurement repetition determination discussed in Section 8.2.3 on page 143 in order to answer **EQ 13.2** (*"What is the impact of the proposed dynamic repetition determination approach on measurement cost and accuracy?"*). The core idea of the presented algorithm is to determine the least number of measurement repetitions that still approximate the true mean of the underlying distribution, that is, the target value of interest. We do so by comparing estimations obtained by *Baloo* with the median of all ten measurement repetitions from the evaluation dataset.

As the true mean of the underlying distribution is unknown, we make the assumption that the median of all ten measurements approximates the true mean of the underlying distribution and serves as a gold standard in this evaluation. We compare *Baloo* with different static methods of always measuring one, two, three, or five repetitions and then returning the median of that value set. As *Baloo* uses probabilistic elements and the selection of the next measurement point is non-deterministic as well and strongly influences the obtained results, we repeat the evaluation 100 times.

Table 13.4: Comparing different baseline approaches with *Baloo* for average accuracy and measurement repetitions.

Approach	MAPE	RMSE	Average # points
Baloo	1.42%	251.8	2.59
1-point	2.75%	538.4	1.00
2-point	2.19%	403.3	2.00
3-point	1.54%	315.4	3.00
5-point	0.76%	148.1	5.00
10-point	0.00%	0.0	10.00

Table 13.4 shows the MAPE and the RMSE of *Baloo* and the four baselines together with the average measurement repetitions required. The reported results are averaged over the 100 repetitions of the evaluation. We observe that the error generally decreases as we increase the number of measurement points. This is expected as more measurement points reduce the impact of random measurement noise. It is also observable that the baseline approaches have a deterministic number of measurement points, as is expected due to the nature of the baseline. *Baloo* outperforms 1-point, 2-point, and even 3-point in average MAPE and RMSE while requiring only 2.59 measurement points on average. This shows that two measurements are sufficient to accurately describe a performance measurement in most cases.

Nevertheless, sometimes more measurements are required. This insight can be supported by comparing the reported MAPEs with the given RMSEs. For MAPE, our approach is only slightly better than 3-point (roughly 8% decrease). This difference is larger when analyzing the RMSE (roughly 20% decrease).

Since the RMSE puts a stronger focus on outliers, we can conclude that our approach is able to correctly determine critical measurement points while keeping the requested measurement to the minimum amount of two measurements when not required. The 5-point baseline and 10-point gold standard consistently achieve lower errors than our proposed approach. This is expected, as they also conduct significantly more measurements. If higher accuracy is required, our approach could be tuned accordingly. This answers **EQ 13.2**.

To summarize, our approach successfully handles the trade-off between required measurement repetitions and target accuracy. It is worth noting that in this experiment, the average number of required measurement points is comparatively low, as all experiments were executed on a private cloud with a relatively low load (see Section 13.1). Therefore, the number of required measurement repetitions, as well as the gain achieved by our approach, might be even higher for other environments [LC16; KKR14].

13.4 Performance Model Construction

In this section, we evaluate the performance model construction techniques and the corresponding performance models. As our approach works with any regression technique, our analysis compares the performance of different machine learning algorithms. For this, we analyze the required amount of measurement configuration points together with the achieved target accuracy on the remaining validation set. We include models created by Linear Regression (LR), Ridge Regression (Ridge), Elastic Net Regression (ElasticNet), Bayesian Ridge Regression (BRR), Huber Regression (HR), Gradient Boosted Decision Trees (GBDT), Random Forest (RF), and SVR. Additionally, to get a better comparison, we add a baseline regressor (Mean) which always predicts the mean of all seen samples.

In this experiment, we vary the target accuracy threshold t_s of the internal MAPE score s based on three-fold cross-validation between 0.1, 0.15, 0.2, 0.25, and 0.3. The maximum ratio r_{max} was set to 0.9, resulting in a cutoff at 81 (0.9 · 90 total) configuration points. For the initial configuration measurement set, we set an *init*-ratio of 0.05, resulting in at least 5 ($\lceil 0.05 \cdot 90 \rceil$) configurations for each approach. We only add one configuration per increment, that is, $ratio = 0.01$, as for our comparatively small example set, the training time was significantly lower than the measurement time and could therefore be neglected. Furthermore, this enables a better analysis of the required configuration points. All algorithm implementations are based on Scikit-learn [Ped+11] and use the defined default parameterization.

The comparison is shown in Figure 13.1. The x-axis depicts the number of configuration points that were measured before the respective workflow terminated, that is, t_s was achieved, or 81 configurations were measured. The y-axis shows the MAPE on the remaining configurations that were not added to the training set. We repeat all experiments 25 times and report the average value as these processes are highly influenced by the random seed.

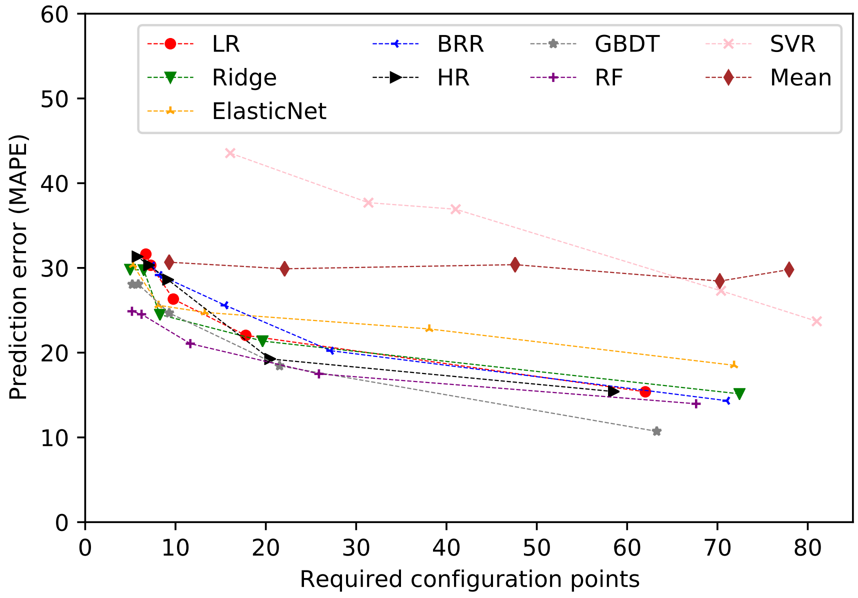

Figure 13.1: Achieved accuracy versus required configuration points for varying target accuracies.

From Figure 13.1, we observe that multiple approaches perform comparably in terms of required configuration points and the achieved prediction error. However, SVR and Mean (baseline) are not able to capture the performance structure very well and, therefore, perform poorly in terms of prediction error and required configuration points. SVR even runs out of measurement options and therefore retrieves the maximum number of configuration points for $t_s = 0.1$.

All other approaches perform similarly well, but we can still identify small differences between them. Increasing the target accuracy has the expected effect of generally reducing the prediction error while increasing the measured configuration points. While GBDT achieves the overall lowest prediction error (10.72%), RF is able to achieve slightly worse results using considerably fewer configuration points for $t_s = 0.15$ and $t_s = 0.2$. For bigger values of t_s, GBDT performs slightly better again. Using this, we can answer **EQ 13.3** (*"What is the most appropriate regression modeling approach?"*) and conclude that RF and GBDT show the best performance on our dataset.

Table 13.5: Detailed performance of all algorithms for a fixed target accuracy t_s of 0.15.

Approach	MAPE	Measurements	Configurations	Time
LR	22.04	43.56	17.78	0.43 s
Ridge	21.37	51.62	19.64	0.52 s
ElasticNet	22.80	98.60	38.10	1.04 s
BRR	20.23	71.16	27.20	0.84 s
HR	19.26	52.78	20.46	0.96 s
GBDT	18.44	55.66	21.52	1.26 s
RF	17.50	66.30	25.88	5.05 s
SVR	27.30	181.88	70.40	2.00 s
Mean	28.44	181.74	70.26	1.89 s

Table 13.5 shows more details on the performance of the individual algorithms for $t_s = 0.15$. In addition to the average achieved error (MAPE) and the required number of configurations (Configurations), we see the average number of total measurement runs conducted (Measurements) and the average time of the workflow execution excluding measurements (Time). We conclude that even unoptimized machine learning approaches are able to achieve prediction errors of around 20% on a dataset consisting of 90 configuration points, made up of 900 individual measurement series while measuring less than 25% (~ 22) of configurations and conducting around 6% (~ 54) of individual measurement runs.

Adding more measurement points increases the accuracy and reduces the error to up to 12%. However, the accuracy gain per added configuration decreases over time, which is consistent with our expectations. The additional computation effort introduced by our framework is negligible (execution times of under 1 second per training process) if we consider that one measurement run takes minutes or even hours to complete. Therefore, the achieved time savings are more than 94%, for an accuracy cost of just 20%. Even by reducing the accuracy cost to 11%, we can still achieve measurement time reductions of over 80% (162.78 measurements on average). We believe this result to be sufficient, considering that average performance in public clouds also regularly fluctuates [LC16; IYE11]. Therefore, this answers **EQ 13.4** (*"What are the cost and accuracy implications of different target thresholds?"*).

13.5 Summary

In this section, we quickly summarize the insights of the presented evaluation and discuss the most relevant threats to the validity of the results, as well as the remaining limitations of the *Baloo* approach. To evaluate our framework, we measured the distributed DBMS Apache Cassandra in our private cloud using 90 different configurations and ten repetitions each, resulting in 900 measurement runs comprising of roughly 450 measurement hours and 9 450 compute hours. We made the resulting dataset publicly available to foster future research in this area [Sey+20].

We compare different robust metrics in order to answer **EQ 13.1** (*"What metrics are suited for summarizing the measurement of one configuration?"*) and find that the 95th percentile (throughput), the trimmed mean (latency), and the winsorized mean are all viable metrics when comparing DBMS runs. In addition, we see that using the proposed dynamic repetition determination minimizes the required amounts of measurements while still maintaining acceptable measurement inaccuracies compared to static baseline approaches (**EQ 13.2**). Lastly, we analyze different performance model construction techniques (**EQ 13.3**) and compare the results of different target thresholds (**EQ 13.4**). The evaluation shows that our highly configurable approach is able to save between 80% and 94% of measurement time for a respective accuracy cost of 11% to 20% when using RF or GBDT.

13.5.1 Threats to Validity

After we summarized the result, we discuss the threats to the validity of the achieved insights in this chapter. We divide this section into internal validity and external validity.

13.5.1.1 Internal Validity

The analysis of Section 13.3 defined the 10-point approach, that is, the median of all ten measurement repetitions, as the gold standard for all measurement repetition approaches. This is a necessary assumption as the true mean of the distribution is unknown at evaluation time. However, this still introduces a threat to the validity of this analysis, as all presented results are based on this assumption. This problem could be mitigated by increasing the measurement repetitions of the gold standard (e.g., 100) in order to increase the confidence in the generated gold standard. However, this would incur massive measurement costs, as is therefore unfeasible for us in this work. Another evaluation approach is to define an artificial distribution and evaluate the approaches by randomly drawing samples from the distribution with the known mean. We purposefully discarded that strategy as this implies defining an underlying distribution, which could also bias the achieved results. Instead, we focus on real-world measurement results in this work.

When analyzing the performance of the different regression techniques for answering **EQ 13.3** (*"What is the most appropriate regression modeling approach?"*), we restrict ourselves to using the standard parameterization. We are convinced that applying parameter optimization (see Section 2.3.3.1 on page 32) or feature engineering (see Section 2.3.3.2 on page 33) methods could additionally increase the performance of all used algorithms. As our analysis presented in Section 13.4 is limited to the default parameterization, the presented ordering might substantially change when tuning the respective approaches. However, we consciously focus on simply showing the applicability of the proposed *Baloo* framework and do not claim to produce the best possible results.

13.5.1.2 External Validity

The dataset introduced in this work contains only measurements for Apache Cassandra, an open-source and distributed NoSQL DBMS. Although we believe that Apache Cassandra is a reasonable choice as it is a well-known representative of a NoSQL DBMS that puts a strong emphasis on distributed deployment, all results insights achieved in this work are limited to the underlying dataset. We purposefully choose a NoSQL DBMS in this work, as NoSQL systems traditionally put a stronger focus on distribution aspects [SD17; Sey+19], which was the focus of our analysis. However, note that both *Baloo* and the used *Mowgli* benchmarking framework work DBMS agnostic and can therefore be applied to other systems as well. However, while we are confident that the concept of *Baloo* can be transferred to other DBMSs, the achieved gain might differ, especially when considering an SQL system.

We evaluate the performance of different robust metrics in Section 13.2 to answer **EQ 13.1** (*"What metrics are suited for summarizing the measurement of one configuration?"*). For this, we compare our dataset with an additional external dataset recorded using the same benchmarking tool (*Mowgli*). Although the observed results are convincingly similar for both datasets, the presented analysis is only applied to two different datasets. As both datasets still concentrate on the Apache Cassandra DBMS, the generalizability of the results of **EQ 13.1** might be limited. Therefore, future work should continue our analysis for other available dataset traces to validate whether our claims can be generalized to more DBMS systems.

Finally, all of our measurement and prediction results are focused on the workload provided by *Mowgli* and YCSB. Therefore, we can not claim any generalizability beyond the applied workload. However, this is due to the lack of an established, real-world benchmark for NoSQL DBMS [Dom+21], which is why YCSB is the de-facto standard [Ren+17]. To that end, we published a vision for realistic and tailored DBMS benchmarking [Dom+21] that will be the target of future work.

13.5.2 Limitations and Assumptions

In this section, we discuss the main assumptions and limitations that we see with *Baloo*. One of the main properties of *Baloo* is that it relies on real benchmarking measurements. This is a conscious choice, as the theoretical and the measured performance of a cloud system might substantially differ [SD17; Wal19]. However, the downsides of the required measurement series are the increased cost implications and time investments. While the actual measurement process is automated, we still need to reserve the respective hardware resource in an environment that closely resembles the target system as well as the time for each benchmark. Hence, *Baloo* offers the configurable cost-accuracy trade-off in order to mitigate this issue.

Another consequence of the large cost associated with each measurement run is the limited size of the presented dataset. Although we varied only four different dimensions, resulting in a total of 90 different configurations, the creation of the dataset required roughly 9 450 compute hours. This is also one reason why we could not easily find comparable datasets, including systematic and repeated measurement series. Therefore, we could not evaluate the applicability of *Baloo* on large configuration spaces. As the regression algorithms used in this work only rely on black-box measurement data and can not utilize domain knowledge, *Baloo* is relatively conservative in reducing the number of required configuration points. Therefore, for huge configuration spaces, the accuracy targets of *Baloo* need to be adapted accordingly, resulting in lower prediction accuracy.

In contrast to the other approaches presented in this work, *Baloo* is not primarily intended for online use. The main reason for this is that *Baloo* needs to be able to actively trigger measurement runs, while the other approaches usually utilize passive monitoring streams from production systems. This furthermore implies that *Baloo* requires an accurate workload model for executing the benchmarking runs. As all configurations are optimized for the conducted measurements, the configuration of the benchmark workload has a huge impact on the respective results.

The presented *Baloo* framework focuses on one-dimensional performance classification. This is sufficient, for example, if the goal is to optimize a DBMS for keeping a latency SLA while minimizing the required cost. However, most DBMS can be measured with more than one performance metric. For example, if both throughput and latency are relevant for the operator, we can utilize multi-target prediction to model multiple relevant performance metrics for the given configuration space at the same time. Therefore, extending the *Baloo* framework for multi-target modeling can be a future research direction.

Finally, Section 13.2 is focused on finding the metric that is suited best to aggregate the complex measurement runs into single measurement values. However, the aggregation of a multivariate time series into a single value necessarily discards useful information about the properties of the underlying benchmark run. This information can be used, for example, to discard measurement runs if anomalies or other failures occur during the benchmark execution. However, as the aggregation of this information into one or several meaningful indices is not trivial, we leave this topic for future work.

Part IV

Conclusion

Chapter 14
Summary

Modern applications are often designed and developed as cloud-native microservice applications. Microservice architectures offer massive benefits for development and maintenance. The paradigm is typically paired with a DevOps culture to further speed up application delivery. In addition, these applications are designed to run natively on public cloud platforms to tackle the unpredictable usage growth and the increased performance and availability requirements of users. Such applications are often referred to as cloud-native applications.

The use of continuously changing and cloud-hosted microservice applications introduces a number of unique challenges for modeling the performance of modern applications. These challenges violate common assumptions of the state of the art and complicate the automated derivation of performance models from monitoring data. Nonetheless, such performance models are an essential tool to manage cloud applications in order to conserve energy and minimize operating costs while still maintaining user experience. Hence, the emergence of modern software paradigms necessitates new research towards the automated learning of such performance models.

In this thesis, we present five techniques for automated learning of performance models for cloud-native software systems. We achieve this by combining machine learning with traditional performance modeling techniques. Depending on the cloud computing model, privacy agreements, or monitoring capabilities of each platform, we identify four main application scenarios where performance modeling, prediction, and optimization techniques can provide great benefits and arrange our contributions accordingly. In this chapter, we summarize the individual contributions and revisit the respective goals and RQs targeted in this work.

Contribution 1: *Monitorless*: Detection of resource saturation using platform-level metrics As our first contribution, Chapter 4 presents *Monitorless*, our approach for utilizing machine learning techniques to bridge the gap between platform-level monitoring and application-specific KPIs. We show that a properly trained machine learning model can utilize platform-level data to inferring the KPI degradation based on resource saturation of arbitrary applications in order to answer **RQ I.1** (*"How can platform-level measurements be utilized to detect resource saturation?"*). Furthermore, answer **RQ I.2** (*"How can we generalize the results to create a generic and holistic prediction model?"*) by creating a diverse test set that covers different scenarios and shows that this model is sufficient for predicting the performance of heterogenous microservice applications. This way, *Monitorless* represents our solution for **Goal I** (*"Design an application-agnostic approach for the detection of resource saturation based on platform-level monitoring data."*).

Chapter 9 evaluates the presented solution using a multi-tier web service and two representative microservice applications that are not included in the training phase (**RQ I.2**). Results show that *Monitorless* infers KPI degradation with an accuracy of 97%. Furthermore, we use the inferred KPI degradations to show that *Monitorless* achieves comparable performance than typical autoscaling solutions, even if these utilize optimized thresholds (**RQ I.1**). Using *Monitorless*, engineers can rely solely on platform-level metrics to fulfill resource-based SLAs and, therefore, remove the need for any application-level monitoring (**Goal I**). In addition, as the model is application-agnostic, there is no need to retrain the provided model for different target applications.

Contribution 2: *SuanMing*: Prediction of performance degradations using application-level tracing Second, we present *SuanMing* in Chapter 5. The goal of our second contribution is to utilize reactive APM tooling to predict and avoid future performance degradations of microservice applications. This is reflected in the two key RQs of Chapter 5. *SuanMing* shows that we can utilize the tracing data to derive request propagation and performance models for the individual components to answer **RQ II.1** (*"How can tracing data be utilized to predict the future performance of a system?"*). Furthermore, we address **RQ II.2** (*"How can we pinpoint the root cause service of a performance problem?"*) by proposing an algorithm that uses both models to filter the root causes for a performance problem. Using these techniques, we achieve **Goal II** (*"Develop an approach for the prediction of performance degradation using application-level tracing."*) by introducing the *SuanMing* framework.

Chapter 10 analyses the performance of *SuanMing* on two realistic microservice applications. We show that our approach predicts (**RQ II.1**) and pinpoints (**RQ II.2**) performance degradations with an accuracy of over 90% and that the overhead of applying our approach is feasible for online environments. One main benefit of *SuanMing* is that operators can add our approach into their already configured monitoring stack to augment the reactive capabilities of their APM tools with a predictive and proactive component that is able to determine and avoid performance degradations before they actually occur (**Goal II**). *SuanMing* requires no additional application data for delivering predictions, as the required information is automatically deducted from the APM data.

Contribution 3: *SARDE*: Continuous and self-adaptive resource demand estimation Our third main contribution targets the area of resource demand estimation. In Chapter 6, we focus on answering **RQ III.1** (*"How can we combine different estimation approaches to efficiently produce continuous resource demand estimations?"*). In order to do so, we present *SARDE*, a framework for self-adaptive resource demand estimation in continuous environments. *SARDE* dynamically and continuously optimizes, selects, and executes an ensemble of resource demand estimation approaches to adapt to changes in the environment. This creates an autonomous and unsupervised ensemble estimation technique, providing reliable resource demand estimations in dynamic environments. Therefore, the approach presented in Chapter 6 represents our solution for answering **RQ III.1**.

We evaluate *SARDE* using two datasets. One set of micro-benchmarks reflecting a multitude of different possible system environments and one dataset consisting of a continuously running application in a changing environment. Our results show that by continuously applying online optimization, selection, and estimation, *SARDE* efficiently adapts to the online trace and reduces the model error using the resulting ensemble technique (**RQ III.1**). In total, *SARDE* achieves an average resource demand estimation error of 15.96%. *SARDE* works as a fully autonomous, situation-aware, and self-adaptive ensemble resource demand estimation approach. Therefore, *SARDE* continuously updates or improves the performance model of a running application and brings us closer to self-aware performance models (**Goal III**).

Contribution 4: *DepIC*: Learning parametric dependencies from monitoring data Our fourth contribution addresses **RQ III.2** (*"How can the impact of parameters on resource demands be identified and characterized?"*). Chapter 7 proposes *DepIC*, our approach to derive parametric dependencies from runtime monitoring data. The *DepIC* approach for Dependency Identification and Characterization utilizes feature selection techniques for identification and an ensemble regression approach for the characterization of dependencies (**RQ III.2**).

Our evaluation shows that *DepIC* achieves an identification precision of 91.7% using the investigated microservice application. Furthermore, Chapter 12 shows that the proposed meta-selector for characterization reduces the prediction error compared to the best individual machine learning approach by 30%. *DepIC* enables the automated extraction of more detailed models while requiring only runtime monitoring data of the managed application and otherwise treats the application as a black-box (**RQ III.2**). Taken together, *DepIC* and *SARDE* achieve **Goal III** (*"Enable the continuous estimation and improvement of performance model parameters using production monitoring data."*) by answering **RQ III.1** and **RQ III.2** and present our effort towards self-aware performance models.

Contribution 5: *Baloo*: Measuring and modeling performance configurations of distributed DBMSs Finally, Chapter 8 presents *Baloo*, the fifth major contribution of this thesis. *Baloo* is a framework for systematically measuring and modeling performance configurations of distributed DBMS in cloud environments. We solve **RQ IV.1** (*"How can the influence of performance variabilities during benchmark measurements be mitigated?"*) by introducing a dynamic measurement repetition procedure automatically analyzing the benchmarking variabilities. Furthermore, **RQ IV.2** (*"How can we analyze a configuration space that is too large to measure exhaustively?"*) is addressed by comparing different machine learning techniques for modeling unknown configurations. Based on the desired target accuracy, *Baloo* dynamically estimates and executes the required number of measurement configurations, as well as measurement repetitions per configuration, and automatically creates a performance model over the whole available configuration space. Therefore, Chapter 8 fulfills **Goal IV** (*"Develop a workflow for modeling configurable, cloud-based, and distributed DBMSs."*).

Chapter 13 shows the evaluation of *Baloo* based on a dataset of 900 configuration measurements. The evaluation shows that the highly configurable approach saves up to 94% of measurement effort for an accuracy cost of 20% or 80% for a cost of 11%. Furthermore, we observe that the measurement repetition algorithm dynamically adapts the number of required repetitions (**RQ IV.1**). Through modeling the configuration space, *Baloo* can quickly extrapolate expected performance results for a given configuration without actually measuring it (**RQ IV.2**). Besides finding the most performant DBMS configuration, it gives a better understanding of the entire configuration space, providing valuable insights for operators and architects when trading performance against other non-functional aspects such as security, reliability, and costs.

Summary In this thesis, we present five core contributions towards automated performance modeling of cloud-native microservice applications. Each contribution is tailored to one of the four specific target scenarios introduced in Chapter 1, depending on the cloud computing model, privacy agreements, or monitoring capabilities of the respective platform. Hence, the contributions are specialized to provide the best modeling, prediction, and optimization results for the respective scenarios. Nevertheless, the goal of all proposed approaches is to provide automated performance modeling techniques that optimize cloud operations for modern microservice applications. Together, they present a holistic solution to a complex problem and significantly advance the research field.

Chapter 15 Open Challenges and Future Work

To conclude, we collect a list of remaining challenges that we did not target in this thesis and give directions or targets for future work in this research area.

Tighter collaboration of individual contributions We already described how the continuous learning capabilities of *SARDE* and *DepIC* for resource demand estimation and dependency learning integrate to realize our vision for self-aware performance models [GEK18]. This also seems natural as both contributions are aimed at the same target scenario and utilize similar monitoring data. As *DepIC* relies on resource demand estimates for identifying and characterizing the dependencies, both tools can be integrated to work together. However, the *SuanMing* framework would also massively benefit if request parameters and parametric dependencies could be included in its performance prediction models. Similarly, the capability of *Baloo* to predict the performance behavior of a certain configuration of a DBMS system can be helpful to determine the performance properties of microservice applications using our *Monitorless* or *SuanMing* approaches. For future work, the main obstacle is to combine the different scenario objectives (e.g., prediction goal, type and amount of available monitoring, or resource constraints) into a meaningful and holistic view. If the individual approaches continue to work as autonomous agents, we could construct a collaborative self-adaptive system with fully autonomous and altruistic agents with maximal knowledge access. These properties are chosen based on a respective taxonomy we helped to derive during this thesis [DAn+19; DAn+20].

Automatic decision framework As an alternative approach to the proposed decentralized collaboration system, we can also introduce a central and holistic learning entity. As we proposed approaches are generally independent but complementary, one target could be the development of a supporting decision framework that helps operators choosing the right approach by providing appropriate guidelines. In addition, one could provide an autonomous decision engine, automatically selecting and executing a suitable approach. This is possible, as all approaches themselves are already designed to run mostly autonomous. We have already shown the benefit of adaptive instrumentation techniques for performance modeling techniques [Maz+20]. Therefore, the approach could aim for dynamically switching the monitoring strategy along with the applied approach, depending on the achieved prediction accuracy. One remaining challenge is the correct configuration and parameterization of the developed approaches, as all offer an additional set of configurable parameters, tweaking the model accuracy and the computational cost. Therefore, the decision framework is also responsible for tuning and changing the respective configuration, depending on the respective scenario and the available resources. Such a meta-learning [Smi09] framework could provide a layer of meta-self-awareness [Kou+17] for the resulting system. Still, it might be able to reuse many of the techniques already introduced during this thesis [GEK18; Gro+21b; Maz+20].

Utilizing model predictions for optimization All of the modeling techniques introduced in this thesis are focused on predicting the performance of a software system. These models are a crucial part of optimizing the resource usage of server systems. However, actual system adaptations and change decisions are usually made by a higher-level entity, such as an autoscaler [Bau+18; Her18] or an optimization engine [Her+20b; Hub+17; Kou+16; Ost+14]. Therefore, next to the core contributions presented in this thesis, we worked towards validating the impact of resource demand estimation for autoscaling [Bau+18] as well as automated network analysis [Her+20c; Her+20a], benchmarking [Her+21], and optimization [Her+20b]. As the scope of this project was to focus on the automated construction of usable performance models, future work might focus on utilizing the resulting predictions in order to optimize system structure and resource usage.

Developing holistic prediction models One central aspect of the discussed optimization engine is various types of performance models. However, optimizing the performance of a cloud system is sometimes not the only interesting aspect. For example, the availability or the resilience of a given application can be of interest to cloud operators and customers as well [Fer+12b]. In addition, the energy efficiency of the underlying server structure is also a major contributor to the overall energy consumption of the infrastructure [Kis19]. While the focus of this thesis was clearly on performance, we also contributed towards models for power consumption of servers [Kis+19b] as well as machine [Züf+21] or HDD [Züf+20] failure prediction. Future work could aim at combining our performance modeling techniques with these and other prediction techniques in order to generate models with a more holistic view [Ber+11; Fer+12b]. This will help to further increase the energy efficiency of cloud systems and generally improve the decision-making of data center optimization techniques.

Deriving confidence values for model predictions All of the developed models represent an abstracted view of reality. Naturally, all modeling approaches introduce a certain amount of prediction error and uncertainty [Wal19]. Therefore, many users are reluctant to trust the predictions of performance models blindly. We validated these assumptions by conducting one academic overview [Gro+20a] and one industrial survey [Bez+19] during our work. Next to providing human-readable explanations, one idea for increasing the trust and the confidence of operators in the respective approaches is to define and calculate confidence values for all models and their resulting predictions. If a reliable confidence measure for a prediction is given, operators could define thresholds until an optimization engine can autonomously execute its decisions on the real system. Therefore, reliable confidences can also be seen as an accelerator for the autonomy of performance models and the respective cloud platforms [Fer+12b].

Support for emerging technologies Many of the techniques proposed in this thesis are specifically designed for and focused on microservice applications. However, as new paradigms such as serverless computing and FaaS are emerging [Eyk+18a; Eis+21b; Eis+21c], future work can investigate the degree to which the introduced approaches can be transferred into the new domain and what assumptions might need to be revisited if any. Furthermore, new trends might open new cloud scenarios or render some of the assumptions we defined in Chapter 1 infeasible. For that reason, we already spent significant effort on analyzing serverless use cases [Eis+21b; Eis+20c; Eis+21c], platforms [Eyk+19], and application performance [Eyk+18a; Eis+20b; Eis+21a] in parallel to the work presented in this thesis. While we are convinced that many of the proposed techniques can be transferred to other application areas, future work will tell if the results achieved in this thesis hold up in other domains as well.

List of Figures

List of Tables

List of Algorithms

Acronyms

AdaBoost Adaptive Boosting. 27, 68–70, 193, 194

API Application Programming Interface. 18, 49, 155, 190

APM Application Performance Monitoring. 9, 19, 21, 22, 73, 74, 188, 244, 245

AUC Area Under Curve. 32, 213, 214

AWS Amazon Web Services. 1, 2, 37

BRR Bayesian Ridge Regression. 25, 26, 176, 179, 180, 182, 234, 236

CART Classification and Regression Trees. 26, 42, 49, 50, 68, 132–134, 136, 193, 194

CI/CD Continuous Integration and Continuous Deployment. 3, 5, 16, 18, 111, 210

CPU Central Processing Unit. 20, 21, 23, 38, 53, 55, 59, 61–63, 65–67, 70, 72, 91, 94, 97, 98, 103, 157–160, 167, 170, 174, 191, 207, 212, 220, 231

CV Coefficient of Variation. 134–136, 144, 146, 231

DBMS Database Management System. v, 5, 8, 10, 11, 35, 48–50, 139–142, 144, 145, 148, 149, 229, 230, 232, 237–240, 246, 249

DML Descartes Modeling Language. 23, 111

ElasticNet Elastic Net Regression. 25, 234, 236

EQ Evaluation Question. 153, 173, 185, 189, 211, 224, 229

FaaS Function-as-a-Service. 18, 19, 251

FN False Negative. 30, 31, 156, 158, 161, 163, 165, 166, 170, 178, 179, 214, 215

FP False Positive. 30–32, 156, 158, 161, 163, 165, 166, 178–180, 214, 215

GBDT Gradient Boosted Decision Trees. 27, 234–237

GP Genetic Programming. 42, 43, 50, 111

HDD Hard Disk Drive. 23, 91, 230, 251

HR Huber Regression. 25, 131, 133, 135, 136, 234, 236

HTTP Hypertext Transfer Protocol. 15, 61

I/O Input-Output. 20, 55, 62, 63, 65

KF Kalman Filter. 44, 45, 198–200

KKF Kumar Kalman Filter. 45, 94, 197–200, 202

kNN k-Nearest Neighbors. 25, 131, 133, 135, 136, 176, 178–180, 182

KPI Key Performance Indicator. 8, 37, 38, 53–55, 57, 58, 72, 157, 161, 169, 244

LAD Least Absolute Deviation. 25, 29, 44

LibReDE Library for Resource Demand Estimation. 45, 94, 191, 209

LIMBO Load Intensity Modeling Tool. 61, 62

Logit Logistic Regression. 25, 68–70, 193, 194

LQN Layered Queueing Network. 22, 23

LR Linear Regression. 25, 26, 42, 44, 50, 131–133, 136, 214, 215, 234, 236

LSQ Least Squares. 25, 30, 44, 45, 94

M5 M5 model tree. 26, 27, 42, 50, 112, 122, 132, 133, 135, 214, 215

MAE Mean Absolute Error. 29, 132–134, 178–180, 186, 222–224

MAPE Mean Absolute Percentage Error. 29, 233–236

MARS Multivariate Adaptive Regression Splines. 42, 43, 50

MSE Mean Squared Error. 29, 30, 186

NN Neural Network. 27, 28, 45, 68–70, 131, 133, 135, 136, 193–195

NoSQL Not only SQL. 48, 61, 230, 238, 239

OS Operating System. 17, 20, 55, 159, 160, 165

PCA Principle Component Analysis. 33, 34, 66, 67

PCM Palladio Component Model. 23, 42, 110, 111

PCP Performance Co-Pilot. 22, 60, 62, 64, 70, 158–160

QN Queueing Network. 22, 23

QPN Queueing Petri Net. 22, 23

RAM Random Access Memory. 20, 53, 55, 62, 63, 65, 67, 70, 130, 157, 159, 174, 191, 212, 231

REST Representational State Transfer. 15, 75, 155, 190, 191, 217

RF Random Forest. 27, 65, 67–70, 72, 112, 123, 132, 133, 135, 156, 170, 176, 178–183, 185, 193–195, 234, 236, 237

Ridge Ridge Regression. 25, 26, 234, 236

RMSE Root Mean Squared Error. 30, 122, 186, 233, 234

ROC Receiver Operating Characteristic. 32, 213, 214

RQ Research Question. 6–8, 72, 243, 244

RR Response time Regression. 44, 45, 94, 197, 198, 202

RTA Response Time Approximation. 43, 45, 94, 115, 197, 198, 202, 204

S3 Stepwise Sampling Search. 47, 100, 193

SAE Sum of Absolute Errors. 29, 30

SDL Service Demand Law. 43, 45, 94, 95, 197, 198, 202

SLA Service Level Agreement. 2, 37, 188, 240, 244

SLO Service Level Objective. 35–38, 163, 164

SPEC RG Standard Performance Evaluation Corporation Research Group. 94, 154, 191

SQL Structured Query Language. 238

SSD Solid State Drive. 23, 91, 230

SSE Sum of Squared Errors. 25, 30, 131

SUS System under Study. 92, 94, 95, 97, 100, 102

SVC Support Vector Classification. 26, 68–70

SVM Support Vector Machine. 26, 45, 193, 194

SVR Support Vector Regression. 26, 131, 133, 135, 136, 176, 179, 180, 182, 222, 224, 234–236

TN True Negative. 30–32, 156, 158, 161, 163, 166, 178, 179, 181, 214, 215

TP True Positive. 30, 31, 156, 158, 161, 163, 166, 178, 179, 186, 214, 215

UR Utilization Regression. 44, 45, 94, 95, 197, 198, 202

USE Utilization, Saturation, Error. 54, 60

VM Virtual Machine. 17, 21, 37, 38, 174, 191, 230, 231

WEKA Waikato Environment for Knowledge Analysis. 132, 134, 212

WKF Wang Kalman Filter. 45, 94, 197–200, 202

XGBoost eXtreme Gradient Boosting. 27, 68–70

YCSB Yahoo! Cloud Serving Benchmark. 61, 62, 230, 239

Bibliography

[Aba+20] Daniel Abadi et al. “The Seattle Report on Database Research”. In: *ACM SIGMOD Record* 48.4 (2020), pp. 44–53. doi: `10.1145/3385658.3385668` (see page 137).

[Aba+16] Martin Abadi et al. “TensorFlow: A system for large-scale machine learning”. In: *12th USENIX Symposium on Operating Systems Design and Implementation* (*OSDI 16*). 2016, pp. 265–283 (see page 68).

[Ack+18] Vanessa Ackermann, Johannes Grohmann, Simon Eismann, and Samuel Kounev. “Black-box Learning of Parametric Dependencies for Performance Models”. In: *Proceedings of 13th International Workshop on Models@run.time* (*MRT*)*, co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems* (*MODELS 2018*). CEUR Workshop Proceedings. 2018 (see pages 111, 131, 132).

[Ale+20] Alexander Alexandrov et al. “GluonTS: Probabilistic and Neural Time Series Modeling in Python”. In: *Journal of Machine Learning Research* 21.116 (2020), pp. 1–6 (see pages 78, 88).

[Alp20] Ethem Alpaydin. *Introduction to Machine Learning*. Adaptive Computation and Machine Learning series. MIT Press, 2020. 712 pp. (see pages 23, 24, 26–28, 30–33).

[Alt92] Naomi S. Altman. “An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression”. In: *The American Statistician* 46.3 (1992), pp. 175–185. doi: `10.2307/2685209` (see pages 24, 131).

[Ama17] Amazon Web Services. *AWS re:Invent 2017: How Amazon Scales Its Infrastructure to Handle Billions of Trans* (*ENT337*). 2017. url: `https://www.youtube.com/watch?v=bIMt0_KLmBQ` (visited on 07/08/2021) (see page 1).

[Ama21a] Amazon Web Services. *Netflix on AWS*. 2021. url: `https://aws.amazon.com/solutions/case-studies/netflix/` (visited on 07/08/2021) (see page 1).

[Ama21b] Amazon.com, Inc. *Amazon Annual Report 2020*. 2021. url: `https://d18rn0p25nwr6d.cloudfront.net/CIK-0001018724/336d8745-ea82-40a5-9acc-1a89df23d0f3.pdf` (visited on 07/08/2021) (see page 1).

[AE15] Anders Andrae and Tomas Edler. “On Global Electricity Usage of Communication Technology: Trends to 2030”. In: *Challenges* 6.1 (2015), pp. 117–157. doi: `10.3390/challe6010117` (see page 2).

[Arm+06] Warren Armstrong, Peter Christen, Eric McCreath, and Alistair P. Rendell. "Dynamic Algorithm Selection Using Reinforcement Learning". In: *2006 International Workshop on Integrating AI and Data Mining*. IEEE, 2006, pp. 18–25. DOI: `10.1109/AIDM.2006.4` (see page 47).

[Aro20] Rohit Arora. *Which Companies Did Well During The Coronavirus Pandemic?* 2020. URL: `https://www.forbes.com/sites/rohitarora/2020/06/30/which-companies-did-well-during-the-coronavirus-pandemic/` (visited on 07/08/2021) (see page 1).

[BHJ16] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture". In: *IEEE Software* 33.3 (2016), pp. 42–52. DOI: `10.1109/ms.2016.64` (see pages 3, 16–18).

[Bau21] André Bauer. "Automated Hybrid Time Series Forecasting: Design, Benchmarking, and Use Cases". en. PhD thesis. Universität Würzburg, 2021. DOI: `10.25972/OPUS-22025` (see page 78).

[Bau+18] André Bauer, Johannes Grohmann, Nikolas Herbst, and Samuel Kounev. "On the Value of Service Demand Estimation for Auto-scaling". In: *Lecture Notes in Computer Science*. Vol. 10740. Lecture Notes in Computer Science. Springer International Publishing, 2018, pp. 142–156. DOI: `10.1007/978-3-319-74947-1_10` (see pages 23, 91, 93, 96, 248).

[Bau+19] André Bauer, Nikolas Herbst, Simon Spinner, Ahmed Ali-Eldin, and Samuel Kounev. "Chameleon: A Hybrid, Proactive Auto-Scaling Mechanism on a Level-Playing Field". In: *IEEE Transactions on Parallel and Distributed Systems* 30.4 (2019), pp. 800–813. DOI: `10.1109/TPDS.2018.2870389` (see page 96).

[Bau+21] André Bauer, Marwin Züfle, Simon Eismann, Johannes Grohmann, Nikolas Herbst, and Samuel Kounev. "Libra: A Benchmark for Time Series Forecasting Methods". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ACM, 2021. DOI: `10.1145/3427921.3450241` (see pages 78, 169).

[Bau+20a] André Bauer, Marwin Züfle, Johannes Grohmann, Norbert Schmitt, Nikolas Herbst, and Samuel Kounev. "An Automated Forecasting Framework based on Method Recommendation for Seasonal Time Series". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ICPE '20. ACM, 2020, pp. 48–55. DOI: `10.1145/3358960.3379123` (see pages 78, 169).

[Bau+20b] André Bauer, Marwin Züfle, Nikolas Herbst, Samuel Kounev, and Valentin Curtef. "Telescope: An Automatic Feature Extraction and Transformation Approach for Time Series Forecasting on a Level-Playing Field". In: *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1902–1905. DOI: `10.1109/icde48307.2020.00199` (see page 184).

[Bau+20c] André Bauer, Marwin Züfle, Nikolas Herbst, Albin Zehe, Andreas Hotho, and Samuel Kounev. “Time Series Forecasting for Self-Aware Systems”. In: *Proceedings of the IEEE* 108.7 (2020), pp. 1068–1093. DOI: 10.1109/JPROC.2020.2983857 (see page 78).

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. “The Palladio component model for model-driven performance prediction”. In: *Journal of Systems and Software* 82.1 (2009). Special Issue: Software Performance - Modeling and Analysis, pp. 3–22. DOI: 10.1016/j.jss.2008.03.066 (see pages 5, 23, 42, 91, 111).

[BCB14] Christoph Bergmeir, Mauro Costantini, and José M. Benítez. “On the usefulness of cross-validation for directional forecast evaluation”. In: *Computational Statistics and Data Analysis* 76 (2014). CFEnetwork: The Annals of Computational and Financial Econometrics, pp. 132–143. DOI: 10.1016/j.csda.2014.02.001 (see pages 28, 174).

[Ber+11] Massimo Bertoncini, Barbara Pernici, Ioan Salomie, and Stefan Wesner. “GAMES: Green Active Management of Energy in IT Service Centres”. In: *Lecture Notes in Business Information Processing*. Springer Berlin Heidelberg, 2011, pp. 238–252. DOI: 10.1007/978-3-642-17722-4_17 (see pages 2, 249).

[Bez+19] Cor-Paul Bezemer, Simon Eismann, Vincenzo Ferme, Johannes Grohmann, Robert Heinrich, Pooyan Jamshidi, Weiyi Shang, André van Hoorn, Mónica Villavicencio, Jürgen Walter, and Felix Willnecker. “How is Performance Addressed in DevOps?” In: *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. ICPE ’19. ACM, 2019, pp. 45–50. DOI: 10.1145/3297663.3309672 (see pages 5, 16, 23, 109, 249).

[Bia+20] Ricardo Bianchini, Marcus Fontoura, Eli Cortez, Anand Bonde, Alexandre Muzio, Ana-Maria Constantin, Thomas Moscibroda, Gabriel Magalhaes, Girish Bablani, and Mark Russinovich. “Toward ML-Centric Cloud Platforms”. In: *Communications of the ACM* 63.2 (2020), pp. 50–59. DOI: 10.1145/3364684 (see pages 4, 5, 22, 37–39, 53, 86).

[Bie+18] André Biedenkapp, Joshua Marben, Marius Lindauer, and Frank Hutter. “CAVE: Configuration Assessment, Visualization and Evaluation”. In: *Learning and Intelligent Optimization*. Ed. by Roberto Battiti, Mauro Brunato, Ilias Kotsireas, and Panos M. Pardalos. Cham: Springer International Publishing, 2018, pp. 115–130. DOI: 10.1007/978-3-030-05348-2_10 (see pages 46, 47).

[Bis+16] Bernd Bischl, Pascal Kerschke, Lars Kotthoff, Marius Lindauer, Yuri Malitsky, Alexandre Fréchette, Holger Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney, and Joaquin Vanschoren. “ASlib: A benchmark library for algorithm selection”. In: *Artificial Intelligence* 237 (2016), pp. 41–58. DOI: 10.1016/j.artint.2016.04.003 (see pages 47, 104).

[BDW16] Thomas Bittman, Philip Dawson, and Michael Warrilow. *Gartner Magic Quadrant for x86 Server Virtualization Infrastructure*. 2016. URL: `https://www.gartner.com/en/documents/3400418/magic-quadrant-for-x86-server-virtualization-infrastruct` (visited on 07/08/2021) (see page 17).

[Bod+09] Peter Bodík, Rean Griffith, Charles Sutton, Armando Fox, Michael Jordan, and David Patterson. "Statistical Machine Learning Makes Automatic Control Practical for Internet Datacenters". In: *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*. HotCloud'09. San Diego, California: USENIX Association, 2009 (see page 37).

[Bol98] Gunter Bolch. *Queueing networks and Markov chains : modeling and performance evaluation with computer science applications*. New York: Wiley, 1998 (see page 190).

[Bon+04] Egor Bondarev, Johan Muskens, Peter With, Michel Chaudron, and Johan Lukkien. "Predicting real-time properties of component assemblies: a scenario-simulation approach". In: *Proceedings. 30th Euromicro Conference, 2004.* IEEE, 2004, pp. 40–47. DOI: `10.1109/eurmic.2004.1333354` (see page 23).

[Bon+05] Egor Bondarev, Peter de With, Michel Chaudron, and Johan Muskens. "Modelling of input-parameter dependency for performance predictions of component-based embedded systems". In: *31st EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 2005, pp. 36–43. DOI: `10.1109/EUROMICRO.2005.40` (see page 109).

[BW80] Taylor L. Booth and Cheryl A. Wiecek. "Performance Abstract Data Types as a Tool in Software Performance Analysis and Design". In: *IEEE Transactions on Software Engineering* SE-6.2 (1980), pp. 138–151. DOI: `10.1109/TSE.1980.230465` (see page 109).

[BGV92] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. "A Training Algorithm for Optimal Margin Classifiers". In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*. COLT '92. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 1992, pp. 144–152. DOI: `10.1145/130385.130401` (see page 26).

[Bot19] Alexei Botchkarev. "A New Typology Design of Performance Metrics to Measure Errors in Machine Learning Regression Algorithms". In: *Interdisciplinary Journal of Information, Knowledge, and Management* 14 (2019), pp. 045–076. DOI: `10.28945/4184` (see pages 27–29).

[Bot99] Léon Bottou. "On-Line Learning and Stochastic Approximations". In: *On-Line Learning in Neural Networks*. USA: Cambridge University Press, 1999, pp. 9–42. DOI: `10.5555/304710.304720` (see page 131).

[Bre96] Leo Breiman. "Bagging predictors". In: *Machine learning* 24.2 (1996), pp. 123–140. DOI: `10.1007/BF00058655` (see pages 27, 123, 132).

[Bre01] Leo Breiman. “Random forests”. In: *Machine learning* 45.1 (2001), pp. 5–32. DOI: 10.1023/A:1010933404324 (see pages 27, 65, 68, 112, 132, 191).

[Bre+84] Leo Breiman, Jerome Friedman, Charles J. Stone, and Richard A. Olshen. *Classification And Regression Trees*. Boca Raton: Routledge, 1984. DOI: 10.1201/9781315139470 (see pages 26, 68, 132, 134, 191).

[BHK11] Fabian Brosig, Nikolaus Huber, and Samuel Kounev. “Automated Extraction of Architecture-Level Performance Models of Distributed Component-Based Systems”. In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’11. USA: IEEE Computer Society, 2011, pp. 183–192. DOI: 10.1109/ASE.2011.6100052 (see pages 5, 40, 42, 109, 110).

[BKK09] Fabian Brosig, Samuel Kounev, and Klaus Krogmann. “Automated Extraction of Palladio Component Models from Running Enterprise Java Applications”. In: *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*. VALUETOOLS ’09. Pisa, Italy: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009. DOI: 10.4108/ICST.VALUETOOLS2009.7981 (see pages 5, 40, 43–45, 91, 94, 109, 110, 115).

[BVK13] Andreas Brunnert, Christian Vögele, and Helmut Krcmar. “Automatic Performance Model Generation for Java Enterprise Edition (EE) Applications”. In: *Computer Performance Engineering*. Springer Berlin Heidelberg, 2013, pp. 74–88. DOI: 10.1007/978-3-642-40725-3_7 (see pages 5, 40, 109, 110).

[Bru+15] Andreas Brunnert et al. *Performance-oriented DevOps: A Research Agenda*. Tech. rep. SPEC-RG-2015-01. SPEC Research Group — DevOps Performance Working Group, Standard Performance Evaluation Corporation (SPEC), 2015 (see page 16).

[Bur+13] Edmund K. Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. “Hyper-heuristics: a survey of the state of the art”. In: *Journal of the Operational Research Society* 64.12 (2013), pp. 1695–1724. DOI: 10.1057/jors.2013.71 (see page 47).

[Cal+12] Radu Calinescu, Carlo Ghezzi, Marta Kwiatkowska, and Raffaela Mirandola. “Self-adaptive software needs quantitative verification at runtime”. In: *Communications of the ACM* 55.9 (2012), pp. 69–77. DOI: 10.1145/2330667.2330686 (see page 92).

[Cap+13] Pedro Capelastegui, Alvaro Navas, Francisco Huertas, Rodrigo Garcia-Carmona, and Juan Carlos Dueñas. “An Online Failure Prediction System for Private IaaS Platforms”. In: *Proceedings of the 2nd International Workshop on Dependability Issues in Cloud Computing*. DISCCO ’13. Braga, Portugal: Association for Computing Machinery, 2013. DOI: 10.1145/2506155.2506159 (see page 39).

[CCT08] Giuliano Casale, Paolo Cremonesi, and Roberto Turrin. "Robust Workload Estimation in Queueing Network Performance Models". In: *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*. IEEE Computer Society, 2008, pp. 183–187. DOI: `10.1109/PDP.2008.80` (see pages 44, 92).

[Cha07] Sung-Hyuk Cha. "Comprehensive Survey on Distance/Similarity Measures between Probability Density Functions". In: *International Journal of Mathematical Models and Methods in Applied Sciences* 1.4 (2007), pp. 300–307 (see page 28).

[CMV13] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. "A Scalable and Nearly Uniform Generator of SAT Witnesses". In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 608–623. DOI: `10.1007/978-3-642-39799-8_40` (see page 144).

[CBK09] Varun Chandola, Arindam Banerjee, and Vipin Kumar. "Anomaly Detection: A Survey". In: *ACM Computing Surveys* 41.3 (2009), pp. 1–58. DOI: `10.1145/1541880.1541882` (see page 186).

[CS14a] Girish Chandrashekar and Ferat Sahin. "A Survey on Feature Selection Methods". In: *Computers and Electrical Engineering* 40.1 (2014), pp. 16–28. DOI: `10.1016/j.compeleceng.2013.11.024` (see pages 112, 121).

[CG16] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 785–794. DOI: `10.1145/2939672.2939785` (see pages 26, 68).

[Cho+15] François Chollet et al. *Keras*. https://keras.io. 2015 (see page 68).

[CH10] Alexander Clemm and Malte Hartwig. "NETradamus: A forecasting system for system event messages". In: *IEEE/IFIP Network Operations and Management Symposium (NOMS)*. Ed. by Yoshiaki Kiriha, Lisandro Zambenedetti Granville, Deep Medhi, Toshio Tonouchi, and Myung-Sup Kim. IEEE, 2010, pp. 623–630. DOI: `10.1109/NOMS.2010.5488430` (see page 39).

[Coc16] Adrian Cockcroft. *Microservices Workshop - Craft Conference*. 2016. URL: `https://www.slideshare.net/adriancockcroft/microservices-workshop-all-topics-deck-2016` (visited on 07/08/2021) (see pages 2, 3, 15, 16).

[Coh+04] Ira Cohen, Moises Goldszmidt, Terence Kelly, Julie Symons, and Jeffrey S. Chase. "Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control". In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, p. 16 (see pages 4, 38, 66).

[Coo+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. “Benchmarking Cloud Serving Systems with YCSB”. In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC ’10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 143–154. DOI: 10.1145/1807128.1807152 (see pages 48, 61, 228).

[CV95] Corinna Cortes and Vladimir Vapnik. “Support-Vector Networks”. In: *Machine Learning* 20.3 (1995), pp. 273–297. DOI: 10.1007/BF00994018 (see pages 25, 131, 191).

[Cor+17] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. “Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP ’17. Shanghai, China: Association for Computing Machinery, 2017, pp. 153–167. DOI: 10.1145/3132747.3132772 (see pages 4, 5, 22, 37, 38, 53).

[CW00] Marc Courtois and Murray Woodside. “Using Regression Splines for Software Performance Analysis”. In: *Proceedings of the 2nd International Workshop on Software and Performance*. WOSP ’00. Ottawa, Ontario, Canada: Association for Computing Machinery, 2000, pp. 105–114. DOI: 10.1145/350391.350416 (see pages 41, 42, 109–111).

[Cox58] David R. Cox. “The Regression Analysis of Binary Sequences”. In: *Journal of the Royal Statistical Society: Series B (Methodological)* 20.2 (1958), pp. 215–232. DOI: 10.1111/j.2517-6161.1958.tb00292.x (see pages 25, 68, 191).

[Cox14] David R. Cox. *The Statistical Analysis of Series of Events*. Springer Netherlands, 2014. 296 pp. (see page 203).

[CC07] Paolo Cremonesi and Giuliano Casale. “How to Select Significant Workloads in Performance Models”. In: *International Conference Computer Measurement Group Proceedings*. 2007, pp. 183–192 (see pages 44, 92).

[CDS10] Paolo Cremonesi, Kanika Dhyani, and Andrea Sansottera. “Service Time Estimation with a Refinement Enhanced Hybrid Clustering Algorithm”. In: *Analytical and Stochastic Modeling Techniques and Applications*. Ed. by Khalid Al-Begain, Dieter Fiems, and William J. Knottenbelt. Vol. 6148. Lecture Notes in Computer Science. Springer, 2010, pp. 291–305. DOI: 10.1007/978-3-642-13568-2_21 (see page 45).

[CS12] Paolo Cremonesi and Andrea Sansottera. “Indirect Estimation of Service Demands in the Presence of Structural Changes”. In: *Ninth International Conference on Quantitative Evaluation of Systems*. IEEE, 2012, pp. 249–259. DOI: 10.1109/QEST.2012.18 (see page 45).

[CS14b] Paolo Cremonesi and Andrea Sansottera. “Indirect Estimation of Service Demands in the Presence of Structural Changes”. In: *Performance Evaluation* 73 (2014), pp. 18–40. doi: 10.1016/j.peva.2013.08.001 (see page 45).

[DAn+19] Mirko D’Angelo, Simos Gerasimou, Sona Ghahremani, Johannes Grohmann, Ingrid Nunes, Evangelos Pournaras, and Sven Tomforde. “On Learning in Collective Self-Adaptive Systems: State of Practice and a 3D Framework”. In: *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. SEAMS ’19. IEEE, 2019, pp. 13–24. doi: 10.1109/seams.2019.00012 (see pages 47, 247).

[DAn+20] Mirko D’Angelo, Sona Ghahremani, Simos Gerasimou, Johannes Grohmann, Ingrid Nunes, Sven Tomforde, and Evangelos Pournaras. “Learning to Learn in Collective Adaptive Systems: Mining Design Patterns for Data-driven Reasoning”. In: *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. IEEE, 2020, pp. 121–126. doi: 10.1109/acsos-c51401.2020.00042 (see pages 47, 247).

[DB13] Jeffrey Dean and Luiz André Barroso. “The Tail at Scale”. In: *Communications of the ACM* 56.2 (2013), pp. 74–80. doi: 10.1145/2408776.2408794 (see pages 1, 2).

[DCS17] Salvatore Dipietro, Giuliano Casale, and Giuseppe Serazzi. “A Queueing Network Model for Performance Prediction of Apache Cassandra”. In: *Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools*. VALUETOOLS’16. Taormina, Italy: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2017, pp. 186–193. doi: 10.4108/eai.25-10-2016.2266606 (see pages 5, 49).

[Dob+13] Florin Dobrian, Asad Awan, Dilip Joseph, Aditya Ganjam, Jibin Zhan, Vyas Sekar, Ion Stoica, and Hui Zhang. “Understanding the Impact of Video Quality on User Engagement”. In: *Communications of the ACM* 56.3 (2013), pp. 91–99. doi: 10.1145/2428556.2428577 (see page 1).

[DD11] Yury Dodonov and Yulia Dodonova. “Robust measures of central tendency: weighting as a possible alternative to trimming in response-time data analysis”. In: *Psikhologicheskie Issledovaniya* 5 (2011), pp. 1–11 (see pages 142, 229).

[Dom+21] Jörg Domaschka, Mark Leznik, Daniel Seybold, Simon Eismann, Johannes Grohmann, and Samuel Kounev. “Buzzy: Towards Realistic DBMS Benchmarking via Tailored, Representative, Synthetic Workloads”. In: *Companion of the ACM/SPEC International Conference on Performance Engineering*. ACM, 2021. doi: 10.1145/3447545.3451175 (see page 237).

[Dra+17] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. "Microservices: Yesterday, Today, and Tomorrow". In: *Present and Ulterior Software Engineering*. Ed. by Manuel Mazzara and Bertrand Meyer. Cham: Springer International Publishing, 2017, pp. 195–216. DOI: 10.1007/978-3-319-67425-4_12 (see pages 2, 73).

[DS98] Norman Draper and Harry Smith. *Applied Regression Analysis, Third Edition*. Wiley, 1998. DOI: 10.1002/9781118625590.ch15 (see pages 24, 25, 131, 134).

[Dru+96] Harris Drucker, Chris J. C. Burges, Linda Kaufman, Alex Smola, and Vladimir Vapnik. "Support Vector Regression Machines". In: *Proceedings of the 9th International Conference on Neural Information Processing Systems*. NIPS'96. Denver, Colorado: MIT Press, 1996, pp. 155–161 (see page 26).

[DK05] Kai-Bo Duan and S. Sathiya Keerthi. "Which Is the Best Multiclass SVM Method? An Empirical Study". In: *Multiple Classifier Systems*. Ed. by Nikunj C. Oza, Robi Polikar, Josef Kittler, and Fabio Roli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 278–285. DOI: 10.1007/11494683_28 (see page 26).

[DTB09] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. "Tuning Database Configuration Parameters with ITuned". In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 1246–1257. DOI: 10.14778/1687627.1687767 (see pages 5, 48, 49).

[Eat12] Kit Eaton. *How One Second Could Cost Amazon $1.6 Billion In Sales*. 2012. URL: https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales (visited on 07/08/2021) (see page 1).

[Ein19] Yoav Einav. *Amazon Found Every 100ms of Latency Cost them 1% in Sales*. 2019. URL: https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/ (visited on 07/08/2021) (see pages 1, 73).

[Eis+20a] Simon Eismann, Cor-Paul Bezemer, Weiyi Shang, Dušan Okanović, and André van Hoorn. "Microservices: A Performance Tester's Dream or Nightmare?" In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ICPE '20. Edmonton AB, Canada: Association for Computing Machinery, 2020, pp. 138–149. DOI: 10.1145/3358960.3379124 (see pages 3, 15–18, 73, 75, 80).

[Eis+21a] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and Samuel Kounev. "Sizeless: Predicting the Optimal Size of Serverless Functions". In: *Proceedings of the 22nd International Middleware Conference*. Middleware '21. ACM, 2021, pp. 248–259. DOI: 10.1145/3464298.3493398 (see pages 18, 249).

[Eis+20b] Simon Eismann, Johannes Grohmann, Erwin van Eyk, Nikolas Herbst, and Samuel Kounev. "Predicting the Costs of Serverless Workflows". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ICPE '20. ACM, 2020, pp. 265–276. DOI: `10.1145/3358960.3379133` (see page 249).

[Eis+19] Simon Eismann, Johannes Grohmann, Jürgen Walter, Jóakim von Kistowski, and Samuel Kounev. "Integrating Statistical Response Time Models in Architectural Performance Models". In: *2019 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2019, pp. 71–80. DOI: `10.1109/icsa.2019.00016` (see pages 22, 111).

[Eis+20c] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina Abad, and Alexandru Iosup. *A Review of Serverless Use Cases and their Characteristics*. Tech. rep. 2020 (see page 249).

[Eis+21b] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. "Serverless Applications: Why, When, and How?" In: *IEEE Software* 38.1 (2021), pp. 32–39. DOI: `10.1109/ms.2020.3023302` (see pages 18, 53, 249).

[Eis+21c] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina Abad, and Alexandru Iosup. "The State of Serverless Applications: Collection, Characterization, and Community Consensus". In: *IEEE Transactions on Software Engineering* (2021). DOI: `10.1109/TSE.2021.3113940` (see pages 18, 249).

[Eis+18] Simon Eismann, Jurgen Walter, Jóakim von Kistowski, and Samuel Kounev. "Modeling of Parametric Dependencies for Performance Prediction of Component-Based Software Systems at Run-Time". In: *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2018, pp. 135–13509. DOI: `10.1109/ICSA.2018.00023` (see pages 22, 23, 109, 111).

[Ela18] Elastisys. *Poor performance cost Swedish e-commerce millions in lost Black Friday sales*. 2018. URL: `https://elastisys.com/poor-performance-cost-swedish-e-commerce-millions-in-lost-black-friday-sales/` (visited on 07/08/2021) (see pages 1, 73).

[EMH19] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. "Neural Architecture Search: A Survey". In: *Journal of Machine Learning Research* 20.55 (2019), pp. 1–21 (see page 47).

[Eme+10] Vincent C. Emeakaroha, Ivona Brandic, Michael Maurer, and Schahram Dustdar. "Low level Metrics to High level SLAs - LoM2HiS framework: Bridging the gap between monitored metrics and SLA parameters in cloud environments". In: *Proceedings of the 2010 International Conference on High Performance Computing and Simulation, HPCS 2010, June 28 - July*

2, 2010, Caen, France. Ed. by Waleed W. Smari and John P. McIntire. IEEE, 2010, pp. 48–54. DOI: `10.1109/HPCS.2010.5547150` (see pages 4, 37).

[Eme+12] Vincent C. Emeakaroha, Marco Aurélio Stelmar Netto, Rodrigo N. Calheiros, Ivona Brandic, Rajkumar Buyya, and César A. F. De Rose. "Towards autonomic detection of SLA violations in Cloud infrastructures". In: *Future Generation Computing Systems* 28.7 (2012), pp. 1017–1029. DOI: `10.1016/j.future.2011.08.018` (see pages 4, 37).

[EFH04] Evgeni Eskenazi, Alexandre Fioukov, and Dieter Hammer. "Performance Prediction for Component Compositions". In: *Component-Based Software Engineering*. Ed. by Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt Wallnau. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 280–293. DOI: `10.1007/978-3-540-24774-6_25` (see page 22).

[EM14] John M. Ewing and Daniel A. Menascé. "A Meta-Controller Method for Improving Run-Time Self-Architecting in SOA Systems". In: *Proceedings of the 5th ACM/SPEC international conference on Performance engineering*. ICPE '14. Dublin, Ireland: Association for Computing Machinery, 2014, pp. 173–184. DOI: `10.1145/2568088.2568098` (see page 47).

[Eyk+19] Erwin van Eyk, Johannes Grohmann, Simon Eismann, André Bauer, Laurens Versluis, Lucian Toader, Norbert Schmitt, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. "The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms". In: *IEEE Internet Computing* 23.6 (2019), pp. 7–18. DOI: `10.1109/mic.2019.2952061` (see pages 18, 249).

[Eyk+18a] Erwin van Eyk, Alexandru Iosup, Cristina L. Abad, Johannes Grohmann, and Simon Eismann. "A SPEC RG Cloud Group's Vision on the Performance Challenges of FaaS Cloud Architectures". In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE '18. ACM, 2018, pp. 21–24. DOI: `10.1145/3185768.3186308` (see pages 18, 249).

[Eyk+17] Erwin van Eyk, Alexandru Iosup, Simon Seif, and Markus Thömmes. "The SPEC cloud group's research vision on FaaS and serverless architectures". In: *Proceedings of the 2nd International Workshop on Serverless Computing*. WoSC '17. Las Vegas, Nevada: Association for Computing Machinery, 2017, pp. 1–4. DOI: `10.1145/3154847.3154848` (see page 18).

[Eyk+18b] Erwin van Eyk, Lucian Toader, Sacheendra Talluri, Laurens Versluis, Alexandru Uta, and Alexandru Iosup. "Serverless is More: From PaaS to Present Cloud Computing". In: *IEEE Internet Computing* 22.5 (2018), pp. 8–17. DOI: `10.1109/mic.2018.053681358` (see page 18).

[FH12] Michael Faber and Jens Happe. "Systematic Adoption of Genetic Programming for Deriving Software Performance Curves". In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*.

ICPE '12. Boston, Massachusetts, USA: Association for Computing Machinery, 2012, pp. 33–44. DOI: `10.1145/2188286.2188295` (see page 43).

[Fan+08] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. "LIBLINEAR: A Library for Large Linear Classification". In: *Journal of Machine Learning Research* 9 (2008), pp. 1871–1874 (see page 68).

[Far+18] Victor A. E. Farias, Flávio R. C. Sousa, José Gilvan R. Maia, João Paulo P. Gomes, and Javam C. Machado. "Regression Based Performance Modeling and Provisioning for NoSQL Cloud Databases". In: *Future Generation Computer Systems* 79.P1 (2018), pp. 72–81. DOI: `10.1016/j.future.2017.08.061` (see pages 5, 49).

[Faz+16] Maria Fazio, Antonio Celesti, Rajiv Ranjan, Chang Liu, Lydia Chen, and Massimo Villari. "Open Issues in Scheduling Microservices in the Cloud". In: *IEEE Cloud Computing* 3.5 (2016), pp. 81–88. DOI: `10.1109/MCC.2016.112` (see pages 2, 15, 17, 73).

[Fer+12a] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware". In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII. London, England, UK: Association for Computing Machinery, 2012, pp. 37–48. DOI: `10.1145/2150976.2150982` (see page 61).

[FPK14] Hector Fernandez, Guillaume Pierre, and Thilo Kielmann. "Autoscaling Web Applications in Heterogeneous Cloud Infrastructures". In: *Proceedings of the 2014 IEEE International Conference on Cloud Engineering*. IC2E '14. USA: IEEE Computer Society, 2014, pp. 195–204. DOI: `10.1109/IC2E.2014.25` (see pages 37, 169).

[Fer+12b] Ana Juan Ferrer et al. "OPTIMIS: A holistic approach to cloud service provisioning". In: *Future Generation Computer Systems* 28.1 (2012), pp. 66–77. DOI: `10.1016/j.future.2011.05.022` (see pages 2, 248, 249).

[Fle20] Flexera. *Flexera 2020 State of the Cloud Report*. 2020. URL: `https://resources.flexera.com/web/pdf/report-state-of-the-cloud-2020.pdf` (visited on 07/08/2021) (see pages 1, 2, 17, 18).

[Fow15] Martin Fowler. *Microservice Trade-Offs*. 2015. URL: `https://martinfowler.com/articles/microservice-trade-offs.html` (visited on 07/08/2021) (see pages 3, 73).

[FHW16] Eibe Frank, Mark A. Hall, and Ian H. Witten. *The WEKA Workbench*. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques", Morgan Kaufmann, Fourth Edition. 2016 (see pages 132, 134, 210).

[Fra+98] Eibe Frank, Yong Wang, Stuart Inglis, Geoffrey Holmes, and Ian H. Witten. "Using Model Trees for Classification". In: *Machine Learning* 32.1 (1998), pp. 63–76. DOI: 10.1023/A:1007421302149 (see page 26).

[Fre+19a] Erik M. Fredericks, Ilias Gerostathopoulos, Christian Krupitzer, and Thomas Vogel. "Planning as Optimization: Dynamically Discovering Optimal Configurations for Runtime Situations". In: *2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE, 2019, pp. 1–10. DOI: 10.1109/saso.2019.00010 (see page 47).

[Fre+19b] Erik M. Fredericks, Christian Krupitzer, Ilias Gerostathopoulos, and Thomas Vogel. *Planning as Optimization: Online Learning of Situations and Optimal Configurations - SASO 2019 - Accompanying material*. Zenodo, 2019. DOI: 10.5281/zenodo.2584266 (see page 47).

[FS97] Yoav Freund and Robert E. Schapire. "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting". In: *Journal of Computer and System Sciences* 55.1 (1997), pp. 119–139. DOI: 10.1006/jcss.1997.1504 (see pages 26, 68).

[Fri01] Jerome H. Friedman. "Greedy function approximation: A gradient boosting machine." In: *The Annals of Statistics* 29.5 (2001). DOI: 10.1214/aos/1013203451 (see page 26).

[Ful15] Scott M. Fulton. *What Led Amazon to its Own Microservices Architecture*. 2015. URL: https://thenewstack.io/led-amazon-microservices-architecture/ (visited on 07/08/2021) (see pages 2, 15, 16).

[GS10] Matteo Gagliolo and Jürgen Schmidhuber. "Algorithm Selection as a Bandit Problem with Unbounded Losses". In: *Learning and Intelligent Optimization*. Ed. by Christian Blum and Roberto Battiti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 82–96. DOI: 10.1007/978-3-642-13800-3_7 (see page 47).

[Gaj+20] Marek Gajda, Peter Cooper, Luca Mezzalira, Richard Rodger, and Sarup Banskota. *State of Microservices 2020*. Ed. by Patryk Mamczur. 2020. URL: https://tsh.io/state-of-microservices (visited on 07/08/2021) (see pages 1–3, 16, 18, 73).

[Gal+18] Michael Galloway, Gabriel Loewen, Jeffrey Robinson, and Susan Vrbsky. "Performance of Virtual Machines Using Diskfull and Diskless Compute Nodes". In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 740–745. DOI: 10.1109/CLOUD.2018.00101 (see page 137).

[Gan+19a] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. "Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS

'19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 19–33. DOI: `10.1145/3297858.3304004` (see pages 5, 37, 39, 73).

[Gan+19b] Yu Gan et al. "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 3–18. DOI: `10.1145/3297858.3304013` (see pages 2, 3, 16–18, 73).

[Gan+14] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang. "Adaptive, Model-driven Autoscaling for Cloud Applications". In: *11th International Conference on Autonomic Computing (ICAC 14)*. Philadelphia, PA: USENIX Association, 2014, pp. 57–64 (see page 37).

[Gan+12] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. "AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers". In: *ACM Transactions on Computer Systems* 30.4 (2012). DOI: `10.1145/2382553.2382556` (see page 37).

[GIT20] Giulio Garbi, Emilio Incerto, and Mirco Tribastone. "Learning Queuing Networks by Recurrent Neural Networks". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ICPE '20. Edmonton AB, Canada: Association for Computing Machinery, 2020, pp. 56–66. DOI: `10.1145/3358960.3379134` (see page 45).

[Gar20] Gartner, Inc. *Gartner Forecasts Worldwide Public Cloud End-User Spending to Grow 18% in 2021*. 2020. URL: `https://www.gartner.com/en/newsroom/press-releases/2020-11-17-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-grow-18-percent-in-2021` (visited on 07/08/2021) (see pages 1, 2).

[Geb+11] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Torsten Schaub, Marius Thomas Schneider, and Stefan Ziller. "A Portfolio Solver for Answer Set Programming: Preliminary Report". In: *Logic Programming and Nonmonotonic Reasoning*. Ed. by James P. Delgrande and Wolfgang Faber. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 352–357. DOI: `10.1007/978-3-642-20895-9_40` (see page 47).

[Gma+08] Daniel Gmach, Stefan Krompass, Andreas Scholz, Martin Wimmer, and Alfons Kemper. "Adaptive Quality of Service Management for Enterprise Services". In: *ACM Transactions on the Web* 2.1 (2008). DOI: `10.1145/1326561.1326569` (see page 37).

[Gov14] Nirmal Govind. *Optimizing the Netflix Streaming Experience with Data Science*. 2014. URL: `https://netflixtechblog.com/optimizing-the-netflix-streaming-experience-with-data-science-725f04c3e834` (visited on 07/08/2021) (see page 1).

[GMS07] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta. "Filling the Gap between Design and Performance/Reliability Models of Component-Based Systems". In: *Journal of Systems and Software* 80.4 (2007), pp. 528–558. DOI: 10.1016/j.jss.2006.07.023 (see page 23).

[Gre13] Brendan Gregg. "Thinking Methodically about Performance". In: *Communications of the ACM* 56.2 (2013), pp. 45–51. DOI: 10.1145/2408776.2408791 (see pages 54, 60).

[Gro+21a] Johannes Grohmann, Simon Eismann, Andre Bauer, Simon Spinner, Johannes Blum, Nikolas Herbst, and Samuel Kounev. *SARDE: Self-adaptive Resource Demand Estimation*. en. 2021. DOI: 10.24433/CO.8429465.V3 (see pages 93, 206).

[Gro+19a] Johannes Grohmann, Simon Eismann, Andre Bauer, Marwin Zuefle, Nikolas Herbst, and Samuel Kounev. "Utilizing Clustering to Optimize Resource Demand Estimation Approaches". In: *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. IEEE, 2019, pp. 134–139. DOI: 10.1109/fas-w.2019.00043 (see page 93).

[Gro+21b] Johannes Grohmann, Simon Eismann, André Bauer, Simon Spinner, Johannes Blum, Nikolas Herbst, and Samuel Kounev. "SARDE: A Framework for Continuous and Self-Adaptive Resource Demand Estimation". In: *ACM Transactions on Autonomous and Adaptive Systems* 15.2 (2021), pp. 1–31. DOI: 10.1145/3463369 (see pages 10, 93, 248).

[Gro+19b] Johannes Grohmann, Simon Eismann, Sven Elflein, Manar Mazkatli, Jóakim von Kistowski, and Samuel Kounev. "Detecting Parametric Dependencies for Performance Models Using Feature Selection Techniques". In: *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. MASCOTS '19. IEEE, 2019, pp. 309–322. DOI: 10.1109/mascots.2019.00042 (see pages 10, 111).

[GEK18] Johannes Grohmann, Simon Eismann, and Samuel Kounev. "The Vision of Self-Aware Performance Models". In: *2018 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2018, pp. 60–63. DOI: 10.1109/icsa-c.2018.00024 (see pages 22, 93, 110, 136, 247, 248).

[GEK19] Johannes Grohmann, Simon Eismann, and Samuel Kounev. "On Learning Parametric Dependencies from Monitoring Data". In: *Proceedings of the 10th Symposium on Software Performance 2019 (SSP'19)*. 2019 (see page 111).

[Gro+20a] Johannes Grohmann, Nikolas Herbst, Avi Chalbani, Yair Arian, Noam Peretz, and Samuel Kounev. "A Taxonomy of Techniques for SLO Failure Prediction in Software Systems". In: *Computers* 9.1 (2020), p. 10. DOI: 10.3390/computers9010010 (see pages 35, 36, 73, 74, 249).

[Gro+17] Johannes Grohmann, Nikolas Herbst, Simon Spinner, and Samuel Kounev. "Self-Tuning Resource Demand Estimation". In: *2017 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 2017, pp. 21–26. DOI: 10.1109/icac.2017.19 (see pages 93, 96, 98).

[Gro+18] Johannes Grohmann, Nikolas Herbst, Simon Spinner, and Samuel Kounev. "Using Machine Learning for Recommending Service Demand Estimation Approaches - Position Paper". In: *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018)*. INSTICC. SCITEPRESS - Science and Technology Publications, 2018, pp. 473–480. DOI: 10.5220/0006761104730480 (see pages 93, 104).

[Gro+19c] Johannes Grohmann, Patrick K. Nicholson, Jesus Omana Iglesias, Samuel Kounev, and Diego Lugones. "Monitorless: Predicting Performance Degradation in Cloud Applications with Machine Learning". In: *Proceedings of the 20th International Middleware Conference*. Middleware '19. ACM, 2019, pp. 149–162. DOI: 10.1145/3361525.3361543 (see pages 8, 22, 55).

[Gro+21c] Johannes Grohmann, Daniel Seybold, Simon Eismann, Mark Leznik, and Jörg Domaschka. *Measuring and Modeling the Performance Configurations of Distributed Database Systems*. en. 2021. DOI: 10.24433/CO.6929232.V3 (see pages 139, 227).

[Gro+20b] Johannes Grohmann, Daniel Seybold, Simon Eismann, Mark Leznik, Samuel Kounev, and Jörg Domaschka. "Baloo: Measuring and Modeling the Performance Configurations of Distributed DBMS". In: *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. MASCOTS '20. IEEE, 2020, pp. 1–8. DOI: 10.1109/MASCOTS50786.2020.9285960 (see pages 11, 100, 139).

[Gro+21d] Johannes Grohmann, Martin Straesser, Avi Chalbani, Simon Eismann, Yair Arian, Nikolas Herbst, Noam Peretz, and Samuel Kounev. "SuanMing: Explainable Prediction of Performance Degradations in Microservice Applications". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ACM, 2021. DOI: 10.1145/3427921.3450248 (see pages 9, 74).

[Gro+21e] Johannes Grohmann, Martin Straesser, Avi Chalbani, Simon Eismann, Yair Arian, Nikolas Herbst, Noam Peretz, and Samuel Kounev. *SuanMing: Explainable Prediction of Performance Degradations in Microservice Applications*. en. 2021. DOI: 10.24433/CO.8530346.V3 (see pages 74, 171).

[Gu+08] Xiaohui Gu, Spiros Papadimitriou, Philip S. Yu, and Shu-Ping Chang. "Online Failure Forecast for Fault-Tolerant Data Stream Processing". In: *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*. ICDE '08. USA: IEEE Computer Society, 2008, pp. 1388–1390. DOI: 10.1109/ICDE.2008.4497565 (see page 39).

[Guo+17] Jianmei Guo, Dingyu Yang, Norbert Siegmund, Sven Apel, Atrisha Sarkar, Pavel Valov, Krzysztof Czarnecki, Andrzej Wasowski, and Huiqun Yu. "Data-efficient performance learning for configurable systems". In: *Empirical Software Engineering* 23.3 (2017), pp. 1826–1867. DOI: `10.1007/s10664-017-9573-6` (see pages 49, 100).

[Gur97] Kevin Gurney. *An introduction to neural networks*. London: CRC Press, 1997 (see pages 27, 68, 131, 191).

[GE03] Isabelle Guyon and André Elisseeff. "An Introduction to Variable and Feature Selection". In: *Journal of Machine Learning Research* 3 (2003), pp. 1157–1182. DOI: `10.5555/944919.944968` (see pages 33, 121, 122, 224).

[HZ19] Huong Ha and Hongyu Zhang. "DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network". In: *Proceedings of the 41st International Conference on Software Engineering*. ICSE '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 1095–1106. DOI: `10.1109/ICSE.2019.00113` (see pages 50, 100).

[Hal+09] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. "The WEKA Data Mining Software: An Update". In: *SIGKDD Explorations Newsletter* 11.1 (2009), pp. 10–18. DOI: `10.1145/1656274.1656278` (see pages 132, 134, 210).

[Hal99] Mark Andrew Hall. "Correlation-based Feature Selection for Machine Learning". PhD thesis. Hamilton, New Zealand: University of Waikato, 1999 (see pages 112, 121, 122).

[Ham09] Dick Hamlet. "Tools and Experiments Supporting a Testing-Based Theory of Component Composition". In: *ACM Transactions on Software Engineering and Methodology* 18.3 (2009), pp. 1–41. DOI: `10.1145/1525880.1525885` (see page 109).

[Han05] Helena Handschuh. "SHA Family (Secure Hash Algorithm)". In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg. Boston, MA: Springer US, 2005, pp. 565–567. DOI: `10.1007/0-387-23483-7_388` (see page 130).

[HBR13] Lucia Happe, Barbora Buhnova, and Ralf Reussner. "Stateful Component-Based Performance Models". In: *Software and Systems Modeling* 13.4 (2013), pp. 1319–1343. DOI: `10.1007/s10270-013-0336-6` (see page 225).

[HR14] Brian Harrington and Roy Rapoport. *Introducing Atlas: Netflix's Primary Telemetry Platform*. 2014. URL: `https://netflixtechblog.com/introducing-atlas-netflixs-primary-telemetry-platform-bd31f4d8ed9a` (visited on 07/08/2021) (see page 3).

[Has+09] Trevor Hastie, Saharon Rosset, Ji Zhu, and Hui Zou. "Multi-class AdaBoost". In: *Statistics and Its Interface* 2.3 (2009), pp. 349–360. DOI: `10.4310/sii.2009.v2.n3.a8` (see pages 26, 191).

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Springer Science & Business Media, 2009. DOI: 10.1007/978-0-387-84858-7 (see page 123).

[HRK11] Malte Helmert, Gabriele Röger, and Erez Karpas. "Fast downward stone soup: A baseline for building planner portfolios". In: *ICAPS 2011 Workshop on Planning and Learning*. Citeseer. 2011, pp. 28–35 (see page 47).

[Hen+18] Abdeltawab Hendawi, Jayant Gupta, Liu Jiayi, Ankur Teredesai, Ramakrishnan Naveen, Shah Mohak, and Mohamed Ali. "Distributed NoSQL Data Stores: Performance Analysis and a Case Study". In: *2018 IEEE International Conference on Big Data (Big Data)*. Los Alamitos, CA, USA: IEEE Computer Society, 2018, pp. 1937–1944. DOI: 10.1109/BigData.2018.8622544 (see page 48).

[Her+17] Nikolas Herbst, Ayman Amin, Artur Andrzejak, Lars Grunske, Samuel Kounev, Ole J. Mengshoel, and Priya Sundararajan. "Online Workload Forecasting". In: *Self-Aware Computing Systems*. Ed. by Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu. Cham: Springer International Publishing, 2017, pp. 529–553. DOI: 10.1007/978-3-319-47474-8_18 (see page 78).

[Her18] Nikolas Roman Herbst. "Methods and Benchmarks for Auto-Scaling Mechanisms in Elastic Cloud Environments". en. PhD thesis. Universität Würzburg, 2018 (see pages 2, 248).

[Her+20a] Stefan Herrnleben, Rudy Ailabouni, Johannes Grohmann, Thomas Prantl, Christian Krupitzer, and Samuel Kounev. "An IoT Network Emulator for Analyzing the Influence of Varying Network Quality". In: *Simulation Tools and Techniques*. SIMUtools 2020. Springer International Publishing, 2020, pp. 580–599. DOI: 10.1007/978-3-030-72795-6_47 (see page 248).

[Her+20b] Stefan Herrnleben, Johannes Grohmann, Pitor Rygielski, Veronika Lesch, Christian Krupitzer, and Samuel Kounev. "A Simulation-Based Optimization Framework for Online Adaptation of Networks". In: *Simulation Tools and Techniques*. SIMUtools 2020. Springer International Publishing, 2020, pp. 513–532. DOI: 10.1007/978-3-030-72792-5_41 (see page 248).

[Her+21] Stefan Herrnleben, Maximilian Leidinger, Veronika Lesch, Thomas Prantl, Johannes Grohmann, Christian Krupitzer, and Samuel Kounev. "ComBench: A Benchmarking Framework for Publish/Subscribe Communication Protocols Under Network Limitations". In: *Performance Evaluation Methodologies and Tools*. Springer International Publishing, 2021, pp. 72–92. DOI: 10.1007/978-3-030-92511-6_5 (see page 248).

[Her+20c] Stefan Herrnleben, Piotr Rygielski, Johannes Grohmann, Simon Eismann, Tobias Hossfeld, and Samuel Kounev. "Model-Based Performance Predictions for SDN-Based Networks: A Case Study". In: *Lecture Notes in*

Computer Science. MMB 2020. Springer International Publishing, 2020, pp. 82–98. DOI: 10.1007/978-3-030-43024-5_6 (see page 248).

[HKP18] Andreas M. Hinz, Sandi Klavžar, and Ciril Petr. *The Tower of Hanoi-Myths and Maths*. Springer International Publishing, 2018. DOI: 10.1007/978-3-319-73779-9 (see page 221).

[HT11] Helmut Hlavacs and Thomas Treutner. "Predicting web service levels during vm live migrations". In: *2011 5th International DMTF Academic Alliance Workshop on Systems and Virtualization Management: Standards and the Cloud (SVM)*. IEEE. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1–10. DOI: 10.1109/svm.2011.6096464 (see pages 4, 38).

[Ho95] Tin Kam Ho. "Random Decision Forests". In: *Proceedings of 3rd International Conference on Document Analysis and Recognition*. ICDAR '95. USA: IEEE Computer Society, 1995, p. 278. DOI: 10.1109/ICDAR.1995.598994 (see pages 27, 112, 123, 132).

[HA04] Victoria Hodge and Jim Austin. "A survey of outlier detection methodologies". In: *Artificial Intelligence Review* 22.2 (2004), pp. 85–126. DOI: 10.1007/s10462-004-4304-y (see page 115).

[HK70a] Arthur E. Hoerl and Robert W. Kennard. "Ridge Regression: Applications to Nonorthogonal Problems". In: *Technometrics* 12.1 (1970), pp. 69–82. DOI: 10.1080/00401706.1970.10488635 (see page 25).

[HK70b] Arthur E. Hoerl and Robert W. Kennard. "Ridge Regression: Biased Estimation for Nonorthogonal Problems". In: *Technometrics* 12.1 (1970), pp. 55–67. DOI: 10.1080/00401706.1970.10488634 (see page 25).

[Hof20] Todd Hoff. *A Short on How Zoom Works*. 2020. URL: http://highscalability.com/blog/2020/5/14/a-short-on-how-zoom-works.html (visited on 07/08/2021) (see page 1).

[Hoo14] Andre van Hoorn. "Model-Driven Online Capacity Management for Component-Based Software Systems". Doktorarbeit/PhD. Kiel, Germany: Faculty of Engineering, Kiel University, 2014 (see pages 5, 19, 22, 23).

[Hoo+09] André van Hoorn, Matthias Rohr, Wilhelm Hasselbring, Jan Waller, Jens Ehlers, Sören Frey, and Dennis Kieselhorst. *Continuous Monitoring of Software Services: Design and Application of the Kieker Framework*. Forschungsbericht. Kiel University, 2009 (see page 114).

[HWH12] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. "Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis". In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ICPE '12. Boston, Massachusetts, USA: Association for Computing Machinery, 2012, pp. 247–248. DOI: 10.1145/2188286.2188326 (see pages 21, 78, 114).

[Hri+99] Curtis E. Hrischuk, C. Murray Woodside, Jerome A. Rolia, and Rod Iversen. "Trace-Based Load Characterization for Generating Performance Software Models". In: *IEEE Transactions on Software Engineering* 25.1 (1999), pp. 122–135. DOI: `10.1109/32.748921` (see pages 5, 40, 109, 110).

[Hub+17] Nikolaus Huber, Fabian Brosig, Simon Spinner, Samuel Kounev, and Manuel Bähr. "Model-Based Self-Aware Performance and Resource Management Using the Descartes Modeling Language". In: *IEEE Transactions on Software Engineering (TSE)* 43.5 (2017), pp. 432–452. DOI: `10.1109/TSE.2016.2613863` (see pages 5, 19, 22, 23, 91, 248).

[Hub64] Peter J. Huber. "Robust Estimation of a Location Parameter". In: *Annals of Mathematical Statistics* 35.1 (1964), pp. 73–101. DOI: `10.1214/aoms/1177703732` (see pages 25, 131).

[HHL11] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. "Sequential Model-Based Optimization for General Algorithm Configuration". In: *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*. LION'05. Rome, Italy: Springer Berlin Heidelberg, 2011, pp. 507–523. DOI: `10.1007/978-3-642-25566-3_40` (see page 47).

[Hut+14] Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. "Algorithm runtime prediction: Methods & evaluation". In: *Artificial Intelligence* 206 (2014), pp. 79–111. DOI: `10.1016/j.artint.2013.10.003` (see page 104).

[HK06] Rob J. Hyndman and Anne B. Koehler. "Another look at measures of forecast accuracy". In: *International Journal of Forecasting* 22.4 (2006), pp. 679–688. DOI: `10.1016/j.ijforecast.2006.03.001` (see page 122).

[Igl+17] Jesus Omana Iglesias, Jordi Arjona Aroca, Volker Hilt, and Diego Lugones. "ORCA: An ORCHestration AUtomata for Configuring VNFs". In: *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. Middleware '17. Las Vegas, Nevada: Association for Computing Machinery, 2017, pp. 81–94. DOI: `10.1145/3135974.3135982` (see page 53).

[INT18] E. Incerto, A. Napolitano, and M. Tribastone. "Moving Horizon Estimation of Service Demands in Queuing Networks". In: *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. Los Alamitos, CA, USA: IEEE Computer Society, 2018, pp. 348–354. DOI: `10.1109/MASCOTS.2018.00040` (see page 44).

[INT21] Emilio Incerto, Annalisa Napolitano, and Mirco Tribastone. "Learning Queuing Networks via Linear Optimization". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ICPE '21. Virtual Event, France: Association for Computing Machinery, 2021, pp. 51–60. DOI: `10.1145/3427921.3450245` (see page 44).

[IYE11] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. "On the Performance Variability of Production Cloud Services". In: *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2011, pp. 104–113. DOI: 10.1109/ccgrid.2011.22 (see pages 138, 143, 235).

[Iqb20] Mansoor Iqbal. *Zoom Revenue and Usage Statistics (2020)*. 2020. URL: https://www.businessofapps.com/data/zoom-statistics/ (visited on 07/08/2021) (see page 1).

[Isr+05] Tauseef A. Israr, Danny H. Lau, Greg Franks, and Murray Woodside. "Automatic Generation of Layered Queuing Software Performance Models from Commonly Available Traces". In: *Proceedings of the 5th International Workshop on Software and Performance*. WOSP '05. Palma, Illes Balears, Spain: Association for Computing Machinery, 2005, pp. 147–158. DOI: 10.1145/1071021.1071037 (see pages 5, 40, 109, 110).

[Jam+18] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonca, James Lewis, and Stefan Tilkov. "Microservices: The Journey So Far and Challenges Ahead". In: *IEEE Software* 35.3 (2018), pp. 24–35. DOI: 10.1109/ms.2018.2141039 (see pages 2, 73, 75).

[Jan+10] Vijay Janapa Reddi, Benjamin C. Lee, Trishul Chilimbi, and Kushagra Vaid. "Web Search Using Mobile Cores: Quantifying and Mitigating the Price of Efficiency". In: *ACM SIGARCH Computer Architecture News* 38.3 (2010), pp. 314–325. DOI: 10.1145/1816038.1816002 (see page 61).

[JS14] Brendan Jennings and Rolf Stadler. "Resource Management in Clouds: Survey and Research Challenges". In: *Journal of Network and Systems Management* 23.3 (2014), pp. 567–619. DOI: 10.1007/s10922-014-9307-7 (see page 2).

[JPG19] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. "Performance Modeling for Cloud Microservice Applications". In: *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. ICPE '19. Mumbai, India: Association for Computing Machinery, 2019, pp. 25–32. DOI: 10.1145/3297663.3310309 (see pages 4, 39).

[JG98] D. N. Joanes and C. A. Gill. "Comparing measures of sample skewness and kurtosis". In: *Journal of the Royal Statistical Society: Series D (The Statistician)* 47.1 (1998), pp. 183–189. DOI: 10.1111/1467-9884.00122 (see pages 103, 104).

[Jon18] Nicola Jones. "How to stop data centres from gobbling up the world's electricity". In: *Nature* 561.7722 (2018), pp. 163–166. DOI: 10.1038/d41586-018-06610-y (see page 2).

[Jud16] Peter Judge. *Netflix completes move to AWS public cloud*. 2016. URL: https://www.datacenterdynamics.com/en/news/netflix-completes-move-to-aws-public-cloud/ (visited on 07/08/2021) (see page 1).

[Jud20a] Peter Judge. *Most of Zoom runs on AWS, not Oracle - says AWS*. 2020. URL: `https://www.datacenterdynamics.com/en/news/most-zoom-runs-aws-not-oracle-says-aws/` (visited on 07/08/2021) (see page 1).

[Jud20b] Peter Judge. *Zoom picks Oracle for some of its cloud infrastructure*. 2020. URL: `https://www.datacenterdynamics.com/en/news/zoom-picks-oracle-its-cloud-infrastructure/` (visited on 07/08/2021) (see page 1).

[Kal+12] Amir Kalbasi, Diwakar Krishnamurthy, Jerry Rolia, and Stephen Dawson. "DEC: Service Demand Estimation with Confidence". In: *IEEE Transactions on Software Engineering* 38.3 (2012), pp. 561–578. DOI: `10.1109/TSE.2011.23` (see page 45).

[Kal+11] Amir Kalbasi, Diwakar Krishnamurthy, Jerry Rolia, and Michael Richter. "MODE: Mix Driven on-Line Resource Demand Estimation". In: *Proceedings of the 7th International Conference on Network and Services Management*. CNSM '11. Paris, France: International Federation for Information Processing, 2011, pp. 1–9. DOI: `10.5555/2147671.2147673` (see page 45).

[Kal+20] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, and Sven Apel. "The Interplay of Sampling and Machine Learning for Software Performance Prediction". In: *IEEE Software* 37.4 (2020), pp. 58–66. DOI: `10.1109/MS.2020.2987024` (see page 144).

[Kel+06] Terence Kelly, Alex Zhang, Terence Kelly, and Alex Zhang. *Predicting performance in distributed enterprise applications*. Tech. rep. HP Labs Tech Report, 2006 (see page 44).

[KC03] Jeffrey O. Kephart and David M. Chess. "The Vision of Autonomic Computing". In: *Computer* 36.1 (2003), pp. 41–50. DOI: `10.1109/MC.2003.1160055` (see page 93).

[Ker+19] Pascal Kerschke, Holger H. Hoos, Frank Neumann, and Heike Trautmann. "Automated Algorithm Selection: Survey and Perspectives". In: *Evolutionary Computation* 27.1 (2019). PMID: 30475672, pp. 3–45. DOI: `10.1162/evco_a_00242` (see pages 41, 46, 47, 102).

[KDK18] Jóakim von Kistowski, Maximilian Deffner, and Samuel Kounev. "Run-Time Prediction of Power Consumption for Component Deployments". In: *2018 IEEE International Conference on Autonomic Computing (ICAC)*. 2018 IEEE International Conference on Autonomic Computing (ICAC) (Trento, Italy). IEEE, 2018, pp. 151–156. DOI: `10.1109/ICAC.2018.00025` (see pages 61, 75, 80, 156, 158, 172).

[Kis+18] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research". In: *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*

(*MASCOTS*). MASCOTS '18. IEEE, 2018, pp. 223–236. DOI: 10.1109/mascots.2018.00030 (see pages 15, 151–153, 158, 171, 180, 209).

[Kis+19a] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. *TeaStore*. Standard Performance Evaluation Corporation Research Group (SPEC RG) accepted Tool. https://research.spec.org/tools/overview/teastore.html. 2019 (see page 152).

[Kis+19b] Jóakim von Kistowski, Johannes Grohmann, Norbert Schmitt, and Samuel Kounev. "Predicting Server Power Consumption from Standard Rating Results". In: *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. ICPE '19. ACM, 2019, pp. 301–312. DOI: 10.1145/3297663.3310298 (see page 249).

[Kis19] Jóakim Gunnarsson von Kistowski. "Measuring, Rating, and Predicting the Energy Efficiency of Servers". en. PhD thesis. Universität Würzburg, 2019. DOI: 10.25972/OPUS-17847 (see pages 210, 248).

[Kis+17] Jóakim Von Kistowski, Nikolas Herbst, Samuel Kounev, Henning Groenda, Christian Stier, and Sebastian Lehrig. "Modeling and Extracting Load Intensity Profiles". In: *ACM Transactions on Autonomous and Adaptive Systems* 11.4 (2017). DOI: 10.1145/3019596 (see page 61).

[Koh21] Shelley E. Kohan. *Amazon's Net Profit Soars 84% With Sales Hitting $386 Billion*. 2021. URL: https://www.forbes.com/sites/shelleykohan/2021/02/02/amazons-net-profit-soars-84-with-sales-hitting-386-billion/ (visited on 07/08/2021) (see pages 1, 2).

[Kot+15] Lars Kotthoff, Pascal Kerschke, Holger Hoos, and Heike Trautmann. "Improving the State of the Art in Inexact TSP Solving Using Per-Instance Algorithm Selection". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2015, pp. 202–217. DOI: 10.1007/978-3-319-19084-6_18 (see page 104).

[KW14] Olga Kouchnarenko and Jean-François Weber. "Adapting Component-Based Systems at Runtime via Policies with Temporal Patterns". In: *10th International Symposium on Formal Aspects of Component Software - Volume 8348*. FACS 2013. Nanchang, China: Springer International Publishing, 2014, pp. 234–253. DOI: 10.1007/978-3-319-07602-7_15 (see page 47).

[Kou06] Samuel Kounev. "Performance Modeling and Evaluation of Distributed Component-Based Systems Using Queueing Petri Nets". In: *IEEE Transactions on Software Engineering* 32.7 (2006), pp. 486–502. DOI: 10.1109/TSE.2006.69 (see page 22).

[KBH14] Samuel Kounev, Fabian Brosig, and Nikolaus Huber. *The Descartes Modeling Language*. Tech. rep. Department of Computer Science, University of Wuerzburg, 2014 (see pages 22, 23).

[Kou+16] Samuel Kounev, Nikolaus Huber, Fabian Brosig, and Xiaoyun Zhu. “A Model-Based Approach to Designing Self-Aware IT Systems and Infrastructures”. In: *Computer* 49.7 (2016), pp. 53–61. DOI: 10.1109/mc.2016.198 (see pages 22, 91, 111, 248).

[Kou+17] Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu. *Self-Aware Computing Systems*. Ed. by Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu. Berlin Heidelberg, Germany: Springer-Verlag GmbH, 2017 (see pages 22, 41, 46, 93, 248).

[KLK20] Samuel Kounev, Klaus-Dieter Lange, and Jóakim von Kistowski. *Systems Benchmarking. For Scientists and Engineers*. Springer International Publishing, 2020. DOI: 10.1007/978-3-030-41705-5 (see page 19).

[Koz08] Heiko Koziolek. “Parameter dependencies for reusable performance specifications of software components”. PhD thesis. 2008. 333 pp. DOI: 10.5445/KSP/1000009096 (see pages 19, 23, 41, 109, 111).

[Koz10] Heiko Koziolek. “Performance Evaluation of Component-Based Software Systems: A Survey”. In: *Performance Evaluation* 67.8 (2010), pp. 634–658. DOI: 10.1016/j.peva.2009.07.007 (see pages 22, 23, 40, 41, 109).

[Kra+09] Stephan Kraft, Sergio Pacheco-Sanchez, Giuliano Casale, and Stephen Dawson. “Estimating Service Resource Consumption from Response Time Measurements”. In: *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*. VALUETOOLS ’09. Pisa, Italy: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009. DOI: 10.4108/ICST.VALUETOOLS2009.7526 (see pages 44–46, 94).

[Kre20] John Kreisa. *Introducing the Docker Index: Insight from the World’s Most Popular Container Registry*. 2020. URL: https://www.docker.com/blog/introducing-the-docker-index/ (visited on 07/08/2021) (see page 17).

[Kro12] Klaus Krogmann. “Reconstruction of Software Component Architectures and Behaviour Models using Static and Dynamic Analysis”. PhD thesis. 2012. 371 pp. DOI: 10.5445/KSP/1000025617 (see page 114).

[KKR10] Klaus Krogmann, Michael Kuperberg, and Ralf Reussner. “Using genetic search for reverse engineering of parametric behavior models for performance prediction”. In: *IEEE Transactions on Software Engineering* 36.6 (2010), pp. 865–877. DOI: 10.1109/tse.2010.69 (see pages 4, 37, 41, 42, 109, 110).

[KKR14] Jörn Kuhlenkamp, Markus Klems, and Oliver Röss. “Benchmarking Scalability and Elasticity of Distributed Database Systems”. In: *Proceedings of the VLDB Endowment* 7.12 (2014), pp. 1219–1230. DOI: 10.14778/2732977.2732995 (see pages 48, 232).

[KTZ09] Dinesh Kumar, Asser Tantawi, and Li Zhang. "Real-Time Performance Modeling for Adaptive Software Systems". In: *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*. VALUETOOLS '09. Pisa, Italy: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009. DOI: `10.4108/ICST.VALUETOOLS2009.7944` (see pages 44, 46, 92, 94).

[KZT09] Dinesh Kumar, Li Zhang, and Asser Tantawi. "Enhanced Inferencing: Estimation of a Workload Dependent Performance Model". In: *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*. VALUETOOLS '09. Pisa, Italy: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009. DOI: `10.4108/ICST.VALUETOOLS2009.7799` (see page 44).

[Kun+12] Sajib Kundu, Raju Rangaswami, Ajay Gulati, Ming Zhao, and Kaushik Dutta. "Modeling Virtualized Applications Using Machine Learning Techniques". In: *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*. VEE '12. London, England, UK: Association for Computing Machinery, 2012, pp. 3–14. DOI: `10.1145/2151024.2151028` (see pages 4, 22, 37).

[KKR09] Michael Kuperberg, Martin Krogmann, and Ralf Reussner. "TimerMeter: Quantifying Properties of Software Timers for System Analysis". In: *2009 Sixth International Conference on the Quantitative Evaluation of Systems*. IEEE, 2009, pp. 85–94. DOI: `10.1109/QEST.2009.49` (see page 43).

[KR10] Miron B. Kursa and Witold R. Rudnicki. "Feature Selection with the Boruta Package". In: *Journal of Statistical Software* 36.11 (2010), pp. 1–13. DOI: `10.18637/jss.v036.i11` (see page 123).

[Kwi19] Aleksandra Kwiecień. *10 companies that implemented the microservice architecture and paved the way for others*. 2019. URL: `https://divante.com/blog/10-companies-that-implemented-the-microservice-architecture-and-paved-the-way-for-others/` (visited on 07/08/2021) (see pages 2, 3, 15).

[Kwo+13] Yongin Kwon, Sangmin Lee, Hayoon Yi, Donghyun Kwon, Seungjun Yang, Byung-Gon Chun, Ling Huang, Petros Maniatis, Mayur Naik, and Yunheung Paek. "Mantis: Automatic Performance Prediction for Smartphone Applications". In: *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*. USENIX ATC'13. San Jose, CA: USENIX Association, 2013, pp. 297–308. DOI: `10.5555/2535461.2535498` (see page 42).

[Laz+84] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. USA: Prentice-Hall, Inc., 1984 (see pages 23, 44, 91).

[Leh06] Erich L. Lehmann. *Nonparametrics*. Springer New York, 2006. 480 pp. (see pages 142, 229).

[LC16] Philipp Leitner and Jürgen Cito. "Patterns in the Chaos—A Study of Performance Variation and Predictability in Public IaaS Clouds". In: *ACM Transactions on Internet Technology* 16.3 (2016), pp. 1–23. DOI: 10.1145/2885497 (see pages 138, 143, 232, 235).

[LF14] James Lewis and Martin Fowler. *Microservices - a definition of this new architectural term*. 2014. URL: https://martinfowler.com/articles/microservices.html (visited on 07/08/2021) (see pages 2, 5, 15, 16, 73, 75, 137).

[Li14] Haifeng Li. *Smile*. https://haifengl.github.io. 2014 (see page 192).

[Li+09] Jim Li, John Chinneck, Murray Woodside, Marin Litoiu, and Gabriel Iszlai. "Performance Model Driven QoS Guarantees and Optimization in Clouds". In: *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*. CLOUD '09. USA: IEEE Computer Society, 2009, pp. 15–22. DOI: 10.1109/CLOUD.2009.5071528 (see page 22).

[LLG18] Luyi Li, Minyan Lu, and Tingyang Gu. "Extracting Interaction-Related Failure Indicators for Online Detection and Prediction of Content Failures". In: *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) (Memphis, TN). IEEE, 2018, pp. 278–285. DOI: 10.1109/ISSREW.2018.00019 (see pages 5, 38).

[Lil00] David J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2000. DOI: 10.1017/cbo9780511612398 (see page 19).

[LCZ18] Jinjin Lin, Pengfei Chen, and Zibin Zheng. "Microscope: Pinpoint Performance Issues with Causal Graphs in Micro-service Environments". In: *Service-Oriented Computing*. Ed. by Claus Pahl, Maja Vukovic, Jianwei Yin, and Qi Yu. Vol. 11236. Cham: Springer International Publishing, 2018, pp. 3–20. DOI: 10.1007/978-3-030-03596-9_1 (see pages 4, 40, 186).

[Lin+15] Marius Lindauer, Holger H. Hoos, Frank Hutter, and Torsten Schaub. "AutoFolio: An Automatically Configured Algorithm Selector". In: *Journal of Artificial Intelligence Research* 53 (2015), pp. 745–778. DOI: 10.1613/jair.4726 (see page 47).

[Lin16] David S. Linthicum. "Practical Use of Microservices in Moving Workloads to the Cloud". In: *IEEE Cloud Computing* 3.5 (2016), pp. 6–9. DOI: 10.1109/MCC.2016.114 (see pages 2, 73).

[LTZ08] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. "Isolation Forest". In: *2008 Eighth IEEE International Conference on Data Mining*. IEEE, 2008, pp. 413–422. DOI: 10.1109/ICDM.2008.17 (see page 143).

[Liu+06] Zhen Liu, Laura Wynter, Cathy H. Xia, and Fan Zhang. "Parameter inference of queueing models for IT systems using end-to-end measurements". In: *Performance Evaluation* 63.1 (2006), pp. 36–60. DOI: 10.1016/j.peva.2004.12.001 (see pages 44–46, 207).

[Liu+03] Zhen Liu, Cathy H. Xia, Petar Momcilovic, and Li Zhang. "Ambience: Automatic model building using inference". In: *Congress MSR03*. 2003 (see page 44).

[Lot17] Angus Loten. *Oracle CEO Hurd Says 80% of Corporate Data Centers Gone by 2025*. Ed. by Steven Rosenbush. 2017. URL: https://www.wsj.com/articles/BL-CIOB-11316 (visited on 07/08/2021) (see page 2).

[Lou+18] Jungang Lou, Yunliang Jiang, Qing Shen, and Ruiqin Wang. "Failure prediction by relevance vector regression with improved quantum-inspired gravitational search". In: *Journal of Network and Computer Applications* 103 (2018). PII: S1084804517303946, pp. 171–177. DOI: 10.1016/j.jnca.2017.11.013 (see pages 4, 38).

[LL17] Scott M. Lundberg and Su-In Lee. "A Unified Approach to Interpreting Model Predictions". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 4768–4777 (see page 169).

[Mac92] David J. C. MacKay. "Bayesian Interpolation". In: *Neural Computation* 4.3 (1992), pp. 415–447. DOI: 10.1162/neco.1992.4.3.415 (see page 25).

[Mah+17] Ashraf Mahgoub, Paul Wood, Sachandhan Ganesh, Subrata Mitra, Wolfgang Gerlach, Travis Harrison, Folker Meyer, Ananth Grama, Saurabh Bagchi, and Somali Chaterji. "Rafiki: A Middleware for Parameter Tuning of NoSQL Datastores for Dynamic Metagenomics Workloads". In: *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. Middleware '17. Las Vegas, Nevada: Association for Computing Machinery, 2017, pp. 28–40. DOI: 10.1145/3135974.3135991 (see pages 5, 49, 138).

[Mal14] Yuri Malitsky. "Evolving Instance-Specific Algorithm Configuration". In: *Instance-Specific Algorithm Configuration*. Cham: Springer International Publishing, 2014, pp. 93–105. DOI: 10.1007/978-3-319-11230-5_9 (see page 47).

[Mar+20] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Rui Xin. "Predicting failures in multi-tier distributed systems". In: *Journal of Systems and Software* 161 (2020). PII: S0164121219302389, p. 110464. DOI: 10.1016/j.jss.2019.110464 (see page 39).

[Mas+20] Eric Masanet, Arman Shehabi, Nuoa Lei, Sarah Smith, and Jonathan Koomey. "Recalibrating global data center energy-use estimates". In: *Science* 367.6481 (2020), pp. 984–986. DOI: 10.1126/science.aba3758 (see page 2).

[MF10] Andréa Matsunaga and José A. B. Fortes. “On the Use of Machine Learning to Predict the Time and Resources Consumed by Applications”. In: *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. CCGRID ’10. USA: IEEE Computer Society, 2010, pp. 495–504. DOI: 10.1109/CCGRID.2010.98 (see page 37).

[MK18] Manar Mazkatli and Anne Koziolek. “Continuous Integration of Performance Model”. In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE ’18. Berlin, Germany: Association for Computing Machinery, 2018, pp. 153–158. DOI: 10.1145/3185768.3186285 (see pages 42, 111).

[Maz+20] Manar Mazkatli, David Monschein, Johannes Grohmann, and Anne Koziolek. “Incremental Calibration of Architectural Performance Models with Parametric Dependencies”. In: *2020 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2020, pp. 23–34. DOI: 10.1109/icsa47634.2020.00011 (see pages 42, 111, 208, 248).

[MP43] Warren S. McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The Bulletin of Mathematical Biophysics* 5.4 (1943), pp. 115–133. DOI: 10.1007/bf02478259 (see page 27).

[Men97] Daniel A. Menascé. “A Framework for Software Performance Engineering of Client/Server Systems.” In: *International Conference Computer Measurement Group Proceedings*. 1997, pp. 460–469 (see pages 40, 41, 109).

[Men02a] Daniel A. Menascé. *Capacity planning for Web services : metrics, models, and methods*. Upper Saddle River, NJ: Prentice Hall, 2002 (see page 191).

[Men02b] Daniel A. Menascé. “Load Testing, Benchmarking, And Application Performance Management For The Web”. In: *International Conference Computer Measurement Group Proceedings*. 2002, pp. 271–281 (see page 19).

[Men08] Daniel A. Menascé. “Computing Missing Service Demand Parameters for Performance Models”. In: *International Conference Computer Measurement Group Proceedings*. 2008, pp. 241–248 (see pages 45, 46, 207).

[MDA04] Daniel A. Menascé, Lawrence W. Dowdy, and Virgilio A. F. Almeida. *Performance by Design: Computer Capacity Planning By Example*. USA: Prentice Hall PTR, 2004 (see pages 23, 44, 91).

[MG98] Daniel A. Menascé and Hassan Gomaa. “On a Language Based Method for Software Performance Engineering of Client/Server Systems”. In: *Proceedings of the 1st International Workshop on Software and Performance*. WOSP ’98. Santa Fe, New Mexico, USA: Association for Computing Machinery, 1998, pp. 63–69. DOI: 10.1145/287318.287331 (see page 40).

[MG00] Daniel A. Menascé and Hassan Gomaa. “A Method for Design and Performance Modeling of Client/Server Systems”. In: *IEEE Transactions on Software Engineering* 26.11 (2000), pp. 1066–1085. DOI: 10.1109/32.881718 (see pages 22, 191).

[Met78] Charles E. Metz. “Basic principles of ROC analysis”. In: *Seminars in Nuclear Medicine* 8.4 (1978), pp. 283–298. DOI: `10.1016/s0001-2998(78)80014-2` (see page 31).

[MP07] Karissa Miller and Mahmoud Pegah. “Virtualization: Virtually at the Desktop”. In: *Proceedings of the 35th Annual ACM SIGUCCS Fall Conference*. SIGUCCS ’07. Orlando, Florida, USA: Association for Computing Machinery, 2007, pp. 255–260. DOI: `10.1145/1294046.1294107` (see page 17).

[MF11] Ahmad Mizan and Greg Franks. “An Automatic Trace Based Performance Evaluation Model Building for Parallel Distributed Systems”. In: *Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering*. ICPE ’11. Karlsruhe, Germany: Association for Computing Machinery, 2011, pp. 61–72. DOI: `10.1145/1958746.1958760` (see pages 5, 40, 109, 110).

[Moh12] Atef Mohamed. “Software Architecture-Based Failure Prediction”. Queen’s University, 2012 (see page 39).

[Mon+21] David Monschein, Manar Mazkatli, Robert Heinrich, and Anne Koziolek. “Enabling Consistency between Software Artefacts for Software Adaption and Evolution”. In: *2021 IEEE 18th International Conference on Software Architecture (ICSA)*. IEEE, 2021. DOI: `10.1109/icsa51549.2021.00009` (see page 42).

[Nad+16] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice Architecture: Aligning Principles, Practices, and Culture*. O’Reilly Media, Inc., 2016. 146 pp. DOI: `10.5555/3002814` (see pages 17, 18).

[NKO19] Luigi Nardi, David Koeplinger, and Kunle Olukotun. “Practical Design Space Exploration”. In: *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2019. DOI: `10.1109/mascots.2019.00045` (see page 50).

[New21] Aaron Newcomb. *Sysdig 2021 container security and usage report: Shifting left is not enough*. 2021. URL: `https://sysdig.com/blog/sysdig-2021-container-security-usage-report/` (visited on 07/08/2021) (see pages 17, 18).

[Ngu+13] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. “AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service”. In: *10th International Conference on Autonomic Computing, ICAC’13, San Jose, CA, USA, June 26-28, 2013*. Ed. by Jeffrey O. Kephart, Calton Pu, and Xiaoyun Zhu. USENIX Association, 2013, pp. 69–82 (see pages 4, 37, 169).

[Nis+13] Rajesh Nishtala et al. "Scaling Memcache at Facebook". In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. nsdi'13. Lombard, IL: USENIX Association, 2013, pp. 385–398. DOI: 10.5555/2482626.2482663 (see page 156).

[Noo15] Omar-Qais Noorshams. "Modeling and Prediction of I/O Performance in Virtualized Environments". PhD thesis. 2015. DOI: 10.5445/IR/1000046750 (see pages 47, 100).

[Noo+13] Qais Noorshams, Dominik Bruhn, Samuel Kounev, and Ralf Reussner. "Predictive Performance Modeling of Virtualized Storage Systems Using Optimized Statistical Regression Techniques". In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. ICPE '13. Prague, Czech Republic: Association for Computing Machinery, 2013, pp. 283–294. DOI: 10.1145/2479871.2479910 (see pages 42, 47, 50, 100).

[Nou+09] Ramon Nou, Samuel Kounev, Ferran Julií, and Jordi Torres. "Autonomic QoS Control in Enterprise Grid Environments Using Online Simulation". In: *Journal of Systems and Software* 82.3 (2009), pp. 486–502. DOI: 10.1016/j.jss.2008.07.048 (see page 43).

[Nov16] Asami Novak. *Going to Market Faster: Most Companies Are Deploying Code Weekly, Daily, or Hourly*. 2016. URL: https://blog.newrelic.com/technology/data-culture-survey-results-faster-deployment/ (visited on 07/08/2021) (see pages 3, 16).

[Nul21] Christopher Null. *10 companies killing it at DevOps*. 2021. URL: https://techbeacon.com/devops/10-companies-killing-it-devops (visited on 07/08/2021) (see pages 3, 16).

[OHa06] Charlene O'Hanlon. "A Conversation with Werner Vogels". In: *Queue* 4.4 (2006), pp. 14–22. DOI: 10.1145/1142055.1142065 (see page 16).

[Oh+17] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. "Finding Near-Optimal Configurations in Product Lines by Random Sampling". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, pp. 61–71. DOI: 10.1145/3106237.3106273 (see page 144).

[Ost+14] Per-Olov Ostberg et al. "The CACTOS Vision of Context-Aware Cloud Topology Optimization and Simulation". In: *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*. IEEE, 2014. DOI: 10.1109/cloudcom.2014.62 (see page 248).

[OY16] Burcu Ozcelik and Cemal Yilmaz. "Seer: A Lightweight Online Failure Prediction Approach". In: *IEEE Transactions on Software Engineering* 42.1 (2016), pp. 26–46. DOI: 10.1109/TSE.2015.2442577 (see page 39).

[Pac+08] Giovanni Pacifici, Wolfgang Segmuller, Mike Spreitzer, and Asser Tantawi. "CPU Demand for Web Serving: Measurement Analysis and Dynamic Estimation". In: *Performance Evaluation* 65.6–7 (2008), pp. 531–553. DOI: `10.5555/1363365.1363465` (see page 44).

[PSF16] Tapti Palit, Yongming Shen, and Michael Ferdman. "Demystifying cloud benchmarking". In: *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. April. Washington, DC, USA: IEEE Computer Society, 2016, pp. 122–132. DOI: `10.1109/ISPASS.2016.7482080` (see pages 61, 151, 156).

[PY10] Sinno Jialin Pan and Qiang Yang. "A Survey on Transfer Learning". In: *IEEE Transactions on Knowledge and Data Engineering* 22.10 (2010), pp. 1345–1359. DOI: `10.1109/TKDE.2009.191` (see page 169).

[Pap+19] Alessandro Vittorio Papadopoulos, Laurens Versluis, André Bauer, Nikolas Herbst, Jóakim Von Kistowski, Ahmed Ali-eldin, Cristina Abad, José Nelson Amaral, Petr Tůma, and Alexandru Iosup. "Methodological Principles for Reproducible Performance Evaluation in Cloud Computing". In: *IEEE Transactions on Software Engineering* (2019), pp. 1–1. DOI: `10.1109/TSE.2019.2927908` (see page 143).

[Pea01] Karl Pearson. "LIII. On lines and planes of closest fit to systems of points in space". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2.11 (1901), pp. 559–572. DOI: `10.1080/14786440109462720` (see pages 33, 66).

[Ped+11] Fabian Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12.85 (2011), pp. 2825–2830 (see pages 64, 68, 143, 174, 232).

[Per+19] Juliana Alves Pereira, Hugo Martin, Mathieu Acher, Jean-Marc Jézéquel, Goetz Botterweck, and Anthony Ventresque. "Learning Software Configuration Spaces: A Systematic Literature Review". In: *CoRR* abs/1906.03018 (2019) (see page 144).

[Per+20] Juliana Alvez Pereira, Mathieu Acher, Hugo Martin, and Jean-Marc Jézéquel. "Sampling Effect on Performance Prediction of Configurable Systems: A Case Study". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ICPE '20. Edmonton AB, Canada: Association for Computing Machinery, 2020, pp. 277–288. DOI: `10.1145/3358960.3379137` (see page 144).

[PCP15] Juan F. Pérez, Giuliano Casale, and Sergio Pacheco-Sanchez. "Estimating Computational Requirements in Multi-Threaded Applications". In: *IEEE Transactions on Software Engineering* 41.3 (2015), pp. 264–278. DOI: `10.1109/TSE.2014.2363472` (see page 45).

[PPC13] Juan F. Pérez, Sergio Pacheco-Sanchez, and Giuliano Casale. "An Offline Demand Estimation Method for Multi-Threaded Applications". In: *Proceedings of the 2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*. MASCOTS '13. USA: IEEE Computer Society, 2013, pp. 21–30. DOI: `10.1109/MASCOTS.2013.10` (see pages 44, 45).

[Pic12] Beau Piccart. "Algorithms for multi-target learning". PhD thesis. Katholieke Universiteit Leuven, 2012 (see page 24).

[Pie20] David Pierce. *Zoom conquered video chat — now it has even bigger plans*. 2020. URL: `https://www.protocol.com/zoom-videoconferencing-history-profit` (visited on 07/08/2021) (see page 1).

[Pit+14] Teerat Pitakrat, Jonas Grunert, Oliver Kabierschke, Fabian Keller, and André van Hoorn. "A Framework for System Event Classification and Prediction by Means of Machine Learning". In: *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools*. VALUETOOLS '14. Bratislava, Slovakia: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2014, pp. 173–180. DOI: `10.4108/icst.valuetools.2014.258197` (see page 39).

[Pit+18] Teerat Pitakrat, Dušan Okanović, André van Hoorn, and Lars Grunske. "Hora: Architecture-aware online failure prediction". In: *Journal of Systems and Software* 137 (2018). PII: S0164121217300390, pp. 669–685. DOI: `10.1016/j.jss.2017.02.041` (see page 39).

[Por+16] Barry Porter, Matthew Grieves, Roberto Rodrigues Filho, and David Leslie. "REX: A Development Platform and Online Learning Approach for Runtime Emergent Software Systems". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 333–348. DOI: `10.5555/3026877.3026904` (see page 46).

[Poz+95] Ennio Pozzetti, Vidar Vetland, Jerome Rolia, and Giuseppe Serazzi. "Characterizing the resource demands of TCP/IP". In: *High-Performance Computing and Networking*. Ed. by Bob Hertzberger and Giuseppe Serazzi. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 79–85 (see pages 40, 41, 44, 109).

[Qui86] John Ross Quinlan. "Induction of decision trees". In: *Machine learning* 1.1 (1986), pp. 81–106. DOI: `10.1007/BF00116251` (see page 26).

[Qui92] John Ross Quinlan. "Learning with continuous classes". In: *5th Australian joint conference on artificial intelligence*. Vol. 92. World Scientific. 1992, pp. 343–348 (see pages 26, 112, 122, 132).

[Qui93] John Ross Quinlan. *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993 (see page 26).

[Red18] Red Hat, Inc. *The path to cloud-native applications*. 2018. (Visited on 07/08/2021) (see pages 3, 16, 18).

[Ren+17] Vincent Reniers, Dimitri Van Landuyt, Ansar Rafique, and Wouter Joosen. “On the State of NoSQL Benchmarks”. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ICPE ’17 Companion. L’Aquila, Italy: Association for Computing Machinery, 2017, pp. 107–112. DOI: 10.1145/3053600.3053622 (see pages 48, 237).

[Reu+16] Ralf H. Reussner, Steffen Becker, Jens Happe, Robert Heinrich, Anne Koziolek, Heiko Koziolek, Max Kramer, and Klaus Krogmann. *Modeling and Simulating Software Architectures: The Palladio Approach*. Cambridge, Massachusetts: The MIT Press, 2016 (see page 91).

[RSG16] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. “"Why Should I Trust You?": Explaining the Predictions of Any Classifier”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 1135–1144. DOI: 10.1145/2939672.2939778 (see page 169).

[Ric76] John R. Rice. “The Algorithm Selection Problem”. In: *Advances in Computers*. Ed. by Morris Rubinoff and Marshall C. Yovits. Vol. 15. Advances in Computers. Elsevier, 1976, pp. 65–118. DOI: 10.1016/s0065-2458(08)60520-3 (see pages 47, 100).

[Rij+17] Jan N. Rijn, Geoffrey Holmes, Bernhard Pfahringer, and Joaquin Vanschoren. “The Online Performance Estimation Framework: Heterogeneous Ensemble Learning for Data Streams”. In: *Machine learning* 107.1 (2017), pp. 149–176. DOI: 10.1007/s10994-017-5686-9 (see page 47).

[Rij+14] Jan N. van Rijn, Geoffrey Holmes, Bernhard Pfahringer, and Joaquin Vanschoren. “Algorithm Selection on Data Streams”. In: *Discovery Science*. Ed. by Sašo Džeroski, Panče Panov, Dragi Kocev, and Ljupčo Todorovski. Cham: Springer International Publishing, 2014, pp. 325–336 (see page 47).

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”. In: *Communications of the ACM* 21.2 (1978), pp. 120–126. DOI: 10.1145/359340.359342 (see page 130).

[RM51] Herbert Robbins and Sutton Monro. “A Stochastic Approximation Method”. In: *Annals of Mathematical Statistics* 22.3 (1951), pp. 400–407. DOI: 10.1214/aoms/1177729586 (see page 131).

[RP17] Roberto Rodrigues Filho and Barry Francis Porter. “Defining Emergent Software Using Continuous Self-Assembly, Perception, and Learning”. English. In: *ACM Transactions on Autonomous and Adaptive Systems* 12.3 (2017), pp. 1–25. DOI: 10.1145/3092691 (see page 47).

[RV95] Jerome Rolia and Vidar Vetland. "Parameter Estimation for Performance Models of Distributed Application Systems". In: *Proceedings of the 1995 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON '95. Toronto, Ontario, Canada: IBM Press, 1995, p. 54 (see pages 44, 46, 91, 94).

[RV98] Jerome Rolia and Vidar Vetland. "Correlating Resource Demand Information with ARM Data for Application Services". In: *Proceedings of the 1st International Workshop on Software and Performance*. WOSP '98. Santa Fe, New Mexico, USA: Association for Computing Machinery, 1998, pp. 219–230. DOI: `10.1145/287318.287366` (see page 44).

[Rol+10] Jerry Rolia, Amir Kalbasi, Diwakar Krishnamurthy, and Stephen Dawson. "Resource Demand Modeling for Multi-Tier Services". In: *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*. WOSP/SIPEW '10. San Jose, California, USA: Association for Computing Machinery, 2010, pp. 207–216. DOI: `10.1145/1712605.1712638` (see page 45).

[Rud19] Anna Rud. *Why and How Netflix, Amazon, and Uber Migrated to Microservices: Learn from Their Experience*. 2019. URL: `https://www.hys-enterprise.com/blog/why-and-how-netflix-amazon-and-uber-migrated-to-microservices-learn-from-their-experience/` (visited on 07/08/2021) (see pages 3, 5, 15, 137).

[RN20] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson series in artificial intelligence. Pearson, 2020. 1136 pp. (see pages 23–33).

[RK13] Piotr Rygielski and Samuel Kounev. "Network Virtualization for QoS-Aware Resource Management in Cloud Data Centers: A Survey". In: *PIK - Praxis der Informationsverarbeitung und Kommunikation* 36.1 (2013), pp. 55–64. DOI: `10.1515/pik-2012-0136` (see page 17).

[RKT15] Piotr Rygielski, Samuel Kounev, and Phuoc Tran-Gia. "Flexible Performance Prediction of Data Center Networks using Automatically Generated Simulation Models". In: *Proceedings of the Eighth EAI International Conference on Simulation Tools and Techniques*. ACM, 2015. DOI: `10.4108/eai.24-8-2015.2260961` (see page 22).

[Sak14] Sherif Sakr. "Cloud-Hosted Databases: Technologies, Challenges and Opportunities". In: *Cluster Computing* 17.2 (2014), pp. 487–502. DOI: `10.1007/s10586-013-0290-7` (see page 137).

[SP19] Areeg Samir and Claus Pahl. "DLA: Detecting and Localizing Anomalies in Containerized Microservice Architectures Using Markov Models". In: *2019 7th International Conference on Future Internet of Things and Cloud (FiCloud)*. 2019 7th International Conference on Future Internet of Things and Cloud (FiCloud) (Istanbul, Turkey). IEEE, 2019, pp. 205–213. DOI: `10.1109/FiCloud.2019.00036` (see pages 4, 37, 40).

[San16] Guilio Santoni. *Microservices Architectures: Become a Unicorn like Netflix, Twitter and Hailo*. 2016. URL: `https://www.slideshare.net/gjuljo/microservices-architectures-become-a-unicorn-like-netflix-twitter-and-hailo` (visited on 07/08/2021) (see page 2).

[Sar+15] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. "Cost-Efficient Sampling for Performance Prediction of Configurable Systems". In: *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. ASE '15. Lincoln, Nebraska: IEEE Press, 2015, pp. 342–352. DOI: `10.1109/ASE.2015.45` (see page 49).

[Sat+11] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. "Finding a "Kneedle" in a Haystack: Detecting Knee Points in System Behavior". In: *Proceedings of the 2011 31st International Conference on Distributed Computing Systems Workshops*. ICDCSW '11. USA: IEEE Computer Society, 2011, pp. 166–171. DOI: `10.1109/ICDCSW.2011.20` (see pages 37, 58).

[SG64] Abraham Savitzky and Marcel J. E. Golay. "Smoothing and differentiation of data by simplified least squares procedures". In: *Analytical Chemistry* 36.8 (1964), pp. 1627–1639. DOI: `10.1021/ac60214a047` (see page 58).

[Sch90] Robert E. Schapire. "The strength of weak learnability". In: *Machine Learning* 5.2 (1990), pp. 197–227. DOI: `10.1007/bf00116037` (see page 26).

[SLB17] Mark Schmidt, Nicolas Le Roux, and Francis Bach. "Minimizing finite sums with the stochastic average gradient". In: *Mathematical Programming* 162.1-2 (2017), pp. 83–112. DOI: `10.1007/s10107-016-1030-6` (see page 68).

[SWH06] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. "Open Versus Closed: A Cautionary Tale". In: *3rd Symposium on Networked Systems Design & Implementation (NSDI 06)*. San Jose, CA: USENIX Association, 2006 (see pages 188, 210).

[Sey17] Daniel Seybold. "Towards a Framework for Orchestrated Distributed Database Evaluation in the Cloud". In: *Proceedings of the 18th Doctoral Symposium of the 18th International Middleware Conference*. Middleware '17. Las Vegas, Nevada: Association for Computing Machinery, 2017, pp. 13–14. DOI: `10.1145/3152688.3152693` (see page 48).

[SD17] Daniel Seybold and Jörg Domaschka. "Is Distributed Database Evaluation Cloud-Ready?" In: *Communications in Computer and Information Science*. Springer International Publishing, 2017, pp. 100–108. DOI: `10.1007/978-3-319-67162-8_12` (see pages 48, 137, 236, 237).

[SD19] Daniel Seybold and Jörg Domaschka. *Mowgli: DBMS Performance and Scalability Evaluation Data Sets*. Zenodo, 2019. DOI: `10.5281/zenodo.3518786` (see pages 228, 229).

[Sey+20] Daniel Seybold, Johannes Grohmann, Simon Eismann, Mark Leznik, Samuel Kounev, and Jörg Domaschka. *Data Sets for Measuring and Modeling the Performance Configurations of Distributed DBMS*. en. Zenodo, 2020. DOI: 10.5281/ZENODO.3854996 (see pages 139, 228, 229, 235).

[Sey+19] Daniel Seybold, Moritz Keppler, Daniel Gründler, and Jörg Domaschka. "Mowgli: Finding Your Way in the DBMS Jungle". In: *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. ICPE '19. Mumbai, India: Association for Computing Machinery, 2019, pp. 321–332. DOI: 10.1145/3297663.3310303 (see pages 48, 137, 140, 142, 228, 236).

[Sha+08] Abhishek B. Sharma, Ranjita Bhagwan, Monojit Choudhury, Leana Golubchik, Ramesh Govindan, and Geoffrey M. Voelker. "Automatic Request Categorization in Internet Services". In: *ACM SIGMETRICS Performance Evaluation Review* 36.2 (2008), pp. 16–25. DOI: 10.1145/1453175.1453179 (see page 45).

[Sha+13] Bikash Sharma, Praveen Jayachandran, Akshat Verma, and Chita R. Das. "CloudPD: Problem Determination and Diagnosis in Shared Dynamic Clouds". In: *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. DSN '13. USA: IEEE Computer Society, 2013, pp. 1–12. DOI: 10.1109/DSN.2013.6575298 (see pages 5, 38).

[SBI15] Siqi Shen, Vincent van Beek, and Alexandra Iosup. "Statistical Characterization of Business-Critical Workloads Hosted in Cloud Datacenters". In: *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. CCGRID '15. Shenzhen, China: IEEE Press, 2015, pp. 465–474. DOI: 10.1109/CCGrid.2015.60 (see page 158).

[Sie+15] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. "Performance-Influence Models for Highly Configurable Systems". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, pp. 284–294. DOI: 10.1145/2786805.2786845 (see pages 49, 100).

[Sig+10] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Tech. rep. Google, Inc., 2010 (see page 78).

[SHP15] Kevin Sim, Emma Hart, and Ben Paechter. "A Lifelong Learning Hyper-Heuristic Method for Bin Packing". In: *Evolutionary Computation* 23.1 (2015), pp. 37–67. DOI: 10.1162/evco_a_00121 (see page 47).

[Sin+16] Ravjot Singh, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E. Hassan. "Optimizing the Performance-Related Configurations of Object-Relational Mapping Frameworks Using a Multi-Objective Genetic Algorithm". In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ICPE '16. Delft, The Netherlands: Association for Computing Machinery, 2016, pp. 309–320. DOI: `10.1145/2851553.2851576` (see page 49).

[Sit+01a] Murali Sitaraman, Greg Kulczycki, Joan Krone, William F. Ogden, and A. L. Narasimha Reddy. "Performance Specification of Software Components". In: *Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context*. SSR '01. Toronto, Ontario, Canada: Association for Computing Machinery, 2001, pp. 3–10. DOI: `10.1145/375212.375223` (see page 109).

[Sit+01b] Murali Sitaraman, Greg Kulczycki, Joan Krone, William F. Ogden, and A. L. Narasimha Reddy. "Performance Specification of Software Components". In: *SIGSOFT Software Engineering Notes* 26.3 (2001), pp. 3–10. DOI: `10.1145/379377.375223` (see page 109).

[Smi09] Kate A. Smith-Miles. "Cross-Disciplinary Perspectives on Meta-Learning for Algorithm Selection". In: *ACM Computing Surveys* 41.1 (2009), pp. 1–25. DOI: `10.1145/1456650.1456656` (see pages 41, 47, 100, 248).

[Spi17] Simon Spinner. "Self-Aware Resource Management in Virtualized Data Centers". PhD thesis. Universität Würzburg, 2017 (see pages 17, 23, 46, 91, 92).

[Spi+15] Simon Spinner, Giuliano Casale, Fabian Brosig, and Samuel Kounev. "Evaluating Approaches to Resource Demand Estimation". In: *Performance Evaluation* 92.C (2015), pp. 51–71. DOI: `10.1016/j.peva.2015.07.005` (see pages 23, 43, 45, 46, 91, 92, 103, 109, 188, 218).

[Spi+14] Simon Spinner, Giuliano Casale, Xiaoyun Zhu, and Samuel Kounev. "LibReDE: A Library for Resource Demand Estimation". In: *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*. ICPE '14. Dublin, Ireland: Association for Computing Machinery, 2014, pp. 227–228. DOI: `10.1145/2568088.2576093` (see pages 45, 92, 94).

[Spi+19] Simon Spinner, Johannes Grohmann, Simon Eismann, and Samuel Kounev. "Online model learning for self-aware computing infrastructures". In: *Journal of Systems and Software* 147 (2019), pp. 1–16. DOI: `10.1016/j.jss.2018.09.089` (see pages 5, 23, 40, 93, 96, 109, 110).

[SGK19] Simon Spinner, Johannes Grohmann, and Samuel Kounev. *LibReDE: Library for Resource Demand Estimation*. Standard Performance Evaluation Corporation Research Group (SPEC RG) accepted Tool. `https://research.spec.org/tools/overview/librede.html`. 2019 (see pages 94, 189).

[SWK16] Simon Spinner, Jürgen Walter, and Samuel Kounev. "A Reference Architecture for Online Performance Model Extraction in Virtualized Environments". In: *Companion Publication for ACM/SPEC on International Conference on Performance Engineering*. ICPE '16 Companion. Delft, The Netherlands: Association for Computing Machinery, 2016, pp. 57–62. DOI: `10.1145/2859889.2859893` (see pages 5, 40, 109, 110).

[Sta21a] Statista GmbH. *Global net revenue of Amazon from 2014 to 2020, by product group*. 2021. URL: `https://www.statista.com/statistics/672747/amazons-consolidated-net-revenue-by-segment/` (visited on 07/08/2021) (see pages 1, 2).

[Sta21b] Statista GmbH. *Number of Netflix paid subscribers worldwide from 1st quarter 2013 to 4th quarter 2020*. 2021. URL: `https://www.statista.com/statistics/250934/quarterly-number-of-netflix-streaming-subscribers-worldwide/` (visited on 07/08/2021) (see page 1).

[SKZ07] Christopher Stewart, Terence Kelly, and Alex Zhang. "Exploiting Nonstationarity for Performance Prediction". In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: Association for Computing Machinery, 2007, pp. 31–44. DOI: `10.1145/1272996.1273002` (see page 44).

[Stu08] Student. "The probable error of a mean". In: *Biometrika* (1908), pp. 1–25 (see page 220).

[SJ11] Charles Sutton and Michael I. Jordan. "Bayesian inference for queueing networks and modeling of internet services". In: *The Annals of Applied Statistics* 5.1 (2011), pp. 254–282. DOI: `10.1214/10-AOAS392` (see page 45).

[TAL15] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. "CherryPick: Tracing Packet Trajectory in Software-Defined Datacenter Networks". In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. SOSR '15. Santa Clara, California: Association for Computing Machinery, 2015. DOI: `10.1145/2774993.2775066` (see page 53).

[The92] Henri Theil. "A Multinomial Extension of the Linear Logit Model". In: *Advanced Studies in Theoretical and Applied Econometrics*. Ed. by Baldev Raj and Johan Koerts. Dordrecht: Springer Netherlands, 1992, pp. 181–191. DOI: `10.1007/978-94-011-2546-8_11` (see page 25).

[The+10a] Eno Thereska, Bjoern Doebel, Alice X. Zheng, and Peter Nobel. "Practical Performance Models for Complex, Popular Applications". In: *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '10. New York, New York, USA: Association for Computing Machinery, 2010, pp. 1–12. DOI: `10.1145/1811039.1811041` (see page 42).

[The+10b] Eno Thereska, Bjoern Doebel, Alice X. Zheng, and Peter Nobel. "Practical Performance Models for Complex, Popular Applications". In: *ACM SIGMETRICS Performance Evaluation Review* 38.1 (2010), pp. 1–12. DOI: `10.1145/1811099.1811041` (see page 42).

[Tib96] Robert Tibshirani. "Regression Shrinkage and Selection via the Lasso". In: *Journal of the Royal Statistical Society. Series B* (*Methodological*) 58.1 (1996), pp. 267–288 (see page 25).

[Tru+11] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. "The SCADS Director: Scaling a Distributed Storage System under Stringent Performance Requirements". In: *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*. FAST'11. San Jose, California: USENIX Association, 2011, p. 12. DOI: `10.5555/1960475.1960487` (see page 37).

[Tuk77] John Tukey. *Exploratory Data Analysis*. Addison-Wesley series in behavioral science Bd. 2. Reading, Mass: Addison-Wesley Publishing Company, 1977 (see pages 142, 229).

[Urg+07] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. "Analytic Modeling of Multitier Internet Applications". In: *ACM Transactions on the Web* 1.1 (2007), p. 2. DOI: `10.1145/1232722.1232724` (see page 43).

[Van+17] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. "Automatic Database Management System Tuning Through Large-Scale Machine Learning". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 1009–1024. DOI: `10.1145/3035918.3064029` (see pages 5, 48, 49).

[vOI19] Vincent van Beek, Giorgos Oikonomou, and Alexandru Iosup. "A CPU Contention Predictor for Business-Critical Workloads in Cloud Datacenters". In: *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems* (*FAS* W*). IEEE. 2019, pp. 56–61. DOI: `10.1109/fas-w.2019.00027` (see pages 5, 38).

[VSS12] P. Suresh Varma, Ashwin Satyanarayana, and M. V. Rama Sundari. "Performance analysis of cloud computing using Queuing models". In: *2012 International Conference on Cloud Computing Technologies, Applications and Management* (*ICCCTAM*). IEEE, 2012, pp. 12–15. DOI: `10.1109/iccctam.2012.6488063` (see page 22).

[Vel+21] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. "White-Box Analysis over Machine Learning: Modeling Performance of Configurable Systems". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering* (*ICSE*). IEEE, 2021. DOI: `10.1109/icse43902.2021.00100` (see pages 43, 50).

[Ven+16] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. "Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics". In: *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*. NSDI'16. Santa Clara, CA: USENIX Association, 2016, pp. 363–378. DOI: `10.5555/2930611.2930635` (see page 53).

[Ver21] Verified Market Research. *Cloud Microservices Market 2020 Trends, Market Share, Industry Size, Opportunities, Analysis and Forecast by 2026*. 2021. URL: `https://www.instanttechnews.com/technology-news/2021/02/23/cloud-microservices-market-2020-trends-market-share-industry-size-opportunities-analysis-and-forecast-by-2026/` (visited on 07/08/2021) (see page 2).

[Von+20] Sonya Voneva, Manar Mazkatli, Johannes Grohmann, and Anne Koziolek. "Optimizing Parametric Dependencies for Incremental Performance Model Extraction". In: *Communications in Computer and Information Science*. Vol. 1269. Communications in Computer and Information Science. Springer International Publishing, 2020, pp. 228–240. DOI: `10.1007/978-3-030-59155-7_17` (see pages 42, 111, 208).

[Wal+18] Jürgen Walter, Simon Eismann, Johannes Grohmann, Dušan Okanovic, and Samuel Kounev. "Tools for Declarative Performance Engineering". In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE '18. ACM, 2018, pp. 53–56. DOI: `10.1145/3185768.3185777` (see page 19).

[WHK17] Jürgen Walter, André van Hoorn, and Samuel Kounev. "Automated and Adaptable Decision Support for Software Performance Engineering". In: *Proceedings of the 11th EAI International Conference on Performance Evaluation Methodologies and Tools*. VALUETOOLS 2017. Venice, Italy: Association for Computing Machinery, 2017, pp. 66–73. DOI: `10.1145/3150928.3150952` (see pages 19, 22).

[Wal+17] Jürgen Walter, Christian Stier, Heiko Koziolek, and Samuel Kounev. "An Expandable Extraction Framework for Architectural Performance Models". In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ICPE '17 Companion. L'Aquila, Italy: Association for Computing Machinery, 2017, pp. 165–170. DOI: `10.1145/3053600.3053634` (see pages 5, 23, 40, 96, 109, 110).

[Wal19] Jürgen Christian Walter. "Automation in Software Performance Engineering Based on a Declarative Specification of Concerns". en. PhD thesis. Universität Würzburg, 2019. DOI: `10.25972/OPUS-18090` (see pages 19, 22, 237, 249).

[Wan+18] Ping Wang, Jingmin Xu, Meng Ma, Weilan Lin, Disheng Pan, Yuan Wang, and Pengfei Chen. "CloudRanger: Root Cause Identification for Cloud Native Systems". In: *Proceedings of the 18th IEEE/ACM International Sym-*

posium on Cluster, Cloud and Grid Computing. CCGrid '18. Washington, District of Columbia: IEEE Press, 2018, pp. 492–502. DOI: `10.1109/CCGRID.2018.00076` (see pages 4, 40).

[Wan+12] Wei Wang, Xiang Huang, Xiulei Qin, Wenbo Zhang, Jun Wei, and Hua Zhong. "Application-Level CPU Consumption Estimation: Towards Performance Isolation of Multi-tenancy Web Applications". In: *2012 IEEE Fifth International Conference on Cloud Computing*. Ed. by Rong Chang. IEEE Computer Society, 2012, pp. 439–446. DOI: `10.1109/CLOUD.2012.81` (see pages 44–46, 91, 94).

[Wan+11] Wei Wang, Xiang Huang, Yunkui Song, Wenbo Zhang, Jun Wei, Hua Zhong, and Tao Huang. "A Statistical Approach for Estimating CPU Consumption in Shared Java Middleware Server". In: *2011 IEEE 35th Annual Computer Software and Applications Conference*. IEEE Computer Society, 2011, pp. 541–546. DOI: `10.1109/COMPSAC.2011.75` (see pages 44, 46, 94).

[WC13] Weikun Wang and Giuliano Casale. "Bayesian Service Demand Estimation Using Gibbs Sampling". In: *Proceedings of the 2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*. MASCOTS '13. USA: IEEE Computer Society, 2013, pp. 567–576. DOI: `10.1109/MASCOTS.2013.78` (see page 45).

[WCS16] Weikun Wang, Giuliano Casale, and Charles Sutton. "A Bayesian Approach to Parameter Inference in Queueing Networks". In: *ACM Transactions on Modeling and Computer Simulation* 27.1 (2016), pp. 1–26. DOI: `10.1145/2893480` (see page 45).

[WPC15] Weikun Wang, Juan F. Pérez, and Giuliano Casale. "Filling the Gap: A Tool to Automate Parameter Estimation for Software Performance Models". In: *Proceedings of the 1st International Workshop on Quality-Aware DevOps*. QUDOS 2015. Bergamo, Italy: Association for Computing Machinery, 2015, pp. 31–32. DOI: `10.1145/2804371.2804379` (see pages 45, 92).

[WAS21] Max Weber, Sven Apel, and Norbert Siegmund. "White-Box Performance-Influence Models: A Profiling and Learning Approach (Replication Package)". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2021. DOI: `10.1109/icse-companion52605.2021.00107` (see pages 43, 50).

[Wes+12] Dennis Westermann, Jens Happe, Rouven Krebs, and Roozbeh Farahbod. "Automated Inference of Goal-Oriented Performance Prediction Functions". In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2012. Essen, Germany: Association for Computing Machinery, 2012, pp. 190–199. DOI: `10.1145/2351676.2351703` (see pages 42, 50).

[Wes14] Peter H. Westfall. “Kurtosis as Peakedness, 1905-2014. R.I.P.” In: *The American Statistician* 68.3 (2014), pp. 191–195. DOI: 10.1080/00031305.2014.917055 (see pages 103, 104).

[Wil+15a] Felix Willnecker, Andreas Brunnert, Wolfgang Gottesheim, and Helmut Krcmar. “Using Dynatrace Monitoring Data for Generating Performance Models of Java EE Applications”. In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ICPE ’15. Austin, Texas, USA: Association for Computing Machinery, 2015, pp. 103–104. DOI: 10.1145/2668930.2688061 (see pages 5, 40, 109, 110).

[Wil+15b] Felix Willnecker, Markus Dlugi, Andreas Brunnert, Simon Spinner, Samuel Kounev, Wolfgang Gottesheim, and Helmut Krcmar. “Comparing the Accuracy of Resource Demand Measurement and Estimation Techniques”. In: *Computer Performance Engineering*. Ed. by Marta Beltràn, William Knottenbelt, and Jeremy Bradley. Cham: Springer International Publishing, 2015, pp. 115–129. DOI: 10.1007/978-3-319-23267-6_8 (see pages 43, 91).

[Wol96] David H. Wolpert. “The Lack of a Priori Distinctions between Learning Algorithms”. In: *Neural Computation* 8.7 (1996), pp. 1341–1390. DOI: 10.1162/neco.1996.8.7.1341 (see pages 91, 126, 133, 224).

[WM97] David H. Wolpert and William G. Macready. “No Free Lunch Theorems for Optimization”. In: *IEEE Transactions on Evolutionary Computation* 1.1 (1997), pp. 67–82. DOI: 10.1109/4235.585893 (see pages 46, 91, 224).

[Woo+07] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. “Black-Box and Gray-Box Strategies for Virtual Machine Migration”. In: *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation*. NSDI’07. Cambridge, MA: USENIX Association, 2007, p. 17. DOI: 10.5555/1973430.1973447 (see pages 4, 37).

[Woo+09] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. “Sandpiper: Black-box and gray-box resource management for virtual machines”. In: *Computer Networks* 53.17 (2009). Virtualized Data Centers, pp. 2923–2938. DOI: 10.1016/j.comnet.2009.04.014 (see pages 4, 37).

[Woo+95] C. Murray Woodside, John E. Neilson, Dorina C. Petriu, and Shikharesh Majumdar. “The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software”. In: *IEEE Transactions on Computers* 44.1 (1995), pp. 20–34. DOI: 10.1109/12.368012 (see pages 40, 41, 109).

[Woo21] Murray Woodside. “Performance Models of Event-Driven Architectures”. In: *Companion of the ACM/SPEC International Conference on Performance Engineering*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 145–149. DOI: 10.1145/3447545.3451203 (see page 22).

[Wu+20a] Li Wu, Johan Tordsson, Alexander Acker, and Odej Kao. "MicroRAS: Automatic Recovery in the Absence of Historical Failure Data for Microservice Systems". In: *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2020. DOI: `10.1109/ucc48980.2020.00041` (see page 40).

[Wu+20b] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. "MicroRCA: Root Cause Localization of Performance Issues in Microservices". In: *IEEE/IFIP Network Operations and Management Symposium (NOMS)*. Budapest, Hungary, 2020, pp. 1–9. DOI: `10.1109/noms47738.2020.9110353` (see pages 4, 40).

[WXZ04] Laura Wynter, Cathy H. Xia, and Fan Zhang. "Parameter Inference of Queueing Models for IT Systems Using End-to-End Measurements". In: *ACM SIGMETRICS Performance Evaluation Review* 32.1 (2004), pp. 408–409. DOI: `10.1145/1012888.1005741` (see pages 44–46).

[XFJ16] Bruno Xavier, Tiago Ferreto, and Luis Jersak. "Time Provisioning Evaluation of KVM, Docker and Unikernels in a Cloud Platform". In: *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. CCGRID '16. Cartagena, Columbia: IEEE Press, 2016, pp. 277–280. DOI: `10.1109/CCGrid.2016.86` (see page 60).

[XYD19] Wen Xiong, Kun Yang, and Hao Dai. "Improving NoSQL's Performance Metrics via Machine Learning". In: *2019 Seventh International Conference on Advanced Cloud and Big Data (CBD)*. Los Alamitos, CA, USA: IEEE Computer Society, 2019, pp. 90–95. DOI: `10.1109/CBD.2019.00026` (see pages 49, 137, 138).

[Xu+08] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. "SATzilla: portfolio-based algorithm selection for SAT". In: *Journal of artificial intelligence research* 32 (2008), pp. 565–606 (see page 47).

[Yan+15a] Rerngvit Yanggratoke, Jawwad Ahmed, John Ardelius, Christofer Flinta, Andreas Johnsson, Daniel Gillblad, and Rolf Stadler. "Predicting real-time service-level metrics from device statistics". In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. May. Washington, DC, USA: IEEE Computer Society, 2015, pp. 414–422. DOI: `10.1109/INM.2015.7140318` (see pages 4, 38).

[Yan+15b] Rerngvit Yanggratoke, Jawwad Ahmed, John Ardelius, Christofer Flinta, Andreas Johnsson, Daniel Gillblad, and Rolf Stadler. "Predicting Service Metrics for Cluster-Based Services Using Real-Time Analytics". In: *Proceedings of the 2015 11th International Conference on Network and Service Management (CNSM)*. CNSM '15. USA: IEEE Computer Society, 2015, pp. 135–143. DOI: `10.1109/CNSM.2015.7367349` (see pages 4, 38, 53).

[YUK06] Li Yin, Sandeep Uttamchandani, and Randy Katz. "An Empirical Exploration of Black-Box Performance Models for Storage Systems". In: *Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation*. MASCOTS '06. USA: IEEE Computer Society, 2006, pp. 433–440. DOI: 10.1109/MASCOTS.2006.12 (see page 37).

[YNM16] Dong Young Yoon, Ning Niu, and Barzan Mozafari. "DBSherlock: A Performance Diagnostic Tool for Transactional Databases". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 1599–1614. DOI: 10.1145/2882903.2915218 (see pages 5, 48, 49).

[Zha+02] Li Zhang, Cathy H. Xia, Mark S. Squillante, and W. Nathaniel Mills. "Workload service requirements analysis: a queueing network optimization approach". In: *Proceedings. 10th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*. IEEE Computing Society, 2002, pp. 23–32. DOI: 10.1109/MASCOT.2002.1167057 (see page 44).

[ZCS07] Qi Zhang, Ludmila Cherkasova, and Evgenia Smirni. "A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications". In: *Proceedings of the Fourth International Conference on Autonomic Computing*. ICAC '07. USA: IEEE Computer Society, 2007, p. 27. DOI: 10.1109/ICAC.2007.1 (see pages 22, 44, 45).

[Zha+15] Yi Zhang, Jianmei Guo, Eric Blais, and Krzysztof Czarnecki. "Performance Prediction of Configurable Software Systems by Fourier Learning". In: *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. ASE '15. Lincoln, Nebraska: IEEE Press, 2015, pp. 365–373. DOI: 10.1109/ASE.2015.15 (see pages 49, 100).

[Zhe+19] Ningxin Zheng, Quan Chen, Yong Yang, Jin Li, Wenli Zheng, and Minyi Guo. "POSTER: Precise Capacity Planning for Database Public Clouds". In: *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2019, pp. 457–458. DOI: 10.1109/PACT.2019.00043 (see pages 5, 49, 138).

[ZWL08] Tao Zheng, C. Murray Woodside, and Marin Litoiu. "Performance Model Estimation and Tracking Using Optimal Filters". In: *IEEE Transactions on Software Engineering* 34.3 (2008), pp. 391–406. DOI: 10.1109/TSE.2008.30 (see pages 44, 46, 91, 92, 94).

[Zhe+05] Tao Zheng, Jinmei Yang, Murray Woodside, Marin Litoiu, and Gabriel Iszlai. "Tracking Time-Varying Parameters in Software Systems with Extended Kalman Filters". In: *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON '05. Toranto, Ontario, Canada: IBM Press, 2005, pp. 334–345 (see pages 44, 92).

[Zhe+15] Tao Zheng, Jinmei Yang, Murray Woodside, Marin Litoiu, and Gabriel Iszlai. “Tracking Time-Varying Parameters in Software Systems with Extended Kalman Filters”. In: *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*. CASCON ’15. Markham, Canada: IBM Corp., 2015, p. 13 (see page 44).

[Zho+18] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. “Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study”. In: *IEEE Transactions on Software Engineering* (2018), p. 1. DOI: `10.1109/TSE.2018.2887384` (see pages 4, 40, 171).

[ZL19] Yuqing Zhu and Jianxun Liu. “ClassyTune: A Performance Auto-Tuner for Systems in the Cloud”. In: *IEEE Transactions on Cloud Computing* (2019), pp. 1–1. DOI: `10.1109/TCC.2019.2936567` (see pages 49, 138).

[Zoo20] Zoom Video Communications, Inc. *90-Day Security Plan Progress Report: April 22*. 2020. URL: `https://blog.zoom.us/90-day-security-plan-progress-report-april-22/` (visited on 07/08/2021) (see page 1).

[ZH05] Hui Zou and Trevor Hastie. “Regularization and variable selection via the elastic net”. In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 67.2 (2005), pp. 301–320. DOI: `10.1111/j.1467-9868.2005.00503.x` (see page 25).

[Züf+21] Marwin Züfle, Joachim Agne, Johannes Grohmann, Ibrahim Dörtoluk, and Samuel Kounev. “A Predictive Maintenance Methodology: Predicting the Time-to-Failure of Machines in Industry 4.0”. In: *2021 IEEE 19th International Conference on Industrial Informatics (INDIN)*. IEEE, 2021, pp. 1–8. DOI: `10.1109/INDIN45523.2021.9557387` (see page 249).

[Züf+20] Marwin Züfle, Christian Krupitzer, Florian Erhard, Johannes Grohmann, and Samuel Kounev. “To Fail or Not to Fail: Predicting Hard Disk Drive Failure Time Windows”. In: *Lecture Notes in Computer Science*. MMB 2020. Springer International Publishing, 2020, pp. 19–36. DOI: `10.1007/978-3-030-43024-5_2` (see page 249).