André Bauer

Automated Hybrid Time Series Forecasting: Design, Benchmarking, and Use Cases

Dissertation, Julius-Maximilians-Universität Würzburg
Fakultät für Mathematik und Informatik, 2020

Gutachter: Prof. Dr. Samuel Kounev, Julius-Maximilians-Universität Würzburg, Germany
Prof. Dr. Erik Elmroth, Umeå University, Sweden
Prof. Dr. John Murphy, University College Dublin, Ireland

Datum der mündlichen Prüfung: 22. Dezember 2020

Abstract

These days, we are living in a digitalized world. Both our professional and private lives are pervaded by various IT services, which are typically operated using distributed computing systems (e.g., cloud environments). Due to the high level of digitalization, the operators of such systems are confronted with fast-paced and changing requirements. In particular, cloud environments have to cope with load fluctuations and respective rapid and unexpected changes in the computing resource demands. To face this challenge, so-called auto-scalers, such as the threshold-based mechanism in Amazon Web Services EC2, can be employed to enable elastic scaling of the computing resources. However, despite this opportunity, business-critical applications are still run with highly overprovisioned resources to guarantee a stable and reliable service operation. This strategy is pursued due to the lack of trust in auto-scalers and the concern that inaccurate or delayed adaptations may result in financial losses.

To adapt the resource capacity in time, the future resource demands must be "foreseen", as reacting to changes once they are observed introduces an inherent delay. In other words, accurate forecasting methods are required to adapt systems proactively. A powerful approach in this context is time series forecasting, which is also applied in many other domains. The core idea is to examine past values and predict how these values will evolve as time progresses. According to the "No-Free-Lunch Theorem", there is no algorithm that performs best for all scenarios. Therefore, selecting a suitable forecasting method for a given use case is a crucial task. Simply put, each method has its benefits and drawbacks, depending on the specific use case. The choice of the forecasting method is usually based on expert knowledge, which cannot be fully automated, or on trial-and-error. In both cases, this is expensive and prone to error.

Although auto-scaling and time series forecasting are established research fields, existing approaches cannot fully address the mentioned challenges: (i) In our survey on time series forecasting, we found that publications on time series forecasting typically consider only a small set of (mostly related) methods and evaluate their performance on a small number of time series with only a few error measures while providing no information on the execution time of the studied methods. Therefore, such articles cannot be used to guide the choice

of an appropriate method for a particular use case; (ii) Existing open-source hybrid forecasting methods that take advantage of at least two methods to tackle the "No-Free-Lunch Theorem" are computationally intensive, poorly automated, designed for a particular data set, or they lack a predictable time-to-result. Methods exhibiting a high variance in the time-to-result cannot be applied for time-critical scenarios (e.g., auto-scaling), while methods tailored to a specific data set introduce restrictions on the possible use cases (e.g., forecasting only annual time series); (iii) Auto-scalers typically scale an application either proactively or reactively. Even though some hybrid auto-scalers exist, they lack sophisticated solutions to combine reactive and proactive scaling. For instance, resources are only released proactively while resource allocation is entirely done in a reactive manner (inherently delayed); (iv) The majority of existing mechanisms do not take the provider's pricing scheme into account while scaling an application in a public cloud environment, which often results in excessive charged costs. Even though some cost-aware auto-scalers have been proposed, they only consider the current resource demands, neglecting their development over time. For example, resources are often shut down prematurely, even though they might be required again soon.

To address the mentioned challenges and the shortcomings of existing work, this thesis presents three contributions: (i) The first contribution—a *forecasting benchmark*—addresses the problem of limited comparability between existing forecasting methods; (ii) The second contribution—*Telescope*—provides an automated hybrid time series forecasting method addressing the challenge posed by the "No-Free-Lunch Theorem"; (iii) The third contribution—*Chamulteon*—provides a novel hybrid auto-scaler for coordinated scaling of applications comprising multiple services, leveraging Telescope to forecast the workload intensity as a basis for proactive resource provisioning. In the following, the three contributions of the thesis are summarized:

Contribution I – *Forecasting Benchmark*
To establish a level playing field for evaluating the performance of forecasting methods in a broad setting, we propose a novel benchmark that automatically evaluates and ranks forecasting methods based on their performance in a diverse set of evaluation scenarios. The benchmark comprises four different use cases, each covering 100 heterogeneous time series taken from different domains. The data set was assembled from publicly available time series and was designed to exhibit much higher diversity than existing forecasting competitions. Besides proposing a new data set, we introduce two new measures that describe different aspects of a forecast. We applied the developed benchmark to evaluate Telescope.

Contribution II – *Telescope*
To provide a generic forecasting method, we introduce a novel machine learning-based forecasting approach that automatically retrieves relevant information from a given time series. More precisely, Telescope automatically extracts intrinsic time series features and then decomposes the time series into components, building a forecasting model for each of them. Each component is forecast by applying a different method and then the final forecast is assembled from the forecast components by employing a regression-based machine learning algorithm. In more than 1300 hours of experiments benchmarking 15 competing methods (including approaches from Uber and Facebook) on 400 time series, Telescope outperformed all methods, exhibiting the best forecast accuracy coupled with a low and reliable time-to-result. Compared to the competing methods that exhibited, on average, a forecast error (more precisely, the symmetric mean absolute forecast error) of 29%, Telescope exhibited an error of 20% while being 2556 times faster. In particular, the methods from Uber and Facebook exhibited an error of 48% and 36%, and were 7334 and 19 times slower than Telescope, respectively.

Contribution III – *Chamulteon*
To enable reliable auto-scaling, we present a hybrid auto-scaler that combines proactive and reactive techniques to scale distributed cloud applications comprising multiple services in a coordinated and cost-effective manner. More precisely, proactive adaptations are planned based on forecasts of Telescope, while reactive adaptations are triggered based on actual observations of the monitored load intensity. To solve occurring conflicts between reactive and proactive adaptations, a complex conflict resolution algorithm is implemented. Moreover, when deployed in public cloud environments, Chamulteon reviews adaptations with respect to the cloud provider's pricing scheme in order to minimize the charged costs. In more than 400 hours of experiments evaluating five competing auto-scaling mechanisms in scenarios covering five different workloads, four different applications, and three different cloud environments, Chamulteon exhibited the best auto-scaling performance and reliability while at the same time reducing the charged costs. The competing methods provided insufficient resources for (on average) 31% of the experimental time; in contrast, Chamulteon cut this time to 8% and the SLO (service level objective) violations from 18% to 6% while using up to 15% less resources and reducing the charged costs by up to 45%.

The contributions of this thesis can be seen as major milestones in the domain of time series forecasting and cloud resource management. (i) This thesis is the first to present a forecasting benchmark that covers a variety of different

domains with a high diversity between the analyzed time series. Based on the provided data set and the automatic evaluation procedure, the proposed benchmark contributes to enhance the comparability of forecasting methods. The benchmarking results for different forecasting methods enable the selection of the most appropriate forecasting method for a given use case. (ii) Telescope provides the first generic and fully automated time series forecasting approach that delivers both accurate and reliable forecasts while making no assumptions about the analyzed time series. Hence, it eliminates the need for expensive, time-consuming, and error-prone procedures, such as trial-and-error searches or consulting an expert. This opens up new possibilities especially in time-critical scenarios, where Telescope can provide accurate forecasts with a short and reliable time-to-result.

Although Telescope was applied for this thesis in the field of cloud computing, there is absolutely no limitation regarding the applicability of Telescope in other domains, as demonstrated in the evaluation. Moreover, Telescope, which was made available on GitHub, is already used in a number of interdisciplinary data science projects, for instance, predictive maintenance in an Industry 4.0 context, heart failure prediction in medicine, or as a component of predictive models of beehive development. (iii) In the context of cloud resource management, Chamulteon is a major milestone for increasing the trust in cloud auto-scalers. The complex resolution algorithm enables reliable and accurate scaling behavior that reduces losses caused by excessive resource allocation or SLO violations. In other words, Chamulteon provides reliable online adaptations minimizing charged costs while at the same time maximizing user experience.

Zusammenfassung

Heutzutage leben wir in einer digitalisierten Welt. Sowohl unser berufliches als auch unser privates Leben ist von verschiedenen IT-Diensten durchzogen, welche typischerweise in verteilten Computersystemen (z.B. Cloud-Umgebungen) betrieben werden. Die Betreiber solcher Systeme sind aufgrund des hohen Digitalisierungsgrades mit schnellen und wechselnden Anforderungen konfrontiert. Insbesondere Cloud-Umgebungen unterliegen starken Lastschwankungen und entsprechenden schnellen und unerwarteten Änderungen des Bedarfs an Rechenressourcen. Um dieser Herausforderung zu begegnen, können so genannte Auto-Scaler, wie z.B. der schwellenwertbasierte Mechanismus von Amazon Web Services EC2, eingesetzt werden, um eine elastische Skalierung der Rechenressourcen zu ermöglichen. Doch trotz dieser Gelegenheit werden geschäftskritische Anwendungen nach wie vor mit deutlich überdimensionierten Rechenkapazitäten betrieben, um einen stabilen und zuverlässigen Dienstbetrieb zu gewährleisten. Diese Strategie wird aufgrund des mangelnden Vertrauens in Auto-Scaler und der Sorge verfolgt, dass ungenaue oder verzögerte Anpassungen zu finanziellen Verlusten führen könnten.

Um die Ressourcenkapazität rechtzeitig anpassen zu können, müssen die zukünftigen Ressourcenanforderungen "vorhergesehen" werden. Denn die Reaktion auf Veränderungen, sobald diese beobachtet werden, führt zu einer inhärenten Verzögerung. Mit anderen Worten, es sind genaue Prognosemethoden erforderlich, um Systeme proaktiv anzupassen. Ein wirksamer Ansatz in diesem Zusammenhang ist die Zeitreihenprognose, welche auch in vielen anderen Bereichen angewandt wird. Die Kernidee besteht darin, vergangene Werte zu untersuchen und vorherzusagen, wie sich diese Werte im Laufe der Zeit entwickeln werden. Nach dem "No-Free-Lunch Theorem" gibt es keinen Algorithmus, der für alle Szenarien am besten funktioniert. Daher ist die Auswahl einer geeigneten Prognosemethode für einen gegebenen Anwendungsfall eine wesentliche Herausforderung. Denn jede Methode hat - abhängig vom spezifischen Anwendungsfall - ihre Vor- und Nachteile. Deshalb basiert üblicherweise die Wahl der Prognosemethode auf Trial-and-Error oder auf Expertenwissen, welches nicht vollständig automatisiert werden kann. Beide Ansätze sind teuer und fehleranfällig.

Obwohl Auto-Skalierung und Zeitreihenprognose etablierte Forschungsgebiete sind, können die bestehenden Ansätze die genannten Herausforderungen nicht vollständig bewältigen: (i) Bei unserer Untersuchung zur Zeitreihenvorhersage stellten wir fest, dass die meisten der überprüften Artikel nur eine geringe Anzahl von (meist verwandten) Methoden berücksichtigen und ihre Performanz auf einem kleinen Datensatz von Zeitreihen mit nur wenigen Fehlermaßen bewerten, während sie keine Informationen über die Ausführungszeit der untersuchten Methoden liefern. Daher können solche Artikel nicht als Hilfe für die Wahl einer geeigneten Methode für einen bestimmten Anwendungsfall herangezogen werden; (ii) Bestehende hybride open-source Prognosemethoden, die sich mindestens zwei Methoden zunutze machen, um das "No-Free-Lunch Theorem" anzugehen, sind rechenintensiv, schlecht automatisiert, für einen bestimmten Datensatz ausgelegt oder haben eine unvorhersehbare Laufzeit. Methoden, die eine hohe Varianz in der Ausführungszeit aufweisen, können nicht für zeitkritische Szenarien angewendet werden (z.B. Autoskalierung), während Methoden, die auf einen bestimmten Datensatz zugeschnitten sind, Einschränkungen für mögliche Anwendungsfälle mit sich bringen (z.B. nur jährliche Zeitreihen vorhersagen); (iii) Auto-Scaler skalieren typischerweise eine Anwendung entweder proaktiv oder reaktiv. Obwohl es einige hybride Auto-Scaler gibt, fehlt es ihnen an ausgeklügelten Lösungen zur Kombination von reaktiver und proaktiver Skalierung. Beispielsweise werden Ressourcen nur proaktiv freigesetzt, während die Ressourcenzuweisung vollständig reaktiv (inhärent verzögert) erfolgt; (iv) Die Mehrheit der vorhandenen Mechanismen berücksichtigt bei der Skalierung einer Anwendung in einer öffentlichen Cloud-Umgebung nicht das Preismodell des Anbieters, was häufig zu überhöhten Kosten führt. Auch wenn einige kosteneffiziente Auto-Scaler vorgeschlagen wurden, berücksichtigen sie nur den aktuellen Ressourcenbedarf und vernachlässigen ihre Entwicklung im Laufe der Zeit. Beispielsweise werden Ressourcen oft vorzeitig abgeschaltet, obwohl sie vielleicht bald wieder benötigt werden.

Um den genannten Herausforderungen und den Defiziten der bisherigen Arbeiten zu begegnen, werden in dieser Arbeit drei Beiträge vorgestellt: (i) Der erste Beitrag - ein *Prognosebenchmark* - behandelt das Problem der begrenzten Vergleichbarkeit zwischen bestehenden Prognosemethoden; (ii) Der zweite Beitrag stellt eine automatisierte hybride Zeitreihen-Prognosemethode namens *Telescope* vor, die sich der Herausforderung des "No-Free-Lunch Theorem" stellt; (iii) Der dritte Beitrag stellt *Chamulteon*, einen neuartigen hybriden Auto-Scaler für die koordinierte Skalierung von Anwendungen mit mehreren Diensten, bereit, der Telescope zur Vorhersage der Lastintensität als Grundlage für ei-

ne proaktive Ressourcenbereitstellung nutzt. Im Folgenden werden die drei Beiträge der Arbeit zusammengefasst:

Beitrag I – *Prognosebenchmark*
Um gleiche Ausgangsbedingungen für die Bewertung von Prognosemethoden anhand eines breiten Spektrums zu schaffen, schlagen wir einen neuartigen Benchmark vor, der Prognosemethoden auf der Grundlage ihrer Performanz in einer Vielzahl von Szenarien automatisch bewertet und ein Ranking erstellt. Der Benchmark umfasst vier verschiedene Anwendungsfälle, die jeweils 100 heterogene Zeitreihen aus verschiedenen Bereichen abdecken. Der Datensatz wurde aus öffentlich zugänglichen Zeitreihen zusammengestellt und so konzipiert, dass er eine viel höhere Diversität aufweist als bestehende Prognosewettbewerbe. Neben dem neuen Datensatz führen wir zwei neue Maße ein, die verschiedene Aspekte einer Prognose beschreiben. Wir haben den entwickelten Benchmark zur Bewertung von Telescope angewandt.

Beitrag II – *Telescope*
Um eine generische Prognosemethode bereitzustellen, stellen wir einen neuartigen, auf maschinellem Lernen basierenden Prognoseansatz vor, der automatisch relevante Informationen aus einer gegebenen Zeitreihe extrahiert. Genauer gesagt, Telescope extrahiert automatisch intrinsische Zeitreihenmerkmale und zerlegt die Zeitreihe dann in Komponenten, wobei für jede dieser Komponenten ein Prognosemodell erstellt wird. Jede Komponente wird mit einer anderen Methode prognostiziert und dann wird die endgültige Prognose aus den vorhergesagten Komponenten unter Verwendung eines regressionsbasierten Algorithmus des maschinellen Lernens zusammengestellt. In mehr als 1300 Experiment-Stunden, in denen 15 konkurrierende Methoden (einschließlich Ansätze von Uber und Facebook) auf 400 Zeitreihen verglichen wurden, übertraf Telescope alle Methoden und zeigte die beste Prognosegenauigkeit in Verbindung mit einer niedrigen und zuverlässigen Ausführungszeit. Im Vergleich zu den konkurrierenden Methoden, die im Durchschnitt einen Prognosefehler (genauer gesagt, den symmetric mean absolute forecast error) von 29% aufwiesen, wies Telescope einen Fehler von 20% auf und war dabei 2556 mal schneller. Insbesondere die Methoden von Uber und Facebook wiesen einen Fehler von 48% bzw. 36% auf und waren 7334 bzw. 19 mal langsamer als Telescope.

Beitrag III – *Chamulteon*
Um eine zuverlässige Auto-Skalierung zu ermöglichen, stellen wir einen hybriden Auto-Scaler vor, der proaktive und reaktive Techniken kombiniert, um verteilte Cloud-Anwendungen, die mehrere Dienste umfassen, koordiniert und kostengünstig zu skalieren. Genauer gesagt, werden proaktive Anpassungen

auf der Grundlage von Prognosen von Telescope geplant, während reaktive Anpassungen auf der Grundlage tatsächlicher Beobachtungen der überwachten Lastintensität ausgelöst werden. Um auftretende Konflikte zwischen reaktiven und proaktiven Anpassungen zu lösen, wird ein komplexer Konfliktlösungsalgorithmus implementiert. Außerdem überprüft Chamulteon Anpassungen im Hinblick auf das Preismodell des Cloud-Anbieters, um die anfallenden Kosten in öffentlichen Cloud-Umgebungen zu minimieren. In mehr als 400 Experiment-Stunden, in denen fünf konkurrierende Auto-Skalierungsmechanismen unter fünf verschiedene Arbeitslasten, vier verschiedene Anwendungen und drei verschiedene Cloud-Umgebungen evaluiert wurden, zeigte Chamulteon die beste Auto-Skalierungsleistung und Zuverlässigkeit bei gleichzeitiger Reduzierung der berechneten Kosten. Die konkurrierenden Methoden lieferten während (durchschnittlich) 31% der Versuchszeit zu wenige Ressourcen. Im Gegensatz dazu reduzierte Chamulteon diese Zeit auf 8% und die SLO-Verletzungen (Service Level Objectives) von 18% auf 6%, während es bis zu 15% weniger Ressourcen verwendete und die berechneten Kosten um bis zu 45% senkte.

Die Beiträge dieser Arbeit können als wichtige Meilensteine auf dem Gebiet der Zeitreihenprognose und der automatischen Skalierung in Cloud Computing angesehen werden. (i) In dieser Arbeit wird zum ersten Mal ein Prognosebenchmark präsentiert, der eine Vielzahl verschiedener Bereiche mit einer hohen Diversität zwischen den analysierten Zeitreihen abdeckt. Auf der Grundlage des zur Verfügung gestellten Datensatzes und des automatischen Auswertungsverfahrens trägt der vorgeschlagene Benchmark dazu bei, die Vergleichbarkeit von Prognosemethoden zu verbessern. Die Benchmarking-Ergebnisse von verschiedenen Prognosemethoden ermöglichen die Auswahl der am besten geeigneten Prognosemethode für einen gegebenen Anwendungsfall. (ii) Telescope bietet den ersten generischen und vollautomatischen Zeitreihen-Prognoseansatz, der sowohl genaue als auch zuverlässige Prognosen liefert, ohne Annahmen über die analysierte Zeitreihe zu treffen. Dementsprechend macht es teure, zeitaufwändige und fehleranfällige Verfahren überflüssig, wie z.B. Trial-and-Error oder das Hinzuziehen eines Experten. Dies eröffnet neue Möglichkeiten, insbesondere in zeitkritischen Szenarien, in denen Telescope genaue Vorhersagen mit einer kurzen und zuverlässigen Antwortzeit liefern kann.

Obwohl Telescope für diese Arbeit im Bereich des Cloud Computing eingesetzt wurde, gibt es, wie die Auswertung zeigt, keinerlei Einschränkungen hinsichtlich der Anwendbarkeit von Telescope in anderen Bereichen. Darüber hinaus wird Telescope, das auf GitHub zur Verfügung gestellt wurde, bereits in einer Reihe von interdisziplinären datenwissenschaftlichen Projekten ein-

gesetzt, z.B. bei der vorausschauenden Wartung im Rahmen von Industry 4.0, bei der Vorhersage von Herzinsuffizienz in der Medizin oder als Bestandteil von Vorhersagemodellen für die Entwicklung von Bienenstöcken. (iii) Im Kontext der elastischen Ressourcenverwaltung ist Chamulteon ein wichtiger Meilenstein für die Stärkung des Vertrauens in Auto-Scaler. Der komplexe Konfliktlösungsalgorithmus ermöglicht ein zuverlässiges und genaues Skalierungsverhalten, das Verluste durch übermäßige Ressourcenzuweisung oder SLO-Verletzungen reduziert. Mit anderen Worten, Chamulteon bietet zuverlässige Ressourcenanpassungen, die die berechneten Kosten minimieren und gleichzeitig die Benutzerzufriedenheit maximieren.

Acknowledgments

This thesis would have been impossible without the aid and support of many people. First of all, I would like to thank my advisor Prof. Samuel Kounev. I first met him while working on my master thesis, and since then he always supported me with advice and encouragement on my journey in the academic world. He has been a constant source of inspiration for me in the past years and his strong faith in me supported me to write this thesis and face all research challenges on the way.

From the Software Engineering Chair at the University of Würzburg, I want to thank my current and former colleagues with whom I had the pleasure to work on many projects: Dr. Nikolas Herbst, Dr. Piotr Rygielski, Dr. Simon Spinner, Dr. Jürgen Walter, Dr. Christian Krupitzer, Lukas Beierlieb, Simon Eismann, André Greubel, Johannes Grohmann, Stefan Herrnleben, Lukas Iffländer, Dennis Kaiser, Robert Leppich, Veronika Lesch, Thomas Prantl, Norbert Schmitt, Maximilian Schwinger, Florian Spiess, Martin Sträßer, and Marwin Züfle. I also want to thank Fritz Kleemann and Susanne Stenglin, who have always been very supportive in helping me. And some colleagues even became friends. I would therefore like to thank you for the many hours of laughter and discussion we spent together.

Further, I would like to thank the SPEC RG Cloud working group for their support and feedback leading to many joint publications. Here, I want to thank to Alexandru Iosup, Ahmed Ali-Eldin, Erwin van Eyk, Alexey Ilyushkin, Alessandro Papadopoulos, and more.

I also want to thank Dr. Valentin Curtef (COSMO CONSULT Data Science GmbH) for sharing his experience in data analytics. During our discussions, he always gave insightful comments and feedback.

Contents

Publication List

Peer Reviewed Journal and Magazine Articles

A. Bauer, M. Züfle, N. Herbst, A. Zehe, A. Hotho, and S. Kounev (2020c). "Time Series Forecasting for Self-Aware Systems". In: *Proceedings of the IEEE* 108.7, pp. 1068–1093.

A. Bauer, N. Herbst, S. Spinner, A. Ali-Eldin, and S. Kounev (2018b). "Chameleon: A Hybrid, Proactive Auto-Scaling Mechanism on a Level-Playing Field". In: *IEEE Transactions on Parallel and Distributed Systems* (*TPDS*) 30.4, pp. 800–813.

N. Herbst, A. Bauer, S. Kounev, G. Oikonomou, E. V. Eyk, G. Kousiouris, A. Evangelinou, R. Krebs, T. Brecht, C. L. Abad, and A. Iosup (2018). "Quantifying Cloud Performance and Dependability: Taxonomy, Metric Design, and Emerging Challenges". In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* (*TOMPECS*) 3.4, p. 19.

F. Metzger, T. Hoßfeld, A. Bauer, S. Kounev, and P. E. Heegaard (2019). "Modeling of Aggregated IoT Traffic and its Application to an IoT Cloud". In: *Proceedings of the IEEE* 107.4, pp. 679–694.

A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. von Kistowski, A. Ali-Eldin, C. Abad, J. N. Amaral, P. Tuma, and A. Iosup (2019b). "Methodological Principles for Reproducible Performance Evaluation in Cloud Computing". In: *IEEE Transactions on Software Engineering* (*TSE*).

E. Van Eyk, J. Grohmann, S. Eismann, A. Bauer, L. Versluis, L. Toader, N. Schmitt, N. Herbst, C. Abad, and A. Iosup (2019). "The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms". In: *IEEE Internet Computing* 23.6, pp. 7–18.

A. Ilyushkin, A. Ali-Eldin, N. Herbst, A. Bauer, A. V. Papadopoulos, D. Epema, and A. Iosup (2018). "An Experimental Performance Evaluation of Autoscalers for Complex Workflows". In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* (*TOMPECS*) 3.2, pp. 1–32.

Peer Reviewed Conference Papers

A. Bauer, M. Züfle, N. Herbst, S. Kounev, and V. Curtef (2020b). "Telescope: An Automatic Feature Extraction and Transformation Approach for Time Series Forecasting on a Level-Playing Field". In: *Proceedings of the 36th IEEE International Conference on Data Engineering (ICDE)*. IEEE, pp. 1902–1905.

A. Bauer, M. Züfle, J. Grohmann, N. Schmitt, N. Herbst, and S. Kounev (2020a). "An Automated Forecasting Framework based on Method Recommendation for Seasonal Time Series". In: *Proceedings of the 11th ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, pp. 48–55.

A. Bauer, V. Lesch, L. Versluis, A. Ilyushkin, N. Herbst, and S. Kounev (2019b). "Chamulteon: Coordinated Auto-Scaling of Micro-Services". In: *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE, pp. 2015–2025.

A. Bauer, J. Grohmann, N. Herbst, and S. Kounev (2018a). "On the Value of Service Demand Estimation for Auto-Scaling". In: *Proceedings of the 19th International GI/ITG Conference on Measurement, Modelling and Evaluation of Computing Systems (MMB)*. Springer, pp. 142–156.

M. Zuefle, A. Bauer, V. Lesch, C. Krupitzer, N. Herbst, S. Kounev, and V. Curtef (2019). "Autonomic Forecasting Method Selection: Examination and Ways Ahead". In: *Proceedings of the 16th IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, pp. 167–176.

V. Lesch, A. Bauer, N. Herbst, and S. Kounev (2018). "FOX: Cost-Awareness for Autonomic Resource Management in Public Clouds". In: *Proceedings of the 9th ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, pp. 4–15.

N. Schmitt, L. Iffländer, A. Bauer, and S. Kounev (2019). "Online Power Consumption Estimation for Functions in Cloud Applications". In: *Proceedings of the 16th IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, pp. 63–72.

D. Seybold, S. Volpert, S. Wesner, A. Bauer, N. Herbst, and J. Domaschka (2019). "Kaa: Evaluating Elasticity of Cloud-Hosted DBMS". in: *Proceedings of the 11th IEEE International Conference on Cloud Computing (CloudCom)*. IEEE, pp. 54–61.

J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev (2018). "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research". In: *Proceedings of the 26th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (*MASCOTS*). IEEE, pp. 223–236.

Peer Reviewed Workshops, Tutorial, and Poster Papers

A. Bauer, M. Züfle, N. Herbst, and S. Kounev (2019c). "Best Practices for Time Series Forecasting (Tutorial)". In: *Proceedings of the 4th IEEE International Workshops on Foundations and Applications of Self* Systems* (*FAS* W*). IEEE, pp. 255–256.

A. Bauer, S. Eismann, J. Grohmann, N. Herbst, and S. Kounev (2019a). "Systematic Search for Optimal Resource Configurations of Distributed Applications". In: *Proceedings of the 4th IEEE International Workshops on Foundations and Applications of Self* Systems* (*FAS*W*), pp. 120–125.

A. Bauer, N. Herbst, and S. Kounev (2017). "Design and Evaluation of a Proactive, Application-Aware Auto-Scaler: Tutorial Paper". In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering* (*ICPE*). ACM, pp. 425–428.

J. Grohmann, S. Eismann, A. Bauer, M. Züfle, N. Herbst, and S. Kounev (2019). "Utilizing Clustering to Optimize Resource Demand Estimation Approaches". In: *Proceedings of the 4th IEEE International Workshops on Foundations and Applications of Self* Systems* (*FAS*W*), pp. 134–139.

S. Eismann, J. v. Kistowski, J. Grohmann, A. Bauer, N. Schmitt, N. Herbst, and S. Kounev (2018a). "TeaStore: A Micro-Service Reference Application for Cloud Researchers". (Poster Paper). In: *Proceedings of 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion* (*UCC Companion*). IEEE, pp. 11–12.

Book Chapters

A. Bauer (2019). "Challenges and Approaches: Forecasting for Autonomic Computing". In: *Organic Computing: Doctoral Dissertation Colloquium 2018*. Ed. by S. Tomforde and B. Sick. Vol. 13. kassel university press GmbH, pp. 3–19.

N. Herbst, A. Bauer, and S. Kounev (2020). "Elasticity of Cloud Platforms". In: *Systems Benchmarking: For Scientists and Engineers*. Ed. by S. Kounev, K.-D. Lange, and J. von Kistowski. Springer, pp. 319–340.

Peer Reviewed Extended Abstracts

M. Züfle, A. Bauer, N. Herbst, V. Curtef, and S. Kounev (2017). "Telescope: A Hybrid Forecast Method for Univariate Time Series". In: *Proceedings of the 4th International Work-Conference on Time Series (ITISE)*.

A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. von Kistowski, A. Ali-Eldin, C. L. Abad, J. N. Amaral, P. Tuma, and A. Iosup (2020). "Methodological Principles for Reproducible Performance Evaluation in Cloud Computing". In: *Software Engineering 2020, Fachtagung des GI-Fachbereichs Softwaretechnik*. Ed. by M. Felderer, W. Hasselbring, R. Rabiser, and R. Jung. Vol. P-300. LNI. Gesellschaft für Informatik e.V., pp. 93–94.

Technical Reports

A. Ilyushkin, A. Bauer, A. V. Papadopoulos, E. Deelman, and A. Iosup (2019). *Performance-Feedback Autoscaling with Budget Constraints for Cloud-Based Workloads of Workflows*. Tech. rep. arXiv:1905.10270. arXiv.

A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. von Kistowski, A. Ali-Eldin, C. Abad, J. N. Amaral, P. Tuma, and A. Iosup (2019a). *Methodological Principles for Reproducible Performance Evaluation in Cloud Computing - A SPEC Research Technical Report*. Tech. rep. SPEC-RG-2019-04. SPEC Research Group — Cloud Working Group, Standard Performance Evaluation Corporation (SPEC).

Peer Reviewed Software Artifacts

J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, L. Bui, and S. Kounev (2019). *TeaStore*. Standard Performance Evaluation Corporation (SPEC) Research Group. Accepted tool. URL: `https://research.spec.org/tools/overview/teastore.html`.

Chapter 1

Introduction

Nowadays, our private and professional lives are interwoven with various IT services and IoT (Internet of Things) applications, which are typically realized as distributed computing systems deployed on cloud computing infrastructures. Due to the high level of digitalization, such systems have to cope with fast living and changing requirements as well as vast amounts of data they have to process. The consolidation of 80% of corporate data centers into cloud infrastructures by 2025 (Rosenbush and Loten, 2017), as predicted by Marc Hurd (former Co-CEO of Oracle Corporation), will push these systems to their limit. The high load paired with frequently changing requirements necessitates a constant adaptation of the underlying computing resources. To face this challenge, smart mechanisms for autonomic cloud resource management can be leveraged to ensure a stable and reliable system operation. Such mechanisms typically monitor the system and the underlying infrastructure, triggering adaptations to account for observed or predicted changes in application workloads. As reacting to changes once they are observed introduces an inherent delay, resource management mechanisms need the ability to "foresee" future workloads and respective resource demands in order to be able to determine necessary adaptations in advance. In other words, accurate forecasting methods are needed to enable proactive resource adaptations.

Besides the domain of cloud resource management, accurate and reliable forecasting methods can be useful in many other domains. Examples comprise any capacity planning (e.g., the number of airplane passengers), weather forecasts, business planning, and many more. By nature, humans have the ability called intuition to "foresee" upcoming events. Fascinated by this skill, people have been trying to "foresee" the future for thousands of years. Historical divination methods were, for instance, hieromancy (investigation of entrails), ornithomancy (examining the flight of birds), or pyro-osteomancy (interpreting cracks in bones caused by heat). Over the past decades, both intuition as well as occult rites have increasingly been replaced by statistical analysis. In other words, predictions are based on the examination and statistical analysis

of past observations. Predictions that explicitly include a time component are known as forecasts. The goal of time series forecasting is to examine past values of a given quantity and build a model allowing to predict how these values will evolve as time progresses. In 1927, the first work on the analysis of time series[1] data (Yule, 1927) was published. Forty-three years later, G. Box and G. Jenkins laid the foundations of time series forecasting by publishing their book "Time Series Analysis Forecasting and Control" (Box and Jenkins, 1970). Today, time series forecasting is an established and essential component in many disciplines, providing means to "foresee" how a time series will evolve as time progresses.

The remainder of this chapter is organized as follows: We first identify and discuss open challenges in Section 1.1. As there are different approaches to tackle the identified problems, we highlight the shortcomings of existing work in Section 1.2. Based on the described problems and shortcomings, we present the goals of this thesis and formulate research questions in Section 1.3. According to the goals, we describe the contributions of this thesis in Section 1.4. Finally, the structure of this thesis is outlined in Section 1.5.

1.1 Problem Statement

During the last two decades, the cloud computing paradigm has grown considerably in importance as it addresses the manageability and efficiency of modern distributed computing systems. More precisely, cloud computing enables fast on-demand access to data center resources and offers a high degree of scalability. Although there are easy-to-deploy autonomic resource management mechanisms, such as the threshold-based mechanism available in Amazon Web Services EC2[2], business-critical applications in cloud environments are typically deployed with excessive amounts of resources to guarantee reliable service operation. According to a recent survey (RightScale, 2019), 35% of cloud costs are wasted due to the lack of trust in such mechanisms. The limited adoption of autonomic resource management mechanisms, so-called auto-scalers, in cloud environments can be explained by the fact that such mechanisms are responsible for managing a dynamic trade-off between customer satisfaction and minimization of resource consumption, and they are consequently subject to high operational risk. In other words, inaccurate or delayed adaptations may result in financial losses. To avoid this, auto-scalers require precise time series

[1] A time series represents a collection of observations (e.g., resource consumption, inventory), where each measurement is labeled with a time stamp.

[2] Amazon auto-scaler: `https://aws.amazon.com/autoscaling/`

forecasting techniques so that they can proactively plan and perform necessary adaptations on time.

There are various forecasting methods available in the literature, which can be applied in many different scenarios. Depending on the specific use case, each method has its advantages and disadvantages, which is consistent with the "No-Free-Lunch Theorem" (Wolpert and Macready, 1997). Although this theorem was initially formulated for optimization problems, stating that there is no algorithm best suited for all scenarios, it can also be applied to the domain of time series forecasting: There is no general forecasting method that always performs best for all time series. Consequently, the selection and configuration of the optimal forecasting method for a given time series is a crucial challenge and thus remains to be a mandatory expert task to avoid trial-and-error. However, expert knowledge can be expensive and susceptible to subjective bias. Moreover, it can require a long time before results are available and cannot be fully automated. To eliminate the human component, the end-to-end process of forecasting a time series (i.e., from method selection through data preprocessing to model building and forecasting of future values) needs to be fully automated. On top of this, many real-world scenarios where forecasting is useful (e.g., auto-scaling) have strict requirements for a reliable time-to-result and forecast accuracy.

1.2 Shortcomings of the State-of-the-Art

As forecasting is a powerful tool in many decision-making fields (Hyndman and Athanasopoulos, 2017), time series forecasting is an established and active field of research, and various forecasting methods have been proposed in the literature. To tackle the challenge stated by the "No-Free-Lunch Theorem" and thus avoid the drawbacks of individual methods, different hybrid forecasting methods that take advantage of at least two methods have been proposed. The M4-Competition[3] demonstrates the success of these methods, as 12 of the 17 most accurate methods were hybrid forecasting methods. However, existing open-source hybrid methods (Bergmeir et al., 2016; Cerqueira et al., 2017; Talagala et al., 2018; Taylor and Letham, 2018; Montero-Manso et al., 2020; Smyl, 2020) are computationally intensive, poorly automated, have an unpredictable time-to-result, or are tailored for a given data set. For instance, the winner of the M4-Competition, a hybrid forecasting method developed by Uber (Smyl, 2020), is tuned for the M4-Competition, exhibits a high variance

[3]The M4-Competition, which took place in 2018, was the fourth Makridakis forecasting competition comprising 100,000 time series.

in the time-to-result, and supports only time series with specific frequencies. Due to its unreliable time-to-result, this method is not suitable for time-critical scenarios (e.g., auto-scaling), where the time-to-result for a forecast is subject to strict constraints. Based on our survey (Bauer et al., 2020c), we identified another shortcoming of existing work: Publications on time series forecasting typically consider only a small set of (mostly related) methods and evaluate their performance on a small number of time series with only a few error measures while providing no information on the time-to-result of the studied methods. In other words, the quality of the evaluations suffers due to the limitations of the applied methodology, and therefore existing work fails to provide a reliable approach to guide the choice of an appropriate forecasting method for a particular use case.

Coming back to the challenges of making autonomic cloud resource management mechanisms (i.e., auto-scalers) reliable and trustworthy, independent of the possible use of forecasting methods in this context, auto-scaling itself has also been a popular research topic in recent years. A number of auto-scaling mechanisms have been proposed in the literature that scale resources in a reactive or proactive manner. Both approaches have specific benefits and drawbacks. Proactive mechanisms can adapt resource allocations in advance, but this is typically done based on forecasting the workload intensity, and therefore the quality of such adaptations depends to a large extent on the accuracy of the employed forecasting method. In contrast, reactive mechanisms may eliminate such uncertainties as adaptations, in this case, are based on the actual observed workload intensity, but adaptations may be triggered too late to avoid performance degradation and possible violation of service level objectives (SLOs). Although a recent survey (Qu et al., 2018) emphasizes the importance of combining reactive and proactive scaling, most existing mechanisms are based either on a proactive or a reactive approach. Existing hybrid auto-scalers (Urgaonkar et al., 2008; Iqbal et al., 2011; Ali-Eldin et al., 2012; Jiang et al., 2013; Wu et al., 2016) lack sophisticated solutions to combine both approaches. For instance, resources are only released proactively, while resource allocation is entirely done in a reactive manner (inherently delayed) (Ali-Eldin et al., 2012). Another shortcoming arises when auto-scalers are deployed in public cloud environments. Existing mechanisms typically do not take the provider's pricing scheme into account while scaling resources, which often results in excessive charged costs. Even though some cost-aware auto-scalers have been proposed (Fernandez et al., 2014; Wu et al., 2016; Naskos et al., 2017), they only consider the current resource demands, neglecting their development over time. For example, resources are often shut down prematurely, even though

they might be required again soon.

1.3 Goals and Research Questions of the Thesis

To sum up, existing work on time-series forecasting and cloud auto-scaling suffers from two major problems. First, no fully automated and generic forecasting approach exists that can effectively combine existing forecasting methods in a way to leverage their strengths and avoid their weaknesses, providing accurate forecasts with a reliable time-to-result. Second, existing cloud auto-scalers are distrusted to provide reliable and cost-effective autonomic resource management for modern cloud environments due to the concern that inaccurate or delayed adaptations may result in financial losses. To tackle both problems, we formulate three goals that are addressed within this thesis:

Goal I: *Provide a forecasting benchmark to establish a level playing field for evaluating and comparing the performance of forecasting methods in a broad setting covering a diverse set of evaluation scenarios.*

Goal II: *Provide a fully automated and generic hybrid forecasting method that automatically extracts relevant information from a given time series and uses it to combine existing methods in a way to provide high forecast accuracy coupled with a low time-to-result variance.*

Goal III: *Develop a hybrid auto-scaler enabling the coordinated scaling of applications comprising multiple services by combining proactive scaling (based on the developed forecasting method) with reactive scaling as a fallback mechanism in order to provide maximum reliability of resource adaptations.*

To achieve Goal I, we state the following two research questions:

RQ 1: *How to automatically compare and rank different forecasting methods on a level playing field based on their performance in a diverse set of evaluation scenarios?*

RQ 2: *What are suitable and reliable measures for quantifying the quality of forecasts?*

Similar to the first goal, we divide Goal II into three research questions:

RQ 3: *How to design an automated and generic hybrid forecasting approach that combines different forecasting methods to compensate for the disadvantages of each technique?*

RQ 4: *How to automatically extract and transform features of the considered time series to increase the forecast accuracy?*

RQ 5: *What are appropriate strategies to dynamically apply the most accurate method within the hybrid forecasting approach for a given time series?*

Lastly, we formulate four research questions to address Goal III:

RQ 6: *What is a meaningful combination of proactive and reactive scaling techniques to minimize the risk of auto-scaling in operation?*

RQ 7: *How can scaling decisions be adjusted so that the charged costs in a public cloud environment are minimized?*

RQ 8: *How to enable coordinated scaling of applications comprising multiple services?*

RQ 9: *What are meaningful measures for assessing the quality of coordinated and cost-aware auto-scaling?*

1.4 Contributions of this Thesis

Based on the formulated goals and research questions, this thesis presents three contributions: (i) The first contribution—a *forecasting benchmark*—addresses the problem of limited comparability between existing forecasting methods; (ii) The second contribution—*Telescope*—provides an automated hybrid time series forecasting method addressing the challenge posed by the "No-Free-Lunch Theorem"; (iii) The third contribution—*Chamulteon*—provides a novel hybrid auto-scaler for coordinated scaling of applications comprising multiple services, leveraging Telescope to forecast the workload intensity as a basis for proactive resource provisioning. In the following, the three contributions of the thesis are summarized:

Contribution I – *Forecasting Benchmark*
To address Goal I, we first surveyed existing work on time series forecasting with the goal to assess how forecasting methods are evaluated in the literature or as part of forecasting competitions. To face the shortcomings of existing evaluation approaches, we propose a novel benchmark that automatically evaluates and ranks forecasting methods based on their performance in a diverse set of evaluation scenarios. The benchmark provides a level playing field for evaluating the performance of forecasting methods in a broad setting. It comprises four different use cases, each covering 100

heterogeneous time series taken from different domains (addressing RQ 1). The data set was assembled from publicly available time series and was designed to exhibit much higher diversity than existing forecasting competitions. Besides proposing a new data set, we introduce two new measures that describe different aspects of a forecast (addressing RQ 2).

The idea of the forecasting benchmark was described as part of our paper published in the Proceedings of the IEEE (Bauer et al., 2020c). Moreover, the benchmark was used to evaluate the novel forecasting method Telescope that was developed as part of this thesis.

Contribution II – *Telescope*
To address Goal II, we introduce a novel machine learning-based forecasting approach that automatically retrieves relevant information from a given time series and splits it into parts, handling each of them separately (addressing RQ 3). More precisely, Telescope automatically extracts intrinsic time series features and then decomposes the time series into components, building a forecasting model for each of them (addressing RQ 4). Each component is forecast by applying a different method and then the final forecast is assembled from the forecast components by employing a regression-based machine learning algorithm. For non-time-critical scenarios, we additionally provide an internal recommendation system that can be employed to automatically select the most appropriate machine learning algorithm for assembling the time series from its components (addressing RQ 5).

In more than 1300 hours of experiments benchmarking 15 competing methods (including approaches from Uber and Facebook) on 400 time series, Telescope outperformed all methods, exhibiting the best forecast accuracy coupled with a low and reliable time-to-result. Compared to the competing methods that exhibited, on average, a forecast error (more precisely, the symmetric mean absolute forecast error) of 29%, Telescope exhibited an error of 20% while being 2556 times faster. In particular, the methods from Uber and Facebook exhibited an error of 48% and 36%, and were 7334 and 19 times slower than Telescope, respectively. When additionally applying the recommendation system, Telescope was able to reduce the forecast error even further down to 19%.

Based on this contribution, publications emerged in the Proceedings of the 4^{th} International Work-Conference on Time Series (Züfle et al., 2017), the Proceedings of the 11^{th} ACM/SPEC International Conference on Performance Engineering (Bauer et al., 2020a), the Proceedings of the 36^{th} International Conference on Data Engineering (Bauer et al., 2020b), and the Proceedings of the IEEE (Bauer et al., 2020c).

Contribution III – *Chamulteon*
To address Goal III, we present a hybrid auto-scaler[4] that combines proactive and reactive techniques to scale distributed cloud applications comprising multiple services in a coordinated and cost-effective manner. More precisely, proactive adaptations are planned based on forecasts of Telescope, while reactive adaptations are triggered based on actual observations of the monitored load intensity. To solve occurring conflicts between reactive and proactive adaptations, a complex conflict resolution algorithm is implemented (addressing RQ 6). Moreover, when deployed in public cloud environments, Chamulteon reviews adaptations with respect to the cloud provider's pricing scheme in order to minimize the charged costs (addressing RQ 7). When scaling an application comprising more than one service, Chamulteon considers all services and their associated scaling decisions in conjunction with an application model to scale the application in a coordinated manner (addressing RQ 8), avoiding unintended effects during the scaling, such as oscillations or bottleneck shifting (Urgaonkar et al., 2005). Furthermore, we propose a set of measures to quantify the quality and cost-efficiency of an auto-scaler (addressing RQ 9).

In more than 400 hours of experiments evaluating five competing auto-scaling mechanisms in scenarios covering five different workloads, four different applications, and three different cloud environments, Chamulteon exhibited the best auto-scaling performance and reliability while at the same time reducing the charged costs. The competing methods provided insufficient resources for (on average) 31% of the experimental time; in contrast, Chamulteon cut this time to 8% and the SLO violations from 18% to 6% while using up to 15% less resources and reducing the charged costs by up to 45%.

The different parts of this contribution resulted in publications in the Proceedings of the 19^{th} International GI/ITG Conference on Measurement, Modelling and Evaluation of Computing Systems (Bauer et al., 2018a), the Proceedings of the 9^{th} ACM/SPEC International Conference on Performance Engineering (Lesch et al., 2018), the IEEE Transactions on Parallel and Distributed Systems (Bauer et al., 2018b), the ACM Transactions on Modeling and Performance Evaluation of Computing Systems (Herbst et al., 2018), and the Proceedings of the 39^{th} IEEE International Conference on Distributed Computing Systems (Bauer et al., 2019b).

[4] Chamulteon is based on the auto-scaler Chameleon, which is preliminary work in collaboration with Nikolas Herbst (Herbst, 2018).

The contributions of this thesis can be seen as major milestones in the domain of time series forecasting and cloud resource management. (i) This thesis is the first to present a forecasting benchmark that covers a variety of different domains with a high diversity between the analyzed time series. Based on the provided data set and the automatic evaluation procedure, the proposed benchmark contributes to enhance the comparability of forecasting methods. The benchmarking results for different forecasting methods enable the selection of the most appropriate forecasting method for a given use case. (ii) Telescope provides the first generic and fully automated time series forecasting approach that delivers both accurate and reliable forecasts while making no assumptions about the analyzed time series. Hence, it eliminates the need for expensive, time-consuming, and error-prone procedures, such as trial-and-error searches or consulting an expert. This opens up new possibilities especially in time-critical scenarios, where Telescope can provide accurate forecasts with a short and reliable time-to-result.

Although Telescope was applied for this thesis in the field of cloud computing, there is absolutely no limitation regarding the applicability of Telescope in other domains, as demonstrated in the evaluation. Moreover, Telescope, which was made available on GitHub, is already used in a number of interdisciplinary data science projects, for instance, predictive maintenance in an Industry 4.0 context, heart failure prediction in medicine, or as a component of predictive models of beehive development. (iii) In the context of cloud resource management, Chamulteon is a major milestone for increasing the trust in cloud auto-scalers. The complex resolution algorithm enables reliable and accurate scaling behavior that reduces losses caused by excessive resource allocation or SLO violations. In other words, Chamulteon provides reliable online adaptations minimizing charged costs while at the same time maximizing user experience.

1.5 Thesis Outline

The remainder of this thesis is split into four parts comprising 12 chapters. Part I introduces the foundations for the following parts and reviews the state-of-the-art in Chapter 2–6. Part II addresses the identified problems and shortcomings of existing work by presenting the three contributions of this thesis in Chapter 7–9. Part III benchmarks and evaluates the contributions of this thesis in Chapter 10–11. Finally, Part IV summarizes the thesis and discusses open challenges in Chapter 12–13.

More precisely, in Chapter 2, we focus on time series analysis. We start with relevant terms and definitions, then briefly introduce spectral analysis,

highlight time series feature engineering, and discuss different time series characteristics. Chapter 3 is about time series forecasting. Consequently, we introduce different forecasting methods from different fields and discuss how to assess the forecasting quality. Chapter 4 deals with the resource management of distributed cloud services. To this end, we briefly introduce queuing theory and discuss how to measure the quality of resource adaptations. Chapter 5 surveys related work on time series forecasting, and Chapter 6 reviews the state-of-the-art in the field of cloud auto-scaling. In Chapter 7, we describe our first contribution—a forecasting benchmark. We motivate the need for a forecasting benchmark, give an overview of the proposed design of a benchmark and its use cases, introduce the underlying data set, explain different evaluation options, propose forecast error measures, and compare the proposed benchmark with other forecasting competitions. In Chapter 8, we present our second contribution, an automated hybrid forecasting approach called Telescope. We start with an overview of Telescope's design, introduce the different phases of its forecasting approach, explain the built-in recommendation system, discuss assumptions as well as limitations, and differentiate the approach from related work. Chapter 9 introduces our last contribution called Chamulteon, a forecasting-based auto-scaling mechanism for distributed cloud applications. We start with an overview of Chamulteon's design, explain how scaling decisions are made, discuss how conflicts are resolved, present Chamulteon's cost-aware component called Fox, introduce new elasticity measures for assessing the quality of the auto-scaling process, discuss assumptions as well as limitations, and delimit the approach from related work. In Chapter 10, we evaluate Telescope using the proposed forecasting benchmark. We start with the presentation of the experimental setup, benchmark existing forecasting methods on different use cases of the forecasting benchmark, investigate the quality of the built-in recommendation system, and compare Telescope against other forecasting methods. Chapter 11 investigates the auto-scaling performance of Chamulteon. In this chapter, we introduce the global experimental setup, analyze the impact of service demand estimation for auto-scaling, benchmark Chamulteon against competing auto-scalers for monolithic applications, evaluate the cost-aware component, and benchmark Chamulteon against competing auto-scalers for applications comprising multiple services. In Chapter 12, we summarize the thesis, and in Chapter 13, we discuss open challenges and give an outlook on future work.

Part I

Foundations and State-of-the-Art

Chapter 2

Time Series Analysis

"Information is the oil of the 21st century, and analytics is the combustion engine". This statement from Peter Sondergaard (Gartner Research) reflects the industry's enthusiasm for data. Primarily, tech companies such as Amazon or Google use the collected data to offer personalized services. However, as with oil that cannot be used in its raw form, data must be "refined" to reveal its value. In other words, the data have to be split up and analyzed. For this purpose, different techniques exist. This thesis considers time series analysis that comprises methods to extract meaningful statistics and other data characteristics.

In this chapter, we briefly introduce the basics of time series analysis and different techniques to extract relevant time series-related information. Moreover, we provide the fundamentals for understanding the Chapters 3, 8, 7 and 10. Note that this chapter is not a prerequisite for understanding the other chapters of this thesis. For a detailed introduction to time series analysis, we refer to the books "Forecasting: Principles and Practice" (Hyndman and Athanasopoulos, 2017) or "Time Series Analysis and its Applications" (Shumway and Stoffer, 2000).

The remainder of this chapter is organized as follows: We first introduce the terms and definitions related to time series analysis in Section 2.1. Then, we present the term spectral analysis and its applications in the context of time series in Section 2.2. Afterwards, we outline feature engineering on time series to leverage additional information in Section 2.3. Finally, we highlight different time series characteristics in Section 2.4.

2.1 Terms and Definitions

A *univariate time series* is an ordered collection of values of a quantity obtained over a specific period or since a certain point in time. In general, observations are recorded in successive and equidistant time steps (e.g., hours). Typically, internal patterns exist, such as autocorrelation, trend, or seasonal variation (see

Section 2.1.1). Mathematically, if $y_t \in \mathbb{R}$ is the observed value at time t and T a discrete set of equidistant time points, a univariate time series is defined by

$$Y := \{y_t : t \in T\}. \tag{2.1}$$

Note that sometimes unevenly distributed time series, where observations are made at irregular intervals, are also called univariate time series. However, these are beyond the scope of this thesis (see Chapter 13).

Although the use of univariate time series is common in practice, these time series are subject to the assumption that the future development of this time series depends only on the historical development of the data and not on other effects. In fact, there are time series where the observations also depend on external factors. For example, a household's electricity demand also depends on the weather and if the current day is a working day or not. Consequently, correlated or dependent observations can be stored together to form a *multivariate time series*. That is, there are multiple observed values for each point in time. Indeed, while using a multivariate time series, more information can be used for the forecasting task. Still, most state-of-the-art methods can only handle univariate time series. Not to narrow the spectrum of methods and to cover classical frameworks, we focus in this thesis on univariate time series. Note that if we use the term time series in this thesis's remainder, we always refer to a univariate time series.

2.1.1 Components of a Time Series

A time series can also be seen as a composition of trend, seasonal, cycle, and irregular components (Hyndman and Athanasopoulos, 2017). The long-term development in a time series (i.e., upwards, downwards, or stagnate) is called *trend*. Usually, the trend is a monotonic function unless external events trigger a break and cause a change in the direction. The presence of recurring patterns within a regular period in the time series is called *seasonality*. These patterns are typically caused by climate, customs, or traditional habits such as night and day phases. The length of a seasonal pattern is called frequency[1]. Rises and falls within a time series without a fixed frequency are called *cycles*[2]. In contrast to seasonality, the amplitude and duration of the cycles vary over time. The remaining part of the time series that is not described by trend, seasonality, or cycles is called the *irregular component*. It usually follows a certain statistical

[1] In the context of time series analysis, the term frequency has a different meaning as, for example, in physics.

[2] Sometimes the trend and cycle components are combined, and the resulting trend-cycle component is referred to as trend for simplicity.

noise distribution and is therefore not predictable by most models. Note that there are time series where some components are absent.

As an illustration of the different presence of time series components, Figure 2.1 shows four examples of time series. The time series in the top left corner shows the half-hourly electricity demand in megawatts in England and Wales from June 5 to 10, 2000. The electricity demand exhibits a strong seasonality and no trend or cycles. The time series in the top right corner reflects the annual numbers of lynx trappings from 1821 to 1934 in Canada. Although the lynx population seems to be seasonal, this time series is cyclic as the width and the amplitude of the peaks vary over time. The time series in the bottom left corner represents the monthly atmospheric concentrations of CO_2 in parts per million from 1959 to 1997. The CO_2 concentrations show both a strong trend and seasonality. The time series in the bottom right illustrates the price changes of GOOG's closing stock prices from the NASDAQ exchange from February 26 to December 22, 2013. In contrast to the other time series, this time series is irregular and exhibits no trend, seasonal, nor cyclic component.

In general, the relationship between the components forming a time series are either *additive* or *multiplicative*. In the case there is a additive relationship, the time series can be written as

$$Y(t) := T(t) + S(t) + C(t) + I(t) \tag{2.2}$$

with $T(t)$ being the trend component, $S(t)$ the seasonal component, $C(t)$ the cycle component, and $I(t)$ the irregular component. Alternatively, the time series can be written as

$$Y(t) := T(t) \cdot S(t) \cdot C(t) \cdot I(t). \tag{2.3}$$

Simply spoken, the type of relationship between the components can be determined by the seasonal pattern or the variation around the trend. More precisely, if the amplitude of the seasonal pattern or the variation evolves with the trend, the relationship between the components is multiplicative. Otherwise, the relationship is additive.

Figure 2.2 shows an example for both types. The upper time series shows the monthly number of passengers of international airlines from 1949 to 1960. In this example, the components have a multiplicative relationship because the annual pattern's amplitude increases with the growing trend. The lower time series reflects the average monthly milk production per cow from 1962 to 1975. In this example, the annual pattern's amplitude remains the same despite the rising trend, indicating an additive relationship between the components. Note that hybrid forms of the relationship between the components are also possible.

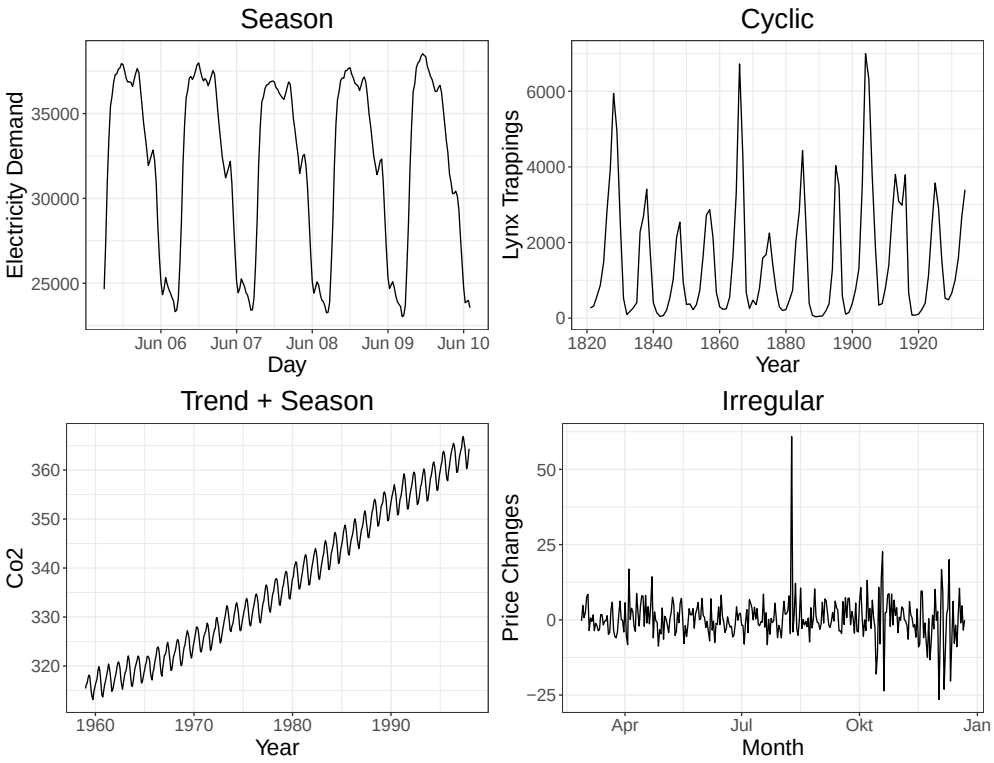

Figure 2.1: Examples of time series with varying components.

2.1.2 Statistical Analysis of Time Series

An important part of *time series analysis* is to examine the observed data and capture the characteristics of the process or the phenomenon generating the time series. In theory, the observations of a phenomenon over time should result in a *deterministic* time series (Makridakis, 1976). That is, there should be no measurement errors/uncertainties, and the generating process should not be affected by external disturbances. In practice, however, these assumptions mostly do not apply. The output of the generating process is mixed with white noise (errors or randomness with zero mean and the values are not correlated with each other), leading to a *stochastic* time series (Makridakis, 1976).

Consequently, a time series can be seen as a part of a random process of the variables $y_1, \ldots, y_n$ sampled at the equidistant time stamps $t_1, \ldots, t_n$ with n being a positive integer and can be written as

$$y_t := x_t + u_t, \tag{2.4}$$

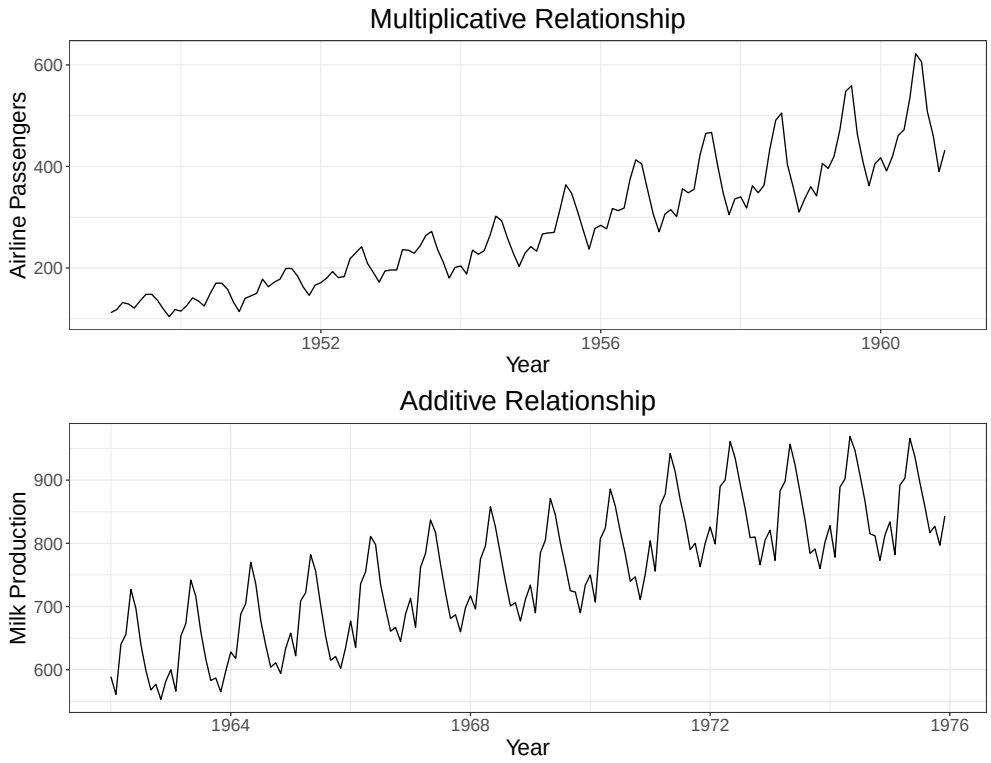

Figure 2.2: Examples for multiplicative and additive relationship between time series components.

where x_t is generated by the process representing the time series and u_t is a white noise term (Makridakis, 1976). Further, a time series can be described as a *joint cumulative distribution function*

$$F_{t_1,\ldots,t_n}(c_1,\ldots,c_n) := P(y_{t_1} \leq c_1,\ldots,y_{t_n} \leq c_n), \tag{2.5}$$

where the values of the time series are jointly less than the constants $c_1,\ldots,c_n$ with n being a positive integer (Shumway and Stoffer, 2000).

2.1.3 Stationarity

One of the most important characteristics of a time series is the *stationarity* since most forecasting methods assume that the time series is either stationary or can be "stationarized". Loosely speaking, the statistical properties (such as mean, variance, and auto-covariance) of a stationary time series do not change

over time. More formally, a stationary time series is shift-invariant in terms of time or mathematically expressed as

$$P(y_{t_1} \leq c_1, \ldots, y_{t_s} \leq c_s) = P(y_{t_{1+\tau}} \leq c_1, \ldots, y_{t_{s+\tau}} \leq c_s) \tag{2.6}$$

for all $s \in \mathbb{N}$, all time stamps $t_1, \ldots, t_s$, all constants $c_1, \ldots, c_s$, and all time-shifts $\tau \in \mathbb{Z}$ (Shumway and Stoffer, 2000). That is, the probabilistic behavior of every subset of the time series is identical to its "time-shifted" subset. For testing whether or not a time series is stationary, a *unit root test*, for instance, the Phillips–Perron test (Phillips and Perron, 1988), can be applied.

As an illustration of stationary and non-stationary time series, Figure2.3 shows three examples of time series. In each plot, the horizontal axis shows the index and the vertical axis the respective observation. The time series in the top plot shows a white noise time series. This time series is stationary since the observations are not correlated and there is no time dependence of the statistical properties. The time series in the second plot is based on the first time series, but additionally exhibits a trend. As the trend introduces a time-dependent mean, the time series non-stationary. The time series in the last plot is also based on the first time series, but the variance was modified. Consequently, the variance is time-dependent, which also leads to a non-stationary time series.

In practice, time series are usually showing a mix of trend and/or seasonal patterns and are thus non-stationary (Adhikari and Agrawal, 2013). To this end, time series are transformed (see Section 2.3.2), seasonally adjusted, made difference-stationary by possibly repeated differencing (see Section 2.3.3), or made trend-stationary by removing the trend.

2.1.4 Time Series Forecasting

The goal of time series forecasting is to examine a time series and predict how this time series will evolve as time progresses. In other words, a mathematical model - dependent on the forecasting method (see Chapter 3) - is build for the plausible description of the observed data. Then, the future development of the time series is estimated based on this model. More formally, we define the forecast for the time $n + k$ based on the historical observations $y_1, \ldots, y_n$ as

$$\hat{y}_{n+k|n} := f(y_n, \ldots, y_1, k) \tag{2.7}$$

with n and k being positive integers and f being the forecasting method capturing the time series model. While performing the forecast, it can be distinguished between *one-step-ahead* and *multi-step-ahead* forecasting. As the name indicates, when performing a one-step-ahead forecast, only the next value

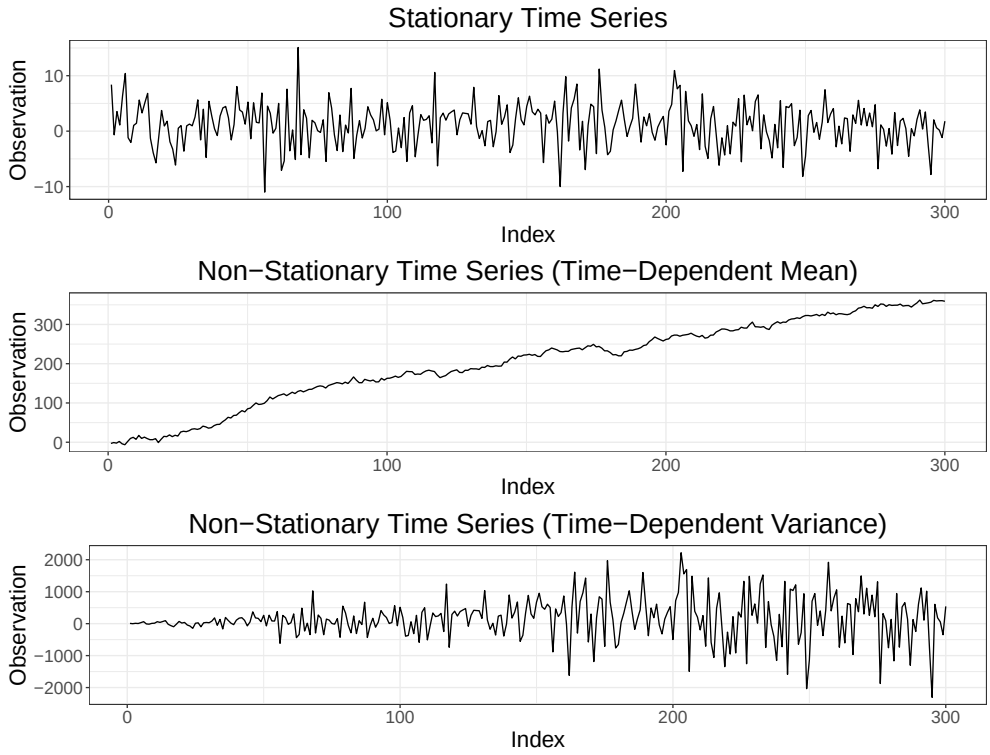

Figure 2.3: Examples for stationary and non-stationary time series.

$\hat{y}_{n+1|n}$ is forecast, that is, $k = 1$. In terms of a multi-step-ahead forecast, the values $\hat{y}_{n+1|n}, \ldots, \hat{y}_{n+h|n}$ are forecast, where h is the *forecast horizon* and represents the number of values that are forecast based on the historical data. In other words, the h values are forecast at once without updating the model with new data.

2.2 Spectral Analysis

In many fields, especially for forecasting, it helps discover the underlying periodicities in the data, that is, knowing the frequencies or the lengths of the seasonal patterns. For instance, if the most dominant frequency is unknown for a given time series, the time series cannot be decomposed (see Section 2.3.1). In this context, the dominant frequency means the most common period, respectively, the seasonal pattern, such as days in a year. Also, if the dominant frequency is available, the information on the next dominant frequency (e.g.,

the week within the year) is helpful. A standard method for this data analysis is the *spectral analysis*, also referred to as analysis in the frequency domain. The key idea is to transform the time series from the time domain to the frequency domain as the spectrum reveals the data's underlying frequencies.

2.2.1 Fourier Terms

Following the discovery of J. Fourier (Fourier, 1822) and the basic principle in mathematics to break down complex objects into more simpler parts, a time series can be approximated by a linear combination of sinusoid terms. More formally, for a time series $y_1, \ldots, y_n$ with uncorrelated zero-mean random variables a_j and b_j, the time series can be expressed as

$$y_t := a_0 + \sum_{j=1}^{\frac{n-1}{2}} \left(a_j \cos\left(2\pi t \frac{j}{n}\right) + b_j \sin\left(2\pi t \frac{j}{n}\right) \right) \tag{2.8}$$

if n is odd. Otherwise, we can write the time series as

$$y_t := a_0 + a_{\frac{n}{2}}(-1)^t + \sum_{j=1}^{\frac{n}{2}-1} \left(a_j \cos\left(2\pi t \frac{j}{n}\right) + b_j \sin\left(2\pi t \frac{j}{n}\right) \right). \tag{2.9}$$

The resulting representation is referred to as *Fourier series* and the sinusoids terms are called *Fourier terms*. The Fourier terms allow, on the one hand, to create an artificial time series based on the relevant terms that resembles the actual time series. On the other hand, the relevant Fourier terms can be used as additional information to fit a better model describing the time series. The fraction $\frac{j}{n}$ can be interpreted as j repeating patterns over n points or as frequency of the cosines/sines function. In this context, the frequencies are referred to as *Fourier frequencies*. These frequencies allow to characterize a time series while looking at the their associated magnitudes. The coefficients a_j and b_j are called *Fourier coefficients*.

2.2.2 Frequency Detection via Periodograms

According to the Fourier analysis, any time series can be decomposed into a number of discrete frequencies, or a spectrum of frequencies over a continuous range. The resulting *spectral density* assigns an intensity to each frequency within the time series. In other words, if a time series is viewed in the form of a frequency spectrum, information such as intrinsic periodic signals are revealed.

An estimate of the spectral density is the *periodogram* (Schuster, 1899). That is, in a time series that is driven by certain seasonal patterns, the periodogram shows peaks at precisely those frequencies. Moreover, the periodogram is based on the *discrete Fourier transformation*. To this end, the first step is to calculate the discrete Fourier transformation

$$\begin{aligned} X(\nu_j) &:= \frac{1}{\sqrt{n}} \sum_{t=1}^{n} e^{-2\pi i t \nu_j} y_t \\ &= \frac{1}{\sqrt{n}} \left(\sum_{t=1}^{n} \cos(2\pi t \nu_j) y_t - i \sum_{t=1}^{n} \sin(2\pi t \nu_j) y_t \right) \end{aligned} \tag{2.10}$$

of the time series $y_1, \ldots, y_n$, where $\nu_j = \frac{j}{n}$ are the Fourier frequencies. Then, the periodogram is calculated as squared modulus of the discrete Fourier transformation as follows

$$\begin{aligned} I(\nu_j) &:= |X(\nu_j)|^2 \\ &= \frac{1}{n} \left| \sum_{t=1}^{n} e^{-2\pi i t \nu_j} y_t \right|^2 \\ &= \frac{1}{n} \left(\left(\sum_{t=1}^{n} \cos(2\pi t \nu_j) y_t \right)^2 + \left(\sum_{t=1}^{n} \sin(2\pi t \nu_j) y_t \right)^2 \right). \end{aligned} \tag{2.11}$$

Large values of $I(\nu_j)$ reveal predominant frequencies within the time series, whereas small values may indicate noise.

Two examples of outcomes of the periodogram are depicted in Figure 2.4. In both plots, the horizontal axis shows the frequencies and the vertical axis the spectrum. The top periodogram exhibits the spectral density of the airline time series (see Section 2.1.1). The most dominant frequency in this example is 12 (the highest spectral value), and the multiples of 12 also show high peaks. At the bottom of the figure, the periodogram of a white noise time series is depicted. In theory, the spectral density should be constant across all frequencies, because a white noise time series is like white light, which is a uniform mixture of all frequencies in the visible spectrum. In this figure, the periodogram is less "noisy". However, it can be seen that many frequencies have almost the same spectrum.

2.3 Time Series Feature Engineering

"At the end of the day, some machine learning projects succeed and some fail. What makes the difference? Easily the most important factor is the features

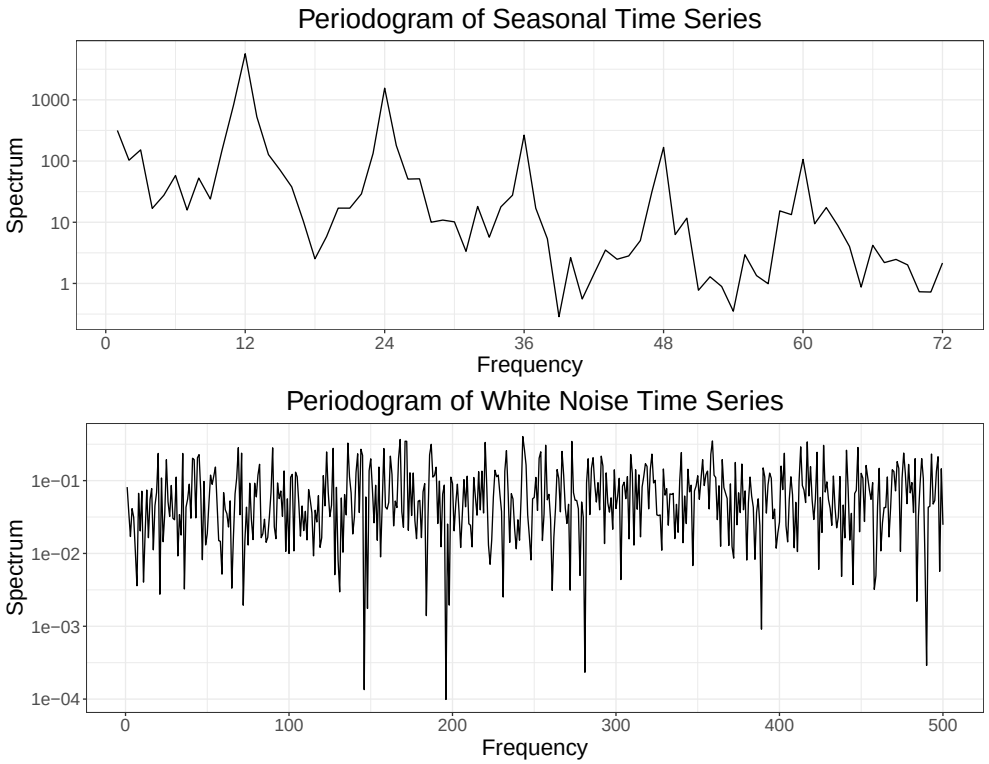

Figure 2.4: Examples of periodograms for a time series with dominant frequency of 12 and a white noise time series.

used" (Domingos, 2012, p. 5). This quote's essence is that the key to success in forecasting and classification is solid feature engineering. By *feature*, we mean an individual measurable property or characteristic of an observed or extracted phenomenon. Simply spoken, a feature is a piece of information (e.g., the seasonal pattern of a time series) that can help model the time series.

2.3.1 Time Series Decomposition

As a time series consists of different components (see Section 2.1.1), a common approach is to break down the time series into its components. The parts can either be used to modify the data (e.g., removing the trend or seasonality) or be used as intrinsic features for augmenting a model capturing the time series.

The "classical" time series decomposition technique uses moving averages with the window size (i.e., the number of observations for the average) of the

seasonal pattern's length to smooth the seasonal influences for calculating the trend. However, while using this approach, the first and last $w/2$ observations for the trend would be unavailable when using a moving average with a window size of w. Further, classical decomposition approaches assume that the seasonal component does not change over time. While this assumption is usually true for short time series, it is less likely in longer time series. An improvement of the classical decomposition is the *X-11 method* (Bell and Hillmer, 1984). This method considers all observations to estimate the trend and can handle slow changes in the seasonal pattern. Based on this method, *X-12* and *X-13* were developed (Dagum and Bianconcini, 2016). However, these methods are designed to handle only time series with monthly or quarterly observations.

A common method that overcomes the limitations of the methods mentioned above is *STL* (*S*easonal and *T*rend decomposition using *L*oess) (Cleveland et al., 1990). STL can handle any type of seasonality, allows the seasonal pattern to change over time, and disassembles the given time series $Y(t)$ into the components trend $T(t)$, season $S(t)$, and irregular $I(t)$. More formally, the resulting decomposition of STL is

$$Y(t) := T(t) + S(t) + I(t). \tag{2.12}$$

Although STL can only handle additive relationships between the components, it can also be applied to multiplicative time series by using the natural logarithm[3] (i.e., logarithm to basis e). More formally, a multiplicative time series

$$Y(t) = T(t) \cdot S(t) \cdot I(t) \tag{2.13}$$

is equivalent to (Hyndman and Athanasopoulos, 2017)

$$\begin{aligned} \ln Y(t) &= \ln\left(T(t) \cdot S(t) \cdot I(t)\right) \\ &= \ln T(t) + \ln S(t) + \ln I(t). \end{aligned} \tag{2.14}$$

Figure 2.5 depicts exemplarily the decomposition of the milk time series (see Section 2.1.1) based on STL. In each plot, the horizontal axis shows the time in years. The vertical axis in the top plot reflects the milk consumption, and the vertical axis in each of the remaining plots shows the range in which the component contributes to the original time series. The second plot exhibits the trend component that reflects the increased milk production over the years. The third plot shows the seasonal component with an annual pattern covering every 12 monthly observations. The plot at the bottom of the figure shows the irregular part. This part is also called the remainder of the time series after the trend and season components are removed.

[3]Note that the logarithm can only be applied if the time series has only values greater than 0.

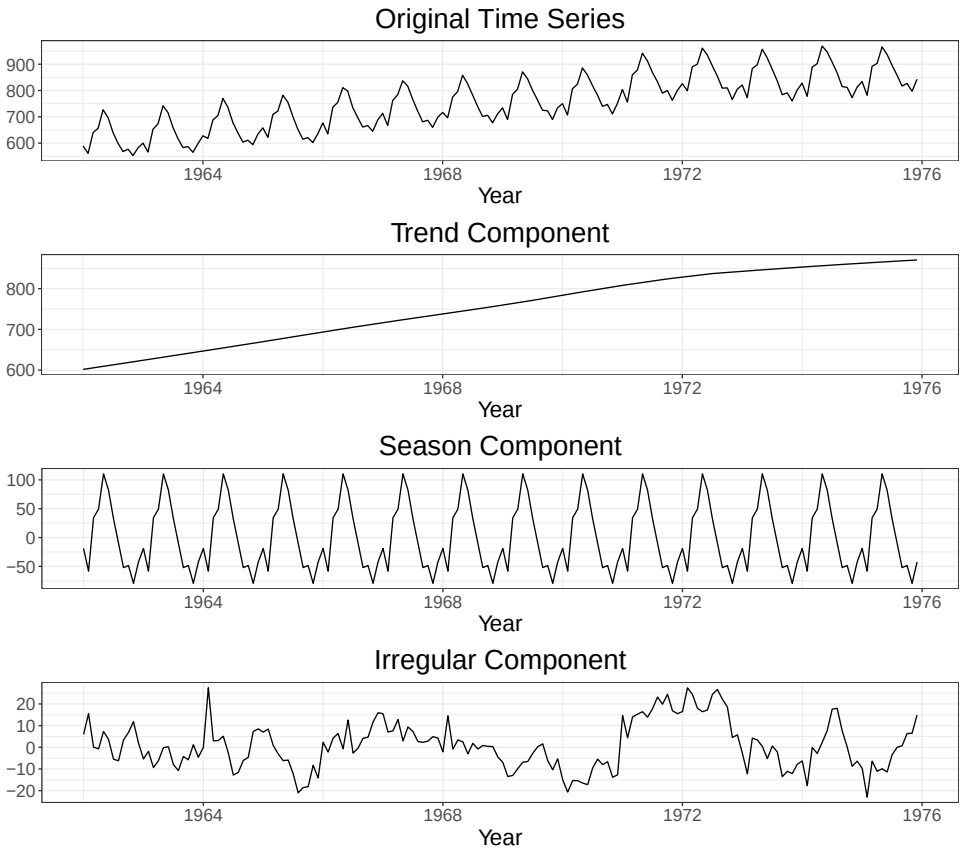

Figure 2.5: Example of STL decomposition.

After applying STL, we can use the components to de-trend or de-seasonalize a time series by removing the the associated component. That is, the *de-trended time series* can be written as

$$Y^T(t) := S(t) + I(t) \tag{2.15}$$

or in the case of multiplicative relationship as

$$\ln Y^T(t) := \ln S(t) + \ln I(t). \tag{2.16}$$

Analogously, the *de-seasonalized time series* can be defined as

$$Y^S(t) := T(t) + I(t) \tag{2.17}$$

or in the case of multiplicative relationship as

$$\ln Y^S(t) := \ln T(t) + \ln I(t). \tag{2.18}$$

Besides the decomposition methods mentioned above (trend, seasonality, and irregular), there are different decomposition models and methods (such as the forecasting method from Facebook (Taylor and Letham, 2018), which decomposes a time series into trend, seasonality, holiday effects, and irregular) leading to diverse components (e.g., season, trend, cycle, events, linear part & non-linear part, ...) and different types of relationship between the components.

2.3.2 Time Series Transformation

As observed data may be quite complex, for example, having high variance and/or multiplicative relationship between the components, an adjustment or simplification of it can improve the forecasting model (Hyndman and Athanasopoulos, 2017). To this end, there are different methods that transform time series. Daily life examples are currency exchange rates (e.g., Euro into US dollar). However, this example is a linear transformation and does not affect the data complexity. More precisely, the type of the distribution is not changed. In practice, non-linear transformations are used. For instance, a common and useful transformation is to apply the logarithm as it stabilizes the variance and eliminates (or reduces) multiplicative effects. Although this method may improve the forecasting model, the transformed data may not be normally distributed, so the improvement may not reach its full potential. The *Box-Cox transformation* (Box and Cox, 1964) tries to transform the data into "normal shape"[4]. To this end, this transformation offers the natural logarithm and power transformations. The Box-Cox transformation is defined as follows

$$w_t := \begin{cases} \ln y_t & \text{if } \lambda = 0, \\ (y_t^\lambda - 1)/\lambda & \text{otherwise,} \end{cases} \tag{2.19}$$

where y_t is the original time series and λ the transformation parameter that determines the function.

Figure 2.6 shows the effect of applying the Box-Cox transformation. The top left corner plot shows the monthly mean relative sunspot numbers from 1749 to 1983, while the bottom left plot depicts the time series after the Box-Cox transformation. The histogram in the top right corner reflects the distribution of the

[4]The advantage of a normal distribution is that mean, median and mode are identical. Furthermore, the distribution can be described by using only mean and variance.

sunspot numbers, while the thick black curve represents a normal distribution based on the statistical characteristics of the data. Analogously, the histogram in the bottom right corner displays the distribution of the transformed time series. After the transformation, the time series exhibits almost a normal distribution compared to the original time series.

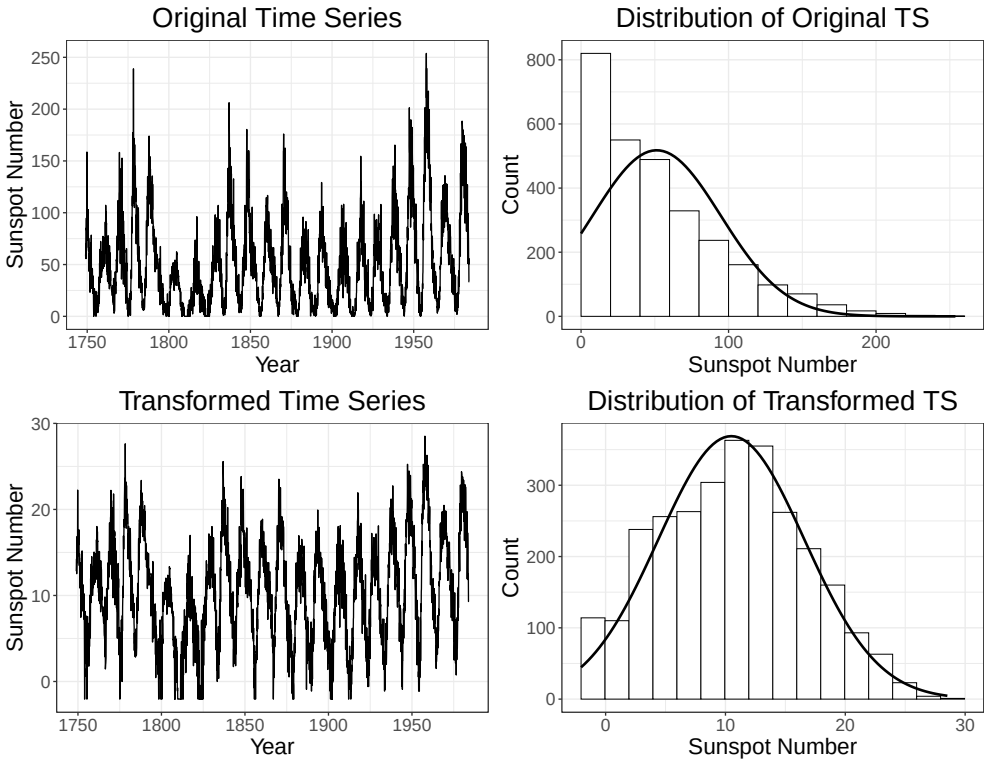

Figure 2.6: Example of Box-Cox transformation.

Note that if a forecast was conducted based on this transformation, the forecast values have to be re-transformed using the same λ to be in the right scale. Consequently, the re-transformation of the time series is defined as

$$y_t := \begin{cases} e^{w_t} & \lambda = 0, \\ (\lambda w_t + 1)^{1/\lambda} & \text{otherwise.} \end{cases} \tag{2.20}$$

As the transformation and therefore the accuracy of the forecasting model depends on the the transformation parameter λ, V. M. Guerrero (Guerrero, 1993) proposes a method that estimates the best λ by minimizing the coefficient of variation for the time series.

2.3.3 Time Series Differencing

Besides stabilizing the variance of a time series by applying, for instance, the logarithm or Box-Cox transformation, differencing a time series can stabilize the mean by eliminating (or reducing) seasonal or trend effects. Moreover, the differentiation of a time series provides useful information, just like the differential calculus in mathematical analysis. A (first-order) *differenced time series* is the change between succeeding observations in the original series and can mathematically be expressed as

$$y'_t := y_t - y_{t-1} \tag{2.21}$$

for $t > 1$. This mathematical expression is also called the *first lag* of the time series. Analogous to the mathematical analysis, a time series can be differenced d times, leading to a d-order differencing. For example, if we want to investigate how observations change compared to observations from a previous season, the seasonal lag can be investigated. More precisely, the time series is differenced m times, where m is the length of the seasonal pattern.

In general, each lag or d-order differencing can be calculated with the *back-shift operator*. To this end, we first define the back-shift operator B as

$$By_t := y_{t-1} \tag{2.22}$$

and

$$B^d y_t := y_{t-d}. \tag{2.23}$$

Then, the d-order differencing of a time series can be expressed as

$$\begin{aligned} y_t^d : &= (1-B)^d y_t \\ &= y_t \sum_{j=0}^{d} \binom{d}{j} 1^{d-j} (-B)^j. \end{aligned} \tag{2.24}$$

For example, the second-order differenced time series $y_t^2 = y_t - 2y_{t-y} + y_{t-2}$.

2.4 Time Series Characteristics

Besides statistical characteristics (mean, standard deviation, skewness, etc.) and the length or frequency of the time series, additional characteristics can be used to describe a time series. Over the last years, numerous time series characteristics have been proposed (Wang et al., 2009; Lemke and Gabrys,

2010b; Fulcher et al., 2013; Hyndman et al., 2015; Kang et al., 2017; Talagala et al., 2018). Consequently, we highlight in the following only the time series characteristics that are used in this thesis:

- *Strength of trend component* and *strength of season component*: The characteristics measure the degree of the trend or seasonality within a time series. The strength of the trend component is measured by

$$1 - \frac{var(I(t))}{var(Y^S(t))} \tag{2.25}$$

and strength of the season component is calculated as

$$1 - \frac{var(I(t))}{var(Y^T(t))}, \tag{2.26}$$

where $Y^S(t)$ is the de-seasonalized time series, $Y^T(t)$ the de-trended time series, and $I(t)$ the irregular component of the time series.

- *Stability* and *lumpiness*: For calculating these characteristics, the time series $y_1, \ldots, y_n$ is broken down into non-overlapping sub-time series

$$(y_1, \ldots, y_l), (y_{l+1}, \ldots, y_{2l}), \ldots, (y_{(\lceil \frac{n}{l} \rceil - 1) * l + 1}, \ldots, y_n) \tag{2.27}$$

with l being a positive integer. For each of these sub-time series, the means and variances are calculated. Then, the stability is the variance of the means of the sub-time series and the lumpiness the variance of the variances of the sub-time series.

- *Spectral entropy*: This characteristic reflects the "forecastability" of the time series. For instance, a white-noise time series, which cannot be forecast, exhibits the highest value due to its flat spectral density (see Section 2.2.2). Consequently, the lower the value, the better the "forecastability". The spectral entropy is computed as

$$-\int_{-\pi}^{\pi} \hat{f}(\lambda) \log \hat{f}(\lambda) d\lambda, \tag{2.28}$$

where $\hat{f}(\lambda)$ is an estimate of the spectral density (Nuttall and Carter, 1982) of the time series.

- *Spikiness*: For calculating this characteristic, only the irregular component of the time series is considered. While one value is left out at a time, the variance of the irregular component's remaining values is calculated. Then, the spikiness is the variance of all leave-one-out variances.

- *Self-similarity*: This characteristic measures how similar the time series is to a part of itself and is expressed by the *Hurst exponent* (Rose, 1996; Willinger et al., 1998). A good estimation for the Hurst exponent fits an autoregressive fractionally integrated moving average (ARFIMA)[5]. More precisely, the Hurst exponent is computed as 0.5 plus the maximum likelihood estimate of the fractional differencing order of the ARFIMA model (Haslett and Raftery, 1989).

- *Non-linearity*: This characteristic measures the degree of the non-linearity of the time series. To determine the degree, Teräsvirta's neural network test for non-linearity is applied (Teräsvirta et al., 1993).

- *Stationarity test statistics*: For testing the stationarity or trend-stationarity of a time series, the Phillips–Perron unit root test (Phillips and Perron, 1988) or Kwiatkowski–Phillips–Schmidt–Shin unit root test (Kwiatkowski et al., 1992) can be conducted, respectively.

- *Auto-correlation coefficients*: This characteristic measures the linear relationship between lagged values of a time series. For a lag k, the k^{th} auto-correlation coefficent r_k reflects the relationship between the observations y_t and y_{t-k} or mathematically expressed as

$$r_k := \frac{\sum_{t=k+1}^{n} (y_t - \bar{y})(y_{t-k} - \bar{y})}{\sum_{t=1}^{n} (y_t - \bar{y})^2}, \tag{2.29}$$

  where $\bar{y}$ is the mean of the time series, n the length of the time series, and k a positive integer (Hyndman and Athanasopoulos, 2017). For instance, the first auto-correlation coefficient of the time series, the differenced time series, or the irregular component provide interesting insights.

- *Partial auto-correlation coefficients*: Similar to the auto-correlation with the exception that for r_k the effects of the lags $1, \ldots, k-1$ are removed.

[5] ARFIMA replaces the parameter of the integrated part of the ARIMA model (see Section 3.1.4) by a non-integer value.

Chapter 3

Time Series Forecasting

Time series forecasting aims to examine past values of a given quantity and build a model allowing to predict how these values will evolve as time progresses. In other words, a forecasting method is applied to describe the historical development of the data and to predict future values. However, selecting a suitable forecasting method for a given scenario is challenging as the forecast accuracy heavily depends on the forecasting method. According to the "No-Free-Lunch Theorem" (Wolpert and Macready, 1997), there is no single forecasting method that works best for all possible time series. Therefore, there are various forecasting methods available in the literature. Depending on the specific use case, each method has its advantages and disadvantages.

In this chapter, we briefly present different state-of-the-art forecasting methods and how to assess the quality of the forecasts. The content of the following sections is the basis for understanding the Chapters 8, 7 and 10. Note that this chapter is not a prerequisite for understanding the other chapters of this thesis. More details on the different classical forecasting methods and forecast error measures can be found in the book "Forecasting: Principles and Practice" (Hyndman and Athanasopoulos, 2017). For further information regarding the machine learning methods, we refer to the books "An Introduction to Statistical Learning" (James et al., 2013) and "Applied Predictive Modeling" (Kuhn and Johnson, 2013)

The remainder of this chapter is organized as follows: We first introduce classical forecasting methods in Section 3.1. Afterwards, we present forecasting methods based on machine learning (see Section 3.2). Finally, we outline how to quantify the forecast accuracy in Section 3.3.

3.1 Classical Forecasting Methods

The forecasting methods described in this section are based on statistical techniques. They require only a time series as input for fitting a model describing the generating process of the time series.

3.1.1 Naïve and sNaïve

The *Naïve* forecast is the simplest way of forecasting future values based on historical data. This method repeats the latest observation for the entire forecasting horizon. Mathematically, the forecast for the time $n + k$ is defined by

$$\hat{y}_{n+k|n} := y_n, \tag{3.1}$$

where n is the number of historical observations, y_n the latest observation, and with k being a positive integer. *sNaïve* is an extension of *Naïve* that integrates a seasonal pattern. Unlike Naïve, the forecast of sNaïve does not correspond to the last value, instead each forecast value is equal to the corresponding observation from the last period. For instance, the forecast for January is identical to the observation made in January last year. More formally, the forecast for the time $n + k$ is calculated as

$$\hat{y}_{n+k|n} := y_{n+k-m\cdot(r+1)}, \tag{3.2}$$

where m is the length of the seasonal pattern (i.e., the number of observations within the period), and r the integer part of $\frac{k-1}{m}$ (Hyndman and Athanasopoulos, 2017). Due to their simplicity, these methods are typically used as baseline methods.

3.1.2 ETS

Many successful forecasting methods are based on the idea of exponential smoothing, that is, using weighted averages of past observations. The simplest and original version is called *Simple Exponential Smoothing* (Brown, 1956) and its one-step-ahead forecast is defined by

$$\hat{y}_{n+1|n} = \alpha y_n + \alpha(1-\alpha)y_{n-1} + \alpha(1-\alpha)^2 y_{n-2} + \cdots, \tag{3.3}$$

where $y_1, \ldots, y_n$ are the historical observations and $0 \leq \alpha \leq 1$ denotes the smoothing factor. Since this method does not take the trend or seasonal component into account, different extensions (Holt, 1957; Winters, 1960) have been proposed over the years. These methods use different ways for combining the components: either additive, multiplicative, or absent. Due to a large number of possible configurations of the methods, a framework (Hyndman et al., 2002) for the automatic selection of the most appropriate method for a given time series was introduced. This framework takes the trend and seasonal component into account, introduces a new component called error, and is thus labeled as

ETS for *E*rror, *T*rend, *S*eason. The options for the components are additive or multiplicative for the error component; additive or absent for the trend component; and absent, additive, or multiplicative for the seasonal component. For example, the simple exponential smoothing has an additive error component, while the trend and season component are absent.

3.1.3 Theta

The idea of the *Theta* method (Assimakopoulos and Nikolopoulos, 2000) is to modify the local curvature of the time series by applying the θ coefficient to the second-order differenced time series. More precisely, the so-called θ-*line* $Z(\theta)$ is determined by solving the equation

$$(1 - B)^2 Z_t(\theta) = \theta(1 - B)^2 y_t, \tag{3.4}$$

where B is the back-shift operator (see Section 2.3.3), $\theta \in \mathbb{R}$, and y_t the observation at time t. A small value of θ (i.e., $\theta < 1$), which represents the long term component, flattens the time series and a high value (i.e., $\theta > 1$), which stands for the short term component, increases the curvature. In the case $\theta = 0$ or $\theta = 1$, the time series is transformed into a linear regression line or the θ-line is equal to the original time series, respectively. The forecasting procedure of the Theta model as suggested by the authors comprises the following steps: (i) Testing whether the time series is seasonal and if so, de-seasonalize the time series by assuming a multiplicative relationship; (ii) decomposing the time series into the θ-lines $Z(0)$ and $Z(2)$; (iii) extrapolating $Z(0)$ and forecasting $Z(2)$ with simple exponential smoothing; (iv) combining the forecasts of $Z(0)$ and $Z(2)$ with equal weights; and finally (v) re-seasonalizing the forecast if the time series is seasonal. Although the proposed model contains only two θ-lines, more lines can be added to extract more information from the data.

3.1.4 ARIMA and sARIMA

Besides exponential smoothing, *ARIMA* (Box and Jenkins, 1970) models are also often used for forecasting. The acronym ARIMA refers to *A*uto*R*egressive *I*ntegrated *M*oving *A*verage. Simply said, an ARIMA model consists of an *autoregressive* $AR(p)$ model and a *moving average* $MA(q)$ model. As the term autoregressive indicates, the observed variable is represented as a linear combi-

nation of its past values. The $AR(p)$ model, where the order p determines the number of past values, can be written as

$$y_t = c + \epsilon_t + \sum_{i=1}^{p} \varphi_i \cdot y_{t-i}, \tag{3.5}$$

where $y_{t-p}, \ldots, y_{t-1}$ are the values considered in the past, $\varphi_1, \ldots, \varphi_p$ the weights of the combination, ϵ_t white noise (i.e., random vector with zero mean, finite variance, and statistical independence), c a constant term, and p being a positive integer. In contrast to the autoregressive model, the moving average $MA(q)$ model uses the past forecast errors to represent the observed value as a linear combination. Here, the order q determines the number of past errors. The $MA(q)$ can be formulated as

$$y_t = c + \epsilon_t + \sum_{j=1}^{q} \Theta_j \cdot \epsilon_{t-j}, \tag{3.6}$$

where $\epsilon_{t-p}, \ldots \epsilon_t$ are the considered forecast (white noise) errors, $\Theta_1, \ldots, \Theta_p$ the weights of the combination, c a constant term, and q being a positive integer. The combination of an $AR(p)$ and $MA(q)$ model is called *ARMA* model. In contrast to the ARMA (Wold, 1938) model, an *ARIMA* model relaxes the requirement for stationary time series through differencing (see Section 2.3.3) that is represented by the 'I' for integrated. To sum up, an ARIMA model can be described by the order p of the AR model, the order q of the MA model, the degree d of the differencing and can be written as

$$(1 - \sum_{i=1}^{p} \varphi_i B^i)(1 - B)^d y_t = c + (1 + \sum_{j=1}^{q} \Theta_j B^j)\epsilon_t, \tag{3.7}$$

where B is the back-shift operator (see Section 2.3.3), c a constant term. In this equation, the first parentheses reflects the $AR(p)$ model, the second parentheses the d differences, and the last parentheses the $MA(q)$ model. An version of ARIMA that is capable of modeling seasonal patterns is *sARIMA*. That is, each non-seasonal component of the ARIMA model is extended with its seasonal counterpart. In addition to the parameters of the ARIMA model, the sARIMA model is specified by the parameters of the seasonal components (P, Q, and D) and the m number of observations per season.

3.1.5 TBATS

Due to its bad performance in detecting complex seasonal patterns, an extension (Livera et al., 2011) of ETS was introduced. This extension comprises

Fourier Terms (see Section 2.2.1), Box-Cox transformations (see Section 2.3.2), and ARMA error corrections. More precisely, each seasonal patterns within a time series is captured by a trigonometric representation based on Fourier Terms, the Box-Cox transformations are used to handle non-linearity, and the ARMA error captures the autocorrelation within the residuals. This extension is called *TBATS* and the acronym stands for the key features: *T*rigonometric, *B*ox-Cox transformed, *A*RMA errors, *T*rend and *S*easonal components.

3.2 Forecasting Methods based on Machine Learning

The methods described in this section are based on regression techniques. Besides the time series, the methods require additional information/features (e.g., time stamps, seasonal patterns, or lags of the time series) to capture the time series' generating process.

3.2.1 CART

The *CART* (Breiman et al., 1984) is a binary tree and its acronym stands for *C*lassification *A*nd *R*egression *T*rees. For building the tree, the data set is divided recursively into two subsets until a predefined termination criterion (e.g., a minimum count of training samples for each leaf node) is met. This procedure is also called *recursive partitioning*. Since a regression tree is created in the case of the time series forecast, the splitting criterion is to find a feature and an associated split value that minimizes the overall sums of errors of the two subsets. In other words, for each value of each feature, the (sub-)data set is divided, and the split with the least error is performed. After the tree is grown, the tree may be too large and be prone to overfitting. In such a case, the tree is *pruned* back to a "smaller" tree.

3.2.2 Evtree

As recursive partitioning trees optimize only per split, the grown trees are only locally optimal. To approach this issue and grow a globally optimal tree, *evtree* (Grubinger et al., 2011) uses an evolutionary algorithm for finding the best partitioning for each split while considering all splits. A population of trees is generated in the first step, with each tree consisting of only a random split (i.e., the feature and the split value are selected randomly). In each iteration, each tree is altered with one of the following actions: *split*, *prune*, *major split rule mutation*, *minor split rule mutation*, and *crossover*. Split selects a random leaf node and performs a further random split to create two new leaf nodes as

successors. Prune performs exactly the opposite and removes two leaf nodes. The split rule generation either changes the feature (major) or the split value (minor) of a split randomly. Crossover swaps two randomly selected sub-trees. For the survivor selection, each tree competes with its most similar offspring to have a fixed population size.

3.2.3 Cubist

Cubist is a rule-based regression method that builds upon the *M5* model tree (Quinlan et al., 1992; Quinlan, 1993). In contrast to classical decision trees that have a single value at each leaf node, Cubist has a multivariate linear regression model at each leaf node. To build a tree structure, rules are defined to partition the data set and then arranged hierarchically. In other words, each branch in the tree is a series of "if-then"-rules, and each rule has an associated multivariate linear model. More formally, each rule has the form if *condition* then *regress* else *apply the next rule*, where the condition can comprise only one feature or a number of features. If a number of features satisfy a rule's condition, the associated linear model is used to predict the value. Otherwise, the next rule is checked and so on.

3.2.4 Random Forest

Random forest (Ho, 1995; Breiman, 2001) is an ensemble method for classification or regression that consists of multiple decision trees. In terms of time series forecasting, random forest consists of an ensemble of regression trees, and the prediction is the average of each tree's output. The advantage of an ensemble of decision trees is the correction of a single tree's proneness to overfit to its training set. However, if the data set has one dominant feature and the other features are less important, most or all trees within the ensemble use this dominant feature in their top split. Consequently, all trees are quite similar and thus highly correlated. To counter this similarity within the ensemble, random forests de-correlates the trees. More precisely, the method trains - as done by an ordinary ensemble - a number of decision trees on the bootstrapped[1] training samples, but for each split, in each tree, only a random sample of the features is considered. In other words, the decision trees within the ensemble focus not only on dominant features but also on features that would not be selected for the top split.

[1] A bootstrapped training sample is a random sample with replacement from the original set (Efron, 1992).

3.2.5 XGBoost

XGBoost (e*X*treme *G*radient *B*oosting) (Chen and Guestrin, 2016) is an ensemble of decision trees based on gradient boosting. In contrast to bagging methods like random forest, the trees are grown sequentially in the boosting approach. Each tree is grown on a modified version of the original data set while using the previously grown trees' information. More precisely, each tree learns from its predecessors and updates the residual errors. That is, each subsequent tree is fit to the current residuals instead of the target. The newly grown tree is then added to the fitted function to update the residuals. This procedure is repeated until the accuracy is no longer improved. Each tree is kept to a minimum to approach the dependence on past trees. Also, the contribution of each tree to the final model is unequally weighted. To reduce overfitting, XGBoost penalizes complex models through regularization objects and applies shrinkage, which reduces each tree's influence to leave space for future trees, and feature subsampling.

3.2.6 SVM and SVR

*SVM*s (Support Vector Machines) (Vapnik, 1995) are typically used for classification and pattern recognition. For example, in binary classification, the basic idea of SVM is to find a linear separator that partitions the data into two classes. More precisely, the separation line is fit so that the margin between the line and the borderline cases is maximized. In other words, the training samples are represented by their feature vectors in high-dimensional space; and the SVM is trained to find a line where all samples from one class are on one side and all other samples are on the other side. Since it is not always possible to separate the two classes linearly, the *kernel trick* (Boser et al., 1992) can be used to transform the feature vector into a higher-dimensional space where the data is linearly separable. To apply the principle of SVM to the prediction of numerical values, *SVR* (Support Vector Regression) (Drucker et al., 1997) was proposed. SVR modifies SVM by introducing a threshold that defines a margin of acceptable errors for numerical predictions.

3.2.7 NNetar

NNetar is a feed-forward neural network and the acronym stands for *N*eural *NET*work *A*uto*R*egression. The model is trained with lagged values (i.e., back-shifted values) of a time series. The network consists of one hidden layer and the number of lags l as well as the number of nodes in the hidden layer n are

automatically selected leading to l–n–1 network (Hyndman et al., 2018). More formally, the model of the time series can be written as

$$y_t = f(\Phi_l(y_{t-1})) + \epsilon_t, \tag{3.8}$$

where ϵ_t is the error at time t, f is the neural network with one hidden layer, and $\Phi_l(y_{t-1}) = (y_{t-1}, \ldots, y_{t-l})$ is a vector containing l lagged values.

3.3 Assessing Forecasting Quality

In principle, a forecast can either be evaluated *a-priori* or *a-posteriori*. In the case of an a-posteriori evaluation, the forecast's accuracy can only be quantified once the future values are available because the data is usually absent at the time of the forecast. In contrast, in an a-priori evaluation, the forecasting method is assessed before the actual forecast is carried out. For this reason, an estimator is required for the forecast accuracy. The straightforward solution is to use the error of the model fitting as an indicator for the forecast accuracy. However, this approach is unreliable, for example, due to overfitting. In other words, a method could perfectly match historical data, but as a result, the method loses its predictability and is not able to capture future values. To tackle such issues, V. Vapnik (Vapnik, 1995) introduces a complex theory that allows setting an upper bound for the forecast error. Expressed in a very simplified manner, this theory implies that

$$\text{forecast error} < \text{model error} + \text{structural risk}. \tag{3.9}$$

In other words, the forecast error is limited by the model error of the fitted model and the theoretical risk that considers the model complexity due to the accompanying overfitting threat. In fact, it is difficult to assess the risk because the methods are quite sophisticated or work like black boxes. Consequently, a more common practice is to split the time series into a *training set* and *test set* as illustrated in Figure 3.1. The training set (blue dashed curve) is used to estimate the parameters of a forecasting method to fit the model to the data. Based on this model, a forecast (red curve) is performed and then compared against the test set (green dashed curve). Since the test data are not used for model fitting, this practice should provide a reliable indicator. Typically, the first 80% of a time series is used as the training set, and the remaining 20% is used for evaluation, that is, as the test set. Note that we use both forecast accuracy and forecast error as terms for quantifying forecasts: the lower the error or the higher the accuracy, the better the forecast. According to R. Hyndman and

G. Athanasopoulos (Hyndman and Athanasopoulos, 2017), there are three types of error measures: (i) *Scale-dependent error measures*, (ii) *percentage error measures*, and (iii) *scaled error measures*.

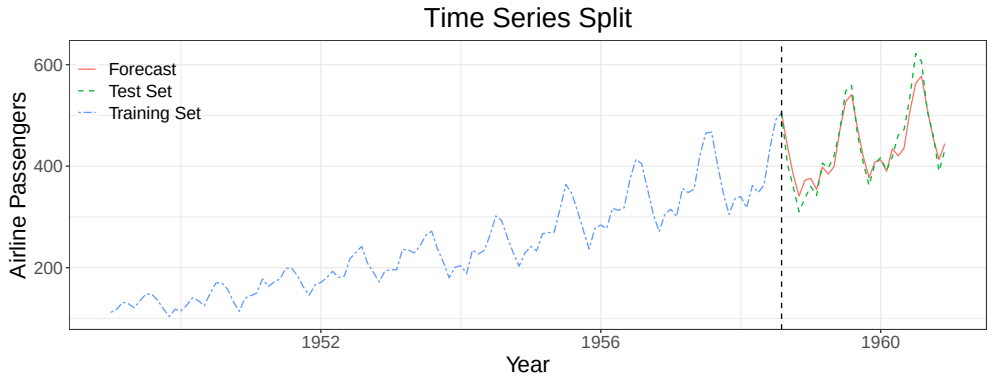

Figure 3.1: Example of a 80%–20% split of a time series.

3.3.1 Scale-dependent Error Measures

The advantage of scale-dependent error measures is that the calculated error has the same scale as the data. In other words, the interpretation of the results is intuitive when comparing different methods on time series with the same scale. However, these measures cannot be used to compare forecasting methods across time series that have different scales. Examples are the *mean forecast error* (MFE), *mean absolute error* (MAE), or *root mean squared error* (RMSE). Formally, these measure are defined as follows

$$\text{MFE} := \frac{1}{k}\sum_{t=1}^{k} y_t - \hat{y}_t, \tag{3.10}$$

$$\text{MAE} := \frac{1}{k}\sum_{t=1}^{k} |y_t - \hat{y}_t|, \tag{3.11}$$

$$\text{RMSE} := \sqrt{\frac{1}{k}\sum_{t=1}^{k} (y_t - \hat{y}_t)^2}, \tag{3.12}$$

where k is the forecast horizon (i.e., the length of the forecast), y_t the actual value at time t, and $\hat{y}_t$ the forecast value at time t.

3.3.2 Percentage Error Measures

On the one hand, percentage error measures are scale-independent and thus can be used to compare forecasting methods across different time series. On the other hand, the forecast error is infinite or undefined if the actual value is zero. Outliers also have a significant influence on the forecast error. Prominent examples are the *mean absolute percentage error* (MAPE), *symmetric mean absolute percentage error* (sMAPE), and the *root mean square percentage error*, which are calculated as follows

$$\text{MAPE} := \frac{100\%}{k} \sum_{t=1}^{k} \frac{|y_t - \hat{y}_t|}{|y_t|}, \tag{3.13}$$

$$\text{sMAPE} := \frac{200\%}{k} \sum_{t=1}^{k} \frac{|y_t - \hat{y}_t|}{|y_t + \hat{y}_t|}, \tag{3.14}$$

$$\text{RMSPE} := \sqrt{\frac{1}{k} \sum_{t=1}^{k} \left(\frac{100\% \cdot |y_t - \hat{y}_t|}{|y_t|} \right)^2}, \tag{3.15}$$

where k is the forecast horizon, y_t the actual value at time t, and $\hat{y}_t$ the forecast value at time t.

3.3.3 Scaled Error Measures

To approach the problem of dividing by the actual value, as occurs with percentage-based error measures, the scaled error measures normalize the forecast error by a baseline. As a result of the normalization, the measure becomes scale-independent. In other words, these measures can be used to compare forecasts across time series that have different scales. However, if the baseline has values that are equal to each other, the forecast error is not defined as a division by zero has to be performed. Examples are the *mean absolute scaled error* (MASE) or *root mean square scaled error* (RMSSE). Mathematically, both measures are defined as follows

$$\text{MASE} := \frac{\frac{1}{k} \sum_{t=1}^{k} |y_t - \hat{y}_t|}{\frac{1}{n-m} \sum_{i=m+1}^{n} |h_i - h_{i-m}|}, \tag{3.16}$$

$$\text{RMSSE} := \sqrt{\frac{1}{k} \sum_{t=1}^{k} \left(\frac{|y_t - \hat{y}_t|}{\frac{1}{n-m} \sum_{i=m+1}^{n} |h_i - h_{i-m}|} \right)^2}, \tag{3.17}$$

where k is the forecast horizon, y_t the actual value at time t, $\hat{y}_t$ the forecast value at time t, m the length of the period ($m = 1$ for non-seasonal time series), n the length of the history, and h_i the historical values at time i.

3.3.4 Discussion of the Measures

There exist different error measures for evaluating forecasting methods. Each measure has its use cases, benefits, and drawbacks. For instance, the MFE shows the error direction while the RMSE does not, or the MAPE and sMAPE do not penalize extreme values, but are scale-independent. A more detailed distinction between different error measures and corresponding discussions can be found in the works of M. Shcherbakov et al. (Shcherbakov et al., 2013), R. Adhikari and R. Agrawal (Adhikari and Agrawal, 2013), or R. Hyndman and A. Koehler (Hyndman and Koehler, 2006). In general, it is impossible to prove the correctness of a measure; instead, it is a joint agreement on how to quantify the given property. To counter the weaknesses of a specific error measure, it is advantageous to take more than one of these measures into account when evaluating forecasts. Since different measures allow different insights and thus, a better understanding of the forecast can be obtained.

Chapter 4

Resource Management of Distributed Cloud Services

Nowadays, various IT services are typically realized as distributed computing systems deployed on cloud computing infrastructures. As a result of this popularity, cloud environments have to cope with load fluctuations and respective rapid and unexpected changes in the computing resource demands. To guarantee a reliably operating service, the computing resource capacity must be adapted according to the time-varying load.

In this chapter, we briefly present the basics of queueing theory, which is a useful technique for capacity planning, and discuss how to assess the quality of resource adaptations in cloud environments. Moreover, we provide the fundamentals for understanding the Chapter 9 and 11. Note that this chapter is not a prerequisite for understanding the other chapters of this thesis. An in-depth treatment of queueing theory can be found in the book "Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications" (Bolch et al., 2006). For further information on cloud elasticity, we refer to the book "Systems Benchmarking: For Scientists and Engineers" (Kounev et al., 2020).

The remainder of this chapter is organized as follows: We start with queueing theory. More precisely, we briefly introduce the characteristics of a queue in Section 4.1.1 and service demand estimation in Section 4.1.2. Then, we outline how the quality of resource adaptations can be quantified. In this context, we present cloud elasticity and its measures in Section 4.2.1 and highlight a cloud elasticity benchmark in Section 4.2.2.

4.1 A Brief Introduction to Basic Queueing Theory

Queueing theory is an essential analytic modeling technique for studying the performance of different systems (e.g., manufacturing lines, telephone networks, traffic systems, or computing systems). The basic idea is to model the system

of interest either as a single queue or as a network of queues. For instance, any resource assigned to an application can be modeled as a queue. Since each queue represents a mathematical model, different analyses can be performed. In the case of the application, the *resource demand* (Kounev et al., 2020), which is the minimum amount of resources required to fulfill a predefined *SLO* (*S*ervice *L*evel *O*bjective) under a given load, can be estimated. In other words, the number of resources that are needed to handle the current load adequately can be determined with queueing theory. Note that there are numerous approaches in the literature for estimating the required number of resources, but the queuing theory is often used in conjunction with these approaches.

4.1.1 Characteristics of a Queue

In general, a queue (illustrated in Figure 4.1), which is also referred to as service station, consists of a waiting line with finite or infinite size and one or more identical servers. A server can only process one request (customer, transaction, or any other unit of work) at a time. Consequently, a server is either in a busy or an idle state. Requests arriving at the queue are processed directly if at least one server is idle. Otherwise, the requests wait in the waiting line - if possible - until a server has finished its current job. After a request has been completely served, it leaves the queue and the next request to be served is selected from the waiting line according to the scheduling strategy.

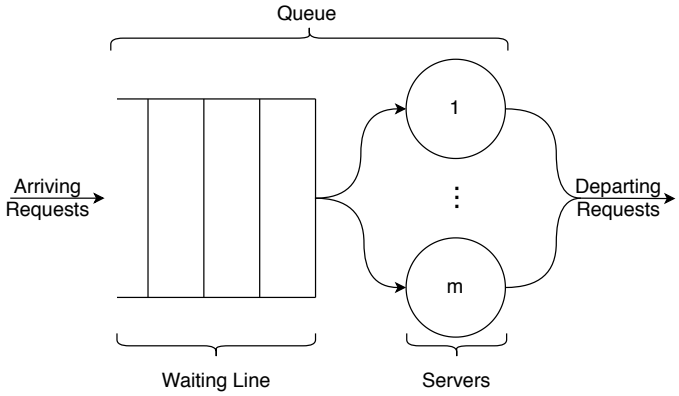

Figure 4.1: Queue with m servers.

While modeling a system as a queue or queueing network, its time-dependent behavior has to be described. Important key figures that describe a queue are the *inter-arrival rate* (i.e., the time between consecutive request arrivals) as

well as the *service time* between successive requests. A sequence of random variables can characterize both key figures. Based on the inter-arrival time, the *arrival rate* (i.e., the average number of requests per time unit) can be calculated. Mathematically, the arrival rate is defined by

$$\lambda = \frac{1}{E[A]}, \tag{4.1}$$

where $E[A]$ is the mean inter-arrival time. Analogously, the *service rate* (i.e., the average number of requests that can be processed per time unit) can be expressed as

$$\mu = \frac{1}{E[B]}, \tag{4.2}$$

where $E[B]$ is the mean service time. Based on the arrival rate and the service rate, different performance measures can be calculated[1]. These measures comprise the probability of the number of jobs in the system, system utilization, throughput, response time, waiting time, and the length of the waiting line. For instance, the system utilization can be computed as

$$\rho = \frac{\lambda}{\mu}. \tag{4.3}$$

Formally, a queue can be characterised by Kendall's notation (Kendall, 1953). This standard notation originally comprised three descriptive factors and was then extended by three additional factors. Consequently, a queue can be described by the six-tuple $A/S/m/K/N/D$: A represents the arrival process (i.e., the distribution of the inter-arrival time), S encodes the service process (i.e., the distribution of the service times), m is the number of servers, K is the size of the waiting line, N is the population size, and D reflects the scheduling discipline. The distributions A and S use codes for describing the distribution. Common used codes are M (exponential/Markovian distribution), D (deterministic distribution), or GI (general independent distribution). K, N, D are optional and if missing, their default values are used. For K as well as for N the default size is infinite and for D the default scheduling discipline is first-come-first-served.

4.1.2 Service Demand Estimation

For some systems, it is sufficient to be modeled as a single queue. However, there are systems with a large number of resources where a network of queues

[1] The system has to be in a steady state before performance measures can be assessed.

is better suited to represent the structure. For instance, a computer may be modeled with a network consisting of three queues: A multi-server queue reflecting the multi-core CPU, a queue representing the disk drive, and another queue modeling the network. More formally, a *queueing network* consists of at least two queues connected with each other. Each queue in the network represents a resource of the system. The incoming requests are grouped into *workload classes*. In each class, the requests have both similar arrival behavior and processing requirements. The requests can be transferred between any two queues of the network. Moreover, a request can be fed back directly to the queue it currently left. Therefore, the total time a request spends across all visits in a queue is a key parameter for performance modeling. This total time is called *service demand* and is the average time a unit of work spends obtaining service from a resource in a system, over all visits at the resource, excluding any waiting times (Lazowska et al., 1984; Menascé et al., 2004). In general, service demands are considered on a per workload class basis. Mathematically, the service demand of a request at queue i can be calculated based on the *service demand law* as

$$D_i = v_i \cdot E[B_i], \tag{4.4}$$

where v_i is the average number of visits per request to queue i and $E[B_i]$ the mean service time at queue i.

In most realistic computing systems, the direct measurement of service demands is not feasible during operation (Spinner et al., 2015) due to instrumentation overheads and possible measurement interference. Nevertheless, it is possible by using statistical estimation approaches to achieve an accuracy comparable to a direct measurement (Willnecker et al., 2015). The advantage of statistical estimation approaches compared to direct measurement techniques is their general applicability and low overheads. Estimation approaches typically rely only on coarse-grained measurements from the system (e.g., CPU utilization and end-to-end average response times) that can be obtained easily with monitoring tools without the need for fine-grained code instrumentation. Several approaches to service demand estimation have been proposed based on different statistical estimation techniques and combined with laws from queueing theory. We refer to Spinner et al. (Spinner et al., 2015) for an overview as well as a classification and experimental evaluation of the different approaches for service demand estimation.

4.2 Assessing the Quality of the Resource Adaptation

Nowadays, cloud services have to cope with the fast-paced changes and requirements of their users. Consequently, resource management systems (e.g., *auto-scalers*) have to adjust the number of provided resources as close as possible to the changing load of the service: On the one hand, the customers pay for the resources; on the other hand, if an application runs on fewer resources than needed, the performance sharply drops below usability thresholds.

The ability of a system to adjust its resources to time-varying load is called *scalability*. In general, scaling is done either vertically or horizontally. *Vertical scaling*, also called scale up/down, is a method that allows the system to add or remove "computing hardware" (e.g., CPUs or memory) to a single resource. *Horizontal scaling*, also called scale out/in, means to add more or remove unnecessary resources.

However, the scalability does not consider aspects such as how fast or how often the system is adjusted. To this end, there exist many approaches in the literature on how to measure the adaptation quality of a system. In this thesis, we focus on the *elasticity* of a cloud system. We choose this measure as, on the one hand, it can be precisely described with mathematical formulas and, on the other hand, it is commonly considered as a central characteristic of the cloud paradigm (Plummer et al., 2009). Note that the scalability of a system is a prerequisite of elasticity.

4.2.1 Definition and Measures of Cloud Elasticity

The term elasticity can be found in various academic fields, such as physics or economics, and has a different meaning in each context. Even in cloud computing, there are different definitions, such as from NIST (Mell and Grance, 2011) or IBM (Schouten, 2012). In this thesis, we use the following definition: Elasticity (in cloud computing) "is the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible" (Herbst et al., 2013, p. 2).

The core idea of quantifying the elastic resource management of an auto-scaler is to compare the *resource supply* and *resource demand* curves, as illustrated in Figure 4.2. The resource supply curve s_t (orange line) represents the monitored number of running resources at time t. The resource demand curve d_t (red line) shows the minimum amount of resources required to meet a predefined SLO under the workload (black dashed line) at time t. As an optimal auto-scaler knows when and the resource curve changes, both curves would be

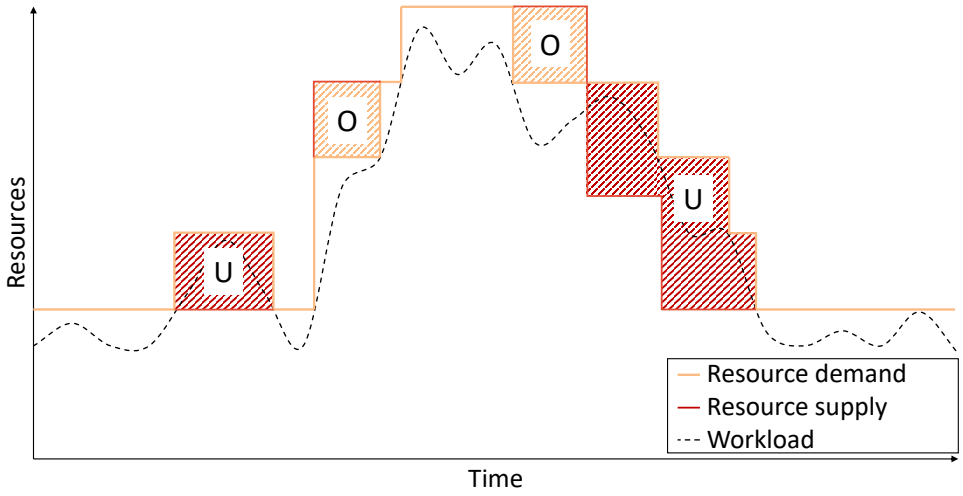

Figure 4.2: Example of supply and demand curves illustrating the idea of elasticity.

identical. In contrast, when using a standard auto-scaler, there are areas where (i) the demand curve is above the supply curve and (ii) the supply curve is above the demand curve. The system is in the first case in an underprovisioned (U) state and in the second case in an overprovisioned (O) state. Based on the wrong-provisioned areas, we can quantify the *accuracy* and *timing* of the auto-scaler.

The timing aspect captures the time shares (width of the U/O areas) in which the system is in an underprovisioned, overprovisioned, or optimal state. In contrast, the accuracy aspect reflects the average deviation (U/O areas) of the amount of supplied resources in relation to the resource demand. To measure these both aspects, we use the *provisioning accuracy* (see Section 4.2.1.1) and the *wrong provisioning time share* (see Section 4.2.1.2) that are also endorsed by the Research Group of the Standard Performance Evaluation Corporation (SPEC) (Herbst et al., 2016). Each of these measures quantifies how the resource supply differs from the resource demand. Consequently, the optimal value is zero, and the higher the value, the worse is the covered elasticity aspect.

4.2.1.1 Provisioning Accuracy

The *provisioning accuracy measure* θ_U and θ_O capture the relative amount of supplied resources that deviate from the demanded resources during the mea-

surement interval. Figure 4.2 illustrates the core idea behind the two measures: θ_U or θ_O is based on the sum of the areas U (i.e., the resource demand exceeds the resource supply) or areas O (i.e., the resource demand curve is below the resource supply curve), respectively. More precisely, the *underprovisioning accuracy* θ_U is the amount of missing resources necessary to fulfill the SLOs in relation to the current demand, normalized by the length of the experiment. Analogously, the *overprovisioning accuracy* θ_O is the amount of resources allocated in excess in relation to the current demand, normalized by the length of the experiment. More formally, the two measures θ_U and θ_O can be defined as

$$\theta_U[\%] := \frac{100}{T} \cdot \int_{t=0}^{T} \frac{\max(d_t - s_t, 0)}{\max(d_t, \varepsilon)} dt, \tag{4.5}$$

$$\theta_O[\%] := \frac{100}{T} \cdot \int_{t=0}^{T} \frac{\max(s_t - d_t, 0)}{\max(d_t, \varepsilon)} dt, \tag{4.6}$$

where s_t and d_t are the resource supply and resource demand at time t, respectively, T the experiment duration, and $\varepsilon > 0$ for avoiding the division by zero if the demand at time t is zero. In this thesis, we set $\varepsilon = 1$. The values of both measures lie in the interval $[0; \infty)$ with 0 indicating that there is no underprovisioning or overprovisioning during the length of the experiment, respectively.

4.2.1.2 Wrong Provisioning Time Share

The *wrong provisioning time share measure* τ_U and τ_O capture the time in which the system is in an underprovisioned or overprovisioned state. As illustrated in Figure 4.2, the core idea of τ_U or τ_O is to sum up the widths of the areas U or areas O, respectively. More precisely, the *underprovisioning time share* τ_U is the time in which resources are missing to meet the SLOs, normalized by the length of the experiment. Analogously, the *overprovisioning time share* τ_O is the time in which more resources than necessary are provisioned, normalized by the length of the experiment. More formally, the two measures τ_U and τ_O can be defined as

$$\tau_U[\%] := \frac{100}{T} \cdot \int_{t=0}^{T} \max(sgn(d_t - s_t), 0) dt, \tag{4.7}$$

$$\tau_O[\%] := \frac{100}{T} \cdot \int_{t=0}^{T} \max(sgn(s_t - d_t), 0) dt. \tag{4.8}$$

where s_t and d_t are the resource supply and resource demand at time t, respectively, and T the experiment duration. The values of both measures lie

in the interval $[0; 1]$ with 0 indicating that there is no underprovisioning or overprovisioning during length of the experiment, respectively.

4.2.2 Elasticity Benchmarking Framework

Comparing the elasticity of different cloud platforms is challenging as the underlying hardware and other influencing factors (e.g., the configuration of the management software) vary across the different platforms. All these (partly unknown) factors have to be considered when measuring the elasticity of a system. To this end, the elasticity benchmarking framework (Herbst et al., 2015), called *BUNGEE*[2], is applied in this thesis. The benefit of using BUNGEE in Chapter 11 is that the framework implements generic and cloud-specific benchmark requirements (Huppler, 2009, 2011; Folkerts et al., 2012) and takes the (partly unknown) factors into account.

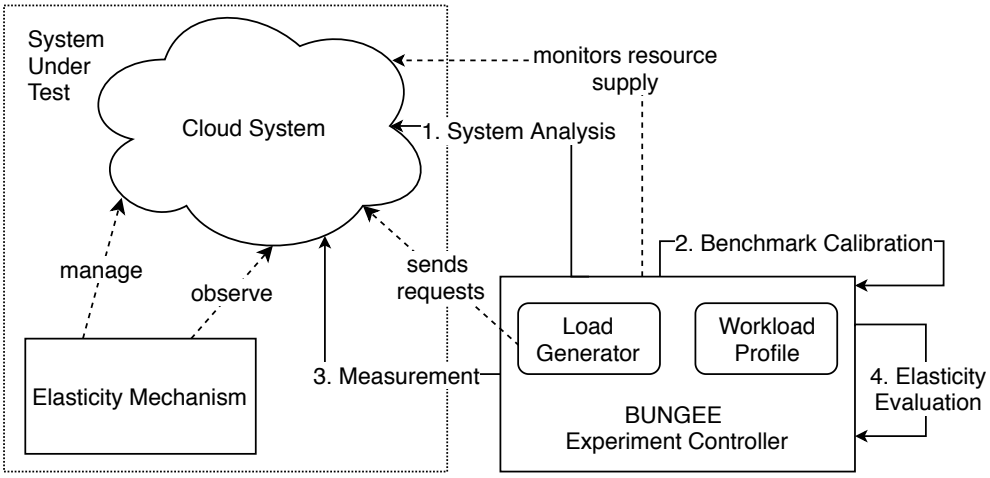

Figure 4.3: Overview of the BUNGEE workflow and experimental environment.

The BUNGEE cloud elasticity benchmark consists of four phases: (i) *System Analysis*, (ii) *Benchmark Calibration*, (iii) *Measurement*, and (iv) *Elasticity Evaluation*. Figure 4.3 illustrates the workflow and the experimental environment of BUNGEE. The experiment environment comprises the system under test (*SUT*) and the *BUNGEE Experiment Controller*. The SUT contains the cloud platform and an adaptation mechanism (e.g., an auto-scaler) that observes the application under stress and elastically manages the number of resources. The

[2]BUNGEE Cloud Elasticity Benchmark: `http://descartes.tools/bungee`

controller automatically executes the four phases and includes a load generator and a predefined workload profile.

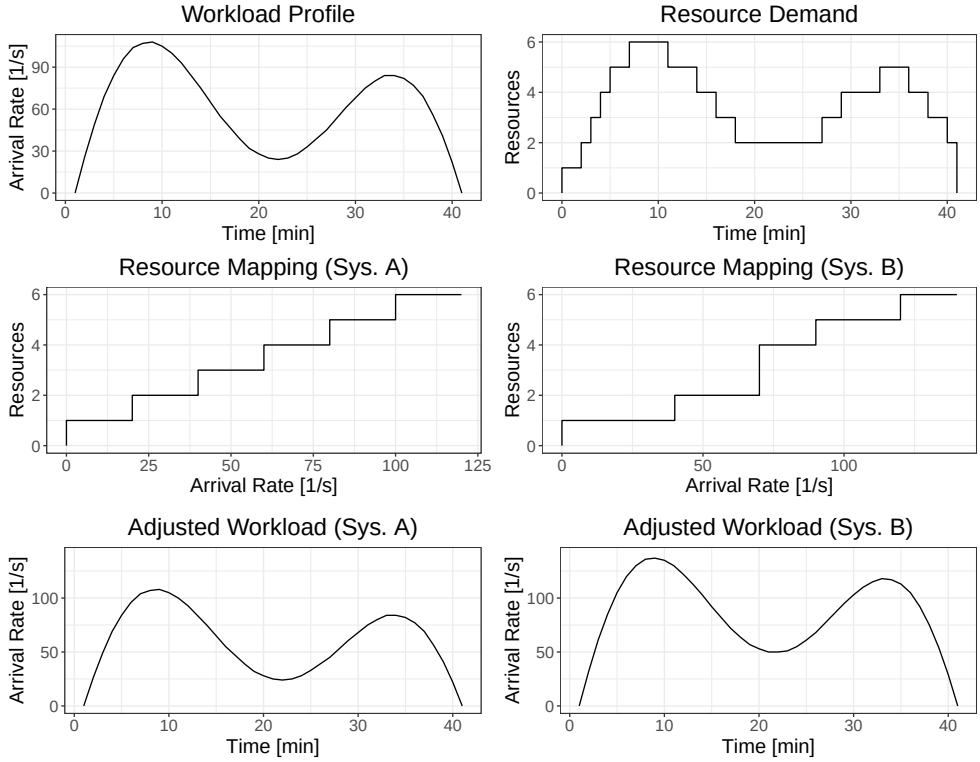

Figure 4.4: Example of workload profile calibration of BUNGEE.

Since the resource demand (i.e., the minimum resources necessary to handle the load without violating the SLOs) of a cloud platform depends on many factors, including the underlying hardware, BUNGEE stresses the cloud platforms, which have to be compared, with different workloads, so that the same variations in resource demand are induced on each platform. Consequently, the first phase analyzes the performance of the used resources and the adaptation/scaling behavior. More specifically, a discrete mapping function is generated that assigns to each load level the minimum amount of resources required to meet the SLOs. For this purpose, a binary search is carried out for all resource amounts, starting with one resource and increasing the number of resources until the maximum load no longer increases or the maximum number of resources is reached. Each binary search is performed by evaluating

whether the SLOs for a starting load level are violated and, depending on this, the load level is iteratively doubled or halved.

In the second phase, *Benchmarking Calibration*, the mapping from the first phase is used to adjust the workload profile on each cloud platform to induce an identical resource demand curve. Figure 4.4 illustrates an example of the calibration of two systems. In the first step, a reference system is selected as a baseline for the comparisons (System A). Starting from this system, the targeted resource demand curve (top right) is derived while taking the scaling behavior of this system into account. Then, the workload profile (top left), which is described by a Descartes Load Intensity Model (DLIM) (Kistowski et al., 2014), is tailored to each cloud platform based on its specific mapping function (second row of the figure) and the demand curve of the baseline system. The resulting calibration of the workload profile (last row of the figure) allows a direct comparison of the elasticity of different cloud platforms.

In the third phase, BUNGEE stresses the cloud platform with a time-varying load according to the tailored workload profile. During the measurement, the amount of supplied resources and the system's performance expressed in SLOs are monitored. Based on the resulting resource supply curve and the resource demand curve obtained in the calibration phase, the elasticity of a cloud platform is quantified in the final phase based on the elasticity measures.

Chapter 5

On the State-of-the-Art in Time Series Forecasting

In 1997, the "No-Free-Lunch Theorem" (Wolpert and Macready, 1997) was postulated. It states that there is not a single optimization algorithm that performs best for all scenarios since improving the performance of one aspect leads typically to a degradation in performance for another aspect. In fact, the theorem also holds for time series forecasting due to the diversity of time series. Considering the inherent drawbacks and limitations of forecasting methods, it can be concluded that there is no single forecasting method that performs best for all kinds of time series. To face the challenge posed by the "No-Free-Lunch Theorem", many hybrid forecasting methods have been proposed in the literature. The underlying idea of such hybrid approaches is to use at least two forecasting techniques to compensate for the limitations of the individual forecasting approaches. The success of this concept can be demonstrated, for example, by investigating the recent M4-Competition (Makridakis et al., 2018b): 12 of the 17 most accurate methods were hybrid forecasting methods.

The proposed hybrid methods can be categorized into three groups, each sharing the same basic concept: (i) *ensemble forecasting*, (ii) *forecasting method recommendation*, and (iii) *time series decomposition*. Consequently, Sections 5.1–5.3 present top cited and recent hybrid approaches. Then, forecasting competitions, in which time series forecasting methods are compared, are introduced in Section 5.4. The delimitation of this thesis from the following approaches is discussed in Section 8.10.

5.1 Ensemble Forecasting

The core idea of the first and historically oldest group called ensemble forecasting is to compute the forecast as a weighted sum of the values derived from applying multiple forecasting methods. The impetus for this idea was provided by an article (Bates and Granger, 1969) published by J. Bates and C. Granger

in 1969. In this work, the authors use a linear combination of two forecasting methods for assembling the final forecast. More precisely, they introduced five different approaches for determining the weight of the ensemble forecast by taking the training error of each method into account. Four approaches rely solely on the error, and one approach also takes the covariance of the training errors of both methods into account. In the evaluation, forecast ensembles consisting of each two forecasting methods (exponentially smoothed forecast and four methods taking serial correlation into account) were compared on two time series against the individual methods.

R. Adhikari et al. also introduced a weighted ensemble (Adhikari et al., 2015) of eight forecasting methods. The underlying idea is to consider the training mean squared error of each method to rank the methods accordingly. More precisely, each method receives a score that is the inverse of its error. Then, the best n methods are selected. The resulting forecast is then the weighted average of the remaining methods where the weights are the normalized scores. The ensemble set contains random walk, SVR, ARIMA, feed-forward neural network, Elman neural network, and generalized regression neural network. To evaluate their approach, the ensemble competed on four time series against each of the used methods and eight state-of-the-art ensemble techniques.

M. Sommer et al. introduced another approach. In their work (Sommer et al., 2016), the authors adapted the extended classifier system for function approximation to enable ensemble forecasting. That is, the approach dynamically learns the optimal weights for each forecasting method based on a combination of gradient-based local learning and a steady-state niche genetic algorithm. To enable good forecasts at the beginning, the forecast is the average of the deployed forecasting methods. At runtime, a recursive least squares absolute algorithm updates the weights according to the previous forecasts' absolute error. The used methods consist of ARIMA, Cubic spline's smoothing, moving average, ETS, and random walk with drift. This approach was compared on ten times series against each of the deployed methods and three state-of-the-art ensemble methods in the evaluation.

In their work (Cerqueira et al., 2017), V. Cerqueira et al. proposed an arbitrated dynamic ensemble approach. That is, the forecast consists of a weighted average of forecasting methods where a meta-learner determines the weights. More precisely, a random forest predicts each method's forecast error based on the training set, chooses the best 50% percent, and uses a softmax function to calculate the weights regarding the predicted error. The set of forecasting methods comprises SVRs, feed-forward neural networks, Gaussian processes, generalized linear models, random forests, generalized boosted regression,

MARS, rule-based regression, and projection pursuit regression, adding up to 40 models. In the evaluation, the authors compared their approach with eight state-of-the-art ensemble methods on 14 time series.

Z. Wang et al. (Wang et al., 2018) introduced three ensemble methods and four weighting approaches. Each ensemble consists of 30 feed-forward neural networks and handles the time series differently. In the first ensemble, each neural network gets a random subset of days as training. In contrast, each day is used in the second ensemble, whereas each neural network gets only a random subset of the day. The last ensemble combines the approaches of both ensembles. The weight is recalculated for each day based on a linear transformation or a soft-max transformation. Moreover, either all or only the n best neural networks are considered. In the evaluation, the different ensembles were evaluated on a single time series and were compared to a feed-forward neural network, SVR, k-nearest-neighbor, bagging forest, boosting forest, random forest, and one naïve approach.

D. Boulegane et al. (Boulegane et al., 2019) extended the approach proposed by V. Cerqueira et al. (Cerqueira et al., 2017). The major changes are two dynamic selection methods and a diversity criteria. The first selection approach considers only methods with a lower predicted error than a threshold that adapts according to the forecast error. The second selection approach uses Bernoulli trial that takes the predicted error and the meta-learner's confidence for each method into account. After the selection of the methods, the pairwise diversities between the methods are determined. If the diversity exceeds a threshold, the method with a higher inter-dependence is removed. The set of methods comprises a normal Hoeffding tree, an adaptive Hoeffding tree, a k-nearest-neighbor, and a weighted k-nearest-neighbor. In the evaluation, the approach competed on 55 time series against the original approach (Cerqueira et al., 2017) and two state-of-the-art ensemble methods.

Building upon the FFORMS approach (see Section 5.2), P. Montero-Manso et al. use time series characteristics to determine the weights for an ensemble forecast (Montero-Manso et al., 2020). More precisely, for each time series from a training set, the characteristics are calculated and each method forecasts this time series. Then XGBoost is used to learn the weights of the ensemble forecast based on time series characteristics and the forecast error of each method. The approach considers 42 characteristics and the following forecasting methods: Naïve, random walk, sNaïve, ARIMA, ETS, TBATS, AR on seasonally adjusted data, and NNetar. The approach took place in the M4-Competition and achieved the second-best point forecast accuracy.

5.2 Forecasting Method Recommendation

The second group forecasting method recommendation builds a rule set for estimating the assumed best forecasting method based on analyzing specific characteristics of the considered time series characteristics. F. Collopy and J. Armstrong introduced the first rule set for weighting forecasting methods based on the characteristics of the given time series in 1992. In their work (Collopy and Armstrong, 1992), the authors manually created an expert system after having interviewed five experts in the field of forecasting from industry and academia. Based on their opinions and knowledge, 99 rules were retrieved based on 18 time series features. The available methods comprise random walk, regression, Brown's linear exponential smoothing, and Holt's exponential smoothing. To evaluate the rule set, the expert system competed on 36 time series with random walk, two methods randomly choosing a forecasting method, and an equally weighted forecasting ensemble. In 2001, M. Adya et al. (Adya et al., 2001) revised this expert system to reduce human intervention, but were not able to completely abandon the experts.

A few years later, X. Wang et al. introduced two approaches to select the best forecasting method based on time series characteristics (Wang et al., 2009). The first approach uses hierarchical clustering and self-organizing maps to group similar time series. The second approach uses a decision tree, namely C4.5, to automatically derive rules. More precisely, for each time series, 13 time series characteristics normalized between 0 and 1 are calculated and forecasts of each method are performed. The set of methods consists of ARIMA, ETS, feed-forward neural network, and random walk. After the forecast of each time series, the forecast accuracy of each method is labeled. The best method is labeled as 1 and the other methods as 0. This classification of the methods and characteristics for each of the 315 time series are used to train the C4.5. Based on this model, static rules were reported. However, the rules must be interpreted with caution as there is no validation of these rules. In our work (Zuefle et al., 2019), we showed that this recommendation is worse than randomly guessing a method.

C. Lemke and B. Gabrys introduced another approach. In their work (Lemke and Gabrys, 2010b), the authors proposed a recommendation technique for ensemble forecasting. More precisely, the recommendation is based on zoomed ranking. The underlying idea is first to calculate the time series characteristics of the training set and determine the methods' performance in question. Then, the distance of the training set and a new time series is computed based on the characteristics. Afterward, the time series are clustered with k-means, and only the time series within the same cluster as the new time series are

considered for the recommendation. On this subset, a ranking of the methods is performed and a convex linear ensemble of the best three methods is then conducted. The authors consider 20 time series characteristics and the following methods: moving average, exponential smoothing, Taylor's exponential smoothing, polynomial regression, Theta, two different ARMA models, random walk, feed-forward neural network, recurrent neural network, and eight ensemble methods. In the evaluation, the recommendation system was compared on 66 time series (NN GC1 competition) against all used methods and ensembles.

A. Widodo and I. Budi (Widodo and Budi, 2013) extended the work of X. Wang et al. (Wang et al., 2009). Their approach uses the same time series characteristics and the four used methods plus three new methods (multiple kernel learning, S-curve, and interpolation based on decomposition). Moreover, a k-nearest-neighbor algorithm is trained on time series characteristics to classify the best method for each time series. To avoid using characteristics that may not be useful for the classification, a principal component analysis is performed. To evaluate the approach, the recommendation system was tested on 111 time series from the M1-Competition and compared to each of its used methods.

In contrast to the approaches introduced in this section, M. Kück et al.(Kück et al., 2016) considered for selecting the best forecasting method for a given time series, a set of error measures in addition to time series characteristics. The set of error measures consists of training and forecast errors in the training set as well as the ranking of the methods leading to 40 error-based information. For the classification, a feed-forward neural network is used to select one of four different exponential smoothing models. In the evaluation, the authors compared their approach on 111 time series (NN3 competition) with different sets of time series characteristics (e.g., the one from X. Wang et al. (Wang et al., 2009)) and their error-based measures against each considered method, two naïve selection methods, and random walk.

In a techpaper (Talagala et al., 2018), T. Talagala et al. introduced the *FFORMS* (feature-based forecast-model selection) approach. This approach maps the best forecasting method to a given time series based on its characteristics. To this end, a random forest is applied as a classifier on 25 to 30 time series characteristics (depending on if the time series is seasonal). To increase the training set and, consequently, the training performance, new time series are simulated by fitting ETS and ARIMA models to the original training set. The methods available for the selection are white noise, five different ARIMA models, random walk, Theta, six different ETS models, AR on seasonally adjusted data, and sNaïve. While using the M1- and M3-Competition, FFORMS

competed against each of the used methods on 4004 time series.

D. Zhang et al. proposed a recommendation system based on time series characteristics and the forecast horizon (Zhang et al., 2020). More precisely, a random forest is used as a classifier to map the best forecasting method to the forecast horizon and the characteristics. To improve the classification accuracy, the minimum redundancy and maximum relevance method combined with a back-search algorithm is applied to omit redundant time series characteristics. The methods in question are a feed-forward neural network, SVR, and extreme learning machine. Moreover, there are 29 time series characteristics and four supported forecast horizons. To evaluate their approach, the authors compared the recommendation system on 522 time series against versions of itself (SVM instead of random forest, using all characteristics, omitting the horizon information), each used forecasting method, and the average of these forecasting methods.

5.3 Time Series Decomposition

The last group time series decomposition either decomposes a time series into components and forecasting methods are applied to each component separately or a time series is forecast with an individual method, and afterward, a second individual method is applied on the residuals. The authors G. Zhang et al. (Zhang, 2003) introduced a hybrid forecasting method based on ARIMA and a feed-forward neural network. The core idea is that a time series can be decomposed into a linear and non-linear part. To this end, the approach first uses an ARIMA model to capture the linear part of the time series. Then, the resulting residuals of the fitting are used to train a neural network to handle the non-linear part of the time series. In the evaluation, the authors compare their method against the individual methods on three time series. Following this idea, P. Pain and C. Lin (Pai and Lin, 2005) use the same approach except that they use SVM to capture the non-linear part. In their evaluation, the authors compared their approach against the individual methods on ten time series.

In their work (Liu et al., 2014), N. Liu et al. introduced an approach based on empirical mode decomposition. More precisely, their approach first uses the empirical mode decomposition to split the time series into a number of intrinsic mode functions. Then, the functions are split into two sets. Each function from the first or the second set is forecast separately by an extended Kalman filter or an extreme learning machine with kernel, respectively. Finally, the forecasts of the functions are assembled to return the forecast of the time series. The parameters (e.g., the size of the first set) of this approach are determined by

particle swarm optimization. The authors tested their approach on four time series against the extreme learning machine with kernel.

The authors I. Khandelwal et al. (Khandelwal et al., 2015) also followed the idea of decomposing a time series into a linear and non-linear part. To this end, the time series is decomposed through a discrete wavelet transformation. More precisely, the time series is decomposed into higher and lower frequency components. Then, these components are reconstructed via the inverse discrete wavelet transformation. Afterward, ARIMA is applied to the reconstructed detailed part. The resulting fitting errors and the reconstructed approximation part are used to train a feed-forward neural network. Finally, both forecasts are added up. In the evaluation, the approach competed on four time series against the approach proposed by G. Zhang et al.(Zhang, 2003) and the individual methods.

C. Bergmeir et al. used in their work (Bergmeir et al., 2016) the idea of bootstrap aggregating. More precisely, a time series is decomposed into its component trend, seasonality (if present), and irregular. For the decomposition, STL or loess is used depending on if the time series is seasonal. Then, 100 different versions of the irregular part are created with moving block bootstrapping. Afterward, the irregular variations are assembled with the remaining components resulting in 100 time series. For each time series, an ETS forecast is performed, and the final forecast consists of the median of the forecasts. To evaluate the approach, 2829 time series from the M3-Competition were used and four variations of the approach were compared with ETS and the results of the competition. In the following, we refer to this approach as *BETS*.

Another approach (Panigrahi and Behera, 2017) using the underlying idea that a time series can be decomposed in a linear and non-linear part was introduced by S. Panigrahi and H. Behera. To this end, the approach first fits an ETS model on the time series and on the resulting fitting errors a feed-forward neural network. In contrast to the work of G. Zhang et al.(Zhang, 2003), the authors split the time series into training and validation data. Moreover, the time series and the errors are both normalized with min-max scaling. Then, the validation data is used to optimize both the ETS and neural network parameters. Finally, both forecasts are de-normalized and assembled. In the evaluation, the authors compared their approach on 16 time series against ARIMA, ETS, a feed-forward neural network, the approach proposed by G. Zhang et al. (Zhang, 2003) and another ARIMA-ANN approach.

The basic idea (Zhang et al., 2017) of the work of J. Zhang et al. is to decompose a time series into a non-linear and periodic part. To this end, the ensemble empirical mode decomposition is applied. The resulting intrinsic mode func-

tions are then forecast by sARIMA or ANFIS (feed-forward neural network with fuzzy inference) depending on if the function is periodic or non-linear. Finally, the functions are assembled to return the future time series. To evaluate their approach, the approach competed against the individual methods on two time series.

In contrast to the methods presented in this section, *Prophet* (Taylor and Letham, 2018), which was introduced by S. Taylor and B. Letham (Facebook), fits an additive regression model with trend, season, and holiday as components. The trend component can either be piece-wise linear or following a logistic growth curve. Both variants support and automatically detect changes in the trend. The seasonal part is represented by a Fourier series. The holiday component can be specified by the user or uses calendar information. For finding all parameters for fitting the model components, the Broyden–Fletcher–Goldfarb–Shanno algorithm is applied. The authors compared their approach against ARIMA, ETS, sNaïve, and TBATS.

In the work of F. Saâdaoui and H. Rabbouch (Saâdaoui and Rabbouch, 2019), the time series is split into trend, seasonal, and irregular parts based on the maximal overlap discrete wavelet transformation. The trend is fitted by a polynomial regression and the periodic part with a sum of sines. Both fitting residuals and the irregular part are added and modeled with an autoregressive fractionally integrated moving average. The resulting errors are then used to train a feed-forward neural network. Finally, all forecasts are summed up. In a case study, the proposed approach was compared on one time series against the stand-alone neural network, a seasonal non-linear autoregressive model, and a multivariate sARIMA.

F. Saâdaoui et al. proposed another approach (Saâdaoui et al., 2019) combing neural networks and decomposition. First, the trend is fitted by polynomial regression. Then, the periodic part is approximated with a sum of sines. Both components are removed from the time series resulting in a de-trended, de-seasonalized, and de-trended-de-seasonalized time series. Afterward, two feed-forward neural networks are trained: one network on the de-trended-de-seasonalized time series and the other network on the de-seasonalized time series. After the forecast of each component, two forecasts are assembled: The first by adding up the forecast of the trend, season, and de-trended-de-seasonalized part; the second by adding up the forecast of the trend and the de-seasonalized part. Finally, the best accurate forecast is chosen. In the evaluation, three variations of the approach were compared on three time series against the stand-alone neural network, one neural network coupled with empirical mode decomposition, and another neural network with wavelet decomposition.

The method (Smyl, 2020) introduced by S. Syml (Uber) deploys exponential smoothing formulas in combination with a recurrent neural network. More precisely, the exponential smoothing formulas are used for on-the-fly deseasonalizing and normalizing the series and the neural network is used for extrapolating the time series. In contrast to the other methods described in this chapter, this method requires a set of time series. The set is used to train the neural network, but the exponential smoothing is trained per time series. Moreover, this approach subsets a time series and trains different models and return the ensemble forecast (either of all or the best n models depending on the time series). This approach was the winner of the M4-Competition. In the following, we refer to this approach as *ES-RNN*.

5.4 Benchmarking of Forecasting Methods

In the last decades, several papers have been published in which forecasting methods have been evaluated either on a small or large scale. However, based on our review (see Section 7.1), we found that the degree of quality of the evaluations suffers on the one hand from the data sets selected and, on the other hand, from the methodology applied. To this end, we focus in this section on benchmarking forecasting methods on a large scale. More precisely, we review forecasting competitions. Moreover, we restrict the scope on competitions involving multiple participants.

One of the best-known forecasting competition series is the Makridakis competitions or also known as M-Competitions. Before the first M-Competition was launched in 1982, S. Makridakis and M. Hibon (Makridakis and Hibon, 1979) assembled 111 time series and compared different methods based on the error metrics MAPE, percentage better, Theil's U-Statistic. Then, in the first M-Competition (Makridakis et al., 1982), S. Makridakis et al. compared forecasting methods on 1001 time series while considering the MAPE, MSE, average ranking, medians of absolute percentage errors, and percentage better. Moreover, the average percentage errors and mean absolute deviations were calculated but not reported. In contrast to its predecessor, the second M-Competition (Makridakis et al., 1993) considered only 26 time series and the measures MAPE, average ranking, percentage better, and mean percentage error. However, this competition lasted almost four years, as participants, starting in 1987, received real-time data and feedback on their submitted forecasts as new data became available. The final forecast was then submitted in the last year. In 1991, the Santa Fe Institute also held a competition comprising six time series and considered the mean squared error and used forecast errors to

compute the likelihood of the data (Gershenfeld and Weigend, 1993). To extend and replicate the former M-Competitions, S. Makridakis and M. Hibon ran their third competition in 1998. The M3-Competition (Makridakis and Hibon, 2000) contained 3003 time series and evaluated the methods based on sMAPE, average ranking, median symmetric absolute percentage error, percentage better, and median relative absolute error. To show the potential of neural networks in terms of forecasting, S. Crone et al. (Crone et al., 2011) used 111 time series from the M3-Competition and evaluated the neural networks' forecasts with the sMAPE, average ranking, median relative absolute error, and MASE. A few years later, the Tourism competition was held in 2010 (Athanasopoulos et al., 2011). The data set contained 1311 time series, and the submitted forecasts were compared regarding the percentage better, MAPE, MASE, median absolute scaled error, and average ranking. To raise the importance of energy forecasting and have a sound benchmark, the Global Energy Forecasting Competition comprising 28 time series was first held in 2012 (Hong et al., 2014). For the evaluation, the RMSE was used. Two years later, the second Global Energy Forecasting Competition was held (Hong et al., 2016). In contrast to the first edition, this competition used for the evaluation the pinball loss function and continuous rank probability score and comprised 15 time series while the data was updated monthly in a rolling manner. The last published competition is the M4-Competition (Makridakis et al., 2018b). S. Makridakis et al. provided 100,000 time series while considering the sMAPE, percentage better, and overall weighted average as evaluation measures.

Chapter 6

On the State-of-the-Art in Cloud Auto-Scaling

With the emergence of cloud computing, both academia and industry have started adopting this paradigm for deploying their applications. One central characteristic of this cloud paradigm is elasticity that allows adapting to workload changes by provisioning and de-provisioning resources automatically. To this end, auto-scaling mechanisms have been a popular research topic over the past decade. There have been multiple recent efforts to survey the state-of-the-art in auto-scaling, for example (i) G. Galante and L. de Bona (Galante and Bona, 2012), (ii) T. Lorido-Botran et al. (Lorido-Botran et al., 2014), (iii) B. Jennings and R. Stadler (Jennings and Stadler, 2015), (iv) C. Qu et al. (Qu et al., 2018), and (v) recently P. Singh et al. (Singh et al., 2019). In each survey, different taxonomy or classification approaches for grouping the mechanisms are proposed.

Based on the mentioned survey articles, this section gives an overview of recent and top-cited auto-scalers for both monolithic applications and distributed applications composed of multiple services or tiers[1]. To group the reviewed mechanisms, we apply the established classification scheme by T. Lorido-Botran et al. (Lorido-Botran et al., 2014). In their survey, the authors proposes a classification of auto-scalers into five groups based on their underlying technique: (i) *control theory*, (ii) *queueing theory*, (iii) *reinforcement learning*, (iv) *threshold-based rules*, and (v) *time series analysis*. To this end, the Sections 6.1–6.5 present the reviewed auto-scalers. After this classification, we discuss the selected approaches based on their cost-efficiency policy. Moreover, we delimit this thesis from the selected techniques in Section 9.7.

6.1 Auto-Scalers based on Control Theory

Auto-scalers from the field of control theory consist of two parts: (i) A model of the application and (ii) the controller. Consequently, the performance of such

[1] In the following, the terms tier and service are used interchangeable.

an auto-scaler depends on both the model and the controller. E. Kalyvianaki et al. (Kalyvianaki et al., 2009) introduced a vertical, proactive mechanism based on control theory. More precisely, three different approaches are proposed: a controller for monolithic applications, a controller for applications consisting of multiple services taking the correlations between the services into account, and an extension of the second controller that self-configures its parameter according to the workload. Each controller uses a simple application performance model that is realized by a one-dimensional random walk. To reduce noise that is caused by workload fluctuations, this model is enhanced with a Kalman filter.

AutoControl (Padala et al., 2009), a vertical, proactive auto-scaler for applications consisting of multiple services, is another representative of this group. The underlying idea is to deploy a two-layer architecture: a controller for each service and one for each virtual node. The controller responsible for a service first models the time-varying relationship between the resource allocation and its normalized performance. Then, the controller predicts the resource allocations regardless of the other services. To allocate the resource requests from the first layer, the controller responsible for a virtual node assigns all requests if possible. Otherwise, the resources are distributed regarding the priority of each service.

Q. Zhu and G. Agrawal introduced another approach. In their work (Zhu and Agrawal, 2012), the authors designed a reactive mechanism for scaling applications comprising multiple services vertically. The aim is to take a fixed time-limit and a resource budget into account to maximize the application performance. Thus, the authors deploy a resource model for each service component that maps the adaptive parameters to system input while considering the resource budget. The proposed controller uses this model to scale the application subjected to time and budget constraints. To reduce the instability of the controller, the controller is combined with a reinforcement learning agent.

In their work (Ali-Eldin et al., 2012), A. Ali-Eldin et al. proposed a mechanism that scales monolithic applications horizontally. The core idea is to model the application as a G/G/n queue and have two adaptive proactive controllers for scaling down and a reactive approach for scaling up. Both controllers are independent of parameters, and thus, any performance metric can be used. The first controller considers the periodical rate of change of the workload. In contrast, the second controller takes the ratio between the change in the workload and the average system service rate over time into account. In the following, we refer to this approach as *Adapt*.

Another auto-scaler of this group, which vertically scales applications consist-

ing of multiple services, was introduced by E. Lakew et al.(Lakew et al., 2017). This mechanism uses a predictive MIMO (Multiple Input Multiple Output) controller to compute CPU and memory usage for future time intervals. More specifically, during an offline phase, a linear MIMO model is created based on how the application reacts to changes in resource allocation. Based on the model, future resource allocations are planned to ensure that the application meets predefined performance. However, only the next planned allocation is performed to respond to unexpected behavior not captured by the model.

6.2 Auto-Scalers based on Queueing Theory

Usually, queueing theory is widely applied to model the relationship between incoming and outgoing jobs in a system. That is, auto-scalers from this field depend on the model of the system for determining the resource demands. A representative of this group, an auto-scaler (Urgaonkar et al., 2008) that scales applications consisting of multiple services horizontally, was proposed by B. Urgaonkar et al. The underlying idea is to model the application as a chain of queues where each service reflects a G/G/1 queue. To estimate the number of required resources, the approach forecasts the workload for the long term based on histograms. In a short-term interval, the auto-scaler checks for unpredictable events or deviations from the forecast load to scale the application reactively. In the following, we refer to this approach as *Hist*.

Another approach (Xiong et al., 2011) was proposed by P. Xiong et al. that scales applications comprising of multiple services vertically and reactively. To estimate the application's performance, the application is modeled as a chain of queues where each service is estimated by an M/G/1 queue. Moreover, the scaling logic consists of two layers. The first layer requests the required amount of resources to guarantee the performance on an overall basis. The second level partitions the total resource budget among the services that can minimize the response time.

A reactive approach (Sharma et al., 2012) introduced by U. Sharma et al. scales applications consisting of multiple services horizontally. The target application is modeled as a chain of queues where each service reflects an M/G/1 queue. The resource provisioning is subjected to the high percentile of response time while minimizing the application's cost. This is done by greedily searching for a resource configuration that has a high utilization but low cost.

In their work (Jiang et al., 2013), J. Jiang et al. proposed another elasticity mechanism for scaling monolithic applications horizontally. To allocate resources proactively, the approach forecasts the workload with linear regression

while having a reactive fallback if the length of the queue of waiting requests exceeds a predefined threshold. Based on the forecast, the number of resources is calculated while modeling each resource as an M/M/n queue. After this estimation, an optimization objective function is used to exploit the cost-latency trade-off based on budget constraints.

A further example is *AutoMAP* (Beltrán, 2015), a reactive auto-scaler for applications comprising several services. To maintain the performance requirements of the application and to minimize the number of resources, the approach supports both vertical and horizontal scaling. The underlying idea is to predict the application's response time while each resource is modeled as a -/G/1 queue and then, to scale according to predefined thresholds. After determining the required resources, a resource configuration is searched that meets the response time but has a lower cost.

6.3 Auto-Scalers based on Reinforcement Learning

In contrast to the first both fields, methods from reinforcement learning do not have explicit knowledge or a model of the application. The goal is to find the best action for each state with a trial-and-error approach. However, the time for convergence on the best actions can take a long time. In their work (Tesauro et al., 2006), G. Tesauro et al. presented a proactive mechanism based on reinforcement learning that scales applications comprising multiple services in a horizontal manner. For learning the best scaling policy, a feed-forward neural network is used. To avoid poor performance during the training phase, an initial policy based on queueing theory is applied. More precisely, the application is modeled as an M/M/1 queue. Instead of learning once, the learning is done iteratively. During the executing of a policy, a new data set is gathered that can be used to train an improved policy.

Another representative of this group is VConf (Rao et al., 2009), an elasticity mechanism for scaling applications consisting of several services vertically and proactively. The goal of this approach is to optimize the summarized performance of each instance while reconfiguring them periodically. For learning the best configurations, a feed-forward neural network is applied. To avoid a long exploration phase at the start and reduce the tested actions during runtime, a model trained from previous information using supervised learning is employed to simulate the application behavior.

N. Dezhabad and S. Sharifian (Dezhabad and Sharifian, 2018) presented another proactive elasticity mechanism for scaling monolithic applications horizontally. The underlying idea is to provide the right number of instances based

on the workload and the proportion of requests each instance can handle. For finding the right scaling action without a complete model of the application, Q-Learning is used. The reward function penalizes high- or low-utilization states and service level agreement violations. To handle heterogeneous workload mixes, a genetic algorithm balances the load to the instances while modeling each instance as an M/G/1 queue.

Another example of this group is *RLPAS* (Benifa and Dejey, 2019), a proactive mechanism for scaling applications consisting of multiple services in a horizontal manner. This approach aims to maximize both the application performance and the instance utilization. To this end, each instance is profiled and performance metrics are gathered. This information is used for learning a model-free State-Action-Reward-State-Action approach that is also capable of forecasting the future demand. To reduce the convergence time of this approach, multiple agents are used for learning the environment in a parallel manner.

6.4 Auto-Scalers based on Threshold-Based Rules

Auto-scalers from this category react to changes in the workload using threshold-based rules. The actions are triggered by performance metrics and associated predefined thresholds. Due to their simplicity and easy setup, commercial cloud providers offer such mechanisms for their clients. One representative of this group was proposed by T. Chieu et al. (Chieu et al., 2009)that scales monolithic applications horizontally and reactively. This algorithm provisions instances based on thresholds or a specific scaling indicator of the application. To this end, the application is continuously monitored, and a moving average is computed for each information. If all instances are above the threshold, a new instance is provisioned. If there are instances below the threshold and at least one instance is idle, the idle instances are removed. In the following, we refer to this approach as *React*.

A reactive mechanism for vertical scaling of a monolithic application was presented by M. Maurer et al. (Maurer et al., 2011). The key idea of this approach is to calculate utilization that is the ratio between the used and provided the number of resources. To scale the application, a lower and upper threshold is defined. If the utilization exhibits the upper bound, new resources are added. Otherwise, resources are removed. To avoid scaling oscillations, a wide span between both thresholds is applied.

In their work (Han et al., 2012), R. Han et al. introduced a reactive auto-scaler for applications comprising multiple services. The goal of the scaling is to find the cheapest configuration that can handle the current workload. To this

end, the approach supports vertical scaling and horizontal scaling based on predefined thresholds. If the vertical up-scaling is not possible, a new instance is added, and then again, vertical scaling is performed. During the down-scaling, the instances or resources that cause the most cost are released horizontally or vertically.

A further example of this group was proposed by A. Naskos et al. (Naskos et al., 2017) and scales monolithic applications horizontally and reactively. To capture the uncertain behavior of the application, the authors apply Markov decision process models. For the scaling, five different policies are conducted. The best action is selected on the policy that has estimated the best application performance. In the case there is more than one option, the policy with the lowest cost is chosen. During a scaling-down, the approach takes pricing schemes into account and releases the instance closest to the charging unit.

6.5 Auto-Scalers based on Time Series Analysis

The key idea in this group is to detect patterns and forecast the future load. As the choice of the most suitable method is crucial, the performance of an auto-scaler using time series analysis depends on the chosen method. W. Iqbal et al. (Iqbal et al., 2011) presented an elasticity mechanism based on time series analysis for scaling applications comprising multiple services horizontally. For ensuring a reliable application performance, resources are added reactively and released proactively. To scale up the application, heuristics and profiling are used. The scale-down is realized proactively to avoid the premature release of required resources. To this end, a polynomial regression with the order two is used. In the following, we refer to this approach as *Reg*.

A further example is *AGILE* (Nguyen et al., 2013), a proactive mechanism that scales applications consisting of multiple services horizontally. The underlying idea is to use a wavelet-based decomposition. Then, each part is forecast separately with a simple Markov model to reduce the error. Finally, the parts are summed up to synthesize the forecast. To reflect the relationship between the workload and the application's performance, online profiling of each service and polynomial curve fitting is used to estimate a black-box performance model.

H. Fernandez et al. (Fernandez et al., 2014) presented another proactive elasticity mechanism that scales monolithic applications in a horizontal manner. To have reliable forecasts, different forecasting methods are used, such as linear regression or Holt's exponential smoothing. Based on the forecasts from the last interval, the method with the best accuracy is chosen for the current forecast. Based on this forecast and the measured computing capacity of

different resource configurations, this method provides the required resources under consideration of their costs and in accordance with an hourly billing interval. In the following, we refer to this approach as *ConPaaS*.

Another approach is *HybridScaler* (Wu et al., 2016), an auto-scaling method for applications consisting of multiple services. This approach uses both horizontal and vertical scaling. That is, mid- to long-term workload changes are forecast with sparse periodic auto-regression to adapt the application proactively and horizontally. To react to unpredictable changes in the workload, the application is scaled vertically. The long-term forecast is also used to allocate resources in accordance with an hourly billing interval.

A further example of this group is a proactive mechanism for scaling applications comprising several services horizontally introduced by R. Khorsand et al. (Khorsand et al., 2018). This approach is aligned with the MAPE-K control loop. In the analysis phase, support vector regression is used for forecasting the future workload. In the plan phase, a fuzzy analytically hierarchy process is deployed for scaling the application based on the forecast. The authors rely on this fuzzy approach as it allows to make decisions in uncertain circumstances.

6.6 Cost-Efficient Auto-Scalers

Although only a few of the reviewed approaches scale an application considering cost, we can classify them into three groups: The first group consists of approaches that take advantage of a heterogeneous resource pool where the resource configurations vary in performance and cost. The idea is to select a resource configuration with the lowest cost while meeting the specified SLAs or application performance. Examples of this group are AutoMAP (see Section 6.2), the approach from R. Han et al. (see Section 6.4), and the method introduced by U. Sharma et al. (see Section 6.2). The auto-scalers of the second group have to handle a trade-off between application performance and predefined budget or runtime constraints. Representatives of this group are the methods introduced by J. Jiang et al. (see Section 6.2), P. Xiong et al. (see Section 6.2), and Q. Zhu and G. Agrawal (see Section 6.1). In the last group, the approaches have knowledge about the charging models of the public cloud, where the application is deployed. Based on these models, the rented instances are used as cost-efficient as possible. Examples of this group are ConPaaS (see Section 6.5), HybridScaler (see Section 6.5), and the approach from A. Naskos et al. (see Section 6.4).

Part II

Contributions

Chapter 7

Forecasting Benchmark

In many domains, the selection of the most appropriate approach is guided by reviewing experimental results performed in either established benchmarks or scientific papers. By benchmark, we refer to an instrument used to evaluate and/or compare systems or methods based on certain properties (Kistowski et al., 2015). Moreover, we consider benchmarks as a key instrument for improvement and competition. In the context of time series forecasting, the key concerns are the forecast accuracy and the time-to-result.

To investigate how both key concerns are assessed in the literature, we reviewed scientific work published during the last decades (see Section 7.1). Although there are forecasting competitions that can be considered as benchmarks, like the well-known M-Competitions[1], these are barely applied in scientific works. More precisely, most of the reviewed papers consider only a small set of (mostly related) methods and evaluate their performance on a small number of time series with only a few error measures while providing no information on the time-to-result of the studied methods. Consequently, such articles fail to provide a reliable approach to guide the choice of an appropriate forecasting method for a particular use case.

To approach the problem of limited comparability between existing forecasting methods, we pose ourselves the following research questions:

RQ 1: *How to automatically compare and rank different forecasting methods on a level playing field based on their performance in a diverse set of evaluation scenarios?*

RQ 2: *What are suitable and reliable measures for quantifying the quality of forecasts?*

Towards addressing the research questions, our contribution is to provide a forecasting benchmark to establish a level playing field for evaluating and comparing the performance of forecasting methods in a broad setting covering

[1]M-Competitions: `https://mofc.unic.ac.cy/history-of-competitions/`

a diverse set of evaluation scenarios. This benchmark automatically evaluates a forecasting method based on the choices of the user. More precisely, the user uploads a code artifact of the forecasting method to be benchmarked. The user then specifies the use case and selects the evaluation type for the benchmarking process. Based on this user input, the method is evaluated and compared to state-of-the-art methods. Besides the design, we propose novel forecast error measures and consider established measures for the evaluation. To provide a diverse set of time series, we assemble a data set comprising 400 publicly available time series taken from different domains. The data set exhibits a higher heterogeneity compared to established forecasting competitions. Moreover, the proposed forecasting benchmark is used to compare Telescope (see Chapter 8) with state-of-the-art forecasting methods in Chapter 10.

The remainder of this chapter[2] is organized as follows: We first perform a literature review in Section 7.1. Then, we introduce the design and the use cases of the forecasting benchmark in Section 7.2. In Section 7.3, we present proposed data set used for the benchmarking. Afterwards, we explain the different evaluation types implemented in the benchmark in Section 7.4. Moreover, we propose new forecast accuracy measures in Section 7.5. Then, we compare our data set with existing forecasting competitions in Section 7.6. Finally, we conclude the chapter in Section 7.7.

7.1 Literature Review

Since time series forecasting is of major importance in many decision-making fields (Hyndman and Athanasopoulos, 2017), there is a broad range of academic work on this topic. To decide which forecasting method is best suited for a particular application, it is typically necessary to rely on the assessments made in scientific papers. We review scientific papers in the field of time series forecasting to investigate how trustworthy and meaningful these state-of-the-art assessments are. More precisely, we collected papers matching the terms *time series forecasting*, *time series analysis*, or *time series prediction* on the academic search engines Google Scholar[3], Mendeley[4], IEEE Xplore[5], and Semantic Scholar[6]. We filtered this initial set of scientific papers by considering only those papers that have been published during the last 40 years and contain an evaluation section.

[2] This chapter is based on our previous work (Bauer et al., 2020c) where we sketch the idea of the benchmark.

[3] Google Scholar: `https://scholar.google.de/`

[4] Mendeley: `https://www.mendeley.com/`

[5] IEEE Xplore: `https://ieeexplore.ieee.org/Xplore/home.jsp`

[6] Semnatic Scholar: `https://www.semanticscholar.org/`

From these papers, we pick out those that have received an average of at least 8.5 citations per year (Google Scholar) to restrict the data set[7] to 100 scientific papers. The data set contains scientific work published between 1982–2019 and cited between 29–2440 times. Moreover, the selected papers cover different topics, such as supply chain management, river flow, tourism, traffic, stock prices, electric/power demand, and many more.

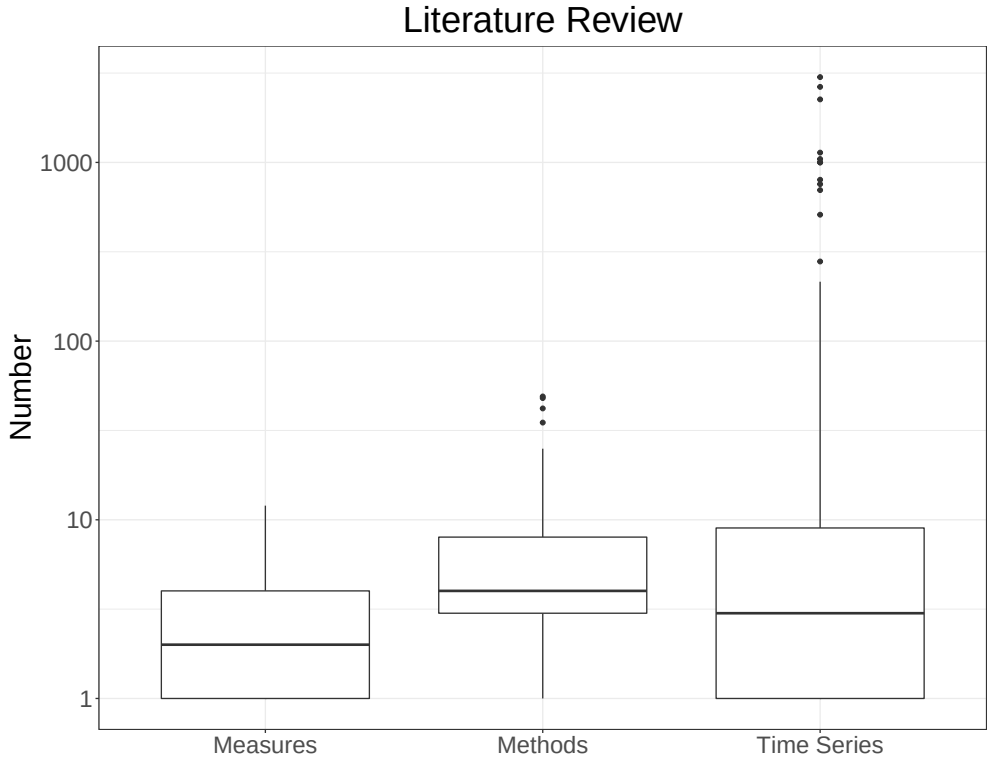

Figure 7.1: Distribution of the used measures, methods, and time series in the evaluation sections from the reviewed 100 scientific papers.

After collecting the data set, we investigated the following questions: (i) How many measures were considered for the evaluation?; (ii) How many forecasting methods were evaluated?; (iii) How many time series were used in the evaluation? To answer these questions, we present in Figure 7.1 the distributions of how many measures, forecasting methods, and time series were used in the reviewed papers as box plots. Note that the vertical axis is depicted in log-scale. The median, that is, 50% of the scientific papers reviewed used at most two

[7]The list of reviewed articles is available at `https://doi.org/10.5281/zenodo.3716035`

measures, considered a maximum of four methods, and used a maximum of three time series to formulate scientific results. Even 75% of the studies examined used a maximum of four measures, at most eight forecasting methods, and a maximum of nine time series. Some papers are outliers and depicted as points in the figure, which used more than 1000 time series. These articles used the M-Competitions[8] or the Watson macroeconomic database. However, those time series have a high degree of similarity (see Section 7.6). For the ranking of the methods, almost all papers consider the MAPE, RMSE, or related error measures, but neglect statistical properties of the error measures such as the standard deviation. On top of this, none of the scientific papers reviewed takes the time-to-result into account as part of their evaluation. In other words, there is a lack of information about the runtimes of the forecasting methods. In fact, the runtime of a forecasting method is irrelevant in non-time-critical scenarios. Still, there are also scenarios (e.g., auto-scaling) with strict deadlines to enable timely and reliable planning.

7.2 Design Overview and Use Cases

Based on our review, we found that the quality of the evaluations suffers, on the one hand, due to the lack of commonly used and representative data sets, and, on the other hand, due to the limitations of the applied methodology, for example, the usage of only a few competing methods or only a few measures. To this end, we design a forecasting benchmark that automatically evaluates and ranks forecasting methods based on their performance in a diverse set of evaluation scenarios. In other words, the benchmark allows comparing a particular forecasting method to state-of-the-art forecasting methods. In more detail, the benchmark offers a broad data set exhibiting a high degree of diversity (see Section 7.6), different measures, and three types of evaluation approaches.

Figure 7.2 illustrates the workflow of the proposed benchmark. First, the user uploads the code artifact (i.e., Docker[9] container) of the forecasting method to the benchmark. Then, the forecasting method is deployed within the benchmark and can only communicate with the benchmark. Afterward, the user specifies for which use case the deployed forecasting method should be evaluated. More precisely, the data set is split into four domains from which the user can choose (see Section 7.3), each covering 100 heterogeneous time series taken from different fields. Moreover, the user has to select the evaluation type

[8] M-Competitions: `https://mofc.unic.ac.cy/history-of-competitions/`
[9] Docker: `https://www.docker.com/`

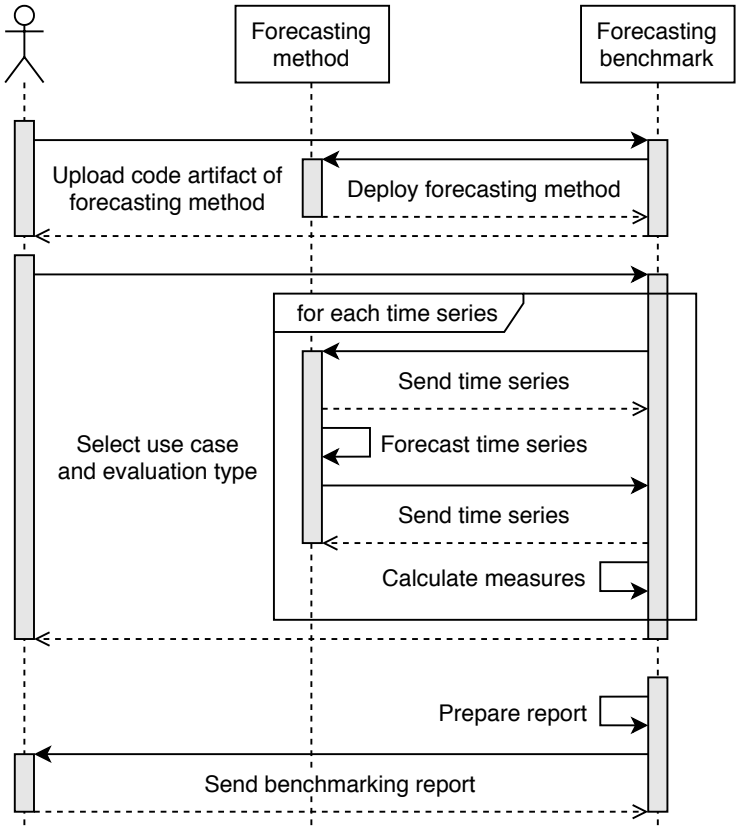

Figure 7.2: Sequence diagram for the usage of the forecasting benchmark.

(see Section 7.4). After the user has selected the settings for the benchmark process, the benchmark shuffles the set of time series within the domain. That is, the deployed forecasting method receives the time series in a random order one after another to mitigate fraud (e.g., optimizing the forecast error based on the order of the time series).

For each time series within the domain, the benchmark splits the time series into a training and test time series. The split depends on the evaluation type (e.g., 80% and 20% in the case of multi-step-ahead forecasts). Then, the training time series and the forecast horizon are passed to the forecasting method. Based on this input, the forecasting method performs a forecast and submits it to the benchmark. Based on the forecast and the test time series, the benchmark calculates the sMAPE (see Section 3.3.2), the MASE (see Section 3.3.3) , and the proposed measures (see Section 7.5). Also, the benchmark records the

time-to-result of the forecasting method. To have comparable time-to-result measures (i.e., being independent of the hardware on which the benchmark runs), the benchmark also performs a forecast with sNaïve (see Section 3.1.1) for normalizing the measured time-to-result. After each time series is forecast, the benchmark creates a report that contains a detailed overview and ranking compared to the state-of-the-art methods. The overview shows the average and the standard deviation of the collected measures. The state-of-the-art methods in competition comprise ETS, NNetar, random forest, sARIMA, sNaïve, SVR, TBATS, Theta, and XGBoost. For details on the methods see Section 3.1 and 3.2. Note that the results (i.e., forecast error measures and normalized time-to-result) of these methods were conducted beforehand and saved within the benchmark.

7.3 Time Series Data Set

Our goal is to establish a level playing field for evaluating the performance of forecasting methods in a broad setting. To this end, a highly heterogeneous data set of time series that covers different aspects is required. In fact, numerous data sets are available online: The M-Competitions (such as M3[10] or M4[11]), the website Kaggle[12], `R` packages, and many more. Usually, the data sets are designed for a specific use case, are very homogeneous, or are based on certain assumptions. For instance, the M3-Competition contains 3003 time series from different domains. However, most time series have a high degree of similarity and a length of less than 100 data points. Although, for instance, the M4-Competition comprises 100,000 time series, these time series are also quite similar and have short frequencies. Consequently, both competitions cannot be used when evaluating forecasting methods for time series with high frequencies. A detailed analysis of the M-Competitions and other forecasting competitions can be found in Section 7.6.

To enable a comprehensive evaluation, we assembled a data set comprising 400 time series[13]. The time series are publicly available and originate from 50 different sources, including also time series from M3 and M4. Figure 7.3 shows the distribution of origins of the time series, which we combined into different groups for the sake of clarity. The groups are authorities from different

[10] M3-Competition: `https://forecasters.org/resources/time-series-data/m3-competition/`

[11] M4-Competition: `https://www.mcompetitions.unic.ac.cy/the-dataset/`

[12] Kaggle: `https://www.kaggle.com/`

[13] The time series are available at Zenodo.org: `https://go.uniwue.de/timeseries`.

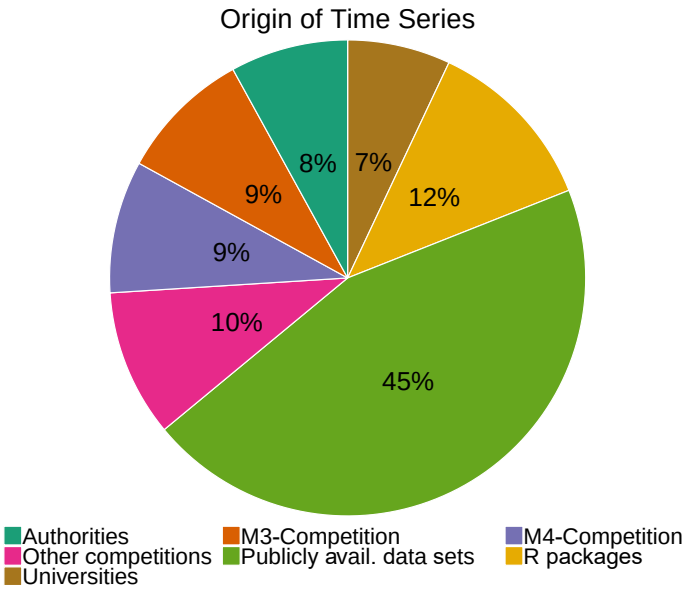

Figure 7.3: Distribution of the time series origins used in the data set.

countries, the M3-Competition, the M4-Competition, other competitions (e.g., Kaggle), various universities, R packages, and other publicly available data sets (not assignable to the other groups).

As the time series are available online and therefore the forecast of each time series is already known, we use a linear transformation to alter each time series. More formally, the transformed time series is

$$Y'(t) = u \cdot Y(t) + v, \tag{7.1}$$

where $Y(t)$ is the original time series, u a normally distributed random variable, and v a uniformly distributed variable. The exact description of the distributions is kept secret for the sake of benchmarking.

During the configuration of the benchmarking process, the user has to specify the use case and the evaluation type (see Section 7.2). More precisely, the user can choose between four different use cases, namely

1. *Economics* (gas, electricity, sales, unemployment, ...),
2. *Finance* (stocks, sales prices, gold, exchange rate, ...),
3. *Human access* (calls, SMS, Internet, requests, ...), and
4. *Nature and demographics* (rain, temperature, birth, death, ...),

each covering 100 heterogeneous time series taken from different fields. The length and frequency distributions of the use cases are shown as cumulative distribution functions in Figure 7.4 and Figure 7.5. Note that the horizontal axes in the figures are depicted in log-scale.

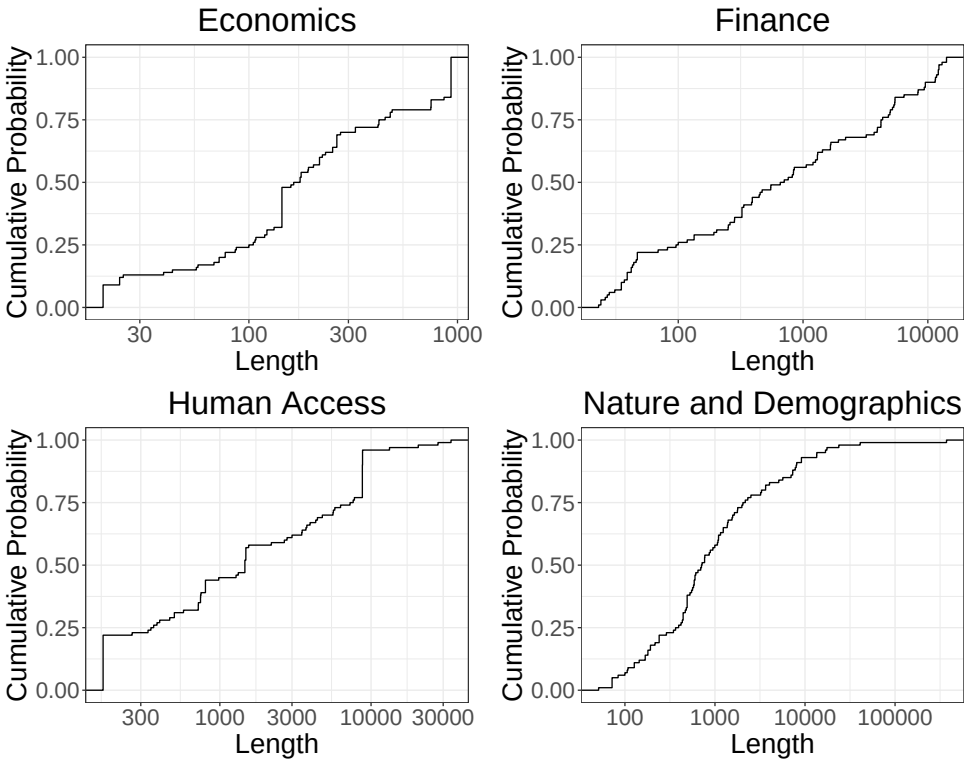

Figure 7.4: Distribution of the time series lengths in each use case.

7.4 Evaluation Types and Rolling Origin Evaluation

To quantify the forecast accuracy of a forecasting method, three different evaluation types are implemented in the forecasting benchmark: (i) *One-step-ahead forecasts*, (ii) *multi-step-ahead forecasts*, and (iii) *rolling origin forecasts*. For the first type, the forecasting method receives all values of the time series except the last one, which must be forecast. In some scenarios (e.g., auto-scaling), where a fine granularity leads to several data points in a short time, planning requires several values in advance. To reflect this need, the second type splits the time series in 80% training and 20% test (see Section 3.3). In other words,

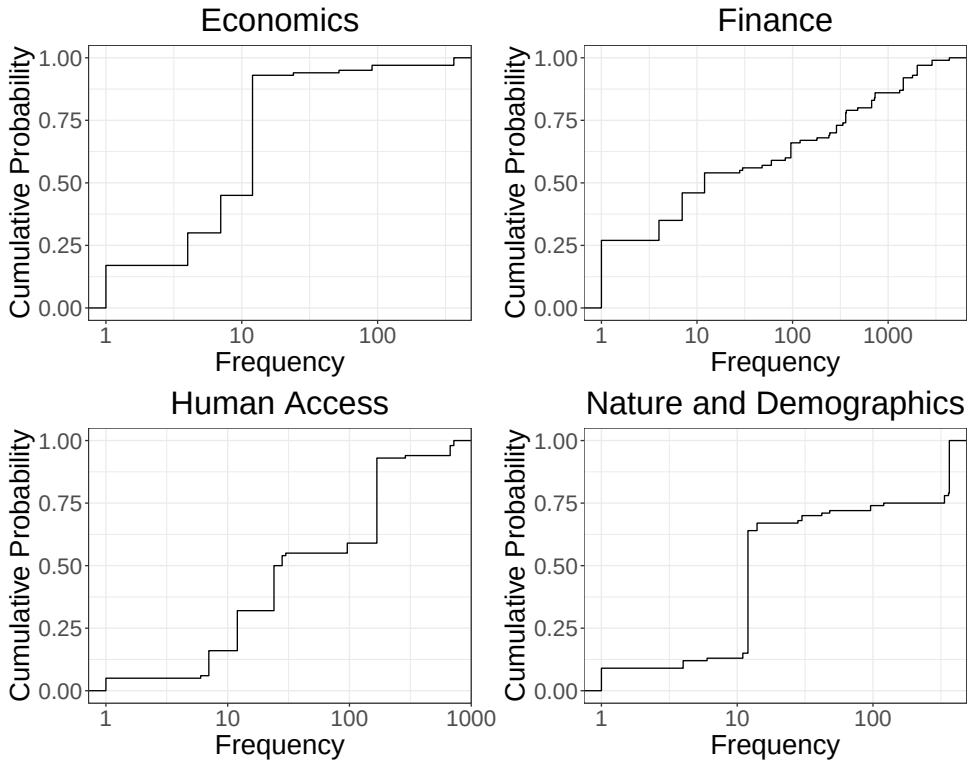

Figure 7.5: Distribution of the time series frequencies in each use case.

the forecast method has to forecast 20% of the time series at once. However, the first two evaluation types perform an "arbitrary" split, resulting in forecasts that are sensitive to occurrences that may only occur in that particular split. To stabilize the assessment of forecasting methods, the third evaluation type is based on *rolling origin* (Hyndman and Athanasopoulos, 2017). The evaluation based on rolling origin is the time series equivalent of cross-validation from the field of machine learning. The term *origin* refers in this context to the training set of the time series, which is successively enlarged. In other words, this technique allows obtaining multiple forecasts, each on the increasing training set of a single time series. Typically, the origin is increased by 1, leading to many forecasts for long time series. Consequently, the proposed forecasting benchmark offers a modified version of this rolling origin approach.

The core idea of the modified rolling origin is illustrated in Figure 7.6. The blue squares represent observations from the training set, the green squares observations from the test set, and the white squares the remaining observations

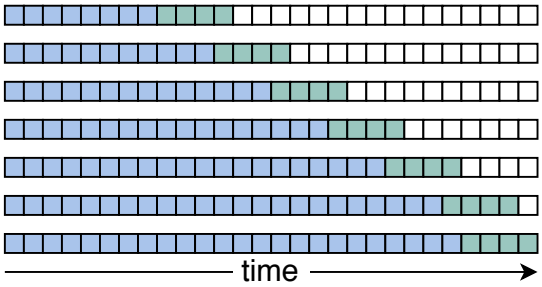

Figure 7.6: Concept of rolling origin forecast implemented in the benchmark.

from the time series. With every iteration except the last one, the training set is increased by a fixed number of observations. In the last iteration, the training set is enlarged so that both sets together cover the entire time series. The detailed rolling origin forecast procedure for a single time series is presented in Algorithm 7.1. As input, the procedure gets the time series *ts*, the horizon *h* (i.e., the number of values that should be forecast at once), and the forecasting method *f*. In the beginning, the rolling origin's start and end are set (Line 1–2). More precisely, the starting point is either at 40% of the time series or at two times the frequency of the time series plus 1, depending on which is greater. We choose these numbers to ensure that a seasonal pattern, if present, can be recognized. As the horizon is 20% of the time series length, the endpoint is set to 80% of the time series. If the numbers of observations between the starting and endpoint are less than or equal to 100 (Line 3–5), the end indices of the origin begin with the starting point and are successively extended by 1 to the endpoint. That is, the end indices contain $r + 1$ but a maximum of 101 points, where $r = end - start$. More formally, the end indices comprise the following points

$$\bigcup_{i=0,\ldots,r} \{y_{start+i}\}\,, \tag{7.2}$$

where y_t is the observation of the time series at time t. If the range defined by the starting and endpoint is greater than 100 points (Line 5–10), the range is divided into 100 parts of equal size. As the length of each split may not be an integer, the length is rounded up, and the resulting integer is referred to as step. Similar to the first case, the end indices of the origin begin with the starting point and are successively increased by the step to the endpoint. Therefore, the end indices contain $q + 1$ but a maximum of 101 points, where $step = \lceil r/100 \rceil$

and $q = \lceil r/step \rceil$. Mathematically, the end indices include the following points

$$\{y_{end}\} \cup \bigcup_{i=0,\ldots,q} \{y_{start+i \cdot step}\}\,. \tag{7.3}$$

After the indices are determined, the algorithm iterates over each index and performs a forecast based on the rolling origin evaluation type (Line 13–21). More precisely, in each iteration, the training set is created, starting with the first observation of the time series and ends with the current index (Line 14). At the same time, the test set (Line 15) comprises the following h observations of the time series (see Figure 7.6). The deployed forecasting method then performs an h-step-ahead forecast based on the training set (Line 17). After the forecast, the time-to-result is determined (Line 18), and the measures (i.e., forecast error measures and normalized time-to-result) are calculated (Line 19). Finally, the measures for each forecast performed are returned. Based on the measures, different statistics such as the average or the standard deviation of each measure can be calculated.

7.5 Proposed Forecast Error Measures

For the assessment of the forecasting method, our benchmark uses the established forecast error measures sMAPE and MASE (see Section 3.3) due to their scale independence. Moreover, we choose these measures, as each can be expressed with a deterministic mathematical expression and are in the interval $[0; \infty)$, where 0 is the optimal value. For the benchmarking report, the average, standard deviation, and the distribution of each measure is output. In addition to these error measures, the benchmark also records the time-to-result for each forecast. Since the time depends on the underlying operating system and hardware, each time series is forecast by the deployed forecasting method and also by sNaïve. Then, the time-to-result of the deployed method is normalized by the reference time of the sNaïve forecast. Besides the sMAPE and MASE, the benchmark also implements two novel forecast error measures to give useful insights into the benchmarked forecasting method. More precisely, we propose the *mean wrong-estimation shares* and the *mean wrong-accuracy shares* that are easy to interpret and scale-independent. The first measure indicates whether the forecasting method under- or over-estimates the future values, while the second measure indicates the extent of the wrong-estimation. Based on these measures, the forecasts could be adjusted to approximate the actual values. For example, if the forecasting method tends, on average, to under-estimate

Algorithm 7.1: Rolling origin forecast.

Input: Time series *ts*, horizon *h*, forecasting method *f*
Result: Measures of *f* on *ts*

```
start = max(floor(0.4 · length(ts)), 2 · frequency(ts) + 1) // minimal
  length of the rolling origin
end = floor(0.8 · length(ts))
if end - start ≤ 100 then // ts[start to end] is too short
    indices = start to end
else // indices are [start, start + step, ..., end]
    step = ceil((end - start)/100) // Creating equidistant steps
    indices = start to end by step
    if end not in indices then
        indices.append(end)
    end
end
measures = [ ]
foreach i in indices do
    hist = ts[1 to i] // i is the end index of the rolling origin
    test = ts[i + 1 to i + h]
    time = getSystemTime()
    forecast = f.forecast(hist, h)
    time = getSystemTime() - time
    m = calcMeasures(forecast, test, time) // calculating the forecast
      error measures and normlazied time-to-result
    measures.appendRow(m)
end
return measures
```

the actual value by 10%, the forecast values can be increased by about 12% to create a safety buffer.

7.5.1 Mean Wrong-Estimation Shares

The core idea of the mean wrong-estimation shares is to capture the tendency of the forecasting method to under- or over-estimate actual values. More precisely, the *mean under-estimation share* ρ_U is the number of forecast values relative to the whole forecast where the forecast value is below the actual value. Analogously, the *mean over-estimation share* ρ_O is the relative number of values where the

forecast value lies over the actual value. Both measures lie in the interval $[0, 1]$. The best value 0 is achieved when the forecasting method does neither under-estimate nor over-estimate the actual values. More formally, the two metrics ρ_U and ρ_O can defined as

$$\rho_U := \frac{1}{k} \cdot \sum_{t=1}^{k} max(sgn(y_t - \hat{y}_t), 0), \tag{7.4}$$

$$\rho_O := \frac{1}{k} \cdot \sum_{t=1}^{k} max(sgn(\hat{y}_t - y_t), 0), \tag{7.5}$$

where k is the forecast horizon (i.e., the length of the forecast), y_t the actual value at time t, and $\hat{y}_t$ the forecast value at time t.

7.5.2 Mean Wrong-Accuracy Shares

In contrast to the mean wrong-estimation shares, the mean wrong-accuracy shares describe how much the forecasting method under- or over-estimate the actual values on average. In other words, the *mean under-accuracy share* δ_U is the mean percentage error between the forecast values and the actual values where the forecasting method under-estimates the actual values. Analogously, the *mean over-accuracy share* δ_O is the mean percentage error where the forecasting method over-estimates the actual values. Values of both measures lie in the interval $[0, \infty)$, where 0 is the best value and indicates that there is no under- or over-estimation. Both measures can be defined[14] as

$$\delta_U := \begin{cases} \frac{1}{k \cdot \rho_U} \cdot \sum_{t=1}^{k} \frac{max(y_t - \hat{y}_t, 0)}{|y_t|}, & \exists t : \hat{y}_t < y_t, \\ 0, & \forall t : \hat{y}_t \geq y_t, \end{cases} \tag{7.6}$$

$$\delta_O := \begin{cases} \frac{1}{k \cdot \rho_O} \cdot \sum_{t=1}^{k} \frac{max(\hat{y}_t - y_t, 0)}{|y_t|}, & \exists t : y_t < \hat{y}_t, \\ 0, & \forall t : y_t \geq \hat{y}_t, \end{cases} \tag{7.7}$$

where k is the forecast horizon (i.e., the length of the forecast), y_t the actual value at time t, and $\hat{y}_t$ the forecast value at time t. Note that the conditions $\forall t : \hat{y}_t \geq y_t$ and $\forall t : y_t \geq \hat{y}_t$ are equal to $\rho_U = 0$ and $\rho_O = 0$, respectively. Analogously, the condition $\exists t : \hat{y}_t < y_t$ and $\exists t : y_t < \hat{y}_t$ are equal to $\rho_U > 0$ and $\rho_O > 0$, respectively.

[14] Note that, like all percentage error measures, neither measure is undefined if $y_t = 0$.

7.6 Comparison with other Forecasting Competitions

During the last decades, several forecasting methods have been proposed and evaluated (see Section 7.1). Although some forecasting competitions have been established, for instance, the M-Competitions initiated by S. Makridakis (see Section 5.4), most of these papers applied their own evaluation methodology. As shown by our review, the level of these evaluations is of poor quality (e.g., only a few competing methods and/or time series). To avoid the weakness of the reviewed scientific papers that we have criticized, we compare the forecasting benchmark in this section with prominent forecasting competitions. More precisely, we compare our assembled data set with publicly available competitions, namely M1 (Makridakis et al., 1982), M3 (Makridakis and Hibon, 2000), M4 (Makridakis et al., 2018b), NN3 (Crone et al., 2011), NN5[15], NNGC1[16], and Tourism (Athanasopoulos et al., 2011). To this end, we first compare the different data sets regarding their time series characteristics. Then, we investigate how similar the data sets are.

7.6.1 Time Series Characteristics

To investigate a time series, we can either examine the observations or the time series characteristics describing the time series. This section investigates the latter to compare our data set to the data sets from the forecasting competitions. More precisely, we examine the distribution of the frequencies, the lengths, and 25 time series characteristics[17]. To be unbiased (i.e., not taking our proposed time series characteristics in Section 8.8.1.1) into account), we consider time series characteristics proposed by R. Hyndman et al. (Hyndman et al., 2015), Y. Kang et al. (Kang et al., 2017), and B. Fulcher et al. (Fulcher et al., 2013).

Table 7.1 shows the frequency distribution of the individual data sets. The competitions M1, M3, M4, NN3, and Tourism comprise only time series with a low frequency. That is, the frequencies range between 1 to 24. More precisely, the M1, M3, and Tourism competitions support only the frequencies 1, 4, and 12, while the M4-Competition additionally supports the frequency of 24. The NN3 and NN5 competitions both support only a single frequency, while the NNGC1 competition supports only the frequencies 365 and 7305. In contrast, our data set supports 37 different frequencies starting from 1 to 4368.

[15] NN5 competition: `http://www.neural-forecasting-competition.com/NN5/`

[16] NNGC1 competition: `http://www.neural-forecasting-competition.com/`

[17] The calculated time series characteristics of the data sets as well as the description of the characteristics are publicly available at Zenodo: `https://zenodo.org/record/4115345`

Table 7.1: Frequency distribution within each data set.

Frequency	Our	M1	M3	M4	NN3	NN5	NNGC1	Tourism
Min.	1	1	1	1	12	365	365	1
1st Qu.	7	4	1	1	12	365	365	1
Median	12	12	4	4	12	365	7305	4
3rd Qu.	168	12	12	12	12	365	7305	12
Max.	4368	12	12	24	12	365	7305	12

Next, we investigate the length distribution as shown in Table 7.2. For instance, the time series from the NN5 competition all have the same length, with 791 observations. The median length of the time series within the M1, M3, NN3, and Tourism competitions is less than or equal to 134. In contrast, the median length of our time series is 570. While examining the interquartile range, our data set has a range of about 3000 observations, while the other data sets have less than 300 observations. Furthermore, our data set also contains the longest time series with 372,864 observations. In comparison, the longest time series from the M4-Competition – which has the longest time series among the other competitions – comprises only 9993 observations. To sum up, our data set shows the highest diversity in terms of the time series' lengths.

Table 7.2: Length distribution within each data set.

Length	Our	M1	M3	M4	NN3	NN5	NNGC1	Tourism
Min.	20	15	20	19	68	791	502	11
1st Qu.	169	44	44	56	69	791	747	27
Median	570	68	69	106	134	791	902	110
3rd Qu.	2074	85	133	252	144	791	1026	199
Max.	372,864	150	144	9933	144	791	1742	333

Besides the lengths and frequencies distribution, we investigate time series characteristics proposed by different scientific works (Fulcher et al., 2013; Hyndman et al., 2015; Kang et al., 2017). The details of the considered characteristics are given in Section 2.4. Since some characteristics such as auto-correlation coefficients are applied to the time series and the first-order and second-order differenced time series, the comparison of the data sets comprises 25 time series characteristics. For the investigation, we apply min-max scaling to all time series characteristics, taking all time series of all considered data sets into account.

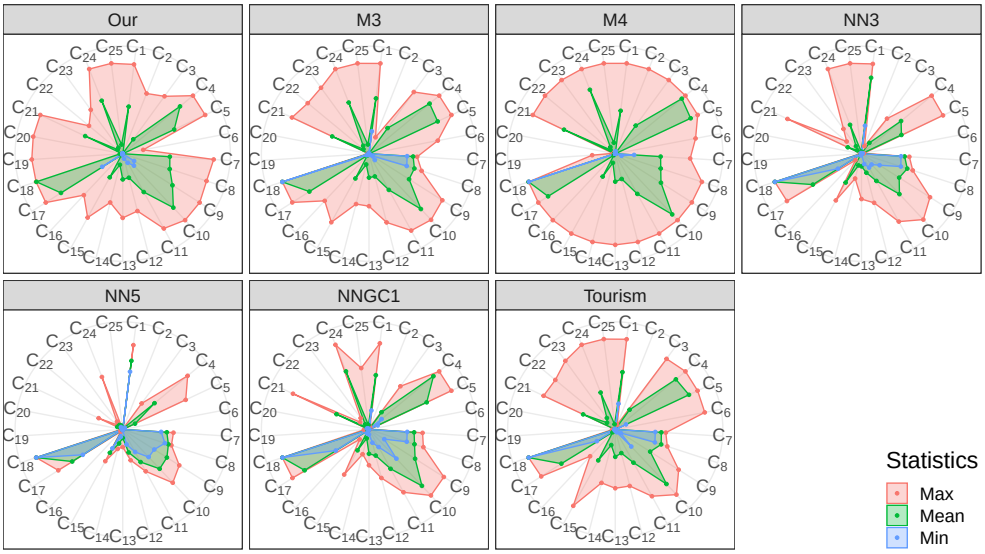

Figure 7.7: Distribution of time series characteristics per investigated data set.

This means that for each time series characteristic, the minimum (0) and the maximum (1) can be located in different data sets. The resulting distribution of each data set is illustrated as a spider-chart in Figure 7.7. Each chart contains 25 edges, each representing a time series characteristic. For each time series characteristic, the red dot represents the largest value in the data set, the green dot the average value, and the blue dot the smallest value. Based on these charts, our data set exhibits a higher diversity of time series characteristics than all other competitions except the M4-Competition. Our data set has the minimum value for 9 out of 25 time series characteristics and the maximum value for 10 out of 25 time series characteristics. The M4-Competition covers the remaining minima and maxima. However, the M4-Competition comprises 100,000 time series. Therefore the likelihood of having a time series with a minimum/maximum for a time series characteristic is higher than in our data set.

7.6.2 Distance between Time Series

To determine how similar time series are to each other in the individual data sets, we calculate the distance between them. If we think of the geometric representation of a time series, we could, for example, consider the Euclidean distance or dynamic time warping. However, the first metric can only be

calculated if the time series are of equal length and the latter is difficult to interpret. To this end, we adopt the idea of the zoomed ranking (Dos Santos et al., 2004) approach applied to time series. More precisely, the distance between two time series is equal to the L_1-norm of their describing time series characteristics. Mathematically, the distance between two time series Y_i and Y_j is defined as (Dos Santos et al., 2004)

$$d(Y_i, Y_j) := \sum_{m=1}^{q} \frac{|c_{Y_i,m} - c_{Y_j,m}|}{\max\limits_{k=\{1,\ldots,s\}/\{i\}}(c_{Y_k,m}) - \min\limits_{k=\{1,\ldots,s\}/\{i\}}(c_{Y_k,m})}, \tag{7.8}$$

where $Y_1, \ldots, Y_s$ are all time series from all considered data sets with s being a positive integer, Y_i and Y_j origin from the same data set, and $c_{Y_i,1}, \ldots, c_{Y_i,q}$ are the describing time series characteristics of time series Y_i with q being a positive integer. The time series characteristics are the same as used in Section 7.6.1, i.e, $q = 25$. The distance between two time series lies in the interval $[0, \infty)$, where 0 indicates that the time series are equal in terms of their describing characteristics. Consequently, the higher the distance, the more heterogeneous are both time series.

To analyze the different data sets, we calculate the distance between each pair of time series within the data set and report the average distance as well as the respective distribution in Table 7.3. The time series in our data set have an average distance of 4.104, while the time series in the other data have an average distance between 1.087 and 3.304. Moreover, the M-Competitions and the NN5 competitions have time series that are identical regarding their describing time series characteristics. Except for the minimal value, our data set exhibits the highest average time series distance for all other quartiles. As for the maximum value, our data set's average distance is at least twice as far as in all other competitions. In summary, our data set exhibits the higher average distance and thus the greater heterogeneity between time series.

Table 7.3: Distance between time series within each data set.

Distance	Our	M1	M3	M4	NN3	NN5	NNGC1	Tourism
Min.	0.003	0.000	0.000	0.000	0.237	0.000	0.018	0.033
1st Qu.	2.697	2.125	2.001	1.669	1.632	0.681	2.077	2.072
Median	3.866	2.950	2.968	2.473	2.797	0.992	2.867	2.899
Mean	4.104	3.186	3.304	2.808	3.112	1.087	2.900	3.024
3rd Qu.	5.296	4.027	4.331	3.594	4.291	1.405	3.660	3.846
Max.	27.898	10.785	10.876	13.583	8.758	3.235	8.395	9.092

7.7 Concluding Remarks

In this chapter, we first surveyed existing work on time series forecasting with the goal of assessing how forecasting methods are evaluated in the literature. Considering the shortcomings of the reviewed articles and the research question RQ 1 "*How to automatically compare and rank different forecasting methods on a level playing field based on their performance in a diverse set of evaluation scenarios?*", we propose a novel benchmark that automatically evaluates and ranks forecasting methods based on their performance in a diverse set of evaluation scenarios. More precisely, the benchmark provides a level playing field for evaluating the performance of forecasting methods in a broad setting by selecting a specific use case and evaluation type. Moreover, we showed that the assembled data set has a higher diversity than established forecasting competitions such as the well known M-Competitions. To address the research question RQ 2 "*What are suitable and reliable measures for quantifying the quality of forecasts?*", the benchmark implements well-established forecast error measures and introduces further two novel measures that give more insights into the benchmarked forecasting method.

Chapter 8

Automated Hybrid Forecasting Approach

Time series forecasting is an essential pillar in many decision-making disciplines (Hyndman and Athanasopoulos, 2017). Accordingly, time series forecasting is an established and active field of research, and thus various methods have been proposed. Due to the variety of approaches, the choice and configuration of the best performing method for a given time series remain a mandatory expert task to avoid trial-and-error. However, expert knowledge can be expensive, may have a subjected bias, and it can take a long time to deliver results.

Thus, the question arises if there is a single forecasting method that performs best for all time series. However, the "No-Free-Lunch Theorem" (Wolpert and Macready, 1997), initially formulated for optimization problems, denies the possibility of such a method. It states that improving one aspect typically leads to a degradation in performance for another aspect. An analogy can be drawn to the domain of forecasting: No forecasting method performs best for all time series. In other words, forecasting methods have their advantages and drawbacks depending on the considered time series.

In fact, different types of hybrid forecasting methods have been proposed in the last years (Fontes and Castro Silva, 2020) to tackle the challenge stated by the "No-Free-Lunch Theorem". The core idea is the usage of at least two methods to minimize the disadvantages of individual methods. For instance, while statistical models have their difficulties with complex patterns, regression-based machine learning methods struggle with non-stationary data (see Section 2.1.3) to extrapolate for a forecast (Sugiyama and Kawanabe, 2012). From our experience, recently presented open-source hybrid methods are computationally intensive, poorly automated, tailored to a particular data set, or they lack a predictable time-to-result. However, many real-world scenarios where forecasting is useful (e.g., auto-scaling) have strict requirements for a reliable time-to-result and forecast accuracy. To achieve a low variance in forecast accuracy, the preprocessing of historical data and the feature handling (extraction, engineering, and selection) must be done in a sophisticated way. On the one hand, the choice

of the essential features is a decisive part. On the other hand, transforming historical data may lead to simpler patterns that usually allows more accurate forecasts (Hyndman and Athanasopoulos, 2017).

To tackle the mentioned challenges, we pose ourselves the following research questions:

RQ 3: *How to design an automated and generic hybrid forecasting approach that combines different forecasting methods to compensate for the disadvantages of each technique?*

RQ 4: *How to automatically extract and transform features of the considered time series to increase the forecast accuracy?*

RQ 5: *What are appropriate strategies to dynamically apply the most accurate method within the hybrid forecasting approach for a given time series?*

Towards addressing the research questions, our contribution is the design of a novel, generic, hybrid forecasting method called *Telescope*[1]. Telescope is a machine learning-based forecasting method that automatically retrieves relevant information from a given time series. More precisely, Telescope automatically extracts intrinsic time series features and then decomposes the time series into components, building a forecasting model for each of them. Each component is forecast by a different method and then the final forecast is assembled from the forecast components by employing a regression-based machine learning algorithm. In other words, we integrate different methods to handle non-stationary data introduced by trend and multiplicative effects.

While Telescope is applied in a non-time-critical scenario and under consideration of the "No-Free-Lunch Theorem", Telescope also extracts time series characteristics and employs the best-suited regression-based machine learning algorithm based on dynamically learned rules. For building the recommendation rules dynamically, a knowledge base is built upon a set of time series. To augment the knowledge base, a time series generator is employed to create numerous new time series with different characteristics out of a small set. Moreover, the Telescope approach and its components are evaluated in Chapter 10.

The remainder of this chapter[2] is organized as follows: We start with the design overview of Telescope in Section 8.1. As the workflow of Telescope can be divided into different phases, Section 8.2–8.6 explain each of the phases. In Section 8.7, the fallback for non-seasonal time series is introduced. To tackle

[1] Telescope at GitHub: https://github.com/DescartesResearch/telescope

[2] This chapter is based on our previous works (Züfle et al., 2017; Bauer et al., 2020a,b,c).

challenge posed by the "No-Free-Lunch Theorem", Telescope has a recommendation system deployed, and Section 8.8 focuses on the meta-learning approach and its components. In Section 8.9, the assumptions and limitations of Telescope are highlighted. We delimit our approach from the reviewed techniques (see Chapter 5) in Section 8.10. Finally, the chapter is summarized in Section 8.11.

8.1 Design Overview

The assumption of data stationarity is an inherent limitation for time series forecasting. Any time series property that eludes stationarity, such as non-constant mean (i.e., trend), seasonality, non-constant variance, or multiplicative effect, poses a challenge for the proper model building (Makridakis et al., 2018a). Consequently, we take all the techniques discussed in Chapter 2 into account to design an automated forecasting workflow called Telescope that automatically transforms the given time series, derives intrinsic features from the time series, selects a suitable set of features, and handles each feature separately. The choice and combinations of different methods and techniques are discussed and evaluated in Chapter 10. Hence, the selection of the deployed methods, as discussed in the following sections, reflects the best configuration. Also, if Telescope is applied in a non-time-critical scenario, the recommendation system dynamically selects the regression-based machine learning method integrated into Telescope for the given time series.

Algorithm 8.1: Telescope forecasting workflow.

Input: Time series ts, horizon h
Result: Forecast of ts

```
[ts, freqs] = Preprocessing(ts) // see Section 8.2
if freqs[1] > 1 then // ts is seasonal
    features = FeatureExtraction(ts) // see Section 8.3
    model = ModelBuilding(ts, features) // see Section 8.4
    forecast = Forecasting(model, h) // see Section 8.5
else
    forecast = ARIMA(ts, h) // see Section 8.7
end
forecast = Postprocessing(forecast) // see Section 8.6
return forecast
```

The workflow of Telescope is briefly illustrated in Algorithm 8.1 and gets as input a univariate time series ts and the horizon h. The horizon specifies

how many values have to be forecast at once. In the first phase (Line 1), the time series is preprocessed and the frequencies of the underlying patterns are extracted. Telescope is intended to handle seasonal time series as many time series are observed or produced by systems subjected to human habits and are thus seasonal. In other words, if a seasonal time series has to be forecast, the second and third phases of Telescope comprise the extracting of relevant intrinsic time series features (Line 3) and building a model that describes the time series based on these features (Line 4). Afterward, the model is used to forecast the behavior of the future time series (Line 5). In the unlikely case where no seasonality exists within a time series(Line 7), the time series is modeled and forecast with ARIMA (see Section 3.1.4). Finally, the forecast is postprocessed according to the preprocessing phase and returned. In the following, each phase is explained in detail.

8.2 Preprocessing

The first phase of Telescope is called *Preprocessing* and the workflow is depicted in Figure 8.1. Orange, rounded boxes represent actions, green hexagons the input of a phase, and blue trapezoids the output of a phase. This phase gets as input the *Time Series*, which is prepared for the following phases. As forecasting methods, especially machine learning methods, struggle with changing variance and multiplicative effects within a time series (Sugiyama and Kawanabe, 2012), the time series is transformed. More precisely, Telescope applies the *Box-Cox Transformation* (see Section 2.3.2) to the *Time Series*. We integrated this transformation step as it reduces both variance and multiplicative effects of the time series, leading to an improved forecast model (Hyndman and Athanasopoulos, 2017; Makridakis et al., 2018a). For estimating the *Transformation Parameter* of the Box-Cox transformation, we apply the method proposed by Guerrero (Guerrero, 1993) and restrict the parameter to values greater than or equal to zero. As the Box-Cox transformation can result in a logarithmic transformation, the time series has to be "real" positive ($\forall t : y_t > 0$). Consequently, the time series is shifted along the ordinate before the transformation if there is at least one value less than or equal to zero. More formally, the shift of the time series $Y(t)$ is defined as

$$\gamma(Y(t)) := \begin{cases} Y(t) + |\min_t(Y(t))| + 1, & \exists t : y_t \leq 0, \\ Y(t), & otherwise. \end{cases} \tag{8.1}$$

In parallel to the transformation, Telescope performs the *Frequencies Estimation*. In short, the main idea of this step is to retrieve the most dominant

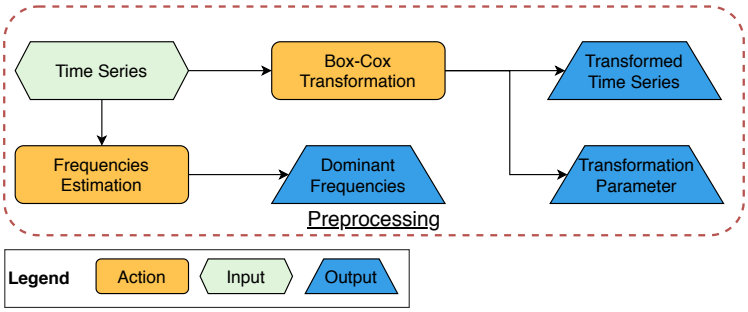

Figure 8.1: Preprocessing phase of Telescope.

frequencies from the input time series by applying a periodogram (see Section 2.2.2). The detailed procedure is depicted in the Algorithm 8.2. As input, the algorithm receives the time series *ts*, an estimation tolerance value *tol*, and a threshold for dominant[3] frequencies *spec_th*. First, the frequencies within the time series and their associated spectrum are estimated. The function *applyPeriodogram*, as the name suggests, applies a periodogram to the time series and sorts the estimated frequencies in descending order according to their spectrum. Accordingly, the most dominant frequency is the first frequency. In general, the i-th most dominant frequency has the i-th highest spectrum. However, finding the highest spectrum does not necessarily indicate the most dominant frequency. If the spectrum is only slightly higher than at the remaining frequencies, the time series is probably random. To this end, a second periodogram is applied on the n-order differenced time series (see Section 2.3.3, where n is equal to the most dominant frequency). Then, the highest spectrum of the original and the differenced time series are compared: If the highest spectrum of the differenced time series is smaller than a fraction of the highest spectrum of the original time, the time series is considered as seasonal and all found frequencies are returned. Otherwise, the time series is handled as non-seasonal and both the function as well as the algorithm return the frequency of 1 (Line 2–4). As the periodogram is only an estimation of the spectral density of a time series, the algorithm tries to match each estimated frequency with reasonable frequencies (e.g., daily, hourly, or yearly). That is, Telescope assumes that reasonable frequencies match with multiples of natural time units. To this end, common minute-based, hour-based, and day-based time units and their combinations are considered (Line 6). Examples for the common time units are a year (365 daily observations), a week (7 daily observations), or

[3]By dominant, we mean the most common period such as days in a year.

Algorithm 8.2: Frequencies Estimation.

Input: Time series *ts*, estimation tolerance *tol*, threshold *spec_th*
Result: Dominant frequencies within *ts*

```
1  [freqs_est, spec] = applyPeriodogram(ts) // frequencies are sorted in
     descending order according to their spectrum
2  if length(freqs_est) == 1 and freqs_est[1] == 1 then // ts is non-seasonal
3  |  return 1
4  end
5  freqs = [ ]
6  freqs_nat = getNaturalFrequencies() // for example, 24 for hourly
     observations or 365 for daily observations
7  for i = 1 to length(freqs_est) do
8  |  if spec[i] < spec_th * max(spec) then // frequency is too less dominant
9  |  |  break
10 |  end
11 |  bounds = [freqs_est[i]*(1-tol), freqs_est[i]*(1+tol)]
12 |  freq = 1
13 |  min = ∞
14 |  foreach natural in freqs_nat do
15 |  |  if natural in bounds then
16 |  |  |  delta = abs(natural - freqs_est[i])
17 |  |  |  if delta < min then // choosing the natural frequency that is
   |  |  |    closest to the estimated frequency
18 |  |  |  |  min = delta
19 |  |  |  |  freq = natural
20 |  |  |  end
21 |  |  end
22 |  end
23 |  freqs.append(freq)
24 end
25 return unique(freqs)
```

an hour (3600 secondly observations). For instance, the value of $96 = 24 \cdot 4$ represents a seasonal pattern of one day (24 hours) while the observations were recorded every 15 minutes (i.e., quarter-hourly). Another example is the value of $336 = 7 \cdot 24 \cdot 2$ that reflects a weekly pattern (7 days) while the observations are taken each 30 minutes (48 half hours a day). In the next step, the algorithm iterates over all estimated frequencies (Line 7–23). In accordance with RQ 4,

Telescope considers only the most dominant frequencies within the time series. If the i-th spectrum is less than the maximal spectrum times the threshold, the loop is terminated (Line 8–10). As the frequencies are ordered according to their spectrum, only frequencies with a low spectrum are omitted. To take the periodogram's inaccuracy into account, a tolerance interval is placed around the currently estimated frequency (Line 11). To match the current frequency with a reasonable frequency, the algorithm iterates over each created natural frequency and checks which natural frequency lies within the tolerance interval (Line 13–21). If a natural frequency lies in the interval, the absolute distance to the estimated frequency is calculated (Line 15). Then, the natural frequency with the lowest distance is determined (Line 16–19) and appended to the list of dominant and reasonable frequencies (Line 22). Finally, the unique set of dominant and reasonable frequencies is returned.

The resulting *Transformed Time Series* from the transformation is then forwarded as input for the phases *Feature Extraction* and *Model Building*, and the *Transformation Parameter* determined during the transformation is passed to the *Postprocessing* phase. The *Dominant Frequencies* are forwarded as input for the *Feature Extraction* phase. In the case that the set of dominant frequencies contains only the frequency of 1 (i.e., the time series is non-seasonal), the *Feature Extraction*, *Model Building*, and *Forecasting* steps are omitted and the *Fallback* phase (see Section 8.7) is performed with the *Transformed Time Series* as input.

8.3 Feature Extraction

The second phase, *Feature Extraction*, is depicted in Figure 8.2, which has the same structure as Figure 8.1. As input, this phase gets the *Transformed Time Series* and the *Dominant Frequencies* from the *Preprocessing* phase. Based on the input, Telescope retrieves intrinsic time series features for tackling typical problems or difficulties that may occur during the modeling of a time series. The first difficulty is that time series may have multiple underlying seasonal patterns (Hyndman and Athanasopoulos, 2017). To this end, Telescope determines for each dominant frequency the associated *Fourier Terms* (see Section 2.2.1) of the *Transformed Time Series* for modelling the different patterns. More precisely, for each dominant frequency, a sine as well as a cosine with the period length of the corresponding frequency are retrieved from the time series.

Another problem are time series violating the stationary property (see Section 2.1.3). Although most forecasting methods assume stationary time series (Brockwell and Davis, 2016), many time series exhibit trend and/or seasonal patterns. That is, in practice, time series are usually non-stationary (Ad-

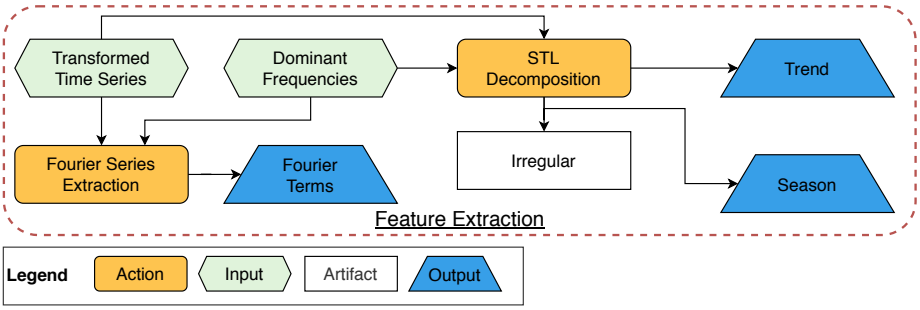

Figure 8.2: Feature extraction phase of Telescope.

hikari and Agrawal, 2013). To handle the non-stationarity, the core idea of Telescope is to decompose the time series and then deal with each part separately. To this end, the *Transformed Time Series* is split into its components *Trend*, *Season*, and *Irregular* with STL (see Section 2.3.1). For the decomposition task, the most dominant frequency is used to specify the length of the seasonal pattern and the extraction of the pattern is set to periodic, that is, we assume that the seasonal pattern does not evolve over time. Although STL can only deal with an additive relationship between the components of a time series, it is not examined whether the time series follows an additive or multiplicative decomposition. More precisely, we assume that the Box-Cox transformation in the *Preprocessing* phase has minimized or removed the multiplicative effects.

The extracted features of this phase that are forwarded to both the *Model Building* and the *Forecasting* phase comprise the *Fourier Terms*, *Trend*, and *Season*. Note that the irregular part of the time series is not considered as a feature since each feature has to be forecast; however, the irregular part is inherently impossible to forecast.

8.4 Model Building

In the third phase called *Model Building*, the model that reflects the time series is build on the inputs *Transformed Time Series*, *Trend*, *Fourier Terms*, and *Season*. To build a suitable model describing the time series, we apply machine learning for finding the relationship between the time series and the intrinsic features. In a time-critical scenario, that is, the forecast is required with a reliable time-to-result, Telescope implements XGBoost (see Section 3.2.5) as regression-based machine learning method. We choose XGBoost since boosting tree algorithms are time-efficient, accurate, and easy to interpret (Ke et al., 2017). Moreover, we

excluded other methods like SVM, Random Forest, or artificial neural networks due to their unfeasible run-time (Makridakis et al., 2018a). In a scenario where the time-to-result is negligible, Telescope uses its recommendation system according to RQ 5 for selecting the most appropriate regression-based machine learning method for the given time series.

8.4.1 Time-Critical Scenario

Figure 8.3, which has the same structure as Figure 8.1, shows the *Model Building* phase in a time-critical scenario. As a strong trend both increases the variance and violates stationarity, the trend introduces challenges for the model building. To this end, the first step is the removal of the trend from the time series. The resulting *De-trended Time Series* is now trend-stationary. Although seasonality can also violate stationarity, machine learning methods are suitable for pattern recognition (Dietterich, 2002). Consequently, XGBoost learns during its training procedure how the *De-trended Time Series* can be described by the intrinsic features *Fourier Terms* and *Season*. More formally, the XGBoost model can be mathematically written as

$$f(Y^T(t), S(t), \{\sin(2\pi\nu_i t)\}_{i=1}^m, \{\cos(2\pi\nu_i t)\}_{i=1}^m) := \hat{Y}^T(t) + \epsilon_t, \tag{8.2}$$

where $Y^T(t)$ is the de-trended time series, $\hat{Y}^T(t)$ the estimation of $Y^T(t)$, $S(t)$ the seasonal component, ϵ_t the estimation error term, and the sinusoids terms represent the Fourier terms of the m most dominant frequencies ν_i. Note that the irregular part of the time series is not explicitly considered a feature to reduce the model error and later the forecast error. That is, the machine learning method learns the irregular part as the difference that is missing to fully recreate the de-trended time series. Finally, the *Model* describing the time series is forwarded to the *Forecasting phase*.

8.4.2 Non-Time-Critical Scenario

The *Model Building* phase in a non-time-critical scenario, which is depicted in Figure 8.4, is identical to the phase in a time-critical scenario, with the exception that no fixed machine learning method is used. In other words, the machine learning method is selected based on the time series. To this end, the *De-trended Time Series* is passed to the recommendation system that extracts characteristics of the time series. Based on these characteristics, the most suitable machine learning method is selected and used to learn how the intrinsic features can describe the time series. The details of the recommendation system are described in Section 8.8.

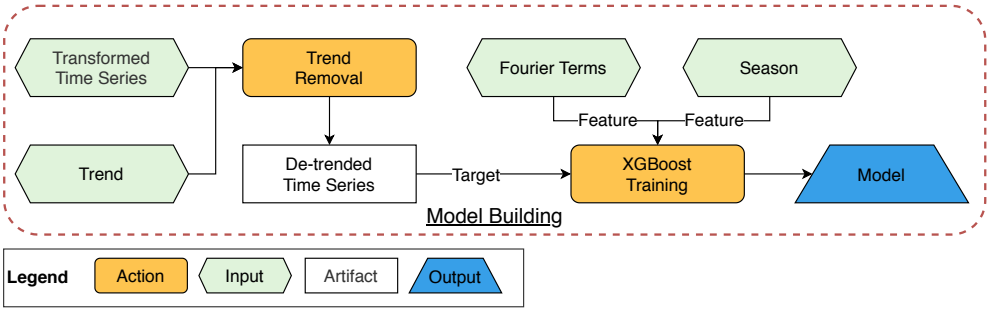

Figure 8.3: Model building phase of Telescope in a time-critical scenario.

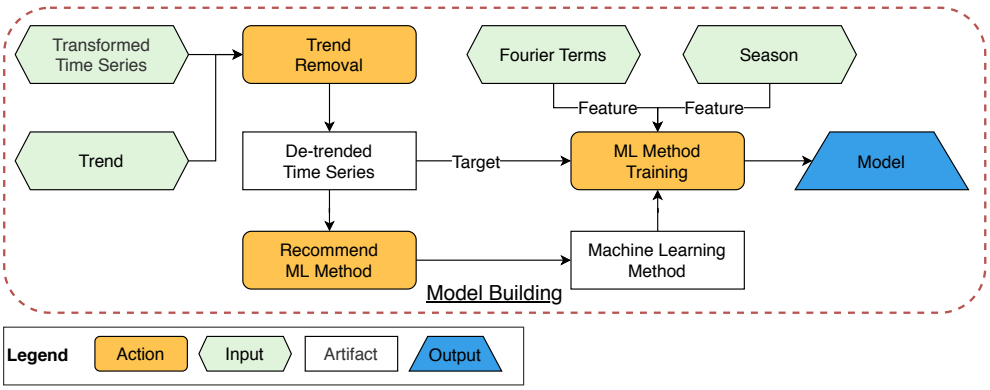

Figure 8.4: Model building phase of Telescope in a non-time-critical scenario.

8.5 Forecasting

In the *Forecasting* phase, which is illustrated in Figure 8.5 (same structure as Figure 8.1), the different components are forecast separately and then, the future time series is assembled. To this end, this phase gets as input the *Trend*, *Fourier Terms*, *Season* as well as the *Model*. As the *Season* and the *Fourier Terms* are recurring patterns per definition, they can be merely continued. More precisely, the seasonal pattern and each Fourier terms is forecast for the time $n + k$ as follows

$$\hat{y}_{n+k|n} := y_{n+k-m\cdot(\lfloor \frac{k-1}{m} \rfloor+1)}, \tag{8.3}$$

where y_t is the observations at time t, n the number of historical observations, m the length of the associated period, and the forecast horizon k being a positive integer. The resulting *Future Season* and *Future Fourier Terms* are used as features

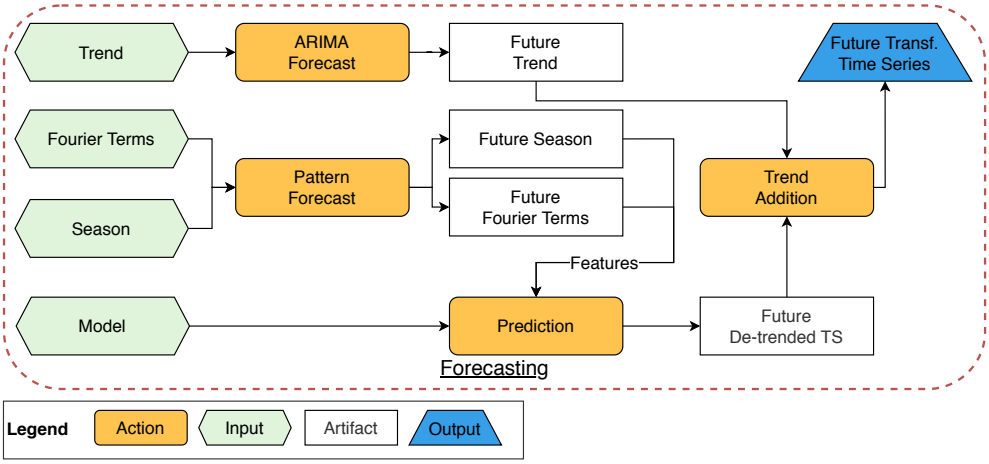

Figure 8.5: Forecasting phase of Telescope.

in conjunction with the *Model* for predicting the future de-detrended time series. More precisely, the machine learning method regresses a new value of the *Future De-trended TS* for each point in time of the forecast based on the corresponding values of the features.

In parallel to the forecast of the recurring patterns, the trend is also forecast. Since the *Trend* contains no recurring patterns, an advanced forecasting method is required to forecast the *Future Trend*. To this end, we apply ARIMA (see Section 3.1.4) as it is able to estimate the trend even from a few points. More precisely, we apply auto.arima[4] (Hyndman and Khandakar, 2008) that automatically finds the most suitable model parameters for a time series. However, before the trend can be forecast, the trend type has to be determined. We assume either a linear or exponential trend. The detailed procedure is depicted in Algorithm 8.3 and gets the trend t and the forecast horizon h as input. As the trend is logarithmized (Line 2), the trend must only contain values greater than zero. To this end, the trend is shifted along the vertical axis (see Equation 8.1) if there is at least one value less than or equal to zero. Then, a linear model is fitted to the positive trend as well as to the logarithmized[5] trend (Line 3–5). Afterwards, the RMSE (see Section 3.3.1) is calculated for both the linear and exponential trend model. If the RSME of the linear model is less than the RSME of the exponential model, Telescope assumes that the trend is linear, and the trend is forecast while applying ARIMA to the original trend (Line 8–9).

[4] Auto.arima conducts a search over possible sARIMA and ARIMA models.
[5] If the trend is exponential, the logarithmized trend results in a linear trend.

Otherwise, the trend is assumed to be exponential. The logarithmized trend is forecast, and the exponential function is applied to re-transform the forecast trend. Moreover, if the original trend was shifted to be positive, the forecast trend is shifted back (Line 10–13). Finally, the *Future Trend* is returned.

After the trend is forecast, the last step of this phase is assembling the forecast of the time series. To this end, the *Future De-trended TS* and the *Future Trend* are summed up. The resulting *Future Transf. Time Series* is then forwarded to the *Postprocessing* phase.

Algorithm 8.3: ARIMA Forecast.

Input: Trend t, horizon h
Result: Forecast of t

```
1  t_pos = shiftTrend(t) // see Equation 8.1
2  t_log = log(t_pos) // logarithmize the trend
3  x = 1 to length(t)
4  fit_lin = fitLinearModel(t_pos, x)
5  fit_exp = fitLinearModel(t_log, x)
6  rsme_lin = RMSE(t_pos, fit_lin.fitted()) // see Section 3.3.1
7  rsme_exp = RMSE(t_pos, exp(fit_exp.fitted())) // exp(log(a)) = a
8  if rsme_lin < rsme_exp then // RSME indicates a linear trend
9  |  forecast = ARIMA(t, h)
10 else // RSME indicates an exponential trend
11 |  forecast = exp(ARIMA(t_log, h))
12 |  forecast = deShiftTrend(forecast, t)
13 end
14 return forecast
```

8.6 Postprocessing

The last phase of Telescope is called *Postprocessing* and its workflow is depicted in Figure 8.6, which has the same structure as Figure 8.1. As the name suggests, this phase is the counterpart of the *Preprocessing* step. More precisely, it gets as input the *Future Transf. Time Series* from the *Forecasting* phase and the *Transformation Parameter* from the *Preprocessing* phase. As the time series was adjusted with the Box-Cox transformation, the *Future Transformed Time Series* has to be re-transformed with the identical transformation parameter from the *Preprocessing* phase. To this end, the inverse Box-Cox transformation with the *Transformation Parameter* is applied to the *Future Transformed Time Series*. If the

time series was shifted along the vertical axis in the first phase, the time series also has to be moved back. Finally, the forecast of the original time series is returned.

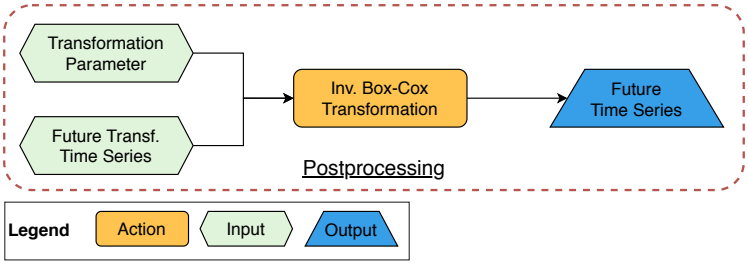

Figure 8.6: Postprocessing phase of Telescope.

8.7 Fallback for Non-Seasonal Time Series

Telescope's core idea is to detect recurring patterns within a time series and use this information to retrieve intrinsic features. However, if Telescope has to forecast a non-seasonal time series, the normal workflow cannot be used as the time series lacks recurring patterns. Moreover, the integrated STL also requires a seasonal pattern to decompose the time series into the components: *Trend*, *Season*, and *Irregular*. Consequently, Telescope requires a fallback for non-seasonal time series. That is, if the *Preprocessing* phase returns only the frequency of 1 as dominant frequency, the *Feature Extraction*, *Model Building*, and *Forecasting* phase are omitted and the fallback is executed instead. The fallback receives as input the transformed time series from the *Preprocessing* phase on which a non-seasonal ARIMA model is trained. More precisely, auto.arima, which automatically finds the most suitable model for the time series, is used for the training. Then, the forecast is performed and the future transformed time series is forwarded to the *Postprocessing* phase, in which the time series is treated as described in Section 8.6.

8.8 Recommendation System for Machine Learning Method

According to the "No-Free-Lunch Theorem", which – simply spoken – states that each method has its advantages and weaknesses depending on the specific use case, it is inadvisable to rely only on one particular method. Typically, the choice of the optimal forecasting method is based on expert knowledge,

which can be expensive, may have a subjective bias, and may take a long time to deliver results. Consequently, we tackle RQ 5 and automate the selection of the best regression-based machine learning method for a given time series. More specifically, we employ a recommendation system based on *meta-learning* to select the most appropriate method based on time series characteristics. In the meta-learning context, the methods, which are available for selection, are referred to as *base-level methods* and characteristics on which the selection is based are called *meta-level attributes*. The underlying selection problem, the methods, attributes, and the recommendation approaches are presented in Section 8.8.1. The employed recommendation system is split in an offline training phase, which is introduced in Section 8.8.2, and the recommendation phase (see Section 8.8.3). Moreover, the recommendation system also uses a time series generator, which is presented in Section 8.8.4, for augmenting the training set.

8.8.1 Meta-Learning for Method Selection

Before the term meta-learning arose, J. Rice was concerned with the problem "Which algorithm is likely to be best suited for my problem?" and therefore, formulated the *algorithm selection problem* (Rice, 1976). Besides the problem formulation, an abstract model for dealing with the selection problem was proposed that comprises four components: (i) the *problem space*, (ii) the *feature space*, (iii) the *algorithm space*, and the (iv) *performance space*. Considering this abstraction (Rice, 1976) and its formal description (Smith-Miles, 2009), we can coin the problem onto the selection problem of this thesis. The first step is the framing of the four components of the abstract model to our specific problem: The problem space Y represents a set of time series; the feature space F contains measurable time series characteristics of each instance of Y, calculated by a deterministic extraction procedure; the algorithm space A is the set of all considered regression-based machine learning methods; the performance space M represents the mapping of each algorithm to the forecast error measure.

Then, the *regression-based machine learning method selection problem* that arises in this thesis can be formally defined as: For a given time series $y \in Y$, with characteristics $f(y) \in F$, find the selection mapping $S(f(y))$ into the algorithm space A, such that the selected algorithm $a \in A$ minimizes the forecast error measure $m(a(y)) \in M$.

8.8.1.1 Meta-Level Attributes

To have an accurate recommendation system for choosing the most appropriate regression-based machine learning method for a given time series, a sound set of characteristics, which describe the time series, is required. To this end, the considered time series characteristics originate from different sources and contain own proposed measures. More precisely, the time series characteristics comprise own proposed measures (O1–O4), statistical information of a time series (S1–S6), characteristics proposed by Lemke and Gabrys (Lemke and Gabrys, 2010a) (L1–L4), and characteristics proposed by Wang et al. (Wang et al., 2009) (W1–W6). In contrast to the work of Wang et al.(Wang et al., 2009), we use the raw values of the characteristics to avoid arbitrary normalization factors. The time series characteristics applied to the meta-learning approach are listed below. Note that these characteristics are only a subset of the original set, selected by, for example, correlation analysis.

O1 The *mean period entropy* quantifies the regularity and unpredictability of fluctuations of the de-trended time series. For this purpose, the approximate entropy of each full period is calculated and then averaged. More formally, the mean period entropy is computed as

$$\frac{1}{\lfloor n/m \rfloor} \sum_{i=1}^{\lfloor n/m \rfloor} AE(p_i), \tag{8.4}$$

where $AE(x)$ is the approximated entropy (Pincus et al., 1991, p.3), p_i the i-th period, n is the length of the de-trended time series, and m the length of each period, that is, the frequency of the de-trended time series.

O2 The *coefficient of entropy variation* describes the standardized entropy distribution over all periods. To this end, the coefficient of variance of the approximate entropy of each full period is determined.

O3 The *mean cosine similarity* states how similar all periods are to each other. The similarity is expressed with the average cosine similarity of each full pair of periods. More formally, the mean cosine similarity is calculated as

$$\frac{1}{\sum_{k=1}^{\lfloor n/m \rfloor - 1} k} \cdot \sum_{i=1}^{\lfloor n/m \rfloor - 1} \sum_{j=i+1}^{\lfloor n/m \rfloor} \frac{p_i \cdot p_j}{||p_i|| \cdot ||p_j||}, \tag{8.5}$$

where p_i is the i-th period, n is the length of the de-trended time series, and m the frequency of the de-trended time series.

O4 The *sinus approximation* quantifies how well the seasonal pattern of the time series can be approximated by a sinus wave. To this end, the Durbin-Watson statistic (Durbin and Watson, 1950) is used to measures the autocorrelation of the resulting fitted errors. More formally, the sinus approximation is expressed as

$$\frac{\sum_{t=2}^{n}(e_t - e_{t-1})^2}{\sum_{t=1}^{n} e_t^2}, \tag{8.6}$$

where e_t is the fitted error at time t and n the length of the de-trended time series.

S1 The *frequency* specifies the length of the most dominant recurring pattern (e.g., 365 daily observations in a yearly pattern) within the de-trended time series. The frequency is estimated while applying a periodogram (see Section 2.2.2).

S2 The *length* counts the total number of observations included in the de-trended time series.

S3 The *standard deviation* measures the amount of variations within the de-trended time series.

S4 The *skewness* reflects the symmetry of the value distribution of the de-trended time series. More formally, the skewness is computed as

$$\frac{1}{n \cdot \sigma^3} \sum_{t=1}^{n} (y_t^s - \mu)^3, \tag{8.7}$$

where y_t^s is the de-trended time series at time t, μ the mean of the de-trended time series, σ the standard deviation of the de-trended time series, and n the length of the de-trended time series.

S5 The *irregular skewness* quantifies the skewness of the irregular part of the time series.

S6 The *irregular kurtosis* reflects the tailedness of the value distribution of the irregular part of the de-trended time series. More formally, the irregular kurtosis can be determined as

$$\frac{1}{n \cdot \sigma^4} \sum_{t=1}^{n} (i_t - \mu)^4, \tag{8.8}$$

where i_t is the irregular part of time series at time t, μ the mean of the irregular part of the time series, σ the standard deviation of the irregular part of the time series, and n the length of the irregular part.

L1 The *2nd frequency* states the second most dominant frequency of the de-trended time series.

L2 The *3rd frequency* refers to the third most dominant frequency of the de-trended time series.

L3 The *maximum spectral value* specifies the maximum spectral value of the periodogram applied to the de-trended time series.

L4 The *number of peaks* reflects how many strong recurring patterns the de-trended time series has. More precisely, the number of peaks in the periodogram that are at least 60% of the maximum value is counted.

W1 The *strength of seasonal component* measures the degree of the seasonality within the de-trended time series (see Section 2.4).

W2 The *serial correlation* describes the correlation of the de-trended time series with itself to an earlier time. The serial correlation is computed as

$$n \cdot \sum_{k=1}^{m} r_k(Y^s)^2, \tag{8.9}$$

where r_k is the auto-correlation coefficient with lag k (see Section 2.4), n the length of the de-trended time series, m the frequency of the de-trended time series, and Y^s the de-trended time series.

W3 The *irregular serial correlation* states the serial correlation of the irregular part of the time series.

W4 The *non-linearity* quantifies the degree of the non-linearity of the de-trended time series (see Section 2.4).

W5 The *irregular non-linearity* reflects the degree of the non-linearity of the irregular part of the time series.

W6 The *self-similarity* measures how similar the de-trended time series is to a part of itself (see Section 2.4).

8.8.1.2 Base-Level Methods

We only consider machine learning methods to learn how the de-trended time series can be described with intrinsic time series features. On the one hand, classical forecasting methods such as ARIMA can typically only process a time series without additional information. That is, the extracted features (see Section 8.3) cannot be used by such methods. On the other hand, machine learning methods can handle any number of features. Consequently, for a possible extension of our approach with external information, these features can easily be added. The implemented regression-based machine learning methods in the meta-learning approach, that is, the base-level methods, are introduced in Section 3.2 and are briefly described below:

- CART is a regression tree that recursively partitions the data set. To prevent the tree from becoming too large, the tree is automatically pruned.
- Cubist is a rule-based regression method. More precisely, the rules are arranged hierarchically, resulting in a tree where each leaf node represents a multivariate linear regression model.
- Evtree implements an evolutionary algorithm for constructing a regression tree that splits the data so that each partition decision is globally optimal.
- NNetar is a feed-forward neural network consisting of one hidden layer and is trained with lagged values of the time series. The number of nodes in the hidden layer and the lags are automatically determined.
- Random forest is an ensemble method comprising multiple regression trees. Each tree is built independently of the others, and for each split, only a random subset of the features is considered.
- SVR uses the same principles as SVM. More precisely, a set of hyperplanes is constructed for separating the data. If the data is not linear in its input space, the values are mapped into a higher dimensional feature space.
- XGBoost is an ensemble method consisting of multiple regression trees and uses gradient tree boosting. That is, each tree is grown with knowledge from the last trained tree.

The selection contains the seven best-performing methods during preliminary experiments. The methods consist of five tree-based machine learning algorithms, a neural network, and a support vector regression method. The focus

of the tree-based methods can be explained by the fact that classification and regression trees are broadly used machine learning methods due to their benefits: (i) white-box modeling, (ii) extraction of interpretable results, (iii) simple decisions, and (iv) fast time-to-result. More precisely, we choose these methods as they are generic and can be applied out-of-the-box.

8.8.1.3 Rule Generation Approaches

To recommend the most appropriate regression-based machine learning method for a given time series, we propose two different methods: (i) a classification-based rule generation approach A_C, and (ii) a regression-based rule generation approach A_R. Both approaches have in common that a random forest[6] (see Section 3.2.4) is used for learning how meta-level attributes (i.e., time series characteristics) can be mapped to the performance of the base-level methods (i.e., regression-based machine learning methods).

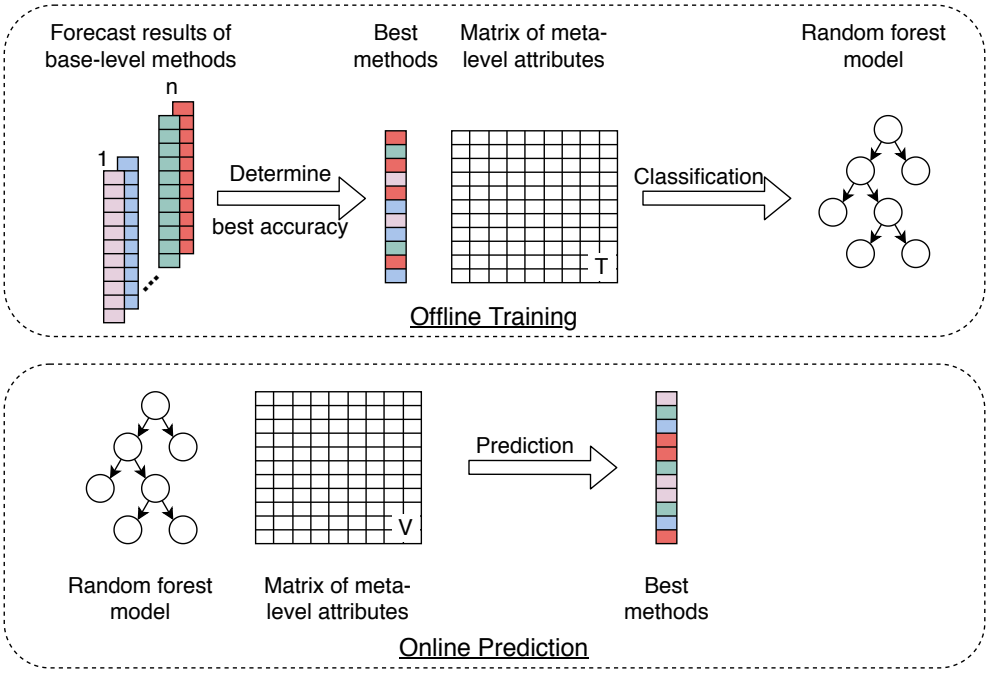

Figure 8.7: Schematic process of the rule generation (classification).

[6]The choice of random forest for the recommendation is based on the extensive experiments in which these combinations yielded the best results.

The idea of the first approach A_C is to map the time series characteristics of a given time series to the regression-based machine learning method with the best accuracy. The schematic workflow of this approach is illustrated in Figure 8.7. In the first step, the forecast results of the base-level methods (see Section 8.8.2) are investigated. Then, a vector is formed containing the most accurate base-level method for each time series. A random forest model is trained based on this vector and the meta-level attributes of each time series. More specifically, a random forest performs a multinomial classification, where the meta-level attributes are the features and the best base-level methods are the target. That is, the resulting model reflects the rules for selecting a base-level method for a specific time series. After the rule generation, the meta-level attributes of new time series can be injected into the model to return the most appropriate base-level methods for each time series.

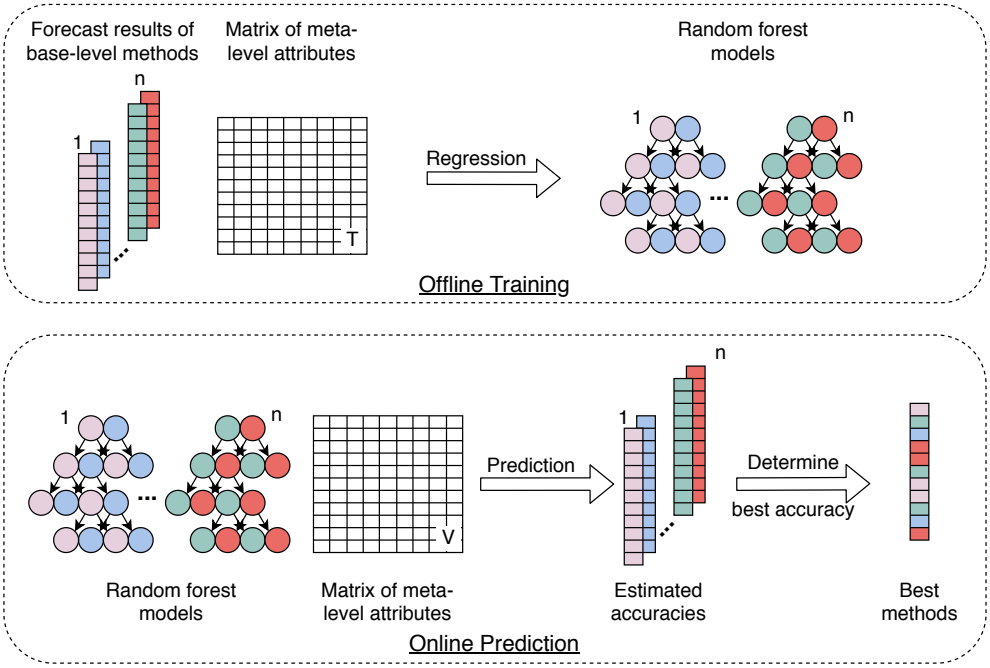

Figure 8.8: Schematic process of the rule generation (regression).

In contrast to the classification-based rule generation, the second approach A_R, as illustrated in Figure 8.8, is based on more than just one method. That is, the idea of this approach is to train a random forest for each base-level method. Each model estimates the forecast accuracy for a given time series. Then, the method that has the best accuracy based on the estimations is selected. More

precisely, the approach calculates for each base-level method and each time series how worse it is compared to the method with the best forecast accuracy. We reflect this deterioration with the *forecast accuracy degradation* that can be calculated as

$$\vartheta_i := \frac{\epsilon_i}{min(\epsilon_1, \ldots, \epsilon_n)}, \tag{8.10}$$

where ϵ_i is the forecast accuracy of the i-th method and n is the number of considered methods. The values of ϑ_i lie in the interval $[1, \infty)$, where 1 indicates that this method has the best forecast accuracy. After the calculation of the degradation, a random forest is used as regressor for each base-level method, where the meta-level attributes are the features and the forecast accuracy degradation vector of the respective method is the target. In other words, the regression task leads to n random forest models, each reflecting the estimate of the forecast accuracy degradation compared to the best method for a given time series. After the training, the meta-level attributes of new time series can be fed to each of the random forest models. Based on these attributes, each model estimates the forecast accuracy degradation vector. Then, the base-level methods with the lowest estimated forecast accuracy degradation are returned for each time series.

8.8.2 Offline Training

In the *Offline Training* phase, which is depicted in Figure 8.9 (orange, rounded boxes represent actions, white boxes artifacts, green hexagons the input of a phase, and blue trapezoids the output of a phase), the rules for recommending a specific method based on time series characteristics are learned or updated either when Telescope is started or when no forecast is currently being conducted. To this end, this phase gets a *Set of Time Series* as input. To retrieve precise rules for the recommendation of the most appropriate regression-based machine learning method for a given time series, a set of time series which may be similar to this time series is required. Therefore, Telescope generates n new time series based on the initial set of time series in this phase. The time series generation is explained in Section 8.8.4 and the parameter n is set in our experiments (see Chapter 10) to 10,000. As the machine learning method has to learn how the de-trended time series can be described by the intrinsic features (see Section 8.4), each time series in the *Extended Set of Time Series*, which comprises the initial time series and the newly generate time series, has to be de-trended. For this purpose, each time series is processed as described in the Sections 8.2, 8.3, and 8.4. That is, each time series is transformed with the Box-Cox transformation, de-trended, and the intrinsic features are extracted.

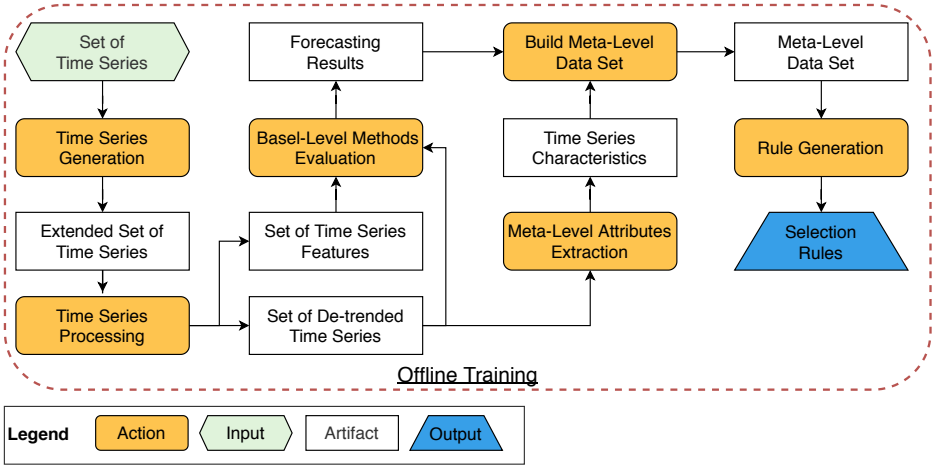

Figure 8.9: Offline training phase of Telescope.

In the *Base-Level Methods Evaluation* step, each base-level method (see Section 8.8.1.2) is trained and evaluated on every de-trended time series and its associated intrinsic features (Fourier terms and seasonal pattern). To this end, the time series is split into history (the first 80% of the time series) and in future/test (the remaining 20%). In parallel, the time series characteristics (see Section 8.8.1.1) of each de-trended time series are extracted. The *Forecasting results* from the *Base-Level Methods Evaluation*, in this case the sMAPE (see Section 3.3.2), and the *Time Series Characteristics* are used to form the *Meta-Level Data Set*. This data set is used in the *Rule Generation* (see Section 8.8.1.3) step to retrieve the rules for the recommendation of the best suited method. The *Selection Rules* are returned and stored so that the *Recommendation* phase can access them when executed.

8.8.3 Recommendation

In the case that Telescope forecasts a time series in a non-time-critical scenario (see Section 8.4), the regression-based machine learning method is selected based on the characteristics of the given time series. The selection process takes place in the *Recommendation* phase, which is illustrated in Figure 8.10 (same structure as Figure 8.9). This phase gets the *De-Trended Time Series* from the *Model Building phase* as input and loads the *Selection Rules*. In the first step, the characteristics (see Section 8.8.1.1) of the de-trended time series are extracted. Then, the selection rules are applied on the *Time Series Characteristics* and the

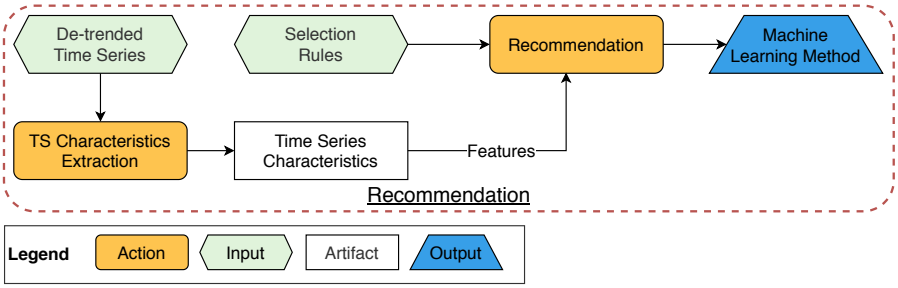

Figure 8.10: Recommendation phase of Telescope.

resulting regression-based machine learning method is returned to the *Model Building phase*.

8.8.4 Time Series Generator

In general, machine learning only works well if the training and test sets exhibit the same characteristics. In other words, the training set should be as representative as possible of the process that created both the training set and the test set. In the context of recommending the most appropriate machine learning method for a given time series, we assume an infinite time series population, from which both the original set of time series and the time series to be forecast originate. Following the premise of machine learning and to be independent of the initial set of time series, Telescope generates based on the initial set a large number of new time series. On the one hand, the generation is done to increase the size of the training set and, on the other hand, to ensure that the time series characteristics are distributed as widely as possible to be as representative as possible of the assumed population. An example output of the time series generation is illustrated in Figure 8.11. The first row shows the initial set of time series (see Section 2.1.1 and 2.3.2 for the descriptions of these time series) while the remaining rows show the time series that were randomly generated based on the initial time series. It can be seen that all newly created time series exhibit a different length, seasonal pattern, and/or trend behavior.

The procedure of the time series generation is presented in Algorithm 8.4 and gets the initial list of time series *list* and the number of time series to be generated n. The algorithm is probabilistic and, unless otherwise specified, each random draw is carried out according to a discrete uniform distribution. The time series generation takes place in a loop (Line 2–28), which is repeated until n valid time series are generated. In the first step, three time series are drawn from the initial set of time series (Line 3). As the drawing of a time

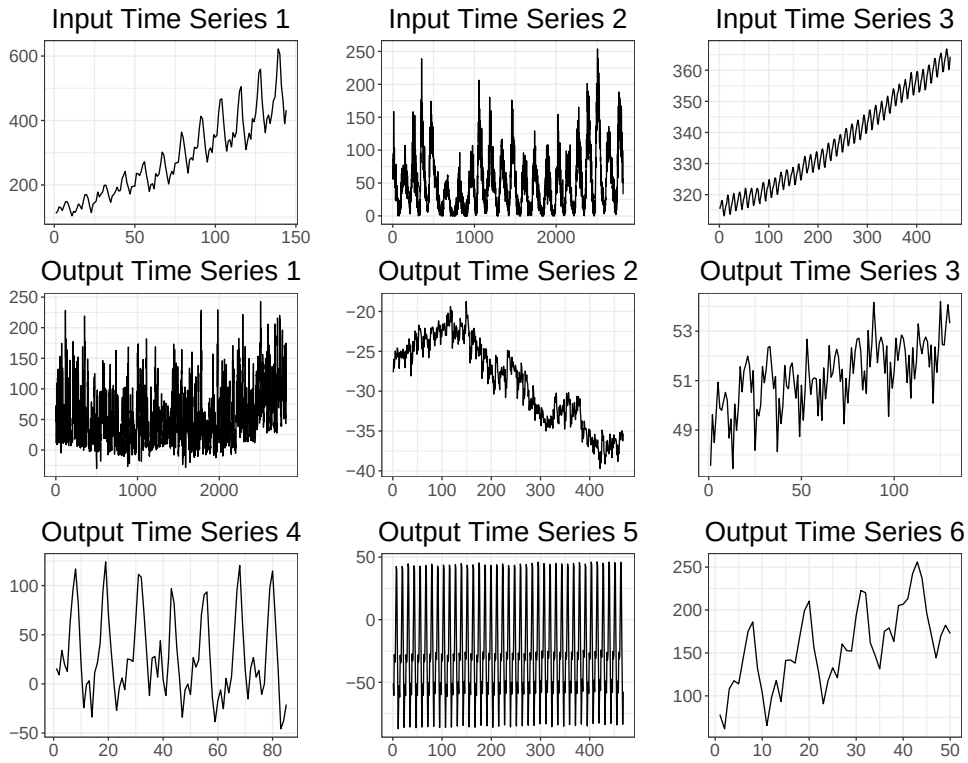

Figure 8.11: Example of six generated time series.

series follows a discrete uniform distribution, it is possible to select the same time series three times. Then, each drawn time series is decomposed with STL (see Section 2.3.1) and the seasonal component of the first time series, the trend component of the second time series, and the irregular part of the last time series are extracted (Line 4–6). During the STL decomposition, the window size of the trend extraction is set randomly, leading to a (slightly) different decomposition in each case. Depending on a random Boolean $\sim B(\frac{1}{3})$ (i.e., Bernoulli distribution with a one-third probability of success), the trend is reduced to a random segment (Line 7–10). The start and end of the segment are randomly drawn from the indices of the trend component. Similar to the trend component, the irregular part can also be reduced to a random segment (Line 11–14), depending on a random Boolean $\sim B(\frac{1}{2})$. In contrast to the trend and irregular component, the random segment selection is omitted for the season component as it is per definition a reoccurring pattern. However, the seasonal pattern is compressed with a probability of 50% (Line 15–19). In the case of compression,

Algorithm 8.4: Time Series Generation.

Input: List of time series *list*, number of new time series n
Result: List with n new time series

```
1  new_list = [ ]
2  while length(new_list) < n do
3      [ts1, ts2, ts3] = drawRandomTS(list) // ~ U(0, |list| − 1)
4      season = stlGetSeason(ts1) // decomposes the time series and gets the
         corresponding component
5      trend = stlGetTrend(ts2)
6      irreg = stlGetIrregular(ts3)
7      if getRandomBoolean() then // ~ B(1/3)
8          indices = getRandomInt(length(trend), 2) // ~ U(0, |trend| − 1)
9          trend = trend[min(indizies) to max(indizies)]
10     end
11     if getRandomBoolean() then // ~ B(1/3)
12         indices = getRandomInt(length(irreg), 2) // ~ U(0, |irreg| − 1)
13         irreg = irreg[min(indizies) to max(indizies)]
14     end
15     if getRandomBoolean() then // ~ B(1/2)
16         freqs = getDivisors(frequency(season))
17         freq = getRandomInt(freqs, 1) // ~ U(0, |freqs| − 1)
18         season = compressPattern(season, freq)
19     end
20     if getRandomBoolean() then // ~ B(1/2)
21         trend = adjustTrend(trend) // increases, decreases, inverts, or
             removes slope of the trend
22     end
23     irreg = MBB(irreg) // moving block bootstrapping
24     ts = assembleTS(season, trend, irreg)
25     if isValid(ts) then
26         new_list.append(ts)
27     end
28 end
29 return new_list
```

the divisors (except 1) of the frequency of the time series are calculated and one divisor is randomly drawn (Line 16–17). Based on this drawn frequency, the seasonal pattern is squeezed. More precisely, the original seasonal pattern

is aggregated so that the length of the seasonal pattern corresponds to the new frequency. Again, depending on a random Boolean $\sim B(\frac{1}{2})$, the trend component is adjusted (Line 20–22). More precisely, the trend is approximated with a linear function. The resulting slope is then multiplied with a random value drawn from $[-2; 1) \cup (1; 2]$. That is, the slope is either made steeper, flattened, inverted, or set to zero. Afterward, the residuals of the fit and the adjusted slope are summed up to form the new trend (Line 21). In the last step before assembling the new time series, a new irregular component is created based on the original irregular part (Line 23). More precisely, the irregular part is bootstrapped with a moving block bootstrapping (Bergmeir et al., 2016). In short, random segments with a fixed length (in our case, the minimum of 25% of the length of the irregular part and the frequency of seasonal component) are drawn and attached to each other, forming a new irregular part (illustrated in Figure 8.12). An example of the moving block bootstrapping is illustrated in Figure 8.12. In the last step, the three components are assembled to a new time series (Line 24), and if the generated time series is valid (Line 25–27), it is added to the list of new time series. As the three components may have different lengths, the resulting time series' length is equal to either the length of the trend or the irregular component, depending on which component is shorter. That is, either the irregular part or the trend part is shortened. The seasonal component is either also shortened or simply continued for the length of the new time series. The generated time series is considered valid if the length is greater than two times the frequency plus one and is unique in the set of time series. STL introduces the restriction regarding the length as it requires at least two full seasonal patterns. Time series classified as invalid are discarded. After n new time series are generated, the list of new time series is returned.

8.9 Assumptions and Limitations

In the development of Telescope, we limit ourselves to univariate time series. In fact, correlated/external data can be used for each time series to improve forecast accuracy. However, the selection and preprocessing of such additional information require domain knowledge. In other words, this knowledge about domain-specific feature engineering cannot yet be fully automated. Consequently, our method would have to be tailored to a specific domain and, therefore, contradict the goal of a generic forecasting approach. Besides this limitation, Telescope has the following assumptions that are either based on the integrated tools or design decisions: (i) The time series must not contain missing values, and each value must be numeric. Indeed, missing values can

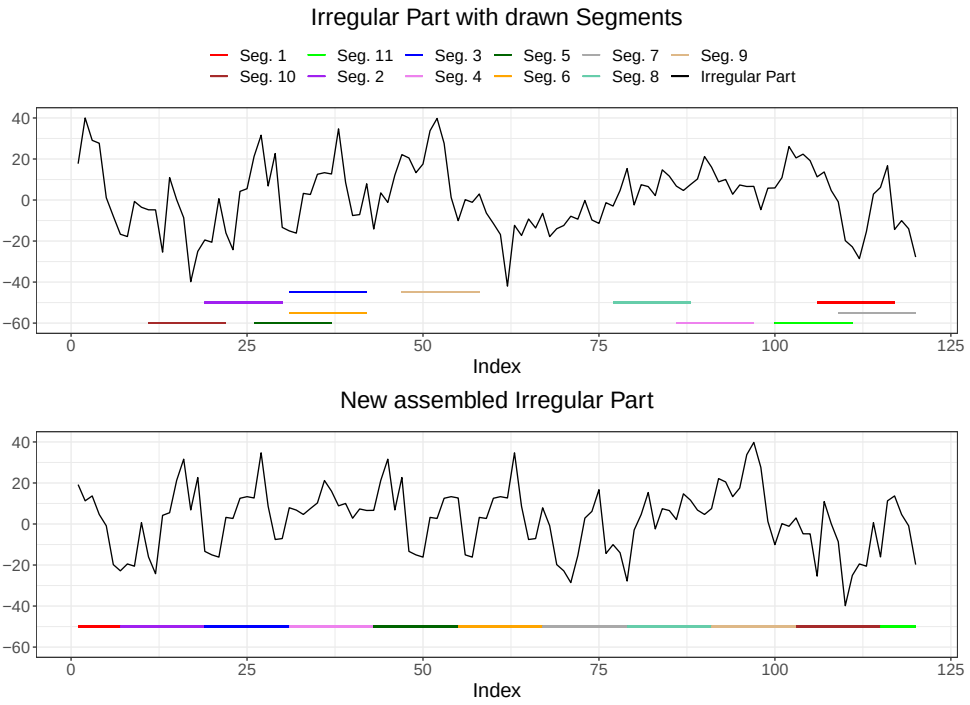

Figure 8.12: Example moving block bootstrapping of an irregular part of a time series.

be interpolated. However, if the gaps are large or frequent, the values must be treated with caution, as they can affect the model's accuracy. (ii) To use the whole Telescope approach and not only the fallback, the time series must be able to be decomposed. That is, the time series have to meet the requirements of the STL decomposition. (iii) For the recommendation of the most appropriate regression-based machine learning method, Telescope assumes that the time series to be forecast and the time series that are used for the training (i.e., the initial time series set plus the generated time series) originate from the same population and therefore, have a similar distribution of time series characteristics. In other words, Telescope assumes the rules that are retrieved based on the training set can also be applied to new time series. (iv) Usually, many systems are driven by human interactions. In other words, the time series produced or observed by these systems are subject to human habits (e.g., day/night phases) and, therefore, seasonal. Consequently, Telescope assumes that the found frequencies within time series are multiples of natural frequencies. (v) We expect that the seasonal pattern does not evolve over time.

8.10 Differentiation from Related Work

To delimit Telescope from the related work, we summarize the reviewed hybrid approaches (see Chapter 5) by listing them in Table 8.1. In this overview, we distinguish (i) whether the approach can only perform one-step-ahead forecasts or can forecast several points at once, (ii) if the approach is generic or is tailored to a special use case, (iii) how many time series were used in the evaluation, (iv) against how many methods the approach competed, and (v) if the approach is open-source.

Table 8.1: Overview of related work on hybrid forecasting methods.

Forecasting Method	Forecasting		Evaluation		Open-Source
	Type	Generic	#Time Series	#Competing Methods	
Telescope	multi-step	✓	400	13	✓
(Bates and Granger, 1969)	multi-step	✓	2	5	✗
(Adhikari et al., 2015)	multi-step	✓	4	14	✗
(Sommer et al., 2016)	one-step	✓	10	8	✗
(Cerqueira et al., 2017)	multi-step	✓	14	8	✓
(Wang et al., 2018)	multi-step	✗	1	7	✗
(Boulegane et al., 2019)	one-step	✓	55	3	✗
(Montero-Manso et al., 2020)	multi-step	✓	100,000	Took place in M4	✓
(Collopy and Armstrong, 1992)	multi-step	✗	126	4	✗
(Wang et al., 2009)	one-step	✓	315	No evaluation	✗
(Lemke and Gabrys, 2010b)	multi-step	✓	288	18	✗
(Widodo and Budi, 2013)	multi-step	✓	3104	5	✗
(Kück et al., 2016)	multi-step	✓	111	7	✗
(Talagala et al., 2018)	multi-step	✓	4004	16	✓
(Zhang et al., 2020)	multi-step	✗	522	4	✗
(Zhang, 2003)	multi-step	✓	3	2	✗
(Pai and Lin, 2005)	multi-step	✓	10	2	✗
(Liu et al., 2014)	multi-step	✗	4	1	✗
(Khandelwal et al., 2015)	multi-step	✓	4	3	✗
(Bergmeir et al., 2016)	multi-step	✗	2829	1	✓
(Zhang et al., 2017)	multi-step	✓	2	3	✗
(Panigrahi and Behera, 2017)	multi-step	✓	16	5	✗
(Taylor and Letham, 2018)	multi-step	✗	unkown	4	✓
(Saâdaoui and Rabbouch, 2019)	multi-step	✗	1	3	✗
(Saâdaoui et al., 2019)	multi-step	✓	3	3	✗
(Smyl, 2020)	multi-step	✗	100,000	Took place in M4	✓

As Telescope is based on time series decomposition and its recommendation part is only used in non-time-critical scenarios, our approach differs considerably from methods from the field of ensemble forecasting. In contrast to the methods originating from time series decomposition that use different decomposition and forecasting techniques, Telescope explicitly decomposes the time series into trend, season, and irregular part. Also, each part is forecast separately using different forecasting methods. Furthermore, our method is designed

for long and seasonal time series. In contrast to the recommendation-based methods that use mainly classical methods, Telescope selects regression-based machine learning methods based on time series characteristics. Furthermore, our approach augments the original time series set by generating new time series from it to increase the data set's diversity.

In terms of evaluation, the reviewed approach use either subsets of the M-Competitions (Lemke and Gabrys, 2010b; Widodo and Budi, 2013; Bergmeir et al., 2016; Kück et al., 2016; Talagala et al., 2018) or a small set of time series. In contrast, our evaluation data set contains 400 time series showing different characteristics and originates from various sources (see Section 7.3). Moreover, most approaches competed with few forecasting methods while we compare our approach with seven (the best of originally 15) forecasting methods covering recent hybrid, as well as established machine learning, and classical forecasting approaches (see Chapter 10). For the ranking of the methods, usually, only the forecast accuracy is considered. In this work, we consider both accuracy and time-to-result measures.

8.11 Concluding Remarks

In this chapter, our contribution addresses the research question RQ 3 "*How to design an automated and generic hybrid forecasting approach that combines different forecasting methods to compensate for the disadvantages of each technique?*" by considering only the given time series and decomposing the time series into components, building a forecasting model for each of them. Moreover, Telescope automatically transforms the time series and retrieves features of time series to increase the accuracy and the information quantity for the model learning and therefore faces the research question RQ 4 "*How to automatically extract and transform features of the considered time series to increase the forecast accuracy?*". In a non-time-critical scenario, Telescope tackles the challenge posed by the "No-Free-Lunch Theorem" by not relying on a single regression-based machine learning method with its possibly inaccurate forecasts. More precisely, for a given time series, a recommendation system employs the best suited regression-based machine learning method for assembling the final forecast. Considering the research question RQ 5 "*What are appropriate strategies to dynamically apply the most accurate method within the hybrid forecasting approach for a given time series?*", Telescope implements two different recommendation approaches. To increase the accuracy of the recommendation and to be independent of the initial set of time series, Telescope also has a time series generator deployed for generating numerous new time series with different characteristics based on the initial set.

Chapter 9

Forecasting-based Auto-Scaling of Distributed Cloud Applications

To face the dynamic behavior and requirements of modern applications, cloud computing emerged as a computing model with a high scalability level. Although there are threshold-based scaling mechanisms such as the auto-scaler available in Amazon Web Services EC2[1], business-critical applications in cloud environments are typically deployed with highly overprovisioned resources to guarantee reliable service operation. This strategy is pursued to avoid negative effects of auto-scaling mechanisms with their possibly wrong or delayed scaling decisions. According to a recent survey (RightScale, 2019), 35% of the cloud costs are wasted partly because of this strategy.

To increase the industry's trust in auto-scaling, current research in the scientific community is focused on novel approaches for reliable techniques. The proposed methods can scale an application reactively or proactively. Both approaches have specific advantages and disadvantages. Proactive mechanisms can scale at an early stage of a load spike, but this is typically done based on forecasting of the workload intensity, and therefore the quality of such adaptations depends to a large extent on the accuracy of the employed forecasting method. This uncertainty may be eliminated with reactive scaling as all adaptations are made based on actual measurements, but such mechanisms can only react after a change in the workload occurs. In fact, an auto-scaler can implement both proactive and reactive mechanisms, but there can be conflicts between the mechanisms. Consequently, the process that decides which mechanism should be applied for scaling the application poses a crucial challenge. Another challenge arises when applications are deployed in a public cloud environment. Namely, the desired effect of adapting an application in response to workload changes can lead to high costs. The reason is that the accounted costs and the charged costs can deviate depending on the cloud provider's pricing scheme. To this end, a trade-off has to be considered as an optimal scaling often increases

[1] Amazon auto-scaler: `https://aws.amazon.com/autoscaling/`

the operational costs while cost-optimal decisions may decrease the scaling performance.

Modern applications are often designed based on a micro-service architecture, and thus they consist of multiple services. To the best of our knowledge, there is no open-source auto-scaler for applications comprising multiple services. Even popular auto-scaling mechanisms such as such as AutoMap (Beltrán, 2015), AGILE (Nguyen et al., 2013), and CloudScale (Shen et al., 2011), are closed-source. To this end, the straightforward implementation of an auto-scaling mechanism for such applications is to instantiate an open-source individual single-service auto-scaler for every service. That is, each service is observed and scaled independently by an auto-scaler. However, this approach can lead to problems like oscillations and bottleneck shifting. B. Urgaonkar et al. provide a detailed example of these problems (Urgaonkar et al., 2005).

To tackle the mentioned challenges, we pose ourselves the following research questions:

RQ 6: *What is a meaningful combination of proactive and reactive scaling techniques to minimize the risk of auto-scaling in operation?*

RQ 7: *How can scaling decisions be adjusted so that the charged costs in a public cloud environment are minimized?*

RQ 8: *How to enable coordinated scaling of applications comprising multiple services?*

RQ 9: *What are meaningful measures for assessing the quality of coordinated and cost-aware auto-scaling?*

Towards addressing the research questions, our contribution is the design of *Chamulteon*. This novel hybrid auto-scaler combines proactive and reactive techniques to scale distributed cloud applications comprising multiple services in a coordinated and cost-effective manner. Chamulteon is based on our original auto-scaler *Chameleon*, which can only scale monolithic applications. More precisely, the Chameleon approach is preliminary work in collaboration with Nikolas Herbst and is used as a basis for this contribution. More details on Chameleon can be found in our previous works (Bauer, 2016; Bauer et al., 2018b) or in the thesis of N. Herbst (Herbst, 2018).

Chamulteon consists of two independent cycles: (i) The reactive cycle that monitors the application and scales reactively in short intervals and (ii) the proactive cycle that predicts the demand at longer intervals for a set of future scaling intervals. More precisely, Chamulteon maintains a performance model of the application, observes and forecasts the request arrival rates with

Telescope (see Chapter 8), and estimates the service time of each service. Furthermore, we propose *Fox*, a cost-aware computing resource management approach. Fox serves as a mediator between an application deployed in a public cloud and an auto-scaler. The main idea is to proactively plan the resource allocation and release according to a predefined charging model. To enable cost-efficient scaling for Chamulteon in public cloud environments, Fox is modified and integrated into Chamulteon. All proposed approaches are evaluated in Chapter 11.

The remainder of this chapter[2] is organized as follows: Section 9.1 introduces the overview of Chamulteon, a brief description of the deployed components, and the changes to the Chameleon[3] approach. Section 9.2 explains the decision making logic of Chamulteon. As Chamulteon leverages both reactive and proactive scaling decisions, conflicts between the decisions may occur. To this end, Section 9.3 focuses on the resolution of the possible conflicts. The cost-aware resource management Fox is introduced in Section 9.4. To evaluate Chamulteon in Chapter 11, Section 9.5 proposes a set of measures to quantify the quality and cost efficiency of an auto-scaler. The assumptions and limitations of the contributions of this chapter are summarized in Section 9.6. In Section 9.7, we differentiate the approach from related work (see Chapter 6). Finally, the chapter is concluded in Section 9.8.

9.1 Overview of the Chamulteon Approach

Chamulteon is a hybrid auto-scaling mechanism for applications comprising different services and is based on a redesign and extension of the underlying architecture and workflow of Chameleon. Chameleon is also an elastic mechanism but can only scale monolithic applications. The Chameleon approach is preliminary work for this thesis and details can be found in our previous works (Bauer, 2016; Bauer et al., 2018b) or in the thesis of N. Herbst (Herbst, 2018).

The Chamulteon approach consists of five basic components: (i) the *controller*, (ii) a *performance data repository*, (iii) the *forecasting component*, (iv) the *service demand estimation component*, and finally (v) the *cost-awareness component*. The central part of Chamulteon is the controller. It communicates with each component and is responsible for scaling each service of the application. Moreover, the

[2] This chapter is based on our previous works (Bauer et al., 2018b; Herbst et al., 2018; Lesch et al., 2018; Bauer et al., 2019b).

[3] Note that as Chameleon is a side-contribution and Chamulteon extends this approach, there is no separate section on Chameleon.

functionality of the controller is split into two parallel sequences: a *reactive cycle* and a *proactive cycle*. The performance data repository contains a time series storage, knowledge about the charging model of the cloud platform provider, and an instance of a descriptive performance model of the dynamically scaled application based on the Descartes Modeling Language (*DML*) (Kounev et al., 2016; Huber et al., 2017). The remaining three components contain external tools and are described in detail in the following sections. Unlike all other components, the cost-awareness component is not mandatory and can be switched on or off accordingly. The functionality of Chamulteon is depicted in Figure 9.1. The red lines show the reactive cycle, and the blue dashed lines represent the proactive cycle. The newly introduced and the changed components – in comparison to the original Chameleon approach – are highlighted with a green dotted line.

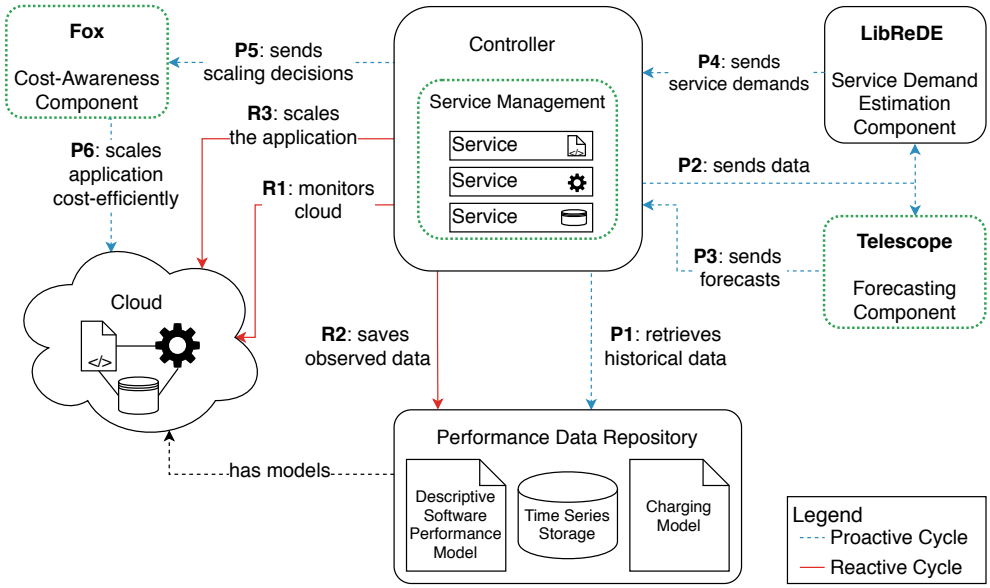

Figure 9.1: Design overview of Chamulteon.

The reactive cycle comprises three tasks: (R1) The controller communicates with the cloud management and periodically retrieves data about the current state of the application and performance measures of the underlying (virtual) hardware at short intervals. To extract the required data, a monitoring agent (Spinner et al., 2016) is deployed within the application's runtime environment. (R2) The gathered information is stored in the performance data

repository for the current time window. (R3) Based on the observed arrival rate and the service demand (which is estimated in (P4) of the proactive cycle), the average system utilization for each service is computed according to queueing theory (see Section 4.1). If the utilization exceeds or falls below a predefined threshold, the controller decides in accordance with the proactive cycle if the respective service needs to be scaled.

In contrast to the reactive cycle, the proactive cycle is executed less frequently as it plans for a set of future scaling intervals. The proactive cycle invokes six actions: (P1) The controller retrieves the historical information and (P2) forwards the data to the forecasting component and the service estimation component. (P3) The arrival rates for the set of future scaling intervals are forecast and sent to the controller. (P4) The service demand for each service is estimated and sent to the controller. (P5) The future system utilization for each service is computed based on the future arrival rates and the service demand. If the utilization exceeds or falls below a predefined threshold, the controller decides if the respective service needs to be scaled. (P6) If activated, the cost-awareness component reviews all planned decisions proposed by the controller and evaluates whether they are cost-efficient or not. That is, scaling decisions may be delayed or omitted to scale the application as cost-efficient as possible.

9.1.1 Forecasting Component

To enable proactive scaling, Chamulteon requires the arrival rates for the next reconfiguration intervals. To this end, Telescope (see Section 8) is deployed in this component. More precisely, Telescope forecasts the arrival rates for a configurable number of future reconfiguration intervals based on the historical arrival rates observed. To minimize the forecasting overhead, only the arrival rates for the entry service are forecast. Moreover, this component is only executed if an earlier forecast has no more predicted values for future arrival rates or a configurable drift between the forecast and the recent monitoring data is detected. To detect the drift between the monitored and forecast values, we compute the forecast accuracy using the MASE measure (see Section 3.3.3).

9.1.2 Service Demand Estimation Component

Due to instrumentation overheads and possibly measurement interferences, the measurement of service demands is not feasible during operation (Spinner et al., 2015). To this end, this component deploys the Library for Resource Demand Estimation (*LibReDE*) (Spinner et al., 2014, 2015) for estimating the service demand. The LibReDE library offers eight different estimation ap-

proaches for service demands on a per request type basis (Spinner et al., 2014). We use the estimator based on the service demand law (Menascé et al., 2004) to minimize the estimation overhead. As input, the average CPU utilization and the throughput of each workload class per service instance are provided. Moreover, LibReDE requires structural knowledge about the application deployment provided by the DML performance model instance.

9.1.3 Cost-Awareness Component

While using auto-scaling in public clouds, the "optimal" adaptation process can lead to high costs as the accounted costs and the charged costs can deviate depending on the cloud provider. For example, if the cloud provider charges service instances hourly, the hour must be paid, although the accounted time is less than one hour. To this end, a cost-aware mechanism called Fox is deployed. More precisely, this component implements only the *Plan* and *Execute* phase of Fox. The details of Fox are described in the Section 9.4. Fox serves as a mediator between an application deployed in a public cloud and Chamulteon. That is, the cost-awareness component leverages knowledge of the charging model of the public cloud and reviews the proactive scaling decisions proposed by Chamulteon to reduce the charged costs to a minimum. In other words, Fox delays or omits the release of service instances to avoid additional charging costs if the service instance will be required again within the charging interval. Using this review logic, the charging interval of each service instance is utilized as efficiently as possible.

9.1.4 Limitations of and Changes to the Chameleon Approach

The original Chameleon is designed only to scale monolithic applications. If this approach should manage an application consisting of several services, a Chameleon instance must be deployed for each service. The application would be scaled in an uncoordinated way, which might lead to bottleneck-shifting and oscillations due to the lack of knowledge over all services. To this end, the first and major change from the original Chameleon is the addition of a *service management* component. This component allows making decisions for each service while considering the other services and their related decisions. For instance, the bottleneck-shifting can be minimized because scaling one service can trigger scaling of succeeding services. Another weakness of the Chameleon approach is the forecasting component: Two forecasting methods, namely sARIMA and TBATS, are deployed. Instead of running the two methods in parallel, the method that is most likely to give the most accurate result

is automatically chosen. This selection is based on the forecasting method recommendation proposed by X. Wang et al. (see Section 5.2). However, both methods have a high variance in their time-to-result (see Chapter 10) and thus, are prone to belated forecasts. To this end, Chamulteon implements the forecasting method Telescope (see Chapter 8) in its forecasting component due to its low and stable time-to-result (see Chapter 10). The last change compared to the original version is the integration of the cost-awareness component. When running an application in the cloud, this component, if enabled, reviews all decisions proposed by the controller and revises them if they are not cost-efficient.

9.2 Decision Making Process

The decision-making process consists of two phases, invoked from corresponding proactive or reactive monitoring cycles for predefined auto-scaling intervals. Both cycles make decisions for each service based on the queueing theory-based utilization. Therefore, Chamulteon transforms the instance of the DML performance model into a product-form queueing network[4] (Huber et al., 2017; Eismann et al., 2018b), whereby each service is modeled as an $M/M/n/\infty$ queue. As each service instance is mapped to exactly one resource instance (e.g., container), the terms resource instance and service instance are interchangeable in the modeling perspective. If the utilization exceeds/undershoots the predefined service thresholds, the required number of service instances is calculated. After all decisions are made, they can further be adjusted for cost-efficiency if the associated component is activated.

Algorithm 9.1 depicts the proactive decision making for a specific time t in the future and service s. The decisions are made based on the forecast arrival rate λ, the estimated service demand μ, and the number of running service instances n (Lines 1–3). The arrival rate for each service is estimated according to the forecast. If the service is the user-facing service, the arrival rate is equal to the forecast value. Otherwise, the forecast arrival rate is estimated based on an invocation graph. This graph is extracted from the DML model that also captures the request types and their control flows. More precisely, the algorithm checks whether there are enough service instances to process the incoming arrival rate for each service. If there are too few service instances, the arrival rate λ is set to the maximum arrival rate that can be served by the

[4] Depending on the application, complex queueing networks may result. In other words, complex control flows can exist where the probability and frequency of visiting each service vary depending on the type of request.

bottleneck service. Otherwise, the arrival rate λ is equal to the forecast arrival rate (Line 4).

Algorithm 9.1: Proactive decision logic.

Input: Service s, time t
Result: Decision for service s at time t in the future

```
1  λ = getForecast(t)
2  μ = getAvgServiceDemand(s)
3  n = getNumInstances(s)
4  λ = estimateArrivals(λ, s) // estimates future arrival rates based on
     invocation graph
5  ρ = λ/(μ·n) // calculate the future average utilization
6  if ρ ≥ ρ_upper then
7      while ρ ≥ ρ_upper and n < maxInstances(s) do
8          ρ = λ/(μ·(++n)) // calculate new average utilization
9      end
10 end
11 else if ρ < ρ_lower then
12     while ρ < ρ_lower and n > minInstances(s) do
13         ρ = λ/(μ·(--n)) // calculate new average utilization
14     end
15 end
16 return decision(n, s, t)
```

Based on this information, the average service utilization ρ is calculated (Line 5). If the calculated utilization exceeds the upper threshold ρ_upper, the number of service instances is theoretically increased and the new average system utilization is computed. This is done iteratively until the number of service instances is equal to the pre-set maximum allowed number of instances for this service or the average system utilization is below the upper threshold (Lines 6–10). Analogously, if the calculated utilization falls below the lower threshold ρ_lower, the number of required service instances is calculated based on the minimum allowed number of instances for this service (Lines 11–15). Finally, a decision with the number of required instances for the specific service at time t in the future is returned. The decision logic for the reactive decisions works analogously, except that reactive decisions only consider the current observed arrival rate.

In contrast to reactive events that are scheduled immediately, proactive de-

cisions are improved before the scheduling. More precisely, the proactive decisions are optimized pairwise per service to reduce oscillations. Basically, there are three possibilities when improving two decisions: (i) Both decisions want to release service instances, (ii) want to add more service instances, (iii) or they have contrary scaling decisions. If one decision intends to keep the current amount of service instances (is interpreted as NOP in the following), no optimization takes place. The resulting six cases are depicted in Figure 9.2, where the solid black line represents the current number of resources, the black dashed line the planned amount, and the grey arrows the scaling decisions.

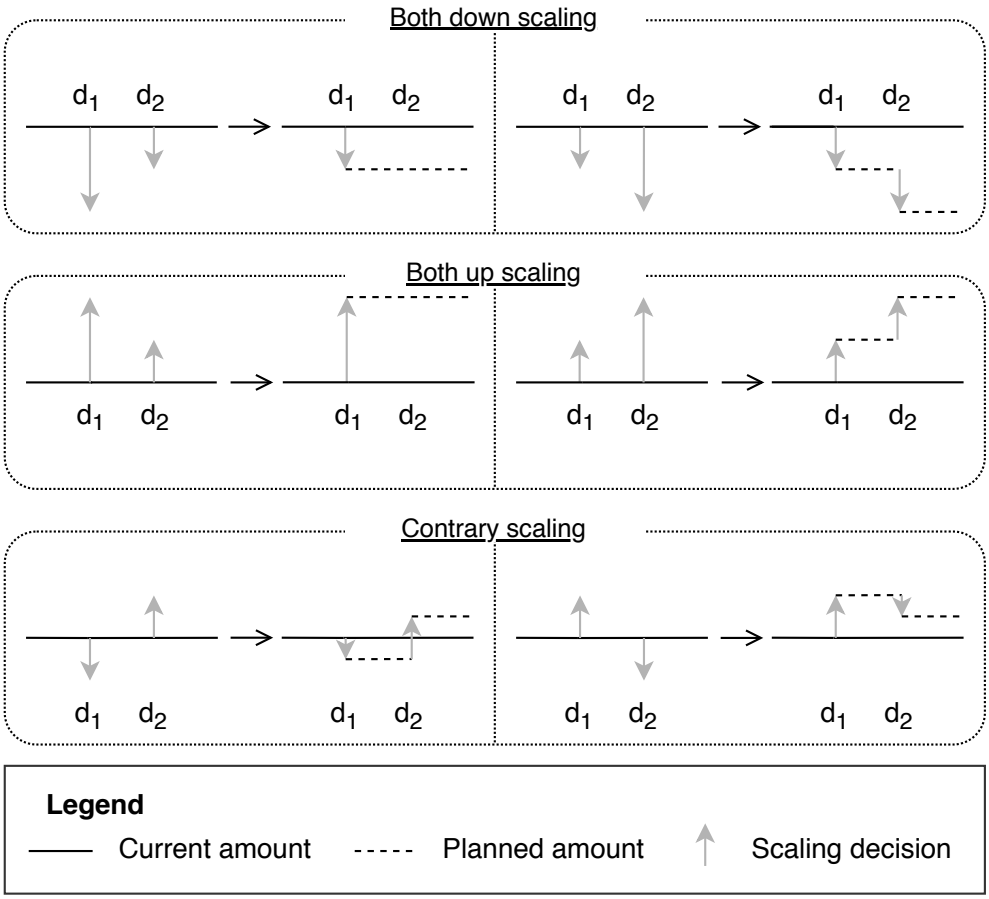

Figure 9.2: Optimization of proactive decisions.

The first possibility (both decisions plan to release service instances; upper rectangle in Figure 9.2) has two cases. Firstly, the first decision wants to release

n service instances and the second one wants to release m service instances, where $n \geq m$. As the down-scaling policy of Chamulteon is conservative, the first decision scales m service instances down and the second one triggers the releasing of 0 service instances ($\neq$ NOP). In the second case (i.e., $m > n$), the first decision scales n service instances down and the second one releases $m - n$ service instances. The second option (both decisions want to add more service instances; middle rectangle in Figure 9.2) has two cases. Firstly, the first decision wants to scale up n service instances and the second one wants to add m more service instances, where $n \geq m$. The resulting first decision allocates n extra service instances. The second decision triggers the allocation of 0 new service instances ($\neq$ NOP). In the second case (i.e., $m > n$), the first decision adds n new service instances and second one allocates $(m - n)$ additional service instances. In the last scenario, the decisions request opposite scaling actions (lower rectangle in Figure 9.2). There are also two cases. Firstly, the first decision wants to release n service instances and the second one wants to allocate m more service instances with $n \geq m$ or $m > n$. To handle the contrary decisions, Chamulteon uses a shock absorption factor $0 < \xi \leq 1$. Thus, the first decisions releases $\lfloor(\xi \cdot d_1)\rfloor$ service instances and the second one scales $\lceil(\xi \cdot (d_1 + d_2))\rceil$ service instances up. The second case is complementary to the first case. The first event allocates $\lceil(\xi \cdot d_1)\rceil$ service instances and the second one releases $\lfloor(\xi \cdot (d_1 + d_2))\rfloor$ VMs. If $\xi = 1$, the contrary actions are executed without modifications. With decreasing ξ the distance between the opposite actions decreases. In other words, ξ influences the degree of oscillation.

9.3 Decision Conflict Resolution

As Chamulteon consists of a proactive and reactive cycle, the controller determines both reactive and proactive decisions, resolves conflicts between decisions, and schedules them accordingly. A decision has information about its type, either *proactive* or *reactive*, the number of required service instances, its trustworthiness, and its planned execution time. A reactive decision should be executed immediately and is always considered as trustworthy. In contrast, proactive decisions have an execution time in the future and are only trustworthy when the model accuracy of the underlying forecast has a MASE (see Section 3.3.3) below a certain threshold. The resolution strategies of possible conflicts are described in the following, and Figure 9.3 shows an example of this process. Note that the resolution takes part per service. That is, the decision of a service has no conflicts with a decision from another service.

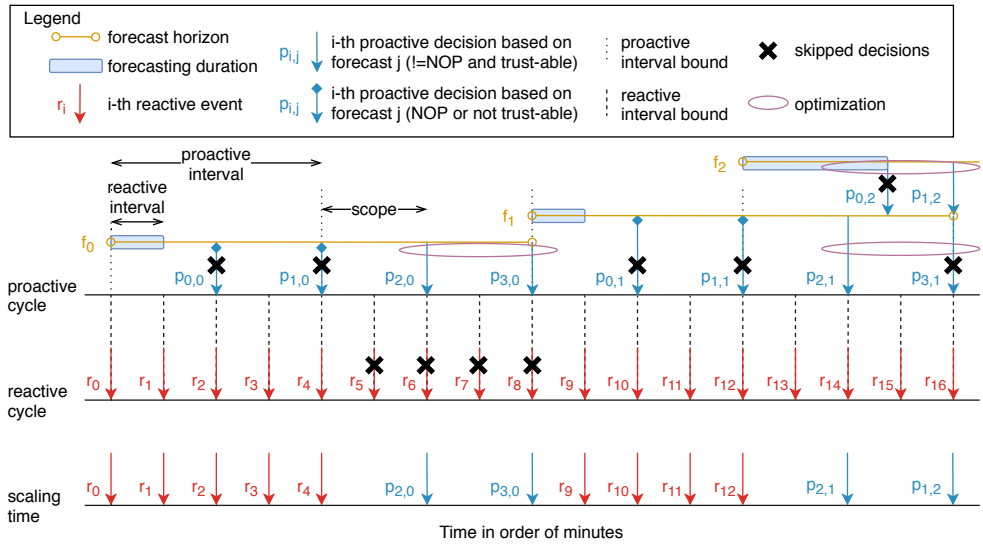

Figure 9.3: Example of Chamulteon's conflict resolution.

9.3.1 Scope Conflict Resolution

Each proactive decision has a scope in which no other decision should be executed. In other words, there is an associated time interval before a decision in which no other decision should occur. Due to the different reconfiguration intervals of reactive and proactive cycles, there may be reactive decisions in the scope of a proactive decision. To resolve this scope conflict, Chamulteon checks whether the proactive decision is trustworthy and wants to scale up or down ($\neq$ NOP). If these conditions are true, the reactive decisions are omitted. Otherwise, the proactive decision is skipped. For instance in Figure 9.3, the reactive decisions r_0, r_1, r_5, and r_6 are triggered in the scope of the proactive decisions $p_{0,0}$ and $p_{2,0}$. As $p_{0,0}$ is a NOP, the reactive decisions r_0 and r_1 are executed while $p_{0,0}$ is skipped. As $p_{2,0}$ is trustworthy and no NOP, r_5 and r_6 are omitted while $p_{2,0}$ is executed.

9.3.2 Time Conflict Resolution

As Chamulteon executes a new forecast as soon as a drift between the last forecast and the monitored arrival rates occurs, there may be proactive decisions based on the previous forecast and decisions based on the newly conducted forecast for the same period. Assuming that decisions based on the newest forecast contain more up-to-date information, the proactive decisions based

on the previous forecast are simply skipped. An example of this resolution is shown in Figure 9.3: The values of the forecast f_1 deviates from the observed arrival rates by more than the tolerance value. To this end, a new forecast f_2 is conducted, although forecast values are left from f_1. Consequently, there is a conflict as the proactive decisions $p_{1,2}$ and $p_{3,1}$ are scheduled at the same time. As $p_{1,2}$ has more recent information (e.g., the current service demand), $p_{1,2}$ is executed and $p_{3,1}$ is skipped accordingly.

9.3.3 Delay Conflict Resolution

This conflict resolution is a legacy of the Chameleon approach: The forecasting component of Chameleon, which has sARIMA and TBATS deployed, exhibits a high variance in time-to-result. Consequently, proactive decisions can be belated. In such a case, the proactive decision for this time generated by the previous forecast is executed. Figure 9.3 shows an example of this resolution: While the forecast f_2 takes too long (the blue box indicates the time-to-result), the proactive decision is belated. To this end, the proactive $p_{2,1}$ generated with forecast f_1 is executed and the currently generated decision $p_{0,2}$ is ignored. In contrast to the Chameleon design, Chamulteon has Telescope deployed within the forecasting component. Although this conflict should not occur while using Telescope, which has a low variance in terms of time-to-result, this resolution is still used for reliability purposes.

9.4 Cost-Aware Resource Management

In public cloud environments, the time a service instance is used and charged may differ depending on the provider's pricing scheme. More precisely, we have to distinguish between two different service instance times: (i) *Accounted instance time* and (ii) *charged instance time*. The accounted instance time is the total runtime of all service instances. The charged instance time is the runtime the public cloud provider charges. Figure 9.4 shows an example of both instance times for an hourly pricing model, which is, for instance, deployed by Amazon Web Services EC2. The red blocks represent the charged instance time and the green blocks the accounted instance time. Service instance 1 (e.g., a VM) has an accounted instance time $a_{0,0}$ of 1.25 on the left and is charged $c_{0,0}$ for two hours as all started hours are charged in full no matter if the resource is stopped earlier. On the right, the accounted instance time $a_{0,1}$ matches the charged instance time $c_{0,1}$ of one hour. The second service instance is started three times and runs only for a few minutes each time. However, it is charged for three full

hours, even if the previous charging interval is still running. The third service instance runs for a bit more than two hours but is charged for three hours. So, all started hours are rounded to a full hour charged instance time. Also, each start of the same service instance is considered to be a completely new service instance without recognition of previous and still running charging intervals.

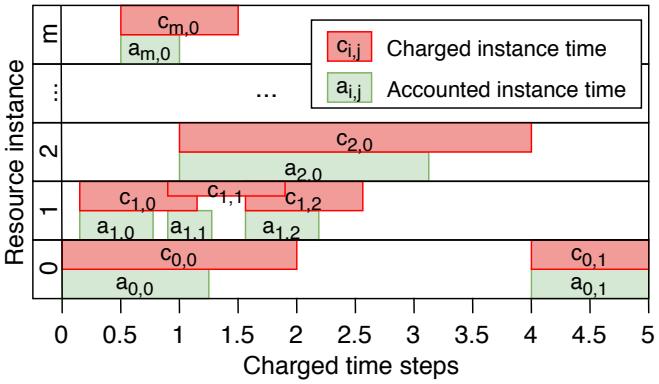

Figure 9.4: Example of instance times that are accounted and charged differently.

To avoid wasting costs introduced by adapting an application in response to changes in the workload, the proposed Fox approach revises the scaling decisions of an elasticity mechanism in accordance with the public cloud provider's pricing scheme.

9.4.1 Design Overview of the Fox Approach

Fox's underlying idea is to operate as a mediator between an auto-scaling mechanism and the application for adapting the associated scaling decisions based on a predefined charging model. More precisely, Fox reviews the auto-scaler's scaling decisions based on future decisions and revises them according to the charging model. For instance, some scaling down decisions are delayed or canceled according to the charging interval. To enable cost-aware scaling, Fox contains a knowledge base, a forecasting component, and an auto-scaler interface. To support as many auto-scalers as possible, Fox assumes homogeneous requests, that is, single class case, and homogeneous resource types in each service. The application, however, can consist of different resources. The knowledge base stores the auto-scaler's future scaling decisions and the charging model of the cloud platform provider. At this moment, two different charging

models are supported: (i) hourly charging and (ii) two-phase charging. As the name suggests, the first scheme charges every started hour for a service instance, regardless of whether or not it was released within the hour. Popular cloud platform providers using the hourly charging are Amazon EC2[5], Oracle Cloud[6], and IBM Cloud[7]. For the two-phase charging model, the pricing scheme that the Google Cloud Platform used in 2017 is implemented: In the first phase, a service instance is charged for each started ten minutes, regardless of whether or not the service instance was released within this interval. If a service instance runs longer than 10 minutes, the second phase is applied, where the service instance is charged every minute.

The working-principle of Fox is based on the *MAPE-K* control loop (Kephart and Chess, 2003) and is depicted in Figure 9.5. In the first phase, Fox monitors the application and gathers information such as arrival rates and saves them into the knowledge base. The monitoring interval is set to two minutes. Then, during the *Analyze* phase, Fox fetches forecast values for the next 30 minutes from the forecast component. The forecast arrival rates are forwarded to the interface for the auto-scaling mechanism. For each forecast, the auto-scaler makes scaling decisions for all services and saves them in the knowledge base. In the *Plan* phase, Fox reviews the scaling decisions based on the decisions found for the future forecasts and changes them according to the charging model. Finally, in the *Execute* phase, Fox scales the application based on the revised scaling decisions. The *Analyze*, *Plan* and *Execute* phases are described in more detail in the following sections.

9.4.2 Analyze

In the *Analyze* phase, Fox sends the observed arrival rate history to the forecaster component and receives the forecast values for the next 30 minutes, that is, 15 forecast values. The forecasting is done every 15 minutes so that an overlap in forecasts exists. This overlap is required since Fox evaluates future events to adapt the scaling decisions. For each forecast value and each service, the auto-scaler is polled for making scaling decisions. The auto-scaler receives the forecast value via the interface, the amount of running service instances and the request rate that a single service instance can handle at the specific service. The amount of running service instances for the first forecast value is the amount of current running service instances. For the following forecast values, the planned amount from previous decisions is used. Based on this

[5]EC2: `https://aws.amazon.com/ec2/pricing/on-demand/?nc1=h_ls`
[6]Oracle: `https://www.oracle.com/cloud/compute/pricing.html`
[7]IBM: `https://cloud.ibm.com/gen1/infrastructure/provision/vs`

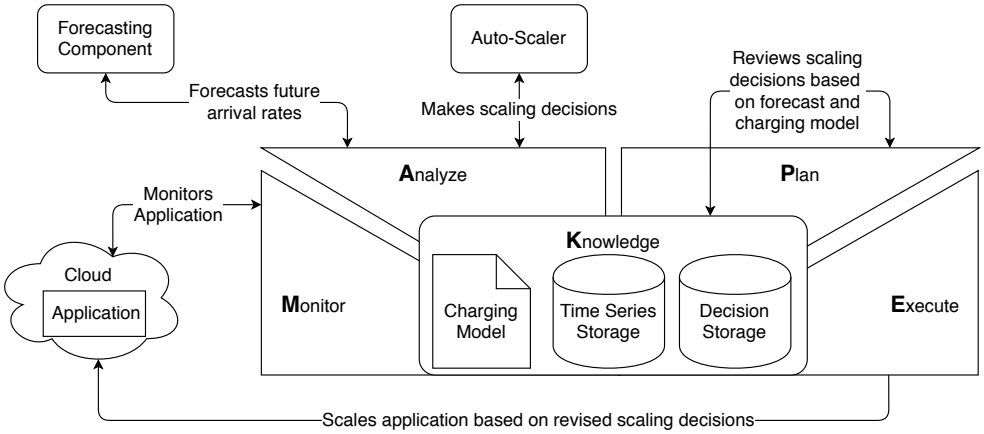

Figure 9.5: MAPE-K cycle of Fox.

information, scaling decisions for each forecast value are made per service and added to the knowledge base. From the second forecaster call on, the overlap of the decisions appears. As the new decisions have more recent information, the old decisions are omitted and replaced by the new ones.

9.4.3 Plan

The idea of Fox is to modify the current scaling decisions based on planned decisions for the future. For example, a down-scaling should be avoided if a service instance is required again soon. In case that an up-scaling should be processed, the decision will not be modified. The decision logic how Fox changes the decisions is depicted in Figure 9.6 and summarized in Algorithm 9.2. First, Fox checks whether the current decision triggers a down-scaling (Line 1). If this holds (both cases in the upper rectangle and the first case in the lower rectangle), all future decisions planned during the next charging interval (Line 2), for instance, one hour, are fetched. Then, Fox iterates over all future decisions (Line 3) and checks whether the number of required resources of the future decision is higher than the number of required resources of the current decision (Line 4). In other words, Fox checks whether the service instances to be released are still required in their charging interval. If this is true, the amount of the current decision is changed to the number of running service instances (left case in the lower rectangle) or the amount of the future decision (first case in the upper rectangle), depending on which one is smaller (Line 5). In case the amount of the future decision is smaller than the amount of the current decision, that is, the service instances to be released are not required again in

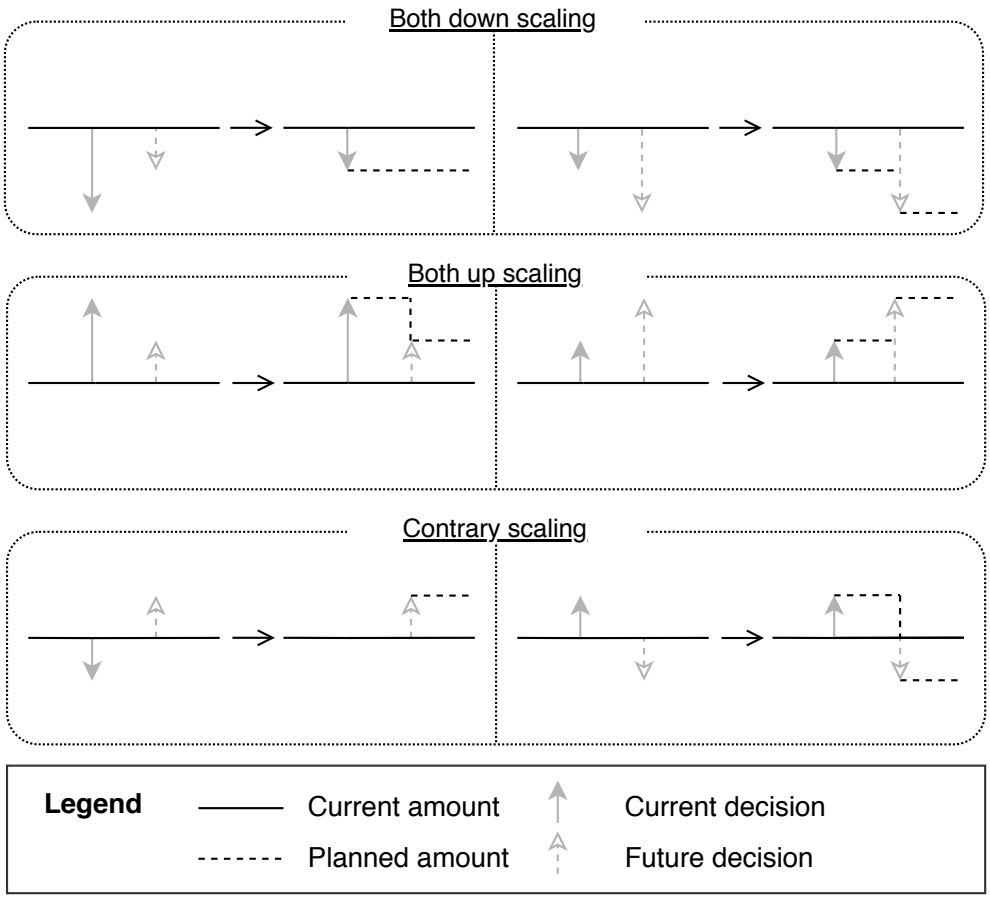

Figure 9.6: Decision logic for comparison to future decisions.

their charging interval, the current decision is not modified. Finally, the revised decision is returned.

9.4.4 Execute

The *Execute* phase is responsible for scaling the application according to the scaling decisions reviewed by Fox. The main task is to decide which service instances should be stopped in case of down-scaling to minimize financial loss. The procedure of this phase works as follows. First, the decisions for the current time are retrieved from the knowledge base. In case of an up-scaling decision, the required service instances are provisioned. In case of a down-scaling decision, service instances that introduce minimum financial

Algorithm 9.2: Revising auto-scaler decisions

```
Input: Decision current, runningInstances run, chargingInterval ci
Result: Revised decision
1 if run > current.amount then // current wants to scale down
2     futures = getFutureDecisionsInInterval(ci)
3     foreach next in futures do
4         if next.amount > current.amount then // see left in the upper &
            lower rectangle in Figure. 9.6
5             current.amount = min(run, next.amount)
6         end
7     end
8 end
9 return current
```

loss if stopped are selected. To determine the service instances that should be stopped, the charging model is taken into account. For the hourly charging model, the runtimes of all service instances are gathered. Then, the service instances closest to the next charging interval, that is, one hour, are selected for down-scaling. For the two-phase charging model, the service instances are sorted descending by their overall runtime so that the service instance which ran longest is at the beginning of the list. Then, the down-scaling amount of service instances is selected from the beginning of the list. So, the service instances with the longest overall runtime are chosen.

9.5 Assessing the Auto-Scaling Quality

Besides the elasticity measures presented in Section 4.2.1, we introduce further measures for capturing the quality of the auto-scaling behavior[8]. Firstly, neither the provisioning accuracy nor the wrong provisioning time share considers oscillations in the adaptation process. To this end, we propose the *instability* measure reflecting unstable behavior. Further, if only individual measures are considered to quantify the quality of auto-scaling behavior, the results may be conflicting. To this end, we define the *auto-scaling deviation* and *elastic speedup* as aggregated elasticity measures for consistent methods. These three elasticity measures were developed in collaboration with Nikolas Herbst (Herbst, 2018). Moreover, these measures are only intended for monolithic applications. There-

[8]We apply the established and the proposed measures in Chapter 11.

fore, we also present the *auto-scaling worst-case deviation*. Lastly, we provide the *cost-saving rate* reflecting the cost-efficiency of an auto-scaler.

9.5.1 Instability

Although the provisioning accuracy (see Section 4.2.1.1) and wrong provisioning timeshare (see Section 4.2.1.2) capture important aspects of the elasticity, cloud platforms can achieve the same value for both measures while their scaling behaviour is different. An example is illustrated in Figure 9.7, where both platforms have the same accuracy and the time in which the platforms are underprovisioned or overprovisioned is exactly the same. However, the difference between the two platforms is the time in which they are stable. To capture this behavior, we define a further elasticity measure called *instability*. The instability υ measures the proportion of time in which the change in resource demand and the change in resource supply have different signs. In other words, the instability describes the time in percentage in which the supply and the demand curves are not changing in the same direction. More formally, the instability

$$\upsilon[\%] = \frac{100}{T} \cdot \int_{t=0}^{T} min\Bigg(\Big|sgn\Big(\frac{d}{dt}s_t\Big) - sgn\Big(\frac{d}{dt}d_t\Big)\Big|, 1\Bigg)dt, \tag{9.1}$$

where s_t is the resource supply at time t, d_t the resource demand at time t, and T the experiment duration. This measure's values lie in the interval $[0; 1]$ with 0, indicating that the changes in demand and supply run parallel during the measurement. In contrast to the accuracy and time share metrics, a value of zero is a necessary but not sufficient requirement for a perfectly elastic behavior. However, the instability is a valuable indicator of the cost incurred by oscillations in the adaptation process.

9.5.2 Auto-Scaling Deviation

The idea of this aggregated measure is to calculate the deviation of the auto-scaler's scaling behavior from the theoretically optimal adaptation process (i.e., the resource demand and resource supply curves are identical at any point in time). The scaling behavior can be described by a vector comprising elasticity measures and system-oriented measures. More precisely, the provisioning accuracy, the wrong provisioning time share, the instability, and the SLO violations are taken into account. Each measure is expressed as a percentage, and the closer the value is to zero, the better the auto-scaler performs in terms of the elasticity aspect described by the measure. To give equal weight to each

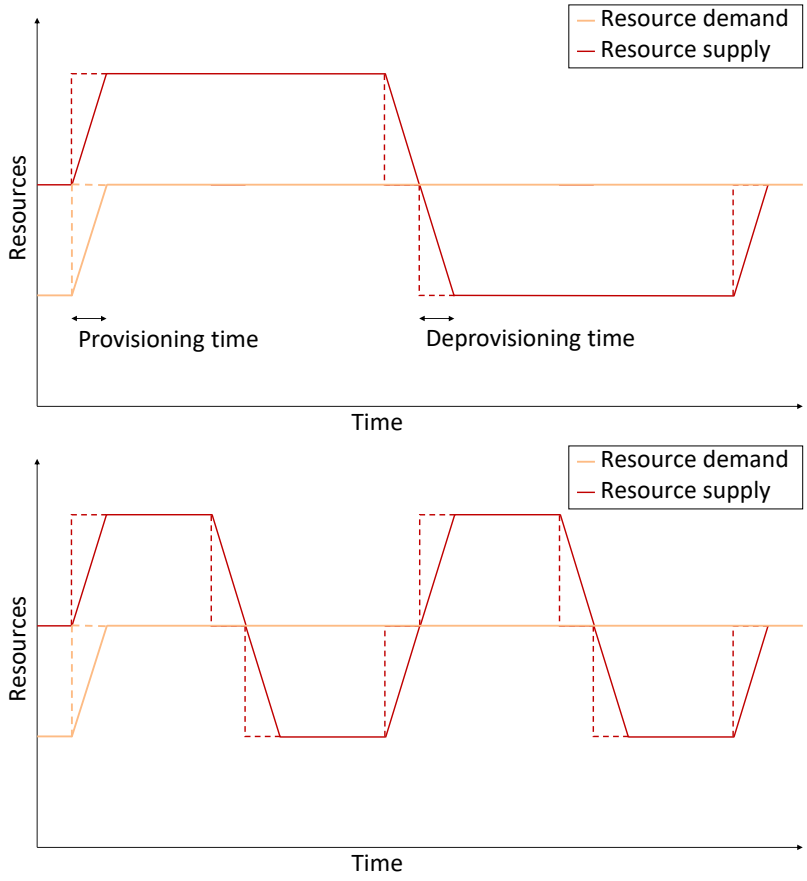

Figure 9.7: Example of two systems with the same elasticity accuracy and time share.

measure considered, the provisioning accuracy and the wrong provisioning time share are mapped to one measure each. For this purpose, a weighted sum for each elasticity aspect is used for both of its associated measures. More formally, the *overall provisioning accuracy* is expressed as

$$\theta[\%] := \gamma \cdot \theta_U + (1 - \gamma) \cdot \theta_O, \tag{9.2}$$

where θ_U is the underprovisioning accuracy, θ_O is the overprovisioning accuracy, and $0 < \gamma < 1$. Analogously, the *overall wrong provisioning time share* is computed as

$$\tau[\%] := \gamma \cdot \tau_U + (1 - \gamma) \cdot \tau_O, \tag{9.3}$$

where τ_U is the underprovisioning time share and τ_O is the overprovisioning time share, and $0 < \gamma < 1$. In both equations, γ represents a penalty factor, that is, a factor reflecting custom requirements. In other words, γ can be set individually to penalize underprovisioning ($\gamma > 0.5$) or overprovisioning ($\gamma < 0.5$). If γ is set to 0.5 (which is also the case in Chapter 11), both the underprovisioning and the overprovisioning are considered equally bad.

For calculation the deviation between an auto-scaler and the theoretically optimal auto-scaler, the *Minkowski distance* is applied. Assuming that the theoretically optimal auto-scaler knows when and how the resource demand changes, the values for the overall provisioning accuracy θ, the overall wrong provisioning time share τ, the instability υ, and the SLO violations ψ are zero. Consequently, if an auto-scaler is compared to the theoretically optimal auto-scaler, the Minkowski distance[9] d_p can be reduced to the p-norm as $\|x - 0\|_p = \|x\|_p$ with $x = (\theta, \tau, \upsilon, \psi)$. More formally, the *auto-scaling deviation* is defined as

$$\sigma[\%] := \|x\|_4 = \left(\theta^4 + \tau^4 + \upsilon^4 + \psi^4\right)^{\frac{1}{4}}. \tag{9.4}$$

The values of this measure lie in the intervals $[0; \infty)$, that is, the closer the auto-scaling deviation is to zero, the closer the behavior of the auto-scaler is to the theoretically optimal auto-scaler.

9.5.3 Elastic Speedup

Besides the auto-scaling deviation, we introduce a further measure which reflects the added value of a special auto-scaler without comparing each elasticity metric separately. Loosely speaking, the *elastic speedup* ϵ calculates for each auto-scaler how the elasticity is affected by its scaling behaviour compared to the case where no auto-scaler is employed To this end, a geometrical mean of the ratio between each elasticity measure pair is calculated. More formally, the elastic speedup is defined as

$$\epsilon := \left(\frac{\tilde{\theta}_U}{\theta_U} \cdot \frac{\tilde{\theta}_O}{\theta_O} \cdot \frac{\tilde{\tau}_U}{\tau_U} \cdot \frac{\tilde{\tau}_O}{\tau_O} \cdot \frac{\tilde{\upsilon}}{\upsilon},\right)^{\frac{1}{5}}, \tag{9.5}$$

where θ_U/ θ_O (under-/ overprovisioning accuracy), τ_U/ τ_O (under-/ overprovisioning time share), and υ (instabillity) are the measured elasticity metrics of the auto-scaler, while $\tilde{\theta}_U$, $\tilde{\theta}_O$, $\tilde{\tau}_U$, $\tilde{\tau}_O$, and $\tilde{\upsilon}$ are the calculated elasticity metrics if no auto-scaler is deployed. The values of this measure lie between 0 and ∞.

[9] The parameter p describes the order of the distance and a popular representative is the Euclidean distance ($p = 2$).

If $\epsilon > 1$, the observed auto-scaler performs better than if no auto-scaler is used. Otherwise, the auto-scaler is equal or worse than a fixed amount of supplied resources. A disadvantage of the elastic speedup is its high sensitivity to values close to zero and being undefined when one or more of the elasticity measures are zero.

9.5.4 Auto-Scaling Worst-Case Deviation

While elasticity metrics are straightforward for monolithic applications, their use becomes more difficult for applications comprising multiple services. In other words, an application consisting of one service has a limited number of possible configurations that can be measured and quantified. In contrast, an application with multiple services has $\prod n_i$ possible resource configurations, where n_i is the maximum allowed number of resources for service i. Moreover, there may be different resource configurations that are equally optimal regarding the served requests. So, it is challenging to determine which of these configurations is the best in terms of elasticity. Conversely, it is just as difficult to determine which auto-scaler is worse. For example, if one auto-scaler overprovisions the first and second service with one resource each and another auto-scaler overprovisions the first service with two resources, but does not overprovision the second service.

To face the issue of how to compare system elasticity in a setting with multiple different services, we propose the *auto-scaling worst-case deviation* ς. The core idea is to compare the auto-scalers in terms of their worst behavior across all services as the services are interdependent and the system performance is limited by the performance of the "weakest" service. In other words, the worst elasticity measures across all services are considered. To this end, the elasticity measures are calculated for each service. Based on these measures the worst performance across all services is selected as follows

$$\hat{\tau}_U[\%] := \max_{1 \leq i \leq n} (\tau_{U,S_i}), \tag{9.6}$$

$$\hat{\tau}_O[\%] := \max_{1 \leq i \leq n} (\tau_{O,S_i}), \tag{9.7}$$

$$\hat{\theta}_U[\%] := \max_{1 \leq i \leq n} (\theta_{U,S_i}), \tag{9.8}$$

$$\hat{\theta}_O[\%] := \max_{1 \leq i \leq n} (\theta_{O,S_i}), \tag{9.9}$$

$$\hat{\upsilon}[\%] := \max_{1 \leq i \leq n} (\upsilon_{S_i}), \tag{9.10}$$

where θ_{U,S_i}/ θ_{O,S_i} is the under-/ overprovisioning accuracy of the i-th service, τ_{U,S_i}/ τ_{O,S_i} the under-/ overprovisioning time share of the i-th service, υ_{S_i} the

instability of the i-th service, and $n > 1$ is the number of different services. Similar to the auto-scaling deviation (see Section 9.5.2), we calculate the *overall worst-case provisioning accuracy* $\hat{\theta}$ and the *overall worst-case wrong provisioning time share* $\hat{\tau}$ as weighted sum to quantify the worst-case deviation. More formally, both measures can be computed as

$$\hat{\theta}[\%] := \gamma \cdot \hat{\theta}_U + (1-\gamma) \cdot \hat{\theta}_O, \tag{9.11}$$

$$\hat{\tau}[\%] := \gamma \cdot \hat{\tau}_U + (1-\gamma) \cdot \hat{\tau}_O, \tag{9.12}$$

where γ is a weighting factor for penalizing underprovisioning $(0.5 < \gamma < 1)$ or overprovisioning $(0 < \gamma < 0.5)$. If γ is set to 0.5 (which is also the case in Chapter 11), both underprovisioning and the overprovisioning are considered equally bad.

For calculating the auto-scaling worst-case deviation of the considered auto-scaler from the theoretically optimal auto-scaler, we apply the Minkowski distance. Under the assumption that the theoretically optimal auto-scaler knows when and how much the resource demand change, the values for worst-case provisioning accuracy $\hat{\theta}$, worst-case wrong provisioning time share $\hat{\tau}$, and worst-case instability $\hat{\upsilon}$ are equal to zero. Consequently, if an auto-scaler is compared to the theoretically optimal auto-scaler, the p-norm can be used (see Section 9.5.2). More formally, the auto-scaling worst-case deviation is defined as

$$\varsigma[\%] := \|(\hat{\theta}, \hat{\tau}, \hat{\upsilon})\|_3 = \left(\hat{\theta}^3 + \hat{\tau}^3 + \hat{\upsilon}^3\right)^{\frac{1}{3}}. \tag{9.13}$$

The values of this measure lie in the intervals $[0; \infty)$, that is, the closer the auto-scaling deviation is to zero, the closer the auto-scaler behavior is to the theoretically optimal auto-scaler.

9.5.5 Cost-Saving Rate

Besides the elasticity measures, we also want to quantify the cost savings of an auto-scaler. The idea of this cost measure is to compare the instance times of an auto-scaler to the instance times of a naïve approach. The naive approach provides all available resources at the beginning of the experiment and has no auto-scaling mechanisms, that is, all provisioned resources run throughout the experiment. More formally, the *cost-saving rate* can be expressed as as

$$\Pi_x[\%] = 100 \cdot \left(\frac{\pi_{s,x}}{\pi_{n,x}} - 1\right), \tag{9.14}$$

where $\pi_{s,x}$ and $\pi_{n,x}$ are the instance times of the auto-scaler and the naïve approach, respectively. $\Pi_x \in (-1, \infty)$ where a negative value indicates that costs are saved in comparison to the naïve approach. Consequently, the lower the value is in the negative range, the more costs are saved. A value greater or equals zero indicates that the auto-scaler spends more or the same cost as the naïve approach. Both types of instance times are considered (see Section 9.4): Π_a and Π_c reflect the cost-saving rate for the accounted instance times and the cost-saving rate for the charged instance times, respectively.

9.6 Assumptions and Limitations

Chamulteon has the following assumptions, which are either based on the integrated tools, design decisions, or have been inherited from Chameleon: (i) To capture an accurate seasonal model, the availability of at least two days of historical data is required. Moreover, a history of at least two years is required to cover holidays and other specials days. (ii) An external monitoring tool has to be employed that retrieves the required information (e.g., request rates). (iii) The application in question has to be request-based due to the service demand estimation. (iv) The DML model, which captures the control flow within the application, has to be either created manually or by an external tool. Moreover, it must be possible to transform the given model into a product-form queueing network (v) While the application can consist of different resources, the resource types for each service are assumed to be homogeneous. Moreover, it is assumed that each instance of a service is hosted solely on a virtual machine or container. There are no further assumptions regarding the deployment, such as if each container has a dedicated physical resource assigned. (vi) The scaling takes place in a horizontal manner. That is, only replicas of the service instances are added or released. In other words, the number of replicates is adapted while the resources allocated to a service instance and the physical nodes are not scaled.

Besides the assumptions, Chamulteon is subject to the following limitations: (i) The cost-awareness component only supports hourly and the two-phase charging model. (ii) The forecasting component does not support a calendar function where special days (e.g., Cyber Monday, i.e., the Monday after Thanksgiving, is a special day for e-commerce in the USA) can be specified. (iii) Due to the integrated tools, Chamulteon has a lower usability than a simple reactive auto-scaling mechanism. However, the used tools can be replaced with other techniques as long as they work compliant with the defined required interfaces. (iv) The forecasting component does not distinguish between different request

types. That is, the forecast includes only the total number of requests without information on the distribution of the different types of requests.

9.7 Distinctive Features of Chamulteon

To delimit Chamulteon from the related work, we summarize the reviewed auto-scalers (see Chapter 6) and give an overview in Table 9.1. This overview distinguishes (i) whether the approach can scale a monolithic application or a distributed application comprising multiple services, (ii) if the scaling is done in a horizontal or/and vertical manner, (iii) whether the approach is reactive or/and proactive, (iv) if the approach is cost-efficient and what group it belongs to (see Section 6.6), (v) whether the evaluation was done experimental or/and simulated, (vi) whether real-world or synthetic workloads were used for the experiment, and (vii) if the approach is open-source.

Table 9.1: Overview of related work on cloud auto-scalers.

Auto-Scaler	Application	Scaling Dimension	Type	Cost-Efficient	Evaluation	Workload	Open-Source
Chamulteon	distributed	horizontal	hybrid	III	experimental	realistic	(✓)[10]
(Kalyvianaki et al., 2009)	distributed	vertical	proactive	–	experimental	synthetic	✗
(Padala et al., 2009)	distributed	vertical	proactive	–	experimental	both	✗
(Zhu and Agrawal, 2012)	distributed	vertical	reactive	II	experimental	real-world	✗
(Ali-Eldin et al., 2012)	monolithic	horizontal	hybrid	–	simulated	realistic	✓
(Lakew et al., 2017)	distributed	vertical	proactive	–	experimental	both	✗
(Urgaonkar et al., 2008)	distributed	horizontal	hybrid	–	experimental	both	✗
(Xiong et al., 2011)	distributed	vertical	reactive	II	experimental	real-world	✗
(Sharma et al., 2012)	distributed	horizontal	reactive	I	both	real-world	✗
(Jiang et al., 2013)	monolithic	horizontal	hybrid	II	experimental	real-world	✗
(Beltrán, 2015)	distributed	both	reactive	I	experimental	synthetic	✗
(Tesauro et al., 2006)	distributed	horizontal	proactive	–	experimental	synthetic	✗
(Rao et al., 2009)	distributed	vertical	proactive	–	experimental	synthetic	✗
(Dezhabad and Sharifian, 2018)	monolithic	horizontal	proactive	–	simulated	real-world	✗
(Benifa and Dejey, 2019)	distributed	horizontal	proactive	–	experimental	real-world	✗
(Chieu et al., 2009)	monolithic	horizontal	reactive	–	experimental	synthetic	✗
(Maurer et al., 2011)	monolithic	vertical	reactive	–	simulated	synthetic	✗
(Han et al., 2012)	distributed	both	reactive	I	experimental	synthetic	✗
(Naskos et al., 2017)	monolithic	horizontal	reactive	III	experimental	real-world	✗
(Iqbal et al., 2011)	distributed	horizontal	hybrid	–	experimental	synthetic	✗
(Nguyen et al., 2013)	distributed	horizontal	proactive	–	experimental	real-world	✗
(Fernandez et al., 2014)	monolithic	horizontal	proactive	III	experimental	real-world	✓
(Wu et al., 2016)	distributed	both	hybrid	III	experimental	real-world	✗
(Khorsand et al., 2018)	distributed	horizontal	proactive	–	simulated	both	✗

Although a recent survey (Qu et al., 2018) highlights the importance of combining reactive and proactive auto-scaling mechanisms, most approaches can scale applications either proactively or reactively (see Table 9.1). In contrast to existing hybrid auto-scalers (Urgaonkar et al., 2008; Iqbal et al., 2011;

[10]The code is currently being prepared for publication.

Ali-Eldin et al., 2012; Jiang et al., 2013; Wu et al., 2016) that combine reactive and proactive mechanisms, Chamulteon (i) leverages long-term forecasts from time series analysis in combination with (ii) predictive models from queueing theory, and also integrates (iii) a reactive fallback mechanism. Due to these two integrated mechanisms, Chamulteon has to resolve conflicts introduced by contradicting reactive and proactive scaling decisions. The reviewed approaches do not explicitly address this issue.

In terms of cost-efficient scaling, Chamulteon belongs to the third group of the cost-aware auto-scaling classification. It combines and extends the example auto-scalers' approaches (Fernandez et al., 2014; Wu et al., 2016; Naskos et al., 2017) of this category. Chamulteon supports more complex charging models, such as the two-phase model that was used in the Google Cloud Platform. Moreover, Chamulteon finds proactive decisions for the future. Based on these future decisions and the knowledge of the charging model, a decision logic is presented when a downscaling is meaningful and when the already running instance should stay running. None of the mentioned auto-scalers support future decisions and reviews the actual decision using them and the knowledge of the charging model.

In contrast to the evaluation/simulation of the reviewed approaches, Chamulteon is evaluated with different realistic experiment setups, resource demands, and real-world traces (see Chapter 11). We identify only two cases of the work comparing multiple and different, proactive auto-scaling policies. The work of Padadopoulos et al. (Papadopoulos et al., 2016) establishes theoretical bounds on the worst-case performance using simulation. The related experimental evaluation in Ilyushkin et al. (Ilyushkin et al., 2018) compares auto-scaler performance for the different types of workflow applications in one deployment. In other words, the reviewed articles lack a sound and comparative evaluation methodology. This observation is also underlined by our former survey (Papadopoulos et al., 2019b).

9.8 Concluding Remarks

In this chapter, our contribution Chamulteon addresses the research question RQ 6 "*What is a meaningful combination of proactive and reactive scaling techniques to minimize the risk of auto-scaling in operation?*" by implementing a proactive and reactive cycle and the associated conflict resolution between proactive and reactive scaling decisions. More precisely, proactive adaptations are planned based on forecasts of Telescope, while reactive adaptations are triggered based on actual observations of the monitored load intensity. Considering the re-

search question RQ 8 "*How to enable coordinated scaling of applications comprising multiple services?*", Chamulteon takes into account all services and their associated decisions coupled with the DML model, which captures the application's internal flow. The contribution Fox tackles the research question RQ 7 "*How can scaling decisions be adjusted so that the charged costs in a public cloud environment are minimized?*" by leveraging the knowledge of the public cloud provider's charging model and reviewing scaling decisions found by an auto-scaler to reduce the charged costs to a minimum. More precisely, Fox delays or omits releases of resources to avoid additional charging costs if the resource is required soon. Lastly, we propose a novel set of elasticity measures and the cost-saving rate to quantify the quality as well as the cost-efficiency of an auto-scaler to answer the research question RQ 9 "*What are meaningful measures for assessing the quality of coordinated and cost-aware auto-scaling?*".

Part III

Benchmarking and Evaluation

Chapter 10

Time Series Forecasting Competition

In this chapter[1], we evaluate[2] and benchmark our contribution Telescope and its components. First, in Section 10.1, we introduce the methods in competition and the applied measures. Then, in Section 10.2, we benchmark different forecasting methods to select the best performing methods as competitors of Telescope. Afterwards, we evaluate the recommendation system, which is integrated in Telescope, in Section 10.3. In Section 10.4, we benchmark Telescope against recent hybrid forecasting methods and the best performing forecasting methods from Section 10.2. Finally, we conclude this chapter in Section 10.5.

10.1 Global Experimental Setup

In this section, we provide information about the experimental setups of the subsequent experiments. Note that each of the following evaluation sections still has a separate experimental description for specific information. In Section 10.1.1, we outline the competing methods and their usage. In Section 10.1.2, we highlight the applied measures for quantifying the experiments.

10.1.1 Methods in Competition

For having a broad and representative forecasting method competition, we compare different methods from different fields. The competitors can be grouped into three categories: (i) "classical" time series forecasting methods, (ii) regression-based machine learning methods, and (iii) hybrid forecasting methods (i.e., taking advantage of at least two methods). For each category, we consider a set of representative methods. A detailed overview of the methods can be found in Section 3 and in Chapter 5. The techniques from the first category are listed and briefly described below:

[1] This chapter is based on our previous works (Bauer et al., 2020a,b,c)

[2] In this chapter, the evaluation is based on the multi-step-ahead forecast.

- ETS builds on the concept of exponential smoothing and is a framework that automatically retrieves the components (trend, season, and error) of a time series and determines the relationships (additive, multiplicative, or not present) between the components.

- sARIMA[3] extends the ARIMA model by adding a seasonal counterpart to each component (autoregressive model for the past values, moving average for the past forecast errors, and time series differencing for stationarity).

- sNaïve repeats past observations for the entire forecast horizon. More precisely, each forecast value is equal to the corresponding observation from the last period.

- TBATS extends ETS using a trigonometric representation based on Fourier series for the seasonal part of the time series and an autoregressive-moving average model for the error corrections. Further, the time series is transformed with Box-Cox transformations.

- Theta first checks whether the time series is seasonal, and if so, de-seasonalizes the time series. The de-seasonalized time series is then split into a short- and a long-term component. Finally, the forecast is the weighted forecast of both components.

The competing methods from the field of machine learning are outlined in the following:

- *GPyTorch* (Gardner et al., 2018) is a Gaussian process library. The basic idea for modeling the time series is to apply Gaussian process inference based on black-box matrix-matrix multiplication.

- NNetar is a feed-forward neural network with one hidden layer. The model is trained with lagged values of the time series, while the number of lags and the number of nodes in the hidden layer are automatically selected.

- Random forest is an ensemble of decision trees based on bagging, that is, the trees are generated in parallel, fully grown, and independent of each other. To reduce overfitting, each tree is trained on a random sample of the features. In the following, we refer to this method as *RF*.

[3] In the experiments, we use auto.arima (Hyndman and Athanasopoulos, 2017) to find the most suitable model for the time series automatically.

- SVR is based on the same principle as SVM's, that is, finding separation lines to group the data into different classes, with the extension to predict numerical values.

- XGBoost is an ensemble of decision trees based on gradient tree boosting, that is, the trees are growing sequentially with knowledge from their preceding tree. To reduce overfitting, XGBoost applies regularization objects, shrinkage, and feature subsampling.

The representatives of the hybrid forecasting methods are shortly described below:

- BETS decomposes the time series into the components trend, season, and irregular. Different versions of the irregular part are then simulated, resulting in different versions of the original time series. Then, these versions are forecast separately by ETS, and the final forecast is the median of all forecasts.

- ES-RNN is a hybrid forecasting method based on time series decomposition developed by Uber. The basic idea is to de-seasonalize the time series using exponential smoothing and to use a neural network for extrapolation of the time series.

- FFORMS is based on forecasting method recommendation. More precisely, a random forest is applied as a classifier to map the most appropriate forecasting method (ETS, NNetar, sARIMA, sNaïve, TBATS, and Theta) to a specific time series described by a set of time series characteristics.

- *Hybrid*[4] performs an ensemble forecast. The considered methods comprise ETS, NNetar, Theta, sARIMA, and TBATS. For the ensemble forecast, each method performs a forecast, and the final forecast is the average of these forecasts.

- Prophet is a hybrid forecasting method based on decomposition developed by Facebook. The idea is to decompose the time series into the components trend, season, holiday, and error. Each component is forecast by a different approach. Then, the forecast parts are assembled to form the final forecast.

[4]Hybrid forecasting method: `https://cran.r-project.org/web/packages/forecastHybrid/vignettes/forecastHybrid.html`

During the measurements, each method was deployed on a VM (Ubuntu 18.04.3, 2 vcores, and 4 GB RAM) in our private CloudStack cluster. The details of the cluster are given in Section 11.1.3. The methods were executed either with R (V 3.4.4), C++ (V 11), or Python (V 3.6.7). Further, each method got a single time series as input if not stated otherwise. Note that using one time series at a time is a major difference to the M4-Competition[5] where it is possible to use the complete training data set (i.e., all time series) for training the algorithms. In our experiments, we perform multi-step-ahead forecasts. That is, each time series was divided into history (first 80% of the time series) and test (remaining 20% of the time series). Based on the history, each method learned a model that was used for forecasting future values of the time series (i.e., the test part) at once with a single execution. This forecasting procedure (i.e., receiving the time series, estimating the parameters, building the model, and forecasting the time series) was repeated ten times for each time series. Consequently, the reported measures were determined on the average values of each time series.

As input for the forecasting task, the classical time series forecasting methods, BETS, NNetar, and Hybrid received the time series and the respective frequency. Prophet also needs the timestamps of the time series. The regression-based machine learning methods except NNetar received the time series and a synthetic seasonal pattern (a vector with the indices modulus the frequency) as input. To achieve a reliable forecast for GPyTorch, we shifted each time series linearly to obtain a mean value of zero (Rasmussen and Williams, 2006). FFORMS was trained on the M4- and M3-Competition and received as input also the time series and the respective frequency. In contrast to the other methods, ES-RNN requires, besides the time series and frequency, a set of time series. To this end, ES-RNN got the same time series several times. Note that we used all methods "out-of-the-box" since the results of the M3-Competition have shown that the methods were kept simple and that complex models do not necessarily perform better (Makridakis and Hibon, 2000). That is, there was no parameter tuning, and the methods were used with their default settings. Recall the “No-Free-Lunch Theorem” (Wolpert and Macready, 1997), stating that improving a method for one aspect leads to deterioration in performance for another aspect. Moreover, also the techniques deployed in Telescope were used with their default settings.

[5] M4-Competition: `https://www.mcompetitions.unic.ac.cy/the-dataset/`

10.1.2 Applied Measures

To compare and quantify the performance of the different forecasting methods, we report forecast error measures (see Section 3.3 and 7.5) and the time-to-result. The time-to-result for a time series reflects the duration in which the forecasting method receives the time series, estimates the parameters, creates the model, and performs the forecast. For the forecast accuracy, we consider the established forecast accuracy measures (see Section 3.3) MASE as well as sMAPE and apply measures proposed in this thesis. Table 10.1 lists a brief description of each measure.

Table 10.1: Overview of the applied measures.

Name	Description
e_{sM}	Forecast accuracy based on the sMAPE. The measures $\overline{e}_{\text{sM}}$, $\tilde{e}_{\text{sM}}$, and $\sigma_{e_{\text{sM}}}$ reflect the average error, median error, and the standard deviation of the error.
e_{MA}	Forecast accuracy based on the MASE. The measures $\overline{e}_{\text{MA}}$ and $\sigma_{e_{\text{MA}}}$ reflect the average error and the standard deviation of the error.
ρ_U, ρ_O	Mean wrong-estimation share as the relative number of forecast values that under- or overestimate the actual values.
δ_U, δ_O	Mean wrong-accuracy share as the mean percentage error during under- or overestimating the actual values.
ϑ	The forecast accuracy degradation reflects the deterioration compared to the best method. The measures $\overline{\vartheta}$, $\tilde{\vartheta}$, and σ_ϑ reflect the average forecast accuracy degradation, the median forecast accuracy degradation, and the standard deviation of the forecast accuracy degradation.
t_{sN}	The time-to-result normalized by the time required by sNaïve. The measures $\overline{t}_{sN}$, $\tilde{t}_{sN}$, and $\sigma_{t_{sN}}$ reflect the average time, median time, and the standard deviation of the time.

10.2 Benchmarking of Forecasting Methods

In this section, we benchmark a set of classical forecasting methods as well as regression-based machine learning methods. First, we describe the subsequent

experiments in Section 10.2.1. Then, we analyze the forecasting performance of the competing methods on the use cases: (i) economics (see Section 10.2.2), (ii) finance (see Section 10.2.3), (iii) human access (see Section 10.2.4), and (iv) nature and demographics (see Section 10.2.5). Afterwards, in Section 10.2.6, we consider all use cases for selecting the competitors for the benchmarking of our Telescope approach. Finally, we sum up the results and discuss threats to validity in Section 10.2.7.

10.2.1 Experimental Description

To investigate the forecast performance of the state-of-the-art forecasting methods and selecting the competing methods for our evolution of Telescope, we use the forecasting benchmark (see Chapter 7). More precisely, we compare on four use cases the forecasting methods (i) ETS, (ii) GPyTorch, (iii) NNetar, (iv) sARIMA, (v) sNaïve, (vi) TBATS, (vii) Theta, (viii) RF, (ix) SVR, and (x) XGBoost. A brief overview of the methods can be found in Section 10.1.1.

10.2.2 Economics Use Case

Table 10.2 and 10.3 show the results of the forecasting competition on the economic use case for the classical time series forecasting methods and the regression-based machine learning methods, respectively. Each row shows a measure (a brief overview of the measures is given in Section 10.1.2). The columns correspond to the methods in competition. The best values (the lower, the better) are highlighted in bold. The most accurate forecasting method based on $\overline{e}_{\text{sM}}$ is ETS (13.02%) followed by sNaïve (13.28%). The most accurate machine learning method is NNetar (16.49%) followed by GPyTorch (17.32%). Concerning $\overline{e}_{\text{MA}}$, the most accurate forecasting method is sARIMA (0.53) followed by TBATS (0.59) and the most accurate machine learning method is XGBoost (0.87) followed by GPyTorch (0.91). For both error measures, all forecasting methods (except sNaïve for $\overline{e}_{\text{MA}}$) exhibit a higher accuracy than the most accurate machine learning method.

As there are different superior methods for both error measures, we investigate measures describing the forecast. More precisely, ρ_O or ρ_U either reflects whether the forecasting method over- or underestimates the future vales. The methods sARIMA, TBATS, and NNetar exhibit almost no tendency ($\rho_O \sim \rho_U$), while the remaining methods tend to underestimate ($\rho_U > 50\%$). Especially, the methods sNaïve, GPyTorch, RF, SVR, and XGBoost underestimate a time series on average more than 2/3 of the forecast horizon. However, while under-

Table 10.2: Comparison of classical time series forecasting methods on the economics use case.

Measures	ETS	sARIMA	sNaïve	TBATS	Theta
$\overline{e}_{\text{sM}}$ [%]	**13.02**	14.88	13.28	14.19	13.40
$\sigma_{e_{\text{sM}}}$ [%]	16.26	25.80	**12.12**	21.06	21.42
$\overline{e}_{\text{MA}}$	0.66	**0.53**	0.90	0.59	0.77
$\sigma_{e_{\text{MA}}}$	1.41	**0.97**	2.20	0.99	1.85
ρ_U [%]	55.52	49.33	67.86	49.71	60.07
ρ_O [%]	44.48	50.67	32.08	50.29	39.93
δ_U [%]	**7.44**	7.45	10.18	9.51	9.05
δ_O [%]	$1.58{\cdot}10^2$	94.40	**90.43**	$1.06{\cdot}10^2$	$2.34{\cdot}10^2$
$\overline{t}_{sN}$	$3.40{\cdot}10^2$	$7.29{\cdot}10^3$	**1.00**	$2.63{\cdot}10^3$	6.04
$\sigma_{t_{sN}}$	$2.55{\cdot}10^2$	$2.36{\cdot}10^4$	**0.00**	$1.86{\cdot}10^3$	18.15

Table 10.3: Comparison of regression-based machine learning methods on the economics use case.

Measures	GPyTorch	NNetar	RF	SVR	XGBoost
$\overline{e}_{\text{sM}}$ [%]	17.32	**16.49**	19.80	25.08	17.58
$\sigma_{e_{\text{sM}}}$ [%]	**13.33**	16.86	18.27	29.23	19.89
$\overline{e}_{\text{MA}}$	1.04	0.91	1.69	2.20	**0.87**
$\sigma_{e_{\text{MA}}}$	2.06	**2.0**	5.71	8.14	2.16
ρ_U [%]	71.80	52.92	73.72	72.57	70.85
ρ_O [%]	28.20	47.08	26.28	27.43	29.15
δ_U [%]	12.11	**7.85**	13.52	16.65	26.04
δ_O [%]	$\mathbf{1.05}{\cdot}10^2$	$2.84{\cdot}10^2$	$1.51{\cdot}10^2$	$1.39{\cdot}10^2$	$2.01{\cdot}10^2$
$\overline{t}_{sN}$	$3.39{\cdot}10^3$	$1.18{\cdot}10^2$	62.26	13.09	**6.41**
$\sigma_{t_{sN}}$	$3.00{\cdot}10^3$	$1.19{\cdot}10^2$	77.32	39.08	**16.90**

estimating a time series, all methods are more accurate than overestimating a time series.

In terms of the time-to-result, the forecasting methods sNaïve (1.00) and Theta (6.04) are by far the fastest methods. Note that the time-to-result of sNaïve was used to normalize the recorded times and thus, sNaïve has a value of 1.00. The remaining forecasting methods are between 100 and 1000 times slower than sNaïve. For instance, sARIMA, which is the most accurate method concerning $\overline{e}_{\text{MA}}$, is the slowest method and also shows by far the most remarkable variation

for the time-to-result. The fastest machine learning method is XGBoost (6.41) followed by SVR (13.09).

10.2.3 Finance Use Case

The results of the forecasting competition on the finance use case are listed in Table 10.4 showing the classical time series forecasting methods and in Table 10.5 showing the regression-based machine learning methods. Each row shows a measure and each column a method. The most accurate values (the lower, the better) are highlighted in bold. Among the forecasting methods, ETS (17.59%) is based on $\overline{e}_{\text{sM}}$ the most accurate method followed by sARIMA (17.83%). The most accurate machine learning method is XGBoost (19.25%) followed by RF (23.39%). In contrast to all other methods, NNetar ($1.33{\cdot}10^3$) is by far the most inaccurate method and exhibits a standard deviation with a value of $4.14{\cdot}10^4$. However, these high values are introduced by a single time series. According to $\overline{e}_{\text{MA}}$, sARIMA (1.58) is the most accurate forecasting method followed by ETS (1.88). Again, XGBoost is the most accurate machine learning method (2.13) followed by GPyTorch (2.36). Like the economics use case, all forecasting methods are more accurate than the machine learning methods.

Table 10.4: Comparison of classical time series forecasting methods on the finance use case.

Measures	ETS	sARIMA	sNaïve	TBATS	Theta
$\overline{e}_{\text{sM}}$ [%]	**17.59**	17.83	24.40	17.95	18.00
$\sigma_{e_{\text{sM}}}$ [%]	17.96	22.04	24.31	**17.44**	18.14
$\overline{e}_{\text{MA}}$	1.88	**1.58**	2.25	1.94	1.96
$\sigma_{e_{\text{MA}}}$	5.32	**4.07**	5.89	5.29	5.24
ρ_U [%]	66.75	64.83	69.88	67.96	70.28
ρ_O [%]	33.25	35.17	30.10	32.04	29.72
δ_U [%]	12.40	**12.24**	17.22	12.71	13.07
δ_O [%]	18.56	25.32	19.79	**15.48**	20.07
$\bar{t}_{sN}$	$1.69{\cdot}10^2$	$2.35{\cdot}10^6$	**1.00**	$5.35{\cdot}10^3$	8.58
$\sigma_{t_{sN}}$	$2.09{\cdot}10^2$	$1.75{\cdot}10^7$	**0.00**	$6.50{\cdot}10^3$	16.72

In contrast to the economics use case, all methods tend heavily to underestimate a time series. More precisely, the forecast of each method is, on average, at least 64% of the horizon below the actual values. Moreover, all methods are exhibiting almost the same accuracy during under- and overestimating. In terms

Table 10.5: Comparison of regression-based machine learning methods on the finance use case.

Measures	GPyTorch	NNetar	RF	SVR	XGBoost
$\overline{e}_{\text{sM}}$ [%]	31.64	$1.33{\cdot}10^3$	24.94	30.44	**19.25**
$\sigma_{e_{\text{sM}}}$ [%]	32.65	$4.14{\cdot}10^4$	23.39	30.69	**19.46**
$\overline{e}_{\text{MA}}$	2.36	2.39	3.62	4.80	**2.13**
$\sigma_{e_{\text{MA}}}$	**5.80**	6.89	11.64	16.59	5.92
ρ_U [%]	70.48	69.85	70.78	69.99	70.66
ρ_O [%]	29.01	30.15	29.22	30.01	29.34
δ_U [%]	20.37	14.97	17.33	20.42	**13.99**
δ_O [%]	17.89	29.24	17.21	**15.50**	22.06
$\overline{t}_{sN}$	$9.95{\cdot}10^3$	$6.78{\cdot}10^2$	$1.64{\cdot}10^3$	$3.60{\cdot}10^2$	**7.27**
$\sigma_{t_{sN}}$	$1.16{\cdot}10^4$	$8.47{\cdot}10^2$	$3.16{\cdot}10^3$	$6.23{\cdot}10^2$	**17.24**

of $\overline{t}_{sN}$, sNaïve (1.00) and Theta (8.58) are the fastest forecasting methods and XGBoost (7.27) and SVR ($3.60{\cdot}10^2$) are the fastest machine learning methods. Also, in this use case, sARIMA, which is the most accurate method according to $\overline{e}_{\text{MA}}$, is, on average, more than one million times slower than sNaïve.

10.2.4 Human Access Use Case

The results for the forecasting competition on the human access use case are shown in Table 10.6 (classical time series forecasting methods) and Table 10.7 (regression-based machine learning methods). Each column shows a method and each row a measure, where the most accurate values (the lower, the better) are highlighted in bold. The most accurate forecasting method based on $\overline{e}_{\text{sM}}$ is sNaïve (23.60%) followed by sARIMA (27.95%). XGBoost (28.45%) and GPyTorch (29.92%) are the most accurate machine learning methods. In terms of $\overline{e}_{\text{MA}}$, XGBoost (0.55) and GPyTorch (0.65) are the most accurate machine learning methods, while sARIMA (0.50) and sNaïve (0.51) swap places. In contrast to the economic and finance use cases, the forecasting methods do not outperform the machine learning methods concerning both accuracy measures.

Regarding the under- and overestimating, all methods behave differently than in the economic and finance use case. More precisely, all methods show almost no tendency to under- or overestimating the future values (i.e., $\rho_O \sim \rho_U$). However, the methods are, on average, far more accurate during underestimation than overestimation. Regarding $\overline{t}_{sN}$, the fastest forecasting method is sNaïve (1.00) followed by Theta (15.33) and the fastest machine learning

Table 10.6: Comparison of classical time series forecasting methods on the human access use case.

Measures	ETS	sARIMA	sNaïve	TBATS	Theta
$\overline{e}_{\text{sM}}$ [%]	46.64	27.95	**23.60**	42.51	31.20
$\sigma_{e_{\text{sM}}}$ [%]	65.27	54.99	**34.91**	$1.46{\cdot}10^{2}$	89.65
$\overline{e}_{\text{MA}}$	1.42	**0.50**	0.51	0.63	0.65
$\sigma_{e_{\text{MA}}}$	2.86	**0.54**	0.89	1.03	1.58
ρ_U [%]	45.68	48.73	53.61	49.40	50.03
ρ_O [%]	54.32	51.27	45.56	50.60	49.97
δ_U [%]	$2.46{\cdot}10^{2}$	52.79	**17.63**	23.55	21.02
δ_O [%]	$7.73{\cdot}10^{3}$	$2.26{\cdot}10^{3}$	$2.02{\cdot}10^{3}$	$\mathbf{1.24}{\cdot}10^{3}$	$6.31{\cdot}10^{3}$
$\overline{t}_{sN}$	$6.90{\cdot}10^{2}$	$9.05{\cdot}10^{5}$	**1.00**	$1.48{\cdot}10^{4}$	15.33
$\sigma_{t_{sN}}$	$1.11{\cdot}10^{3}$	$4.58{\cdot}10^{6}$	**0.00**	$1.52{\cdot}10^{4}$	25.44

Table 10.7: Comparison of regression-based machine learning methods on the human access use case.

Measures	GPyTorch	NNetar	RF	SVR	XGBoost
$\overline{e}_{\text{sM}}$ [%]	29.92	33.09	75.60	$1.47{\cdot}10^{2}$	**28.45**
$\sigma_{e_{\text{sM}}}$ [%]	50.22	67.99	$6.17{\cdot}10^{2}$	$1.08{\cdot}10^{3}$	**42.73**
$\overline{e}_{\text{MA}}$	0.65	0.66	0.82	1.01	**0.55**
$\sigma_{e_{\text{MA}}}$	1.02	0.91	1.44	1.83	**0.87**
ρ_U [%]	52.69	48.58	43.79	47.38	52.35
ρ_O [%]	47.13	51.42	56.21	52.62	47.65
δ_U [%]	19.85	24.81	**18.32**	23.03	68.87
δ_O [%]	$\mathbf{1.95}{\cdot}10^{3}$	$2.59{\cdot}10^{3}$	$3.82{\cdot}10^{3}$	$3.59{\cdot}10^{3}$	$2.58{\cdot}10^{3}$
$\overline{t}_{sN}$	$1.41{\cdot}10^{4}$	$9.60{\cdot}10^{2}$	$2.40{\cdot}10^{3}$	$1.06{\cdot}10^{3}$	**6.72**
$\sigma_{t_{sN}}$	$1.11{\cdot}10^{4}$	$1.01{\cdot}10^{3}$	$4.90{\cdot}10^{3}$	$2.20{\cdot}10^{3}$	**13.30**

method is XGBoost (6.72) followed by NNetar ($9.60{\cdot}10^{2}$). Also, in this use case, sARIMA is the most accurate method according $\overline{e}_{\text{MA}}$, but exhibits the highest mean value ($9.05{\cdot}10^{5}$) and the highest standard deviation ($4.58{\cdot}10^{6}$) for the time-to-result.

10.2.5 Nature and Demographics Use Case

The results of the last use case, nature and demographics, are shown in Table 10.8 for the classical time series forecasting methods and in Table 10.9 for the regression-based machine learning methods. Each row reflects a measure and each column a method. The most accurate values (the lower, the better) are highlighted in bold. The most accurate forecasting method based on $\overline{e}_{\text{sM}}$ is TBATS (19.85%) followed by Theta (21.79%), while the most accurate machine learning method is GPyTorch (24.00%) followed by RF (26.53%). Among the forecasting methods, sARIMA (0.31) and TBATS (0.36) are the most accurate methods regarding $\overline{e}_{\text{MA}}$. XGBoost (0.47) and GPyTorch (0.48) are the most accurate machine learning methods. Similar to the human access use case, the forecasting methods and machine learning methods exhibit a comparable accuracy for both measures.

Table 10.8: Comparison of classical time series forecasting methods on the nature and demographics use case.

Measures	ETS	sARIMA	sNaïve	TBATS	Theta
$\overline{e}_{\text{sM}}$ [%]	38.54	21.87	26.00	**19.85**	21.79
$\sigma_{e_{\text{sM}}}$ [%]	$1.05{\cdot}10^2$	28.47	39.54	**22.52**	24.41
$\overline{e}_{\text{MA}}$	0.83	**0.31**	0.42	0.36	0.40
$\sigma_{e_{\text{MA}}}$	2.75	**0.28**	0.64	0.53	0.58
ρ_U [%]	44.95	46.18	53.38	51.01	50.39
ρ_O [%]	55.05	53.82	46.39	48.99	49.61
δ_U [%]	15.12	**14.34**	16.51	15.21	14.80
δ_O [%]	$1.60{\cdot}10^2$	51.15	58.83	**33.51**	40.68
$\overline{t}_{sN}$	$5.02{\cdot}10^2$	$1.25{\cdot}10^5$	**1.00**	$7.61{\cdot}10^3$	1.04
$\sigma_{t_{sN}}$	$5.17{\cdot}10^2$	$5.11{\cdot}10^5$	**0.00**	$1.03{\cdot}10^4$	19.31

As in the human access use case, all methods neither tend to under- nor overestimate the future vales (i.e., $\rho_U \sim \rho_O$). However, while underestimating a time series, all methods are more accurate than overestimating a time series. On average, the fastest forecasting methods are sNaïve (1.00) and Theta (1.04) and the fastest machine learning methods are XGBoost (6.54) and SVR ($1.04{\cdot}10^2$). The remaining methods are at least 100 times slower than sNaïve. For example, sARIMA is 125,000 times slower than sNaïve.

Table 10.9: Comparison of regression-based machine learning methods on the nature and demographics use case.

Measures	GPyTorch	NNetar	RF	SVR	XGBoost
$\overline{e}_{\text{sM}}$ [%]	**24.00**	28.79	26.53	31.79	30.11
$\sigma_{e_{\text{sM}}}$ [%]	33.53	45.96	**27.13**	69.59	45.73
$\overline{e}_{\text{MA}}$	0.48	0.52	0.57	0.54	**0.47**
$\sigma_{e_{\text{MA}}}$	0.81	0.75	0.98	1.27	**0.74**
ρ_U [%]	51.83	42.75	48.00	50.82	48.79
ρ_O [%]	48.17	57.25	52.00	49.18	51.21
δ_U [%]	16.02	15.23	**15.04**	16.92	17.84
δ_O [%]	62.92	86.13	78.17	**49.07**	53.36
$\overline{t}_{sN}$	$1.01{\cdot}10^4$	$6.11{\cdot}10^2$	$1.58{\cdot}10^3$	$1.04{\cdot}10^2$	**6.54**
$\sigma_{t_{sN}}$	$1.02{\cdot}10^4$	$8.39{\cdot}10^2$	$4.72{\cdot}10^3$	$4.50{\cdot}10^3$	**14.60**

10.2.6 Overall Evaluation

In this section, we investigate the overall performance of the forecasting methods for choosing the most accurate methods for the competition with our contribution Telescope in Section 10.4. To this end, Table 10.10 shows the performance of the classical time series forecasting methods on all use cases and Table 10.11 the results of the regression-based machine learning methods on all use cases. Each row shows a measure and each column a method. The best values (the lower, the better) are highlighted in bold. Considering all use cases, the most accurate forecasting method based on both accuracy measures is sARIMA (20.63%; 0.73). For the machine learning methods, XGBoost (23.85%; 1.00) is the most accurate method regarding both accuracy measures. To this end, we select sARIMA and XGBoost as representatives of the forecasting methods and the machine learning methods to compete against Telescope, respectively.

By taking all use cases into account, we can investigate the overall tendency of the methods. All methods tend to underestimate the future vales ($\rho_O > 50\%$). However, methods like ETS, NNetar, sARIMA, and TBATS exhibit only a slight difference between ρ_O and ρ_U. In contrast, the remaining methods underestimate, on average, almost 3/5 of the future values. Also, during the underestimation, the methods are, on average, more accurate than overestimating a time series. In terms of the time-to-result, XGBoost (6.73) is the fastest method of the machine learning methods, while sARIMA ($8.48{\cdot}10^5$) is among all methods the slowest. Moreover, all methods (except XGBoost and Theta) are at least 100 times slower than sNaïve.

Table 10.10: Comparison of classical time series forecasting methods on all use cases.

Measures	ETS	sARIMA	sNaïve	TBATS	Theta
$\overline{e}_{\text{sM}}$ [%]	28.95	**20.63**	21.82	23.62	21.10
$\sigma_{e_{\text{sM}}}$ [%]	64.57	35.63	**30.07**	76.00	48.95
$\overline{e}_{\text{MA}}$	1.20	**0.73**	1.02	0.88	0.94
$\sigma_{e_{\text{MA}}}$	3.43	**2.17**	3.27	2.82	2.97
ρ_U [%]	53.22	52.27	61.18	54.52	57.69
ρ_O [%]	46.78	47.73	38.53	45.48	42.31
δ_U [%]	70.44	21.71	15.38	15.25	**14.48**
δ_O [%]	$2.02{\cdot}10^3$	$6.07{\cdot}10^2$	$5.46{\cdot}10^2$	$\mathbf{3.48}{\cdot}10^2$	$1.65{\cdot}10^3$
$\overline{t}_{sN}$	$4.25{\cdot}10^2$	$8.48{\cdot}10^5$	**1.00**	$7.59{\cdot}10^3$	10.10
$\sigma_{t_{sN}}$	$6.65{\cdot}10^2$	$9.08{\cdot}10^6$	**0.00**	$1.08{\cdot}10^4$	20.46

Table 10.11: Comparison of regression-based machine learning methods on all use cases.

Measures	GPyTorch	NNetar	RF	SVR	XGBoost
$\overline{e}_{\text{sM}}$ [%]	25.72	$3.52{\cdot}10^2$	36.72	58.63	**23.85**
$\sigma_{e_{\text{sM}}}$ [%]	35.40	$2.07{\cdot}10^4$	$3.10{\cdot}10^2$	$5.41{\cdot}10^2$	**34.67**
$\overline{e}_{\text{MA}}$	1.13	1.12	1.68	2.14	**1.00**
$\sigma_{e_{\text{MA}}}$	**3.23**	3.71	6.64	9.45	3.27
ρ_U [%]	61.70	53.53	59.07	60.19	60.66
ρ_O [%]	38.13	46.47	40.93	39.81	39.34
δ_U [%]	17.09	**15.71**	16.05	19.26	31.69
δ_O [%]	$\mathbf{5.34}{\cdot}10^2$	$7.48{\cdot}10^2$	$1.02{\cdot}10^3$	$9.47{\cdot}10^2$	$7.13{\cdot}10^2$
$\overline{t}_{sN}$	$9.39{\cdot}10^3$	$5.92{\cdot}10^2$	$1.42{\cdot}10^3$	$6.19{\cdot}10^2$	**6.73**
$\sigma_{t_{sN}}$	$1.03{\cdot}10^4$	$8.39{\cdot}10^2$	$3.85{\cdot}10^3$	$2.56{\cdot}10^3$	**15.59**

10.2.7 Summary of the Results and Threats to Validity

While comparing the forecast error, the machine learning methods exhibit a higher forecast error than the classical forecasting methods. This observation is in line with the high forecast error of pure machine learning methods in the M4-Competition (Makridakis et al., 2018b). However, the machine learning methods are faster than the classical forecasting methods. Considering the individual methods over all four use cases, no method performs best for all use

cases (recall the "No-Free-Lunch Theorem" (Wolpert and Macready, 1997)). For the classical forecasting methods, sARIMA is, on average, the most accurate method, although it is not the best method for any use case. In terms of time-to-result, however, sARIMA is, on average, almost one million times slower than sNaïve. In contrast, the most accurate machine learning method XGBoost is on average 6.73 times slower than sNaïve. Moreover, XGBoost is, for two use cases, the best machine learning method. The baseline method sNaïve achieves only good forecasts when a strong seasonality within a time series is present.

Although we compare the methods on a broad competition comprising a wide range of domains containing 400 different time series, the results may not be generalizable to all time series from all domains. Moreover, we use all methods "out-of-the-box" with their respective default setting. Thus, the individual methods' performance may be different if parameter tuning would have been performed before the forecasting task. We also investigate (i) whether the differences for the observed forecast accuracy are statistically significant and (ii) whether the differences for the measured time-to-result are statistically significant. Consequently, we apply the Friedman test (Friedman, 1937) that is a non-parametric statistical test. More precisely, the test ranks the forecasting methods for each time series separately and compares the methods' average ranks. If there is a tie, average ranks are assigned. Based on this test, we formulate the following hypotheses:

$H_{0,1}$: The methods perform equally regarding the forecast error.

$H_{0,2}$: The methods perform equally regarding the time-to-result.

We conduct both hypotheses with a significance level of 1%. The resulting p-values $p_1 < 2 \cdot 10^{-16}$ and $p_2 < 2 \cdot 10^{-16}$ indicate that both hypotheses can be rejected. Thus, the differences in the exhibited performance of the forecasting methods are statistically significant.

10.3 Evaluation of the Forecasting Method Recommendation

In this section, we investigate and assess the recommendation system, which is integrated in Telescope. First, we describe the subsequent experiments in Section 10.3.1. Then, in Section 10.3.2, we examine the performance of each method that is available for the recommendation. The accuracy of the different recommendation approaches are analyzed in Section 10.3.3. As our recommendation system augments the training set, we investigate the generation of new time series in Section 10.3.4. Finally, we sum up the results and discuss threats to validity in Section 10.3.5.

10.3.1 Experimental Description

To investigate the accuracy of our recommendation system deployed in Telescope, we compare our recommendation approaches based on different feature sets and against selection strategies. Further, we assembled a heterogeneous data set disjointed from the forecasting benchmark data set to have broad and sound experiments. The data set[6] consists of 150 real-world and publicly available time series. The time series are gathered from various sources (Wikipedia Project-Counts, Internet Traffic Archive, `R` packages, Kaggle, Datamarket, etc.) and covering different use cases (e.g., Internet accesses, sales volume). For the recommendation, we split the data set into 100 training and 50 test time series. Moreover, we divide the data set 100 times into unique training and test sets to avoid arbitrary splits. In other words, the recommendation approaches and strategies are trained and evaluated on 100 different splits. Also, we ensure that each time series is at least once in a test split. For the experiments, our recommendation system further augments the training set as described in Section 8.8.4. That is, the time series generator uses the 100 time series in each training split, so that the augmented training set then comprises 10,000 time series. However, before assessing the recommendation of the forecasting methods, we investigate the performance of the regression-based machine learning methods as a reference for the recommendation process. More precisely, we analyze how accurate each of the methods performs on the data set. To this end, we consider the following shares on the 100 splits:

- *Best in split*: "How often the method is the best method in each split"
  In each split, we examined the distribution of the most accurate methods. Then, the method that was most often the most accurate method in the split is credited with this split. In other words, a method with $x\%$ was the most accurate method in $x\%$ of the splits.

- *Lowest in split*: "How often the method has, on average, the lowest forecast accuracy degradation in each split"
  In each split, we calculated the forecast accuracy degradation per time series for each method based on the most accurate method. Then, the method with the lowest average forecast degradation within the split is credited with this split. Consequently, a value of $x\%$ means that the method had in $x\%$ of the splits the lowest average forecast degradation.

- *Total best*: "How often the method is the most accurate method over all time series"

[6] Time series data set is available at `https://zenodo.org/record/3508552`

For each time series in each split, we determine the most accurate method. Then, this method is credited with the respective time series. A value of $x\%$ means that the method was the most accurate method on $x\%$ of all time series.

To investigate the accuracy of our recommendation system deployed in Telescope, we compare our recommendation approaches (see Section 8.8.1.3) based on different feature sets and against selection strategies. Both the approaches and strategies are briefly described in the following:

- The idea of A_C is to use a random forest for mapping the time series characteristics of a given time series to the most accurate regression-based machine learning method.
- A_R trains a random forest for each regression-based machine learning method that estimates the forecast accuracy for a given time series. Then the method with the best-estimated accuracy is selected.
- S^* selects the most accurate method for each time series a-posteriori. Note that this strategy is a theoretical construct since the most accurate method is unknown at the time of the forecast.
- The idea of S_L is based on Lowest split and selects the method that exhibited the best average accuracy per training split.
- S_B bases on Best in split and selects the method that was most often the most accurate method per training split.

The approaches $A_C^\dagger$ and $A_R^\dagger$ work like their counterparts A_C and A_R, but instead of the time series characteristics proposed in this thesis (see Section 8.8.1) the same characteristics as in Section 7.6.1 are used. The methods for selection comprise (i) CART, (ii) Cubist, (iii) Evtree, (iv) NNetar, (v) RF, (vi) SVR, and (vii) XGBoost. A detailed description of each method can be found in Section 3.2.

10.3.2 Performance of the Regression-based Machine Learning Methods

In this section, we analyze the performance of each regression-based machine learning method that can be selected during the recommendation process. To this end, we report the shares (Best in splits, Lowest in splits, and Total best) in Table 10.12 for the training data set and in Table 10.13 for the test set. While the distribution of Total best is almost the same (the only exceptions are Evtree

and SVR) for both sets, the distribution varies considerably on a per split basis, that is, Best in split. More precisely, the methods NNetar (43%) and SVR (57%) dominate the training splits. However, in the test splits, NNetar is on 78% of the splits the most accurate method, while SVR drops to 7%. The remaining splits are governed by Cubist (4%) and Evtree (11%). An even more significant discrepancy between training and test set can be observed for Lowest in splits. While Cubist exhibits the lowest forecasting degradation in 93% of the training splits, it has the lowest value in only 8% of the test splits. Another example is Evtree. This method has the lowest forecasting degradation in no training split, but in 32% of the test splits (the maximum among all methods). Based on these results, we can conclude that the dynamic choice of the most accurate method is a crucial task with considerable potential. Even the choice of a method based on straightforward decision logic (e.g., selecting the method that was, on average, the best method in the training data) can lead to poor forecasts.

Table 10.12: Investigation of the machine learning methods on the training sets.

	CART	Cubist	Evtree	NNetar	RF	SVR	XGBoost
Best in split [%]	0	0	0	43	0	57	0
Lowest in split [%]	7	93	0	0	0	0	0
Total best [%]	10	18	8	27	3	27	7

Table 10.13: Investigation of the machine learning methods on the test sets.

	CART	Cubist	Evtree	NNetar	RF	SVR	XGBoost
Best in split [%]	0	4	11	78	0	7	0
Lowest in split [%]	19	8	32	0	23	5	13
Total best [%]	7	15	18	26	4	18	12

10.3.3 Analysis of the Recommendation Approaches

The results of the comparison between the different recommendation approaches and strategies are listed in Table 10.14. Each row shows a measure, and each column a recommendation approach/strategy. As measures, we consider the forecast accuracy degradation ϑ over all 100 splits and the distribution of how often the approach/strategy selects (i) the best method, (ii) the second-best method, and (iii) the worst method. As S^* is a theoretical construct and acts as

a baseline for the recommendation, the best values of the remaining approaches and strategies are highlighted in bold. More precisely, S^* has a-posteriori knowledge and exhibits, thus the best values. However, this method is impractical in practice, as this strategy selects machine learning methods after the forecast has already been made and can be compared with actual values. That is, the remaining six methods remain in fair competition. The lowest $\overline{\vartheta}$ is shown by A_R (1.084) followed by $A_R^\dagger$ (1.106) and A_C (1.143). In other words, the methods chosen by A_R are, on average, 8.4% less accurate than always selecting the most accurate method. The worst methods regarding the $\overline{\vartheta}$ are $A_R^\dagger$ (1.513) and S_B (1.360). In contrast, $A_R^\dagger$ has the second-best $\tilde{\vartheta}$ (1.0009), while A_C (1.0008) exhibits the best value. In terms of σ_ϑ, A_R exhibits also the lowest value, while S_B, $A_C^\dagger$, and $A_R^\dagger$ are showing a high variation.

Table 10.14: Comparison of the recommendation approaches.

	S^*	S_L	S_B	A_C	A_R	$A_C^\dagger$	$A_R^\dagger$
$\overline{\vartheta}$	1.000	1.173	1.360	1.143	**1.084**	1.513	1.106
$\tilde{\vartheta}$	1.000	1.005	1.003	**1.001**	1.004	1.001	1.002
σ_ϑ	0.000	0.806	3.335	0.848	**0.533**	4.189	1.096
Best method [%]	100.00	14.68	20.64	**31.20**	15.78	29.84	18.72
2^{nd} best method [%]	0.000	19.28	14.76	17.16	17.02	19.98	17.58
Worst method [%]	0.000	**7.08**	28.68	24.50	10.96	23.16	9.70

Besides the statistical measures of ϑ, we investigate the accuracy of the selected methods. For instance, A_C selects in 31.20% of the time series the most accurate method and in 48.36% of the time series the most accurate or second most accurate method, but has at the same time a chance of 24.50% to recommend the worst method. In contrast, S_L selects the best method or the worst method only in 14.68% or 7.08% of the time series, respectively. A_R, which is according the $\overline{\vartheta}$ the most accurate approach and has the lowest variation, recommends only in 15.78% of the time series the most accurate method, in 32.08% of the time series either the best or second-best method, and in 10.96% of the time series the worst method. In fact, no strategy or approach is showing a high chance of selecting the best method and a negligible change for choosing the worst method.

10.3.4 Training Set Augmentation

Typically, machine learning has the inherent limitation of only predicting what has been learned during the training phase. That is, they only perform well if the training and test set have the same characteristics. As our recommendation system is based on machine learning, this restriction also holds for our approach. To counter this, Telescope augments the training set to be as representative as possible. Thus, we investigate in this section the time series generation. More precisely, we compare the time series characteristic distribution within the original training set and in the augmented training set.

Table 10.15: Excerpt of the distribution of the time series characteristics of the original and the augmented training set.

	Length		Standard deviation		Str. of Seasonal Comp.		Serial correlation		Number of peaks	
	Original	Augmented	Original	Augmented	Original	Augmented	Original	Augmented	Original	Augmented
Min.	31	28	0	0	0.17	0.01	0.00	0.00	1	1
Median	476	154	5	5	0.60	0.58	0.19	0.08	1	1
Mean	$3.71{\cdot}10^3$	439	$2.98{\cdot}10^4$	$5.62{\cdot}10^{27}$	0.59	0.59	0.23	0.16	2	3
Max.	$2.92{\cdot}10^4$	$2.92{\cdot}10^4$	$2.23{\cdot}10^6$	$4.49{\cdot}10^{33}$	0.93	1.00	0.96	1.00	12	598

For almost all characteristics, the newly generated time series in the augmented training set extend the spectrum of the data with respect to both the maximum value (80%) and the minimum value (55%). For the sake of clarity, Table 10.15 shows only an excerpt of the time series characteristic distribution. Every two columns show the distribution in the original training set and the augmented training set. And the rows show the minimum, the median, the mean, and the maximum value of the respective time series characteristic. For example, in the original training set, the maximum number of peaks (i.e., the number of strong recurring patterns within a time series) was 12. After generation, there is at least one time series with 598 strong recurring patterns. Another example is the strength of the seasonal component. In the original training set, the seasonal part dominates between 17% and 93% of a time series. In the augmented training set, a time series may contain only a very weak seasonal pattern (1%) or consist entirely of the pattern (100%).

10.3.5 Summary of the Results and Threats to Validity

Comparing the performance of regression-based machine learning methods on the training and test set, we have shown no method is superior to the other methods. This result is in accordance with the "No-Free-Lunch Theorem" and demonstrates that selecting the most suited method for a given method is a crucial task. In terms of the recommendation accuracy, our proposed rec-

ommendation approaches outperform the straightforward strategies. More precisely, the best performance is exhibited by the regression-based recommendation approach. Note that S^* is a theoretical construct using a-posteriori knowledge. Therefore, it acted as a baseline and took not part in the competition. Since every machine learning task depends on its training data, we also investigated the augmentation of the training set and showed that the generation of new time series extends the spectrum for all time series characteristics. In summary, our recommendation approach is able to augment the training set with time series having diverse characteristics and exhibits a high accuracy in selecting the most suitable method for a given time series.

To minimize the risk of an arbitrary selection of the data set, we consider time series that have already been used in textbooks, scientific articles, in competitions, or in R. We also chose regression-based machine learning methods that are easy to understand, generic, and can be used "out-of-the-box". Moreover, we evaluated our recommendation approaches based on our own proposed time series characteristics and on a set of characteristics that have been used in scientific articles (Fulcher et al., 2013; Hyndman et al., 2015; Kang et al., 2017). Additionally, we investigate whether the differences in the observed recommendation performance and the resulting forecast accuracy of the different methods are statistically significant. For this purpose, we use the Friedman test (Friedman, 1937). More precisely, we apply a non-parametric statistical test that ranks (average ranks are assigned for ties) the recommendation approaches separately for each time series. In other words, the test compares the average ranks of the recommendation approaches. Subsequently, we can formulate the null hypothesis for the forecast accuracy as follows:

$$H_0 : \text{The recommendation approaches perform equally.}$$

We conduct the hypothesis with a significance level of 1%. The resulting p-value $p < 2 \cdot 10^{-16}$ indicates that the hypothesis can be rejected, that is, the differences in the performance of the recommendation approaches are statistically significant.

10.4 Benchmarking the Telescope Approach

In this section, we benchmark our Telescope approach against recent hybrid forecasting methods. First, we describe the experiments in Section 10.4.1. Then, we asses the performance of Telescope and the competing methods in Section 10.4.2. A more detailed investigation of the forecasting methods takes place in Section 10.4.3. Beside the forecast performance, we analyze the repeatability

of the forecasting methods in Section 10.4.4. Afterwards, we exchange methods within Telescope to investigate different configurations in Section 10.4.5. Lastly, we summarize the results and discuss threats to validity in Section 10.4.6.

10.4.1 Experimental Description

To benchmark Telescope with recent hybrid forecasting methods and the best performing forecasting well-established forecasting methods, we use the forecasting benchmark (see Chapter 7) to compare these methods. The competing methods comprise (i) BETS, (ii) ES-RNN, (iii) FFORMS, (iv) Hybrid, (v) Prophet, (vi) sARIMA, and (vii) XGBoost. A brief overview of the methods can be found in Section 10.1.1. In this evaluation, we investigate both the pure Telescope approach and Telescope using the recommendation system. For the training of the recommendation system, we use the same data set as described in Section 10.3.1. Note that the benchmark and this training data set consist of different time series. Moreover, we use the time series generator (see Section 8.8.4) to augment the training data set from 150 time series to 10,000. In the following, we refer to Telescope using the recommendation system as *Telescope**.

10.4.2 Forecasting Method Competition

In this section, we compare Telescope over all use cases of the benchmark (i.e., 400 time series) against five recent hybrid forecasting methods and the best method from the field of "classical" time series forecasting and machine learning. As an example, Figure 10.1 shows the competing forecasting methods on the airline passenger time series (see Section 2.1.1). This time series, which is often used as a baseline, shows that all forecasting methods are correctly configured and perform reasonable forecasts. Therefore, we investigate the results (i.e., forecast error and time-to-result) of these methods over all time series in the following.

Table 10.16 shows the performance for all competing methods in competition averaged over all time series of all use cases. Each row represents a measure, each column a method, and the best values (the lower, the better) are highlighted in bold. The most accurate forecasting method based on $\overline{e}_{\text{sM}}$ is Telescope* (19.46%) followed by Telescope (19.95%) and sARIMA (20.63%). The highest error is shown by ES-RNN (47.87%). The lowest error based on $\sigma_{e_{\text{MA}}}$ is exhibited by sARIMA (0.73) closely followed by Telescope* (0.77) and Telescope (0.77). Again ES-RNN has the highest error (24.78). As the errors calculated by sMAPE and MASE showing almost the same ranking, we are

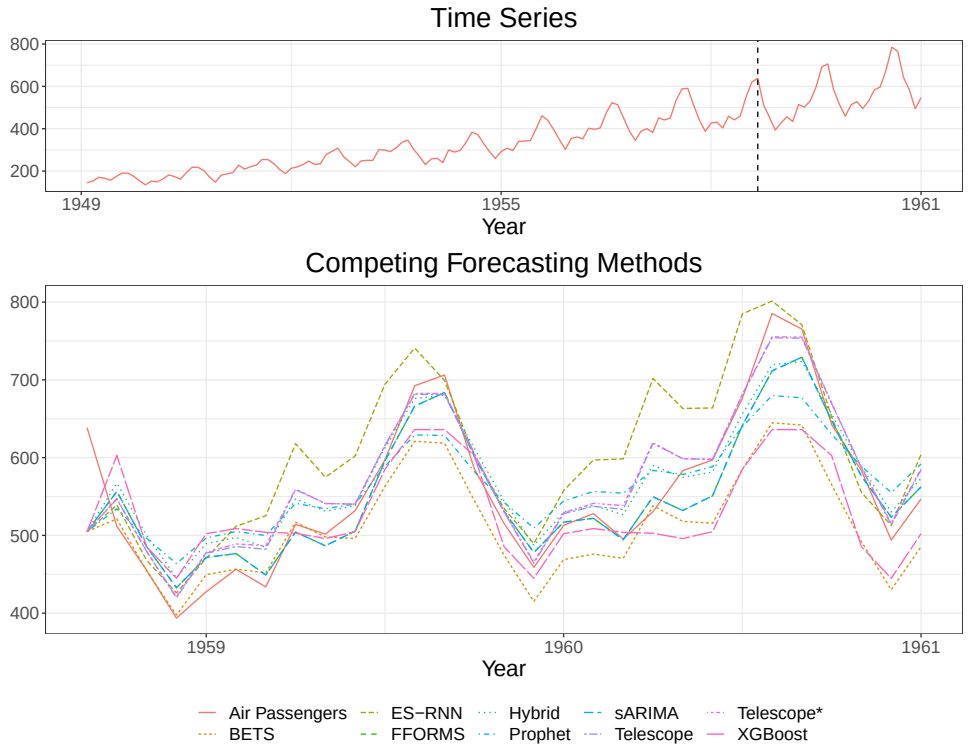

Figure 10.1: Forecasts for all methods in competition on the airline passengers time series.

Table 10.16: Forecast error and time-to-result comparison on all time series.

Measures	BETS	ES-RNN	FFORMS	Hybrid	Prophet	sARIMA	Telescope	Telescope*	XGBoost
$\overline{e}_{\text{sM}}$ [%]	25.52	47.87	21.15	27.89	35.56	20.63	19.95	**19.46**	23.85
$\sigma_{e_{\text{sM}}}$ [%]	34.25	66.99	36.87	89.76	$2.30{\cdot}10^2$	35.63	31.35	**27.96**	34.67
$\overline{e}_{\text{MA}}$	1.16	24.78	0.83	2.83	0.87	**0.73**	0.77	0.77	1.00
$\sigma_{e_{\text{MA}}}$	6.97	$2.17{\cdot}10^2$	2.35	54.08	2.37	**2.17**	2.23	2.23	3.27
ρ_U [%]	52.07	52.37	53.43	53.48	52.11	52.27	48.42	48.40	60.66
ρ_O [%]	47.93	47.62	46.53	46.52	47.88	47.73	51.57	51.60	39.34
δ_U [%]	53.56	21.94	**13.87**	26.54	16.62	21.71	63.73	42.83	31.69
δ_O [%]	$4.97{\cdot}10^2$	$1.50{\cdot}10^3$	$1.67{\cdot}10^3$	$7.49{\cdot}10^2$	$7.62{\cdot}10^2$	$6.07{\cdot}10^2$	$3.35{\cdot}10^2$	$\mathbf{3.00}{\cdot}10^2$	$7.13{\cdot}10^2$
$\overline{t}_{sN}$	$6.14{\cdot}10^4$	$1.61{\cdot}10^6$	$5.23{\cdot}10^5$	$1.09{\cdot}10^6$	$2.04{\cdot}10^3$	$8.48{\cdot}10^5$	$1.43{\cdot}10^2$	$3.21{\cdot}10^3$	**6.73**
$\sigma_{t_{sN}}$	$1.07{\cdot}10^6$	$4.92{\cdot}10^6$	$7.93{\cdot}10^6$	$8.08{\cdot}10^6$	$6.15{\cdot}10^3$	$9.08{\cdot}10^6$	98.67	$1.33{\cdot}10^4$	**15.59**

using $\overline{e}_{\text{sM}}$ in the following sections as the main error measure due to its more intuitive interpretation.

Besides the forecast accuracy, we investigate the tendency of the forecasting method to either under- (ρ_U) or overestimate (ρ_O) the actual values. All

methods except XGBoost are showing almost no tendency ($\rho_U \sim \rho_O$). However, Telescope and Telescope* tend to overestimate ($\rho_O > 50\%$) while the other methods tend to underestimate ($\rho_U > 50\%$). Moreover, all methods are more accurate while underestimating the actual values than while overestimating the actual values. Regarding δ_U, FFORMS (13.87%) is the most accurate method followed by Prophet (16.62%), while Telescope (63.73%) is the worst method. In contrast, Telescope* ($3.00{\cdot}10^2$) is the most accurate method regarding δ_O followed by Telescope ($3.35{\cdot}10^2$) while FFORMS ($1.67{\cdot}10^3$) exhibits the worst value.

As the time-to-result is a crucial requirement in time-critical scenarios, the forecasting methods are also compared based on their time-to-result. Note that the time-to-result per time series was normalized with the time to result of *sNaïve*[7]. By far, the fastest method is XGBoost (6.73). Telescope ($1.43{\cdot}10^2$) has the second-lowest time-to-result while Telescope* is, on average, ten times slower. The slowest method is ES-RNN ($1.61{\cdot}10^6$). Although sARIMA has the third-lowest forecast accuracy, it is, on average, almost 6000 times shower than Telescope and almost 300 times slower than Telescope*. More precisely, the maximum actual forecast time of sARIMA for a time series was 465,574 seconds, which corresponds to almost 5.5 days.

According to RQ 3, our contribution Telescope is designed to produced stable forecasting results. Consequently, we also investigate the variation of the forecast error and the time-to-result as a crucial property. The lowest standard deviation regarding $\overline{e}_{\text{sM}}$ is shown by Telescope* (27.96%) followed by Telescope (31.35%) and BETS (34.25%). In contrast, the highest variation has Prophet ($2.30{\cdot}10^2$). In terms of the time-to-result, XGBoost has the lowest variation (15.59) followed by Telescope (98.67). The highest values for $\sigma_{t_{sN}}$ are exhibited by sARIMA ($9.08{\cdot}10^6$), Hybrid ($8.08{\cdot}10^6$), and FFORMS ($7.93{\cdot}10^6$).

Although the mean and standard deviation are useful statistical measures, we also examine the distributions of the forecast errors based on sMAPE and the time-to-result. Figure 10.2 illustrates the forecast error distribution of all methods. Each distribution is depicted as a box plot, where the horizontal axis shows the different methods and the vertical axis the forecast error in log-scale. The methods FFORMS, sARIMA, Telescope, and Telescope* are showing almost the same distribution (i.e., quite short and similar interquartile ranges) with only a few outliers. Although Hybrid shows a similar distribution between the 25th and 75th quantiles, there are many outliers above the upper whisker. Another group with similar distribution comprises the methods BETS, Prophet, and XGBoost. The method with no outliers above the upper whisker but with

[7] *sNaïve* needs on average 0.01 seconds per forecast.

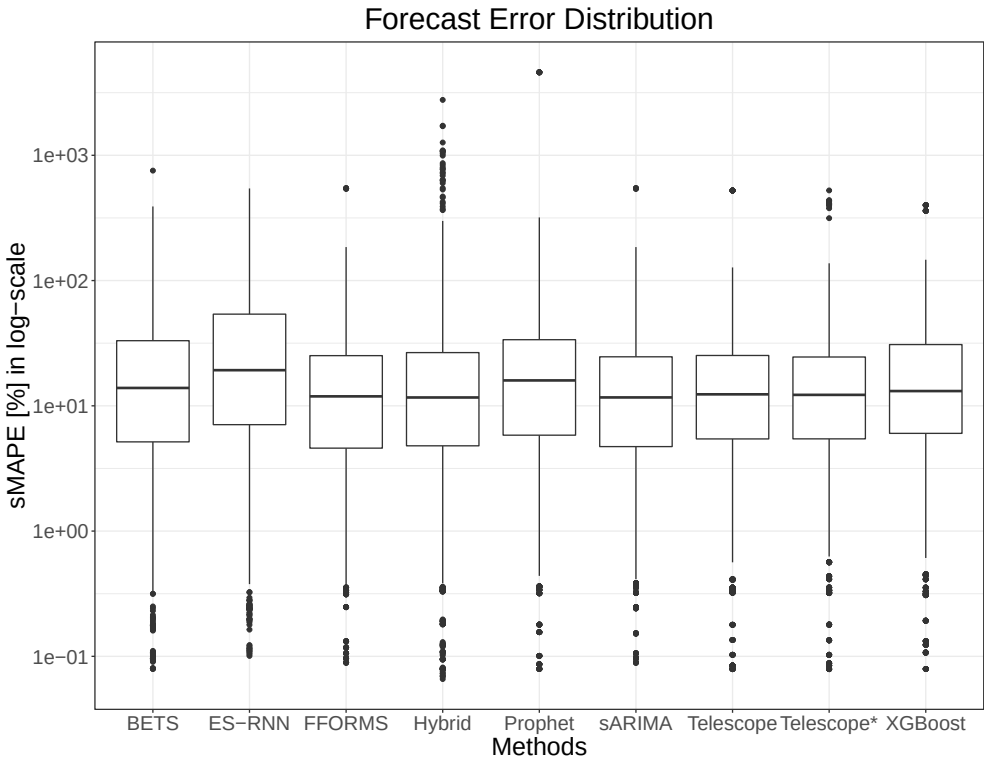

Figure 10.2: Forecast error (sMAPE) distribution.

the longest interquartile range is ES-RNN. However, the fairly similar error distributions are consistent with the "No-Free-Lunch Theorem", which states that there is no forecasting method that works best for all scenarios. Figure 10.3 depicts the time-to-result distribution of all methods. Again, each distribution is illustrated as a box plot, and the vertical axis shows the time-to-result in log-scale. In contrast to the error distribution, the time-to-result distributions are completely different. XGBoost or Telescope exhibit a low variation in the time-to-result. In contrast, FFORMS and sARIMA have a wide range of the time-to-result. Consequently, both methods may be impractical for time-critical scenarios. Telescope* has, in comparison to Telescope, also a huge variation regarding the time-to-result. Thus, Telescope is used in time-critical scenarios and Telescope* in non-time-critical scenarios.

To investigate the trade-off between forecast error and time-to-result, we compare the methods in a 2-dimensional space spanned by the forecast error and time-to-result. We compute the median time-to-result $\tilde{t}_{sN}$ and median

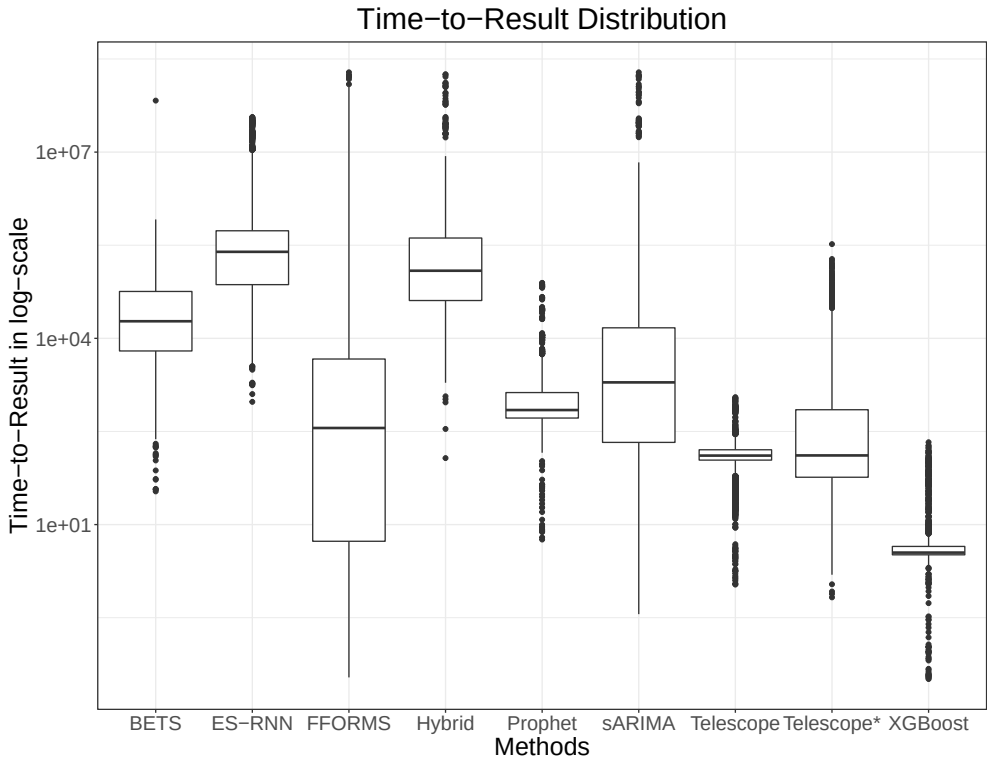

Figure 10.3: Time-to-result distribution.

forecast error $\tilde{e}_{\text{sM}}$ over all methods. Based on these both values, we can sort the forecasts of each method for each time series (t_{sN}, e_{sM}) in one of the four quadrants: (i) $[\tilde{t}_{sN}; \infty[\times [\tilde{e}_{\text{sM}}; \infty[$, (ii) $[0; \tilde{t}_{sN}[\times [\tilde{e}_{\text{sM}}; \infty[$, (iii) $[0; \tilde{t}_{sN}[\times [0; \tilde{e}_{\text{sM}}[$, or (iv) $[\tilde{t}_{sN}; \infty[\times [0; \tilde{e}_{\text{sM}}[$. The best trade-off is achieved in the 3^{rd} quadrant as both the forecast error and time-to-result are lower than their median values. A semi-good performance is reflected by the 2^{nd} and 4^{th} quadrant as either the time-to-result or the forecast error is lower than the associated median value. The worst performance is achieved by forecasts in the 1^{st} quadrant. Here, both the forecast error and time-to-result are worse than their median values. Figure 10.4 depicts each forecast as a point in the 2-dimensional space. The vertical axis represents the forecast error and the horizontal axis the time-to-result. Both axes are in log-scale. The vertical dashed line represents $\tilde{t}_{sN}$ and the horizontal axis $\tilde{e}_{\text{sM}}$. Each forecasting method is depicted in an individual color and point shape. The methods XGBoost, Telescope, and Telescope* have compact clusters in the 2^{nd} and 3^{rd} quadrant. In contrast, the methods BETS,

ES-RNN, and Hybrid have far-reaching clusters in the 1^{st} and 4^{rd} quadrant. However, almost all methods have time series in each quadrant.

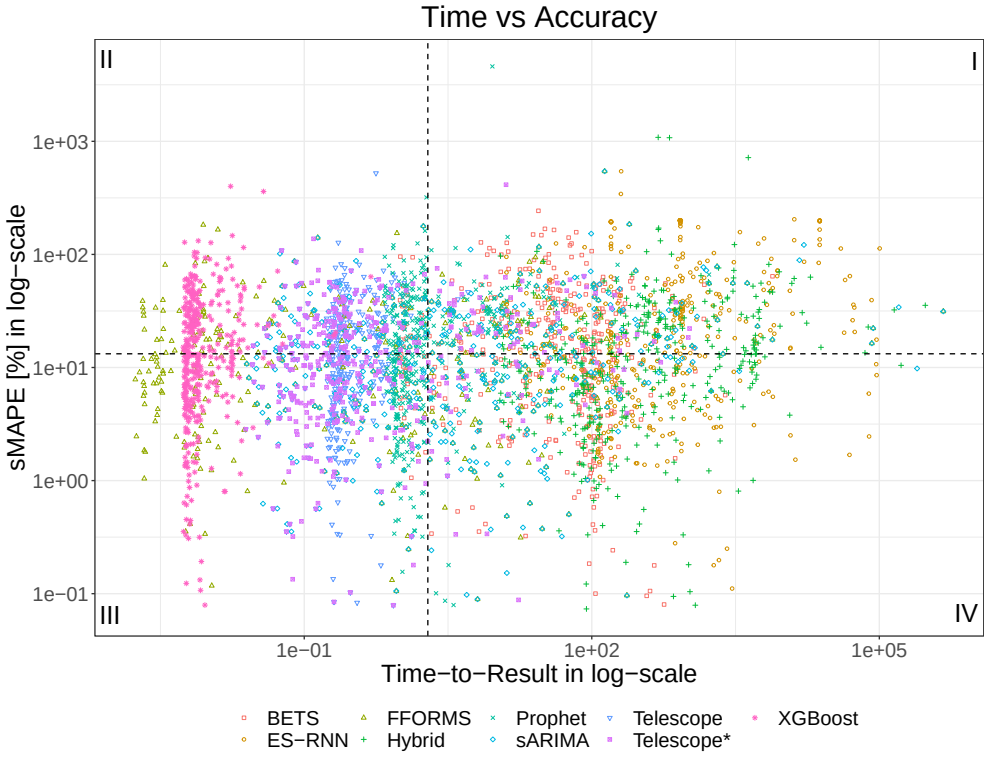

Figure 10.4: Forecast error vs. time-to-result for all methods.

Table 10.17 shows the distribution of the time series for each forecasting method in each quadrant. Each row represents a quadrant and each column a method. For instance, XGBoost has all forecasts equally distributed in the 2^{nd} and 3^{rd} quadrant. That is, all forecasts have a lower time-to-result than $\tilde{t}_{sN}$. In contrast, ES-RNN and Hybrid have all of their forecasts in the 1^{st} and 4^{rd} quadrant. Consequently, all forecasts having a longer time-to-result than $\tilde{t}_{sN}$. Telescope has 98% of its forecast in the 2^{nd} and 3^{rd} quadrant. More precisely, 51% of these forecasts are located in the 3^{rd} quadrant. In other words, Telescope has the highest number of forecasts exhibiting the best trade-off. The second most time series in the 3^{rd} quadrant has XGBoost (50%) followed by Telescope* (42%). In contrast, ES-RNN has 60% of its forecast in the 1^{st} quadrant. That is, for these time series ES-RNN exhibits the worst trade-off.

Table 10.17: Distribution of time series in each quadrant for each forecasting method.

		BETS	ES-RNN	FFORMS	Hybrid	Prophet	sARIMA	Telescope	Telescope*	XGBoost
QI:	$[\tilde{t}_{sN};\infty[\times[\tilde{e}_{\text{sM}};\infty[$	46%	60%	21%	45%	22%	30%	1%	15%	0%
QII:	$[0;\tilde{t}_{sN}[\times[\tilde{e}_{\text{sM}};\infty[$	4%	0%	26%	0%	33%	17%	47%	33%	50%
QIII:	$[0;\tilde{t}_{sN}[\times[0;\tilde{e}_{\text{sM}}[$	5%	0%	34%	0%	36%	22%	51%	42%	50%
QIV:	$[\tilde{t}_{sN};\infty[\times[0;\tilde{e}_{\text{sM}}[$	45%	40%	19%	55%	10%	31%	1%	9%	0%

10.4.3 Detailed Examination

Although our contribution Telescope can handle both seasonal and non-seasonal time series, it is intended for long and seasonal time series. To this end, we analyze the forecasting performance of all methods on seasonal (85% of the data set) and non-seasonal (15%) time series. Table 10.18 lists the results for seasonal and non-seasonal time series. Each row shows a measure and each column a method. The best values (the lower, the better) are highlighted in bold. The lowest forecast error on average for seasonal time series is exhibited by Telescope* (19.77%) followed by Telescope (20.33%) and sARIMA (21.10%). However, Telescope is about 6300 and Telescope* is about 260 times faster than sARIMA. In the case of non-seasonal time series, FFORMS (16.47%) has the lowest forecast error followed by Telescope* (17.65%) and Telescope (17.65%). Note that both Telescope and Telescope* are using the same fallback for non-seasonal time series. Since the fallback, which comprises ARIMA, has a lower error than the sARIMA (17.85%), we are able see the impact of the Preprocessing (see Section 8.2) and Postprocessing (see Section 8.6) phase of Telescope. In both cases, XGBoost exhibits the lowest time-to-result followed by Telescope.

Table 10.18: Forecast error and time-to-result comparison on seasonal and non-seasonal time series.

	Measures	BETS	ES-RNN	FFORMS	Hybrid	Prophet	sARIMA	Telescope	Telescope*	XGBoost
Seasonal	$\bar{e}_{\text{sM}}$ [%]	26.79	52.30	21.94	29.40	38.12	21.10	20.33	**19.77**	24.74
	$\sigma_{e_{\text{sM}}}$ [%]	33.24	71.18	39.11	92.16	$2.49{\cdot}10^2$	37.65	32.83	**28.96**	36.73
	$\bar{t}_{sN}$	$7.14{\cdot}10^4$	$1.63{\cdot}10^6$	$6.11{\cdot}10^5$	$1.26{\cdot}10^6$	$2.22{\cdot}10^3$	$9.92{\cdot}10^5$	$1.56{\cdot}10^2$	$3.74{\cdot}10^3$	**6.90**
	$\sigma_{t_{sN}}$	$1.16{\cdot}10^6$	$4.92{\cdot}10^6$	$8.57{\cdot}10^6$	$8.73{\cdot}10^6$	$6.62{\cdot}10^3$	$9.81{\cdot}10^6$	93.05	$1.44{\cdot}10^4$	**15.71**
Non-Seasonal	$\bar{e}_{\text{sM}}$ [%]	18.00	21.74	**16.47**	18.98	20.53	17.85	17.65	17.65	18.59
	$\sigma_{e_{\text{sM}}}$ [%]	20.12	18.73	18.96	19.60	21.47	20.46	21.04	21.04	**18.16**
	$\bar{t}_{sN}$	$2.53{\cdot}10^3$	$1.54{\cdot}10^6$	$1.74{\cdot}10^2$	$9.55{\cdot}10^4$	$9.26{\cdot}10^2$	84.73	65.04	65.04	**5.75**
	$\sigma_{t_{sN}}$	$4.42{\cdot}10^3$	$4.90{\cdot}10^6$	$8.20{\cdot}10^2$	$2.07{\cdot}10^5$	$1.07{\cdot}10^3$	$1.25{\cdot}10^2$	95.05	95.05	**14.83**

In addition to the performance of the forecasting methods for seasonal and non-seasonal time series, we examine the strengths and disadvantages of the forecasting methods more precisely. To this end, we split the time series into four classes: (i) short time series (< 2000) with a short period (< 96), (ii)

short time series with a long period (≥ 96), (iii) long time series (≥ 2000) with a short period, and (iv) long time series with a long period, containing 270, 34, 14 and 82 time series, respectively. Table 10.19 compares the different forecasting methods based on this classes. Each row shows a measure and each column a method. The best values (the lower, the better) are highlighted in bold. For all classes, XGBoost is the fastest method, on average, followed by Telescope. In 3 of 4 classes, Telescope* is the third-fastest method. The lowest error in the first class exhibits Telescope* (16.66%) followed by sARIMA (16.68%) and Telescope (16.75%). In the second class, Telescope* has the lowest error (19.20%) followed by Telescope (19.72%) and Hybrid (21.59%). In the third class, Telescope* is again the most accurate method (19.41%) followed by Telescope (19.71%) and FFORMS (21.78%). In the last class, Telescope* shows the lowest error (28.82%) followed by Telescope (30.59%) and sARIMA (31.79%). In summary, our approach provides good forecasts for all classes while having a low time-to-result compared to the next best method.

Table 10.19: Forecast error and time-to-result comparison on different time series classes.

		BETS	ES-RNN	FFORMS	Hybrid	Prophet	sARIMA	Telescope	Telescope*	XGB
(short ts; short p)	$\sigma_{e_{sM}}$ [%]	16.92	29.12	17.11	25.73	21.03	16.68	16.75	**16.66**	19.12
	$\bar{t}_{sN}$	$7.47{\cdot}10^4$	$4.49{\cdot}10^5$	$3.66{\cdot}10^3$	$1.03{\cdot}10^5$	$6.62{\cdot}10^2$	$7.28{\cdot}10^3$	$1.15{\cdot}10^2$	$5.73{\cdot}10^2$	**6.37**
(short ts; long p)	$\sigma_{e_{sM}}$ [%]	32.91	51.06	25.43	21.59	25.71	24.57	19.76	**19.20**	22.81
	$\bar{t}_{sN}$	$4.83{\cdot}10^3$	$1.84{\cdot}10^6$	$1.24{\cdot}10^4$	$2.63{\cdot}10^5$	$9.40{\cdot}10^2$	$2.34{\cdot}10^4$	$1.37{\cdot}10^2$	$7.18{\cdot}10^2$	**6.48**
(long ts; short p)	$\sigma_{e_{sM}}$ [%]	23.09	66.23	21.78	26.39	36.31	21.84	19.71	**19.41**	26.35
	$\bar{t}_{sN}$	$2.11{\cdot}10^5$	$1.00{\cdot}10^7$	$6.76{\cdot}10^3$	$7.68{\cdot}10^5$	$7.40{\cdot}10^3$	$5.80{\cdot}10^3$	$1.84{\cdot}10^2$	$2.13{\cdot}10^3$	**6.64**
(long ts; long p)	$\sigma_{e_{sM}}$ [%]	51.20	$1.05{\cdot}10^2$	32.56	37.90	87.42	31.79	30.59	**28.82**	39.40
	$\bar{t}_{sN}$	$1.57{\cdot}10^4$	$3.91{\cdot}10^6$	$2.53{\cdot}10^6$	$4.76{\cdot}10^6$	$6.10{\cdot}10^3$	$4.10{\cdot}10^6$	$2.28{\cdot}10^2$	$1.31{\cdot}10^4$	**8.07**

10.4.4 Repeatability

Besides the accuracy and time-to-result of the forecast, we investigate how stable forecasts are performed. To this end, we calculate for each method and for each time series, the standard deviation of the forecast error based on sMAPE over the ten repetitions. Table 10.20 shows for each method the average values for the variation of the forecast error. Prophet and XGBoost have an average standard deviation of 0%, indicating that both methods performed the same forecasts for each of the ten repetitions. FFORMS, sARIMA, and Telescope exhibit, on average, a forecast error standard deviation of 0.01%. This negligible variation is introduced by sARIMA as it is used in Telescope as a building block and as a possible method in FFORMS. Telescope* has a variation of 0.46% that occurs mainly while NNetar is selected as machine learning method. Similarly,

the high value shown by Hybrid can also be explained by the usage of NNetar in the ensemble forecast.

Table 10.20: Average standard deviation in % of the sMAPE per time series within the 10 repetitions.

BETS	ES-RNN	FFORMS	Hybrid	Prophet	sARIMA	Telescope	Telescope*	XGB
1.35	0.92	0.01	14.40	0.00	0.01	0.01	0.46	0.00

10.4.5 Investigation of Alternative Building Blocks

In this section, we investigate the forecast performance of Telescope while using different building blocks to ensure that our approach reflects the best possible configuration. More precisely, we changed (i) the forecasting method for the trend (ARIMA), (ii) we switched the regression-based machine learning method (XGBoost), and (iii) we deactivated the Box-Cox transformation. We consider ETS and Theta for the trend forecast. For the machine learning method, we take all base-level methods (CART, Cubist, Evtree, NNetar, RF, SVR) into account. Details of the methods can be found in Section 3. The results of the different versions of Telescope are listed in Table 10.21. Each row shows a measure and each column a configuration. The best values (the lower, the better) are highlighted in bold. The most accurate forecast is exhibited by Telescope using RF (19.69%) followed by Telescope using ETS (19.83%) and Telescope (19.95%). However, the version with RF is 6.6 times slower than the original version. The second most accurate version is almost two times slower than the original and shows a higher variance in the time-to-result. Moreover, deactivating the Box-Cox transformation decreases the forecast accuracy by far and also increases the time-to-result. Considering these results, we can conclude that the configuration we have chosen shows the best trade-off and thus overcomes the other methods.

Table 10.21: Testing different building blocks of Telescope.

Measures	Telescope	w/ ETS	w/ Theta	w/ CART	w/ Cubist	w/ Evtree	w/ NNetar	w/ RF	w/ SVR	w/o Box-Cox
$\bar{e}_{\text{sM}}$ [%]	19.95	19.83	27.81	21.01	20.00	20.31	27.25	**19.69**	19.99	35.01
$\sigma_{e_{\text{sM}}}$ [%]	31.35	31.18	$1.86{\cdot}10^2$	40.88	29.96	36.87	$1.25{\cdot}10^2$	**28.22**	31.97	$2.40{\cdot}10^2$
$\bar{t}_{sN}$	$1.43{\cdot}10^2$	$2.11{\cdot}10^2$	**1.16**$\cdot 10^2$	$1.77{\cdot}10^2$	$1.94{\cdot}10^2$	$3.02{\cdot}10^4$	$1.09{\cdot}10^4$	$9.50{\cdot}10^2$	$2.43{\cdot}10^3$	$8.80{\cdot}10^2$
$\sigma_{t_{sN}}$	**98.67**	$1.89{\cdot}10^2$	$1.23{\cdot}10^2$	$1.12{\cdot}10^2$	$1.17{\cdot}10^2$	$2.49{\cdot}10^4$	$5.52{\cdot}10^3$	$6.55{\cdot}10^2$	$1.95{\cdot}10^3$	$2.01{\cdot}10^2$

10.4.6 Summary of the Results and Threats to Validity

Benchmarking our Telescope approach against recent hybrid forecasting methods and the best methods from Section 10.2, we showed that both Telescope and Telescope* outperforms the state-of-the-art. More precisely, Telescope* achieves the best forecast accuracy based on the sMAPE followed by Telescope. Regarding the forecast error based on MASE, both Telescope versions are the second most accurate method. Although our approach is intended for seasonal time series, it exhibits the second-best forecast accuracy on non-seasonal time series. Moreover, Telescope is, on average, up to 6000 times faster than the third most accurate method. In all experiments, Telescope* is more accurate than Telescope, but has, on average, a higher time-to-result as well as variation in the time-to-result. Also, we show that the chosen configuration of Telescope has the best trade-off between forecast accuracy and time-to-result. The third most accurate method, sARIMA, suffers from a high variation in the time-to-result. The winner of the M4-Competition, ES-RNN, is tailored for cross-learning to the M4-Competition data and therefore has the highest average forecast error. In summary, Telescope exhibits the best forecast accuracy coupled with a low and reliable time-to-result.

Although our data set comprises a broad spectrum with 400 different time series, the evaluation results may not be generalized to all time series from all areas. Besides the data set, we also try to have a sound set of recent hybrid forecasting methods based on different techniques. To this end, we also consider methods developed by Facebook and Uber. However, we use all methods with their default settings. Consequently, the observed results may differ if the forecasting methods are tuned to each time series. Moreover, our classification of the time series into long time series and time series with long periods may also affect the results. As Telescope achieves on the whole data set the best performance, the ranking inside the classes may only change. Lastly, we analyze whether the observed forecast accuracy as well as the measured time-to-result are statistically significant. To this end, we apply a non-parametric statistical test. More formally, we use the Friedman test (Friedman, 1937), which ranks the forecasting methods separately for each time series and compare the average ranks of the methods. In case of a tie, average ranks are assigned. Thus, we formulate the following hypotheses:

$$H_{0,1} : \text{The methods perform equally regarding the forecast error.}$$
$$H_{0,2} : \text{The methods perform equally regarding the time-to-result.}$$

We conduct both hypotheses with a significance level of 1%. The resulting p-values $p_1 < 2 \cdot 10^{-16}$ and $p_2 < 2 \cdot 10^{-16}$ indicate that both hypotheses can be

rejected. Thus, the differences in the exhibited performance of the forecasting methods are statistically significant.

10.5 Concluding Remarks

In this chapter, we benchmarked our contribution Telescope against state-of-the-art methods from the fields of (i) "classical" forecasting methods, (ii) regression-based machine learning methods, and (iii) recent hybrid forecasting methods. For this benchmarking process, we performed 72,000 forecasts consuming 13,560 (instance) hours with 15 state-of-the-art methods. In the first benchmarking experiments, we observed that the machine learning methods exhibit a higher forecast error than classical forecasting methods. Moreover, no forecasting method outperforms the other methods for all use cases. Both observations are in accordance with previous articles (Wolpert and Macready, 1997; Makridakis et al., 2018b). Then, we evaluated the recommendation system integrated into Telescope to investigate RQ 5 "*What are appropriate strategies to dynamically apply the most accurate method within the hybrid forecasting approach for a given time series?*". Here, the results underline the challenge of selecting the most appropriate method for a given time series. Nevertheless, the recommendation system outperforms the straightforward selection strategies. In the last experiments, we benchmarked Telescope against the best methods from the first benchmarking round and recent hybrid forecasting methods. On average, our approach achieves a higher forecast accuracy than the state-of-the-art, has the second-lowest variance in the time-to-result, and is, on average, 6000 times faster than the next most accurate method. Moreover, Telescope* is more accurate in all experiments than Telescope, but is on average slower and has a higher variation in the time-to-result. These results show that the design of Telescope (i.e., combining different methods) lead to accurate and reliable forecasts (RQ 3 "*How to design an automated and generic hybrid forecasting approach that combines different forecasting methods to compensate for the disadvantages of each technique?*"). To investigate RQ 4 "*How to automatically extract and transform features of the considered time series to increase the forecast accuracy?*", we analyzed the forecasting methods for different subsets of the time series. For non-seasonal time series, Telescope's fallback (i.e., applying Box-Cox transformation and ARIMA) exhibits the second-best accuracy and is more accurate than sARIMA. For seasonal time series, Telescope (i.e., extracting features, applying Box-Cox transformation, and employing XGBoost) has the best accuracy and is more accurate than XGBoost. When investigating different building blocks of our contribution, Telescope without using the Box-Cox transformation is less accu-

rate than the original version. Based on these observations, we can conclude that the automatic transformation and extraction of intrinsic time series features enhance the explanation of the time series.

Chapter 11

Elastic Resource Management

In this chapter[1], we evaluate and benchmark our contribution Chamulteon and its components. First, we introduce the deployed workloads, applications, environments, auto-scaler, and applied measures for the following experiments in Section 11.1. Then, we investigate the benefit of using service demand as scaling indicator for auto-scaling in Section 11.2. As Chamulteon bases on the preliminary contribution Chameleon[2], we benchmark this mechanism in Section 11.3. Afterwards, we evaluate the cost-awareness component Fox in Section 11.4. In Section 11.5, we benchmark our contribution Chamulteon. Finally, we conclude this chapter in Section 11.6.

11.1 Global Experimental Setup

This section provides the information about the experimental setup, which are used in the subsequent evaluation sections. Note that each subsequent evaluation section has a separate experimental description for specific information. In Section 11.1.1, the workloads for stressing the applications are described. Then, the deployed applications are introduced (see Section 11.1.2) and the deployment environment (see Section 11.1.3) in which the applications are hosted. Afterwards, the competing auto-scalers are presented in Section 11.1.4. Finally, the applied measures for the comparison of the auto-scalers are highlighted in Section 11.1.5.

11.1.1 Workload Description

To benchmark our contributions in representative experiments, authentic workloads with time-varying load intensity profiles are required. For this purpose,

[1] This chapter is based on our previous works (Bauer et al., 2018a,b; Lesch et al., 2018; Bauer et al., 2019b).

[2] Note that the experiments of Chameleon were done in collaboration with Nikolas Herbst.

we gathered existing traces from real-world systems that show different behavior and can last up to several months. For a practical experiment run duration of a few hours, we randomly selected subsets (one to three days) from the real-world traces. Further, we accelerated the playback of these subsets so that one day corresponds to one to 3.2 hours of experiment time. In other words, the resource demand changes during the experiments in the order of minutes. By doing so, we have covered balanced time intervals for a realistic setup for the auto-scaling mechanisms. In our opinion, a higher time acceleration factor would make the experiments unrealistic since, for example, the increased changes in the demand could exceed the provision rate of the underlying hardware. In the following, the real-world traces applied in the following experiments are depicted in Figure 11.1 and are listed below:

- The *BibSonomy* trace contains HTTP requests to servers of the social bookmarking system BibSonomy (Benz et al., 2010) during April 2017.
- The *FIFA*[3] trace, which is widely known and analyzed (Arlitt and Jin, 2000), contains HTTP requests to the FIFA servers during the FIFA World Cup between April and June 1998.
- The *IBM* trace contains transactions on a z10 mainframe CICS (Customer Information Control System) installation during a month.
- The *Retailrocket*[4] trace contains HTTP requests that were sent to servers of an anonymous real-world e-commerce website in June 2015.
- The *Wiki*[5] trace contains the page requests to all German Wikipedia projects during December 2013.

11.1.2 Deployed Applications

Besides the authentic workloads, we benchmarked our contributions to different applications. More precisely, we designed different services for the experiments, which are based on existing applications. Each applications is written in Java and was set up on either a JBoss WidlFly[6] or Apache Tomcat[7] applications server. In the following, the deployed services are listed:

[3]FIFA Source: `http://ita.ee.lbl.gov/html/contrib/WorldCup.html`
[4]Retailrocket Source: `https://www.kaggle.com/retailrocket/ecommerce-dataset`
[5]Wikipedia Source: `https://dumps.wikimedia.org/other/pagecounts-raw/2013/`
[6]Wildfly: `https://www.wildfly.org/`
[7]Tomcat: `http://tomcat.apache.org/`

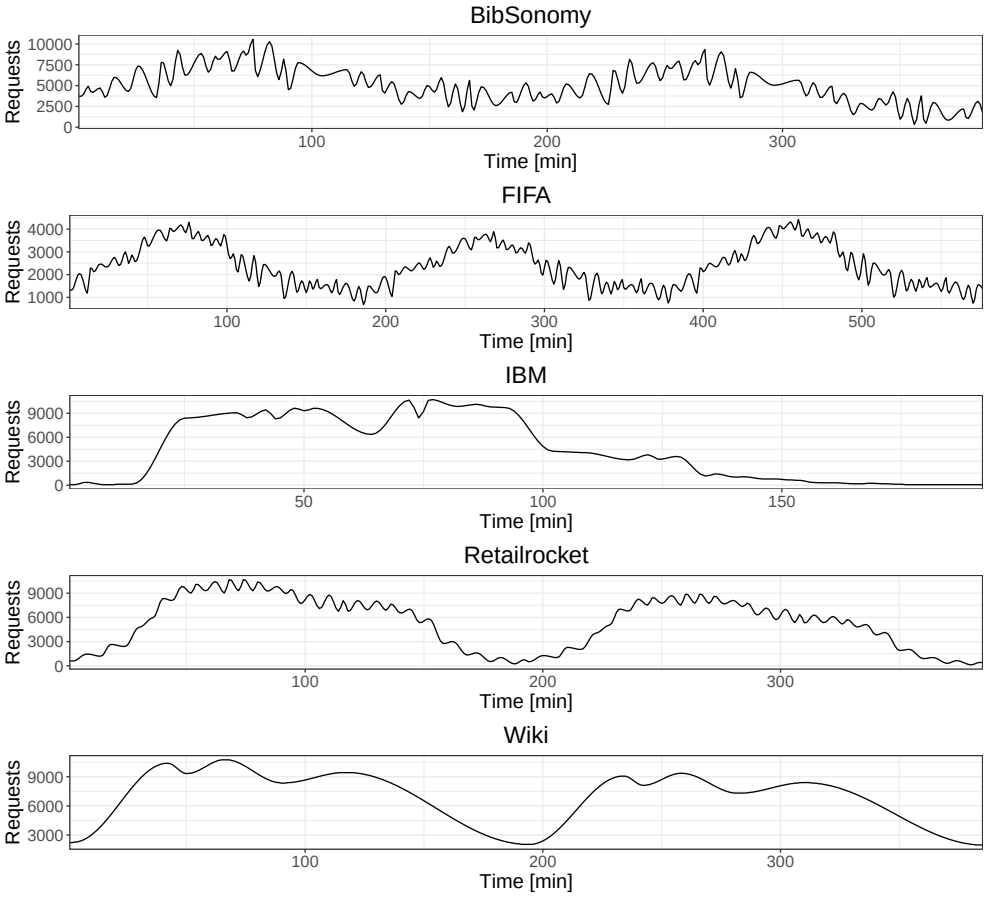

Figure 11.1: Overview of the real-world traces.

- The *Content* application simulates a content-delivery service provider with limited capacities. More specifically, an incoming HTTP request attempts to read a file from a limited pool of randomly generated files. Each file can only be read by one request at a time and is locked as soon as it is read. That is, an incoming HTTP request either reads a file successfully or waits until a file is unlocked.

- The *LU* application is a re-implementation of the LU worklet from SPEC's Server Efficiency Rating Tool SERT™2. The application calculates the lower-upper matrix decomposition (Bunch and Hopcroft, 1974) of random generated $n \times n$ matrix, where the incoming HTTP request defines n. In the experiments, we used a fixed n.

- The *Mixed* application comprises the hardware contention of the LU application and software contention of the Content application. More precisely, an incoming HTTP request first calculates the lower-upper matrix decomposition and then tries to read a file from the limited pool.
- The *Verification* application represents a lightweight micro-service application comprising three different services: (i) *UI* service, (ii) *Validation* service, and (iii) *Data* service. The UI forwards each request to the validation service to check its validity. Then, the request is redirected to the data service. This last service provides the requested data and sends the response to the UI for rendering the content.

For estimating the service demands[8] (see Section 4.1.2) of each of these applications, we collected the average CPU utilization and the throughput of each workload class per instance and applied LibReDE (Spinner et al., 2014). The estimated service demands for the applications and services are listed in Table 11.1.

Table 11.1: Estimated service demand of each deployed application.

	Content	**Data**	**LU**	**Mixed**	**UI**	**Validation**
Service demand [s]	0.15	0.04	0.1	0.25	0.059	0.1

11.1.3 Deployment Description

In addition to the authentic workloads and different applications, we also used different deployments for the benchmarking of our contributions. More precisely, the LU application is hosted on three different platforms; the Verification application is either deployed on VMs or Docker containers; the Verification and LU applications are run on different instance configurations. The considered platforms comprise

- A CloudStack-based private cloud (*CSPC*),
- Amazon Web Services EC2 (*AWS EC2*), and
- the DAS-4 (Distributed ASCI Supercomputer 4) IaaS cloud of a medium-scale multi-cluster experimental environment (*MMEE*) (Bal et al., 2016).

[8]The service demand captures the average time required from each service for processing a request, excluding any waiting times.

We used the CSPC for experiments in a controlled environment. In contrast, the AWS EC2 and the MMEE environment were considered for scenarios with background noise. In the CSPC environment, the applications were deployed in a private Apache CloudStack[9] cloud. More precisely, CloudStack manages eight identical virtualized Xen-Server (v6.5) hosts (HP DL160 Gen9 with eight physical cores @2.4Ghz (Intel E5-2630v3)). We deactivated hyper-threading to limit VM overbooking and rely on a constantly stable performance per VM. Dynamic frequency scaling was enabled as default, and further CPU-oriented features were not changed. The hosts have each 2×16GB RAM (DIMM DDR4 RAM operated @ 1866 MHz) deployed. The specifications of the VMs are listed in Table 11.2.

Table 11.2: Specifications of the VMs.

	c.small	**c.medium**	**c.large**	**m4.large**	**d.small**
Platform	CSPC	CSPC	CSPC	AWS EC2	MMEE
Operating System	Ubuntu 16.06	CentOS 6.5	Debain 4.9	CentOS 6.5	Debian 8
vCPU	1 core	2 cores	2 Cores	2 cores	2 cores
Memory	2GB	4GB	8GB	8GB	2GB
Application Server	Tomcat 7	WildFly 10	Tomcat 8.5	WildFly 10	WildFly 10

Note that for all experiments, the load driver, the experiment controller, and the auto-scaling mechanism were not part of the system-under-test and, thus, were located outside the three platforms. Moreover, we initialized all VMs before the experiments to avoid measurement perturbations due to VM image copying. At the beginning of each experiment, only a certain amount of instances were running while the remaining instances were shut-down.

11.1.4 Deployed Auto-Scaling Mechanisms

During the experiments, our contributions competed against state-of-the-art open-source auto-scalers. Each auto-scaler is described in detail in Chapter 6 and briefly below:

- Adapt (Ali-Eldin et al., 2012) is a hybrid auto-scaler based on control theory. The main idea is to have two proactive controllers for scaling down and one reactive scaling approach. The auto-scaler considers mainly the changes in the workload for scaling decisions.

[9] Apache CloudStack: `https://cloudstack.apache.org/`

- ConPaaS (Fernandez et al., 2014) is a proactive auto-scaler based on time series analysis. To have a reliable forecast, the auto-scaler performs different forecasting methods and selects the method that exhibited the best accuracy in the last scaling interval.
- Hist (Urgaonkar et al., 2008) is a hybrid auto-scaler based on queueing theory. To scale applications proactively, the load is forecast based on histograms. In contrast, reactive scaling takes place to correct inaccurate decision based on the forecast.
- React (Chieu et al., 2009) is a reactive auto-scaler based on threshold-based rules. Up-Scaling takes only place if all service instances exceed the threshold. In contrast, down-scaling takes place if at least one service instance is below the threshold.
- Reg (Iqbal et al., 2011) is a hybrid auto-scaler based on time series analysis. The up-scaling reactively takes place, while the down-scaling is done reactively. The down-scaling works similarly to React, and the proactive up-scaling bases on polynomial regression.

Besides the state-of-the-art auto-scalers, we implemented a mechanism based on the AWS EC2 auto-scaler using CPU utilization as a scaling indicator. As thresholds, we used 80% CPU utilization for scaling up and 60% for scaling down while adding/removing the fixed amount of one service instance. We refer to this mechanism as *T-Hold*.

Each auto-scaler was called at fixed intervals, received a set of input values, and then returned the number of service instances that had to be removed or added. The state-of-the-art auto-scalers got (i) the accumulated number of requests during the last interval, (ii) the estimated service demand per request determined by LibReDE as used in Chamulteon, and (iii) the number of currently running service instances as input. T-Hold got (i) the current CPU utilization and (ii) the number of currently running service instances as input. For the auto-scalers Adapt, ConPaaS, Hist, React, and Reg, we used the implementations[10] provided by A. Ali-Eldin with the same configurations as used in their simulative evaluation (Papadopoulos et al., 2016).

As popular auto-scalers, which can scale applications comprising multiple services, such as AutoMap, AGILE, and CloudScale (Shen et al., 2011; Nguyen et al., 2013; Beltrán, 2015), are closed-source, we extended the state-of-the-art open-source auto-scalers to enable scaling of such applications. To this end,

[10]Competing auto-scalers: `https://github.com/ahmedaley/Autoscalers` (Papadopoulos et al., 2016)

we deployed for each service an instance of the associated auto-scaler. We also modified the number of arrivals for each service. In other words, the first service receives the current observed request rate as input. As input for the following instances, the number of requests is calculated using the following formula

$$\mathrm{r}(i) := \begin{cases} \lambda & \text{if } i = 1, \\ \min(n(i-1) \cdot s(i-1), r(i-1)) & \text{if } i > 1, \end{cases} \tag{11.1}$$

where λ is the measured arrival rate at the entry-service of the application, $r(i)$ the request rate at the i-th service, $n(i)$ the number of instances of the i-th service, and $s(i)$ the service rate[11] at the i-th service. In other words, the request rate for the i-th service is set as the minimum of the number of service instances of the $(i$-1$)$-th service multiplied by the service rate per instance and the request rate for the $(i$-1$)$-th service. Simply put, if the $(i$-1$)$-th service's capacity is exceeded, the maximum request rate that this service can process is passed on to the i-th service. Otherwise, the incoming request rate is forwarded to the i-th service.

11.1.5 Applied User and System Measures

To compare and quantify the performance of different auto-scalers, we used a combination of system- and user-oriented measures. Table 11.3 lists a brief description of each measure. The system-oriented measures consist of elasticity measures (see Section 4.2 and 9.5). As user-oriented metrics, we report the number of average adaptations, the number of average services instances, the cost-saving rate, percentage of SLO (service level objective) violations, and user satisfaction. We reflect the user satisfaction with the Application Performance Index (*Apdex*), which is an open standard measure developed by a consortium of companies that measures user satisfaction on a uniform scale from 0% to 100% (Sevcik, 2005). The best score of 100% is achieved when all requests are served within the agreed response time (SLO). In addition to SLO violations, which reflect whether a request is served within the predefined time, this metric can be used to gain additional insight into how bad the violations are from the users' perspective. Mathematically, the Apdex can be calculated as

$$Apdex[\%] := \frac{\nu + 0.5 \cdot \kappa}{\Omega}, \tag{11.2}$$

[11] The service rate is the inverse of the service demand.

where ν is the number of satisfied requests, that is, requests within the SLO, κ the number of tolerating requests, that is, requests that exceed the SLO within a toleration interval, and Ω the number of total sent requests.

Table 11.3: Overview of the applied measures.

Name	Description
θ_U, θ_O	Provisioning accuracy as the relative amount of service instances that are under- or overprovisioned normalized by time.
τ_U, τ_O	Wrong provisioning time share as the relative amount of time in under- or overprovisioned state.
υ	Instability as the relative amount of time in which demand and supply are not parallel.
ϵ	The elastic speedup score compared to when no auto-scaling takes place.
σ	Auto-scaling deviation from the theoretically optimal auto-scaler.
ς	Auto-scaling worst-case deviation from the theoretically optimal auto-scaler.
SLOs	The relative amount of SLO violations.
Apdex	The user satisfaction.
#Adaptations	The total number of scaling adaptations.
Avg. #Inst.	The average number of concurrently running service instances during the experiment.
Π_a	Accounted cost-saving rate compared to using all service instances throughout the experiment.
Π_a	Charged cost-saving rate compared to using all service instances throughout the experiment.

11.2 The Impact of Service Demand Estimation

In this section, we investigate the benefits of using service demand and CPU utilization as scaling indicator. First, we describe the experiment in Section 11.2.1. Then, we compare an auto-scaling approach based on service demand and an approach based on CPU-utilization on a hardware contention scenario (see Section 11.2.2), on a software contention scenario (see Section 11.2.3), and on a scenario that exhibits both hardware as well as software contention (see

Section 11.2.4). Finally, we sum up the results and discuss threats to validity in Section 11.2.5.

11.2.1 Experimental Description

To investigate the value of service demand estimation for auto-scaling, we compare two approaches with identical underlying decision logic under time-varying load, namely the Retailrocket (see Section 11.1.2) trace. The scaling logic is based on simple threshold-based mechanisms, such as implemented on AWS EC2. The first approach gets the measured average CPU utilization, and the second approach gets the average system utilization based on queueing theory (see Section 4.1) as input. The reason for choosing CPU utilization as scaling indicator are: (i) In many cases, the bottleneck resource may not be known at configuration time, (ii) using IO metrics requires an in-depth knowledge of the IO characteristics of the underlying hardware, which does not seem feasible in cloud deployments, and finally (iii) the CPU utilization is easy to measure. In contrast, the average system utilization based on the queuing theory has no limit (CPU utilization is limited to 100%) and thresholds can be defined independently of the application. However, determining the average system utilization based on queueing theory requires application-level metrics such as response times and throughput per class of request.

Algorithm 11.1 illustrates the simplified decision logic underlying both approaches. As input parameters, the algorithm gets the current average system utilization $\overline{\rho}$ and the number of currently running instances run. In contrast to the CPU utilization-based approach, where $\overline{\rho}$ is equal to the measured average CPU utilization, the service demand-based approach uses the average system utilization based on the queueing theory, which provides a good trade-off between estimation time and accuracy (Grohmann et al., 2017). The system utilization ρ is the arrival rate multiplied by the estimated service demand or the highest service demand if multi-tier systems are scaled (Bolch et al., 2006). First, the algorithm checks if the average system utilization per instance $\overline{\rho}$ falls below the predefined down-scaling threshold (Line 2). If this is true, the new average system utilization per VM is calculated after decreasing the number of instances iteratively (Line 3–6). Then, the algorithm checks if $\overline{\rho}$ exceeds the predefined up-scaling threshold (Line 8). If it holds, the new average system utilization per instance is computed after iteratively increasing the number of instances (Line 9–12). Since the algorithm investigated $\overline{\rho}$ twice, the algorithm prevents that too much instances are released, and thus, the $\overline{\rho}$ exceeds the up-scaling threshold. Finally, the number of instances that are required (delta > 0) or that can be released (delta < 0) are returned.

Algorithm 11.1: Decision logic

Input: Average system utilization $\overline{\rho}$, runningInstances *run*
Result: DeltaInstances *delta*

```
1  delta = 0
2  if ρ̄ <= down_threshold then // down_threshold is predefined
3      while ρ̄ <= down_threshold do
4          delta−−
5          ρ̄ = ρ̄ * (run / (run + delta)) // calculates the new average
               utilization
6      end
7  end
8  if ρ̄ > up_threshold then // up_threshold is predefined
9      while ρ̄ > up_threshold do
10         delta++
11         ρ̄ = ρ̄ * (run / (run + delta)) // calculates the new average
               utilization
12     end
13 end
14 return delta
```

In the following experiments, the CPU utilization-based approach and the service demand-based approach scaled three different applications (see Section 11.1.2): (i) The LU application with its performance limited by hardware contention, (ii) the Content application restricted by software contention, and finally (iii) the Mixed application exhibiting both hardware and software contention. Each application was deployed on c.medium VMs in the CSPC environment (see Section11.1.3) and stressed by the Retailrocket trace, which was sped-up so that each recorded point is equal to one minute leading to 192 minutes experiment time. Consequently, every minute, VM specific information (such as the amount of running VMs and the average CPU utilization) and application-specific information (such as request arrival rates) were gathered and passed to the auto-scalers. For the service demand-based approach, the estimation of service demand was updated online every 10 minutes because the estimated service demand is not expected to change significantly soon. As a baseline scenario called *No AS*, we run 15 VMs throughout the experiment duration.

11.2.2 Hardware Contention Scenario

The results for the hardware contention scenario are shown in Figure 11.2. This diagram is divided into three sub-figures: The first plot shows the scaling behavior of both approaches, the second plot the input (i.e., the average utilization per instance) for each approach, and the last plot the estimated service demand. For each plot, the horizontal axis shows the time of the measurement in minutes. In the first plot, the amount of demanded resources (determined by BUNGEE, see Section 4.2.2) is depicted as black curve, the amount of supplied resources using the service demand-based approach as blue curve, and the amount of supplied resources using the CPU utilization-based approach as red curve. Both methods tend to overprovision the system during the decreasing daily decline. However, the auto-scaler based on CPU utilization overprovisioned more instances during this period compared to the service demand-based approach. The service demand-based auto-scaler can also handle the increasing load during each day more efficiently than the CPU utilization-based auto-scaler. More precisely, there is almost no underprovisioning, and the approach tends to overprovision slightly.

The observations during the auto-scaling can be explained by looking at the plot in the middle of Figure 11.2. Here, the blue curve shows the average system utilization, the red curve the average CPU utilization, and the black dashed or dot-dashed line represents the threshold for up-scaling (90%) or down-scaling (70%), respectively. In contrast to the CPU utilization limited by 100%, the system utilization is not limited upwards by a fixed value. That is, the system utilization can have values higher than 100%. For instance, at minute 115, where the system is in an underprovisioned state for both approaches, the CPU utilization is 100% and the system utilization is 160%. Consequently, the service demand-based approach assigns three instances to handle this utilization and the CPU utilization-based approach only one instance.

To enable a quantitative comparison, we compare both approaches with the baseline approach No AS (i.e., no scaling takes place) and report the elasticity as well as user-oriented measures in Table 11.4. Each row shows a measure (a brief overview of the measures is given in Section 11.1.5), and each column refers to an auto-scaling approach. The best values are highlighted in bold. Comparing only the individual measures of elasticity, the service demand-based approach exhibits the best results for θ_O, τ_U, and τ_O. Consequently, it also achieves the highest ϵ (1.41) and the lowest σ (41.83%). As the CPU utilization-based approach also has an elastic speed-up greater than 1, both methods are more efficient than the baseline approach. Also, both approaches use fewer instances and achieve higher SLO conformance. In terms of SLO

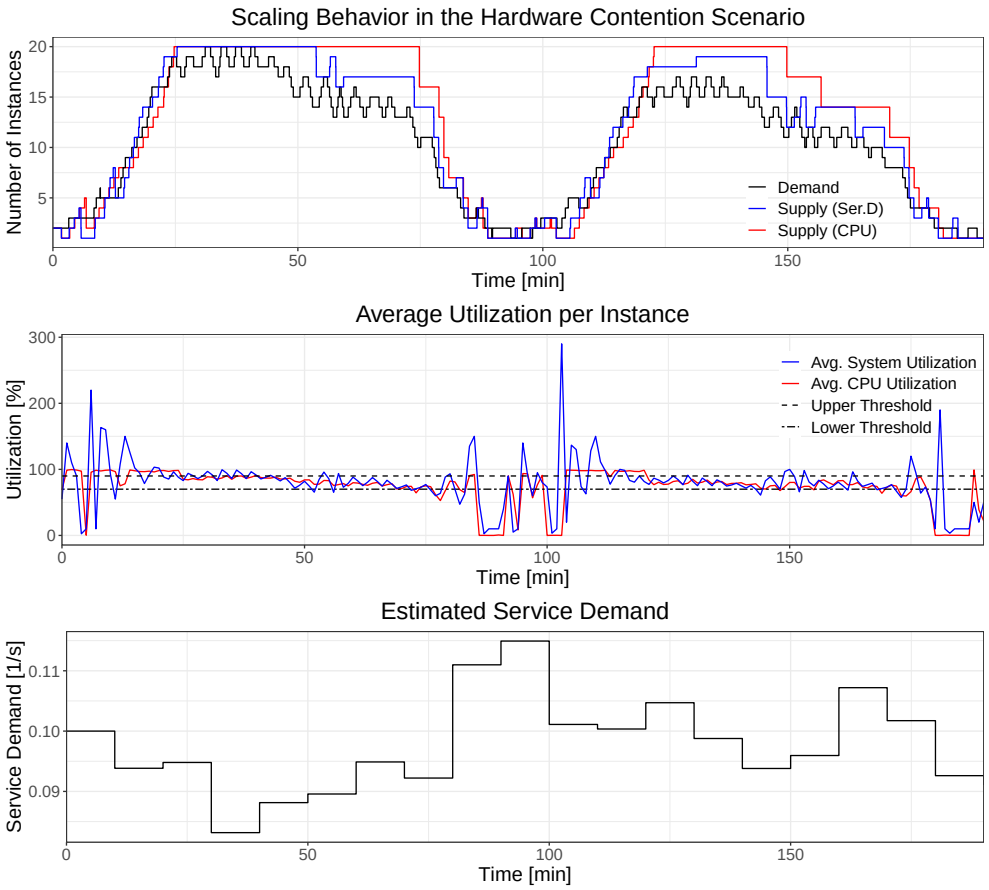

Figure 11.2: Scaling behavior in the hardware contention scenario.

violations, the service demand-based auto-scaler has the lowest value for SLOs (8.40%) and the highest value for Apdex (93.17%).

11.2.3 Software Contention Scenario

Exactly as in Section 11.2.2, the results for the software contention scenario are illustrated in Figure 11.3. In each plot, the horizontal axis shows the time of the measurement in minutes. The scaling behavior of both approaches is depicted in the upper plot. Again, the amount of demanded resources is represented as black curve, the amount of supplied resources using the service demand-based approach as blue curve, and the amount of supplied resources using the CPU utilization-based approach as red curve. As the software contention

Table 11.4: Results for the hardware contention scenario.

Measure	CPU Util.-based	Ser. Demand-based	No AS
θ_U [%]	7.46	7.54	**3.20**
θ_O [%]	23.57	**14.60**	203.28
τ_U [%]	22.20	**19.10**	22.89
τ_O [%]	62.65	**62.56**	67.38
υ [%]	21.25	22.63	**16.61**
ϵ	1.26	**1.41**	1.00
σ [%]	43.34	**41.83**	105.14
SLOs [%]	12.67	**8.40**	45.72
Apdex [%]	88.58	**93.17**	58.59
#Adaptations	62	67	0
Avg. #Instances	7.93	8.58	15.00

scenario causes only minimal load on the CPU, we calibrated and adjusted the thresholds of the CPU utilization-based approach for up-scaling (grey dashed line in the middle plot) to 30% and down-scaling (grey dot-dashed line in the middle plot) to 5%. Despite the adaptation, the CPU utilization-based approach is unable to provide the required amount of resources. As a result, the system is underprovisioned for almost the entire measurement. In contrast, the thresholds for the service demand-based approach are unchanged, leading to a similar scaling behavior as in the hardware contention scenario. Moreover, the service demand-based approach provides sufficient resources so that the supply curve closely follows the demand curve.

To assess the scaling behavior of both approaches, we calculated the respective elasticity and user-oriented measures. The measures of both approaches and the No scaling scenario are shown in Table 11.5. Each row shows a measure (the best values are highlighted in bold). The columns represent both auto-scaling approaches as well as the No scaling scenario. In terms of the single elasticity measures, the service demand-based approach has for θ_O the best value and the second-best values for the remaining measures. Consequently, this approach exhibits the lowest value for σ (36.81%) and highest value for ϵ (1.57). Both measures are also reflected by the lowest SLO value for (6.27%) and the highest value for Apdex (94.20%). In contrast, the CPU utilization-based approach provides in 86.23% of the measurement too less resources. Thus, this approach has the highest SLO value for (97.25%), the lowest value for Apdex (3.2%), and ϵ is less than 1 (i.e., the scaling behavior is worse compared to the

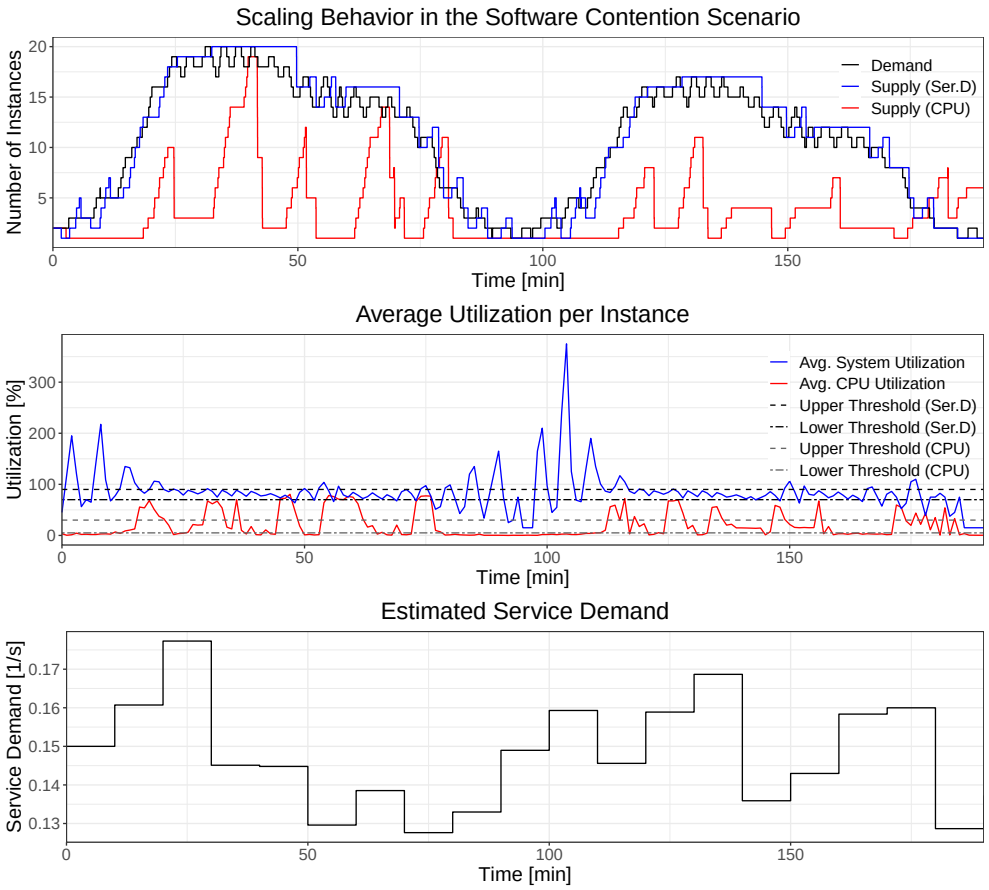

Figure 11.3: Scaling behavior in the software contention scenario.

baseline approach).

11.2.4 Mixed Contention Scenario

Similar to the hardware and software contention scenario, the results of the mixed contention scenario are depicted in Figure 11.4. In each of the three sub-plots, the horizontal axis shows the experiment time in minutes. The scaling behavior of both approaches is shown in the first sub-plot. Again, the amount of supplied resources using the service demand-based approach is depicted as blue curve, the amount of supplied resources using the CPU utilization-based approach as red curve, and the amount of demanded resources as black curve. In this scenario, we also calibrated and adjusted the thresholds for the

Table 11.5: Results for the software contention scenario.

Measure	CPU Util.-based	Ser. Demand-based	No AS
θ_U [%]	60.45	7.64	**3.20**
θ_O [%]	24.67	**8.36**	203.28
τ_U [%]	86.83	27.02	**22.89**
τ_O [%]	**9.29**	43.37	67.38
υ [%]	23.49	22.23	**16.61**
ϵ	0.90	**1.57**	1.00
σ [%]	99.58	**36.81**	104.16
SLOs [%]	97.25	**6.27**	8.64
Apdex [%]	3.2	**94.20**	92.36
#Adaptations	71	75	0
Avg. #Inst.	6.69	8.21	15.00

CPU utilization-based approach for up-scaling (grey dashed line in the middle plot) to 55% and down-scaling (grey dot-dashed line in the middle plot) to 40%. Based on this adjustment, the CPU utilization-based approach is able to cover the resource demand closer than in the previous scenarios. However, during the increasing load of the first day, the approach has problems meeting the required resources, and thus, the system is in an underprovisioning state. Like in the previous scenarios, the thresholds for the service demand-based approach are the same, and again, the approach tends to overprovision slightly with almost no underprovisioning.

The scaling behavior of both approaches is quantified by the measures listed in Table 11.6. Again, each row shows a measure (elasticity and user-oriented) where the best values are depicted in bold. The columns represent the CPU utilization-based approach, the service demand-based approach, and the No scaling scenario. In terms of single elasticity measures, the CPU utilization-based approach exhibits in 2 of 5 the best values. In contrast, the service demand-based approach only has the best value for τ_U and has in 3 of 4 remaining measures the second-best values. Consequently, the service demand-based approach exhibits the value of ϵ (1.43) and has only slightly worse σ (40.89%) than the CPU utilization-based approach (39.86%). Moreover, the service demand-based approach has the lowest SLO violations (4.77%), the highest Apdex (96.39%), and provisions, on average, the least amount of resources. Although the CPU utilization based approach has an ϵ greater than 1 (i.e., having a better scaling behavior than the baseline approach), it exhibits higher

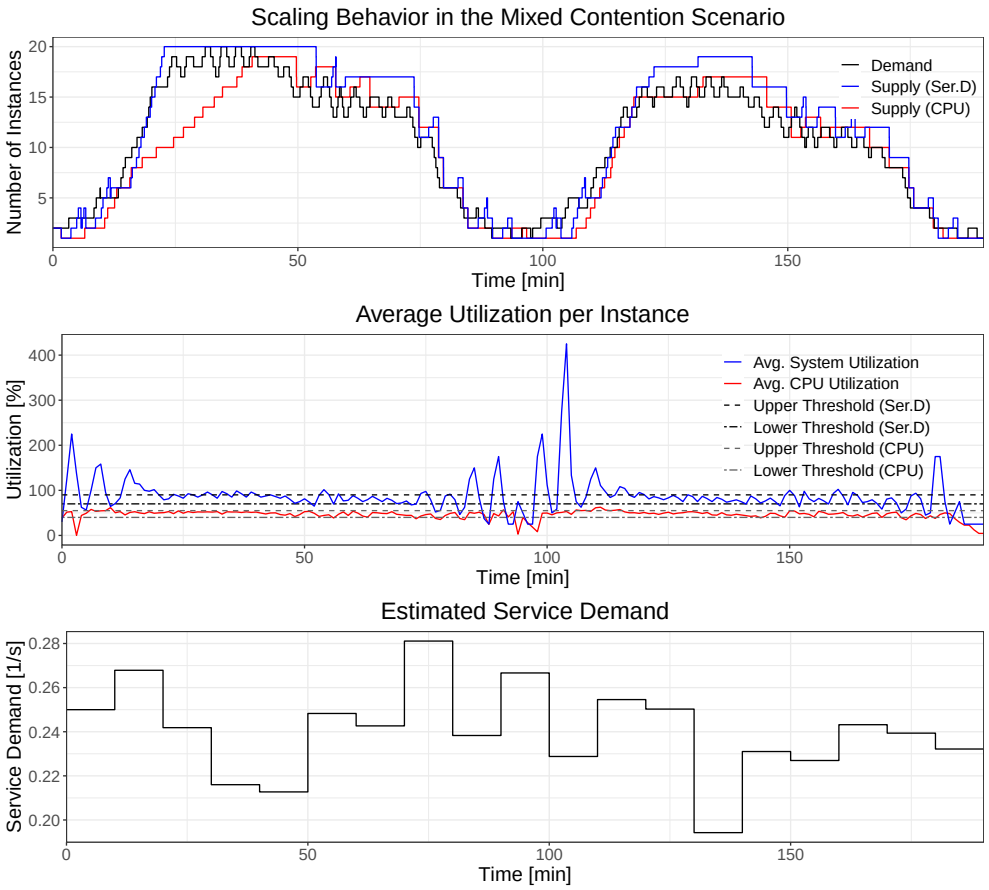

Figure 11.4: Scaling behavior in the mixed contention scenario.

SLOs (11.96%) and less Apdex (89.02%) than the baseline approach (i.e., no scaling takes place).

11.2.5 Summary of the Results and Threats to Validity

Comparing the different scenarios, the service demand-based auto-scaler behaves similarly in each scenario and per day. Also, this approach exhibits the best values for elasticity and user-oriented measures. In contrast, the CPU utilization-based approach behaves differently in each scenario. Although the CPU usage-based approach has problems in the software conflict scenario, it is still more efficient than not using auto-scaling in the hardware contention and mixed contention scenarios. Note that the elasticity metrics for the baseline

Table 11.6: Results for the mixed contention scenario.

Measure	CPU Util.-based	Ser. Demand-based	No AS
θ_U [%]	14.61	7.13	**3.20**
θ_O [%]	**7.05**	14.12	203.28
τ_U [%]	42.86	**19.28**	22.89
τ_O [%]	**35.03**	59.90	67.38
υ [%]	20.91	23.92	**16.61**
ϵ	1.39	**1.43**	1.00
σ [%]	**39.86**	40.89	104.19
SLOs [%]	11.96	**4.77**	6.40
Apdex [%]	89.02	**96.39**	94.26
#Adaptations	62	75	0
Avg. #Inst.	9.51	8.96	15.00

scenario are, by definition, identical in all experiments. In terms of applicability, the CPU utilization-based auto-scaler is easy to setup and needs no further instrumentation. For determining the service demand, a complex but non-intrusive instrumentation is required. The service demand must either be determined in advance, assuming that it is static, or the demand for services must be estimated on-the-fly. For this purpose, the estimator needs structural application knowledge and information that may require basic instrumentation of the application to monitor high-level metrics such as throughput and response times. In summary, the service demand is an independent and reliable input for automatic scaling, but involves a high instrumentation overhead.

Due to overbooking of resources and background traffic, CPU utilization measurements in public cloud infrastructures are both unstable and unreliable (Iosup et al., 2011). To avoid this performance variability, we conducted the experiments in our private cloud environment under controlled conditions. Based on our experience with experiments in public clouds (see Section 11.3.3), auto-scaling based on CPU utilization is supposed to perform worse than under controlled conditions, while it has little impact on the service demand-based approach. We have also chosen two similar days of the Retailrocket trace and performed long experiments to validate the measurements internally or have the second day as a repetition of the first. Since the defined thresholds influence the scaling behavior, we cannot prove that we have chosen the optimal ones.

11.3 Benchmarking of the Chameleon Approach

In this section, we benchmark the Chameleon[12] approach against different state-of-the-art auto-scalers. First, we describe the experiments in Section 11.3.1. Then, we explain how to interpret the experimental results in Section 11.3.2. To investigate how the auto-scalers behave on different platforms, we investigate their scaling in Section 11.3.3. Section 11.3.4 lists the results from all experiments and ranks the auto-scalers. Finally, we summarize the results and discuss threats to validity in Section 11.3.5.

11.3.1 Experimental Description

To benchmark Chameleon under authentic conditions, we compare Chameleon, on the one hand, against the autos-scalers (i) Adapt, (ii) ConPaaS, (iii) Hist, (iv) T-Hold, and (v) Reg; on the other hand, we deployed the LU application in our controlled CSPC environment (c.medium) and in AWS EC2 (m4.large) as well as in MMEE (d.small) to have experiments with background noise. The details of the deployment can be found in Section 11.1.3. For stressing the LU application, we considered all workloads (see Section 11.1.1) for comparing the scaling of the used mechanisms to different demand curves. In these experiments, each trace was accelerated so that one day in the workload takes 3.2 hours of experiment time. In other words, every two minutes, VM specific information (such as the amount of running VMs and the average CPU utilization) and application-specific information (such as request arrival rates) were gathered and passed to the auto-scalers. As a baseline scenario called *No AS*, we run 9 VMs throughout the experiment duration.

11.3.2 Introduction to the Results

In total, we benchmark Chameleon in seven different settings. For the sake of clarity, we now present only two settings in more detail so that the results presented in the following sections are easier to interpret. The settings comprise the Wiki and Retrailrocket trace. Both experiments were conducted in the CSPC environment. The scaling behavior of all auto-scalers for the Wiki trace is depicted in Figure 11.5. The figure shows the scaling of Chameleon (top left), Adapt (middle left), Hist (bottom left), ConPaaS (top right), Reg (middle right), and T-Hold (bottom right). In each subplot, the horizontal axis shows the experiment time in minutes; the vertical axis shows the number

[12] As Chameleon is preliminary work for Chamulteon, these experiments were done in collaboration with Nikolas Herbst (Herbst, 2018).

of concurrency running instances (i.e., VMs). The black curves represent the resource demand (determined by BUNGEE, see Section 4.2.2) and the blue curves the amount of supplied VMs of each auto-scaler. If the supply curve falls below the demand curve, the system is in an underprovisioned state. In case the supply curve exceeds the demand curve, the system is in an overprovisioned state.

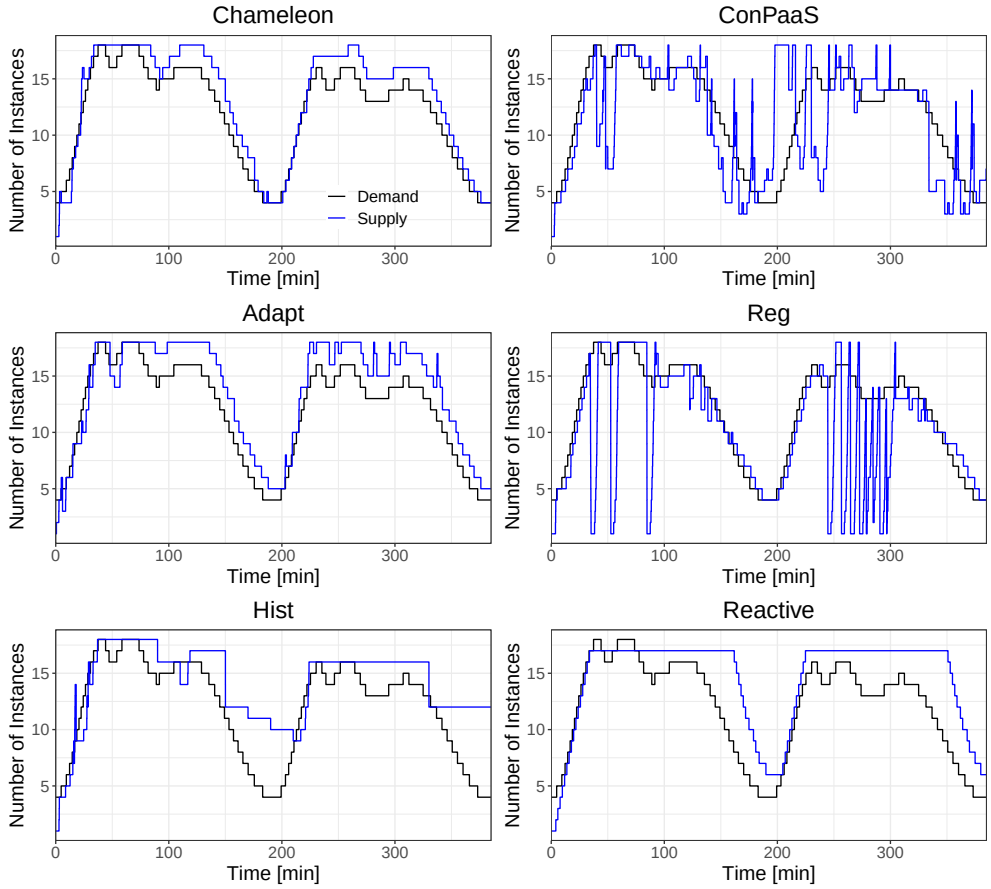

Figure 11.5: Comparison of the auto-scalers on the Wiki trace.

Comparing the scaling behavior of the auto-scalers for the Wiki trace, a first observation is that the auto-scaler can be divided into two groups: (i) tendency to overprovision the system (Hist, T-Hold, Adapt and Chameleon), (ii) tendency to underprovision the system (ConPaaS and Reg). Chameleon is the first auto-scaler to meet the increasing demand at the beginning of the experiment

and then tends to supply slightly more VMs than required until the end of the experiment. Adapt behaves similarly to Chameleon, but allocates more VMs. T-Hold's supply curve closely follows the demand curve during the increasing load, but the down-scaling is delayed. Hist approximately meets the demand and keeps the amount of supplied VMs for a longer time (30 to 60 minutes) until it drops to the current demand. In contrast to the other auto-scalers, Reg and ConPaaS show a high oscillation rate during the measurement.

To enable a quantitative comparison of the scaling behaviors, we calculated elasticity as well as user-oriented measures listed in Table 11.7. Each row shows a measure (a brief overview of the measures is given in Section 11.1.5), and each column refers to an auto-scaler, with the last one corresponding to the no auto-scaling scenario. The best values are highlighted in bold. Comparing individual measures, only the aspect characterized by the respective measure is considered and can lead to inconsistent rankings. For example, Chameleon has the best values for θ_U, τ_U, SLOs, and Apdex, while Reg has the best values for θ_O and τ_O, but has the most SLOs violations compared to all other auto-scalers. To this end, we quantify the performance of the auto-scalers based on σ as well as ϵ. Here, Chameleon exhibits the best values for both of these aggregated measures. The good scaling is also supported by the fact that our approach also shows the best values for SLOs and Apdex. In the case of ϵ, the scaling behavior of Chameleon is 2.30 times more efficient compared to the baseline scenario, while the other auto-scalers reach at most a value of 1.56. ConPaaS (0.91), for instance, is less efficient than no auto-scaling takes place.

Table 11.7: Results for the Wiki trace.

Measure	Chameleon	Adapt	Hist	ConPaas	Reg	T-Hold	No AS
θ_U [%]	**1.57**	1.68	2.37	14.69	16.08	2.10	25.93
θ_O [%]	11.15	17.51	33.55	15.67	**4.34**	28.94	19.38
τ_U [%]	**5.70**	9.16	12.75	47.41	51.04	12.27	70.48
τ_O [%]	73.25	80.94	71.95	32.07	**25.24**	80.77	26.28
υ [%]	5.83	7.09	4.75	12.66	12.88	4.97	**2.94**
ϵ	**2.30**	1.78	1.51	0.91	1.19	1.56	1.00
σ [%]	**39.49**	45.24	42.75	62.54	81.71	46.67	88.13
SLOs [%]	**2.95**	15.47	11.86	59.73	80.71	7.15	85.95
Apdex [%]	**97.22**	85.40	88.84	42.89	21.65	93.23	15.21
#Adaptations	61	66	26	112	102	49	0
Avg. #Inst.	10.80	12.34	11.57	11.19	9.38	10.45	9.00

Similar to Figure 11.5, Figure 11.6 shows the scaling behavior of the auto-

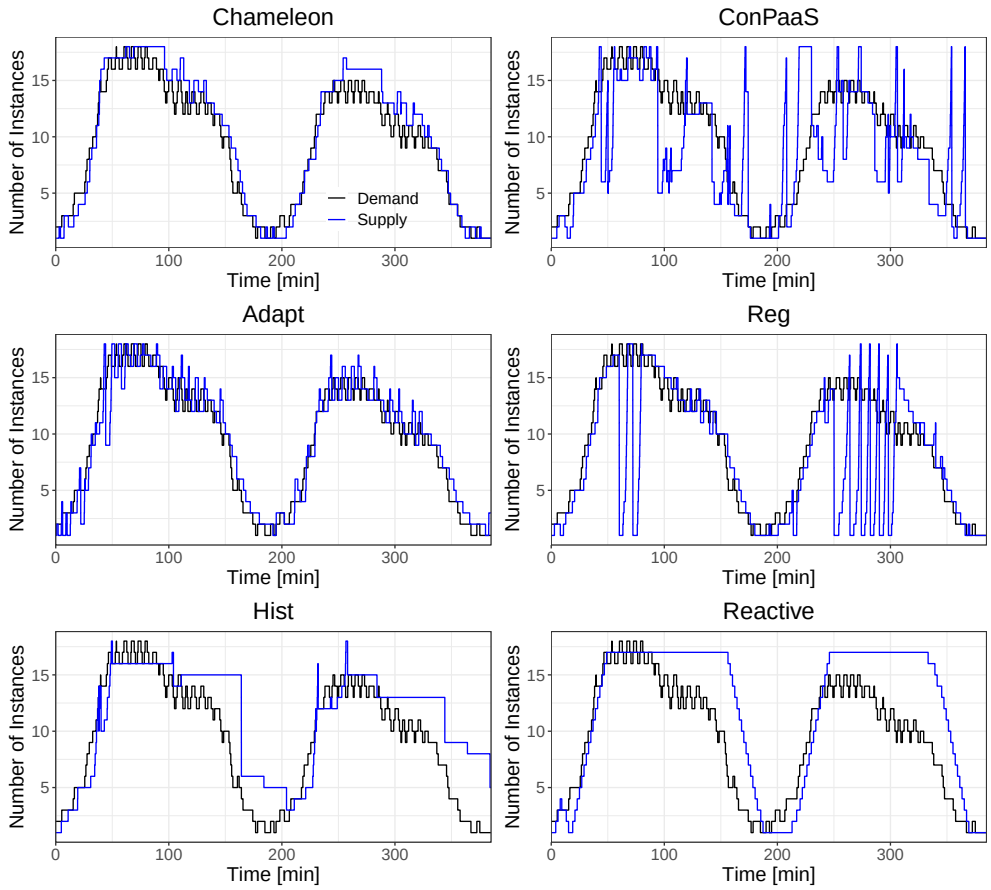

Figure 11.6: Comparison of the auto-scalers on the Retailrocket trace.

scalers for the Retailrocket trace. Each subplot shows an auto-scaler, where the black curves represent the resource demand, the blue curve the supplied VMs, and the horizontal axis the experiment time in minutes. In this scenario, the auto-scalers behave similarly as observed for the Wiki trace: Chameleon and Adapt are able to meet the resource demand by slightly overprovision the system; T-Hold shows a fast up-scaling but a slow down-scaling; Hist can roughly supply the demand and holds the number of resources for a certain time; Reg and ConPaaS are subject to strong oscillations during measurement. The scaling behaviors of each auto-scaler are quantified by the measures listed in Table 11.8. Again, each row shows a measure (the best values are highlighted in bold), and each column represents an auto-scaler. Chameleon has the best

values for ϵ, σ, SLOs, and Apdex.

Table 11.8: Results for the Retailrocket trace.

Measure	Chameleon	Adapt	Hist	ConPaas	Reg	T-Hold	No AS
θ_U [%]	6.37	6.55	**5.22**	21.05	19.37	8.50	17.68
θ_O [%]	11.71	16.45	76.29	22.41	**8.44**	44.64	110.38
τ_U [%]	**19.53**	31.76	24.59	59.90	44.38	25.02	55.75
τ_O [%]	51.34	44.60	59.42	**20.74**	29.21	62.92	38.81
υ [%]	10.98	14.82	9.46	18.24	15.73	9.68	**7.22**
ϵ	**2.06**	1.68	1.41	1.23	1.56	1.39	1.00
σ [%]	**35.59**	40.49	49.39	63.93	60.51	45.53	90.52
SLOs [%]	**8.68**	26.56	16.26	60.87	58.20	15.20	82.06
Apdex [%]	**91.89**	75.38	84.42	41.09	44.19	85.41	19.37
#Adaptations	82	125	36	122	113	69	0
Avg. #Inst.	9.301	11.009	11.013	10.196	9.0686	8.2361	9.00

11.3.3 Auto-Scaling on Different Platforms

To investigate how the auto-scalers behave on different platforms, the auto-scalers are investigated in the CSPC, AWS EC2, and MMEE environment (see Section 11.1.3). Moreover, the FIFA trace is used for inducing the same resource demand in all three scenarios. Note that although the resource demand curves are identical, the sent request differ due to different resource mappings (see Section 4.2.2). As an example, the resulting scaling behaviors[13] of Chameleon and T-Hold are illustrated in Figure 11.7. The left column shows Chameleon and the right column T-Hold on all three platforms. In each subplot, the horizontal axis shows the experiment time in minutes, and the vertical axis shows the number of concurrently running VMs. The black curves represent the resource demand and the blue curves the amount of supplied VMs of each auto-scaler. While Chameleon scales almost identically in the CSPC and MMEE scenarios and similarly in the AWS EC2 scenario, T-Hold shows a different behavior for each platform. This can be explained as Chameleon is based on service demand (see Section 11.2), and T-Hold is based on CPU utilization. If the system is in an underprovisioned state, the CPU utilization drops significantly for some time, and thus, T-Hold scales down because the CPU utilization suggests a low load. Afterward, T-Hold allocates more resources until the CPU utilization drops again. Since these drops in CPU utilization seem to follow a pattern,

[13] Note that the behavior of all auto-scalers is quantified in Section 11.3.4.

we assume that AWS EC2 performs migrations in the background to move the VMs from overloaded hosts to hosts with lower load, and therefore, the CPU utilization drops. In the MMEE scenario, T-Hold has problems to supply sufficient resources during the increasing load of each day, but shows the same delayed down-scaling as in the CSPC scenario.

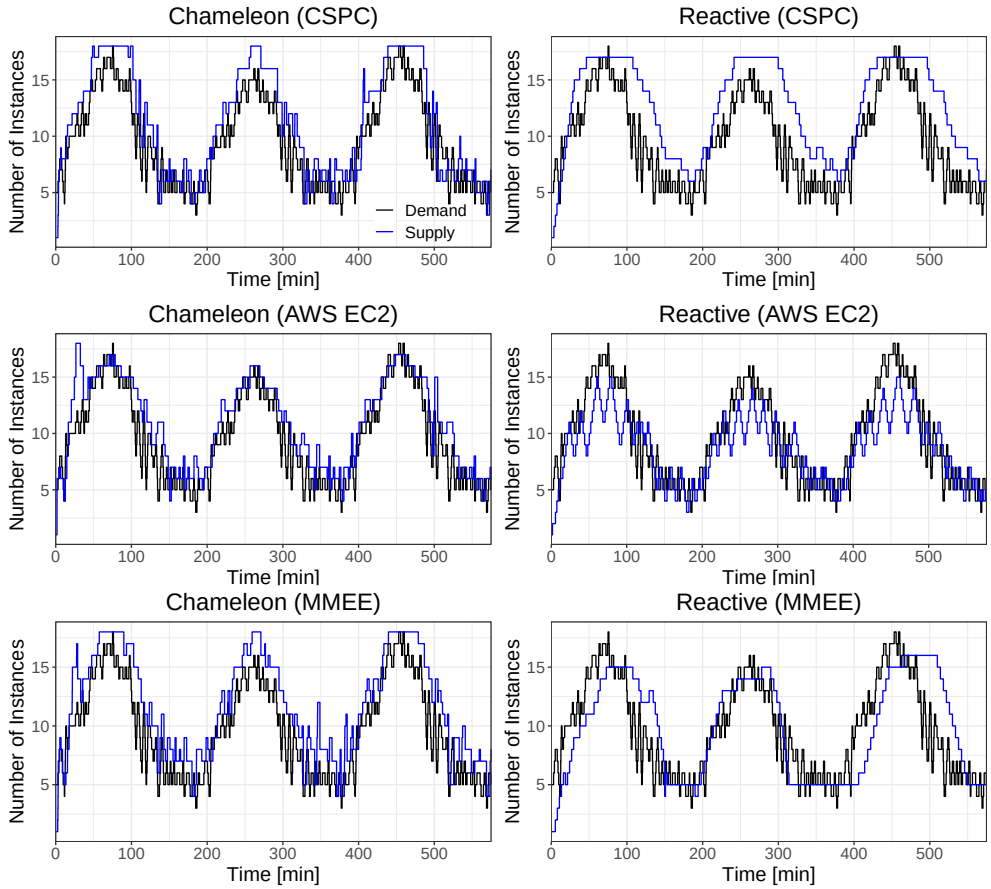

Figure 11.7: Comparison of the Chameleon and T-Hold on different platforms.

The associated measures of the scaling behaviors are listed in Table 11.9. Each row shows a measure (user-oriented and elasticity), and each column one of both auto-scalers on a different platform. Chameleon shows on all platforms a low variation for almost all single elasticity measures, and also the number of adaptations is nearly the same. In contrast, T-Hold shows a high variation in all single elasticity measures across all platforms. The results and comparison

of the other auto-scalers are listed in the next section.

Table 11.9: Comparison of the scaling behavior of Chameleon and T-Hold across different platforms.

Measure	Chameleon			T-Hold		
	CSPC	AWS EC2	MMEE	CSPC	AWS EC2	MMEE
θ_U [%]	3.23	1.64	1.66	1.56	13.36	11.83
θ_O [%]	21.95	21.31	28.28	40.20	9.19	16.52
τ_U [%]	13.09	11.04	6.45	5.20	57.49	46.85
τ_O [%]	74.05	67.92	84.97	87.14	25.49	38.12
υ [%]	16.36	15.95	16.76	14.68	19.02	13.96
ϵ	1.58	1.93	1.92	1.93	1.27	1.18
σ [%]	43.88	39.81	46.53	46.76	54.80	50.10
SLOs [%]	8.65	5.04	20.95	2.70	49.30	41.49
Apdex [%]	91.79	95.38	79.34	97.41	51.70	59.26
#Adaptations	122	126	131	81	254	135
Avg. #Inst.	9.53	9.65	9.88	10.49	8.84	10.98

11.3.4 Overall Evaluation

Since we have discussed only selected experiments in the previous sections, this section focuses on all experiments and presents an overview of all results. In our experimental setup, each auto-scaler was investigated on five different traces and three different platforms. In total, the experiments lasted over 400 hours, during which approximately 107 million requests were sent, and the auto-scalers made 5000 adjustments. Table 11.10 shows ϵ, σ, SLOs, and Apdex for all scenarios. Each row corresponds to a measure and each column to an auto-scaler. The best values are highlighted in bold. In five of seven scenarios, Chameleon exhibits the best ϵ, σ, SLOs, and Apdex. In the last two scenarios, Chameleon has the best σ and the second-best values for the remaining measures.

To rank the auto-scalers, we determine the average rank regarding the four listed measures in Table 11.10. In terms of ϵ, Chameleon is the best auto-scaler followed by T-Hold, Adapt, Hist, Reg, and ConPaaS. By taking σ into account, Chameleon is again the best auto-scaler followed by Hist, Adapt, T-Hold, ConPaaS, and Reg. SLOs and Apdex show both the same ranking:

Table 11.10: Comparison of the auto-scalers over all experiments.

Experiment	Measure	Chameleon	Adapt	Hist	ConPaaS	Reg	T-Hold
Wikipedia	ϵ	**2.30**	1.78	1.51	0.91	1.19	1.56
	σ [%]	**39.49**	45.24	42.75	62.54	81.71	46.67
	SLOs [%]	**2.95**	15.47	11.86	59.73	80.71	7.15
	Apdex [%]	**97.22**	85.40	88.84	42.89	21.65	93.23
IBM	ϵ	**3.10**	1.73	1.79	1.79	2.47	1.74
	σ [%]	**29.05**	42.54	43.40	42.14	65.79	44.50
	SLOs [%]	**9.57**	29.92	11.44	37.41	65.32	31.44
	Apdex [%]	**91.13**	71.71	89.25	64.38	37.96	63.32
BibSonomy	ϵ	1.27	1.12	1.18	0.83	1.13	**1.33**
	σ [%]	**43.05**	59.39	48.28	48.31	56.17	48.60
	SLOs [%]	16.20	54.85	13.99	31.75	50.74	**6.47**
	Apdex [%]	84.73	47.49	87.05	69.42	51.10	**93.79**
Retailrocket	ϵ	**2.06**	1.68	1.41	1.23	1.56	1.39
	σ [%]	**35.59**	40.49	49.39	63.93	60.51	45.53
	SLOs [%]	**8.68**	26.56	16.26	60.87	58.20	15.20
	Apdex [%]	**91.89**	75.38	84.42	41.09	44.19	85.41
FIFA CSPC	ϵ	1.58	1.33	1.37	0.98	1.04	**1.93**
	σ [%]	**43.88**	49.68	46.27	59.09	67.30	46.76
	SLOs [%]	8.65	43.67	12.01	53.23	64.28	**2.70**
	Apdex [%]	91.79	59.55	88.67	49.08	38.11	**97.41**
FIFA AWS EC2	ϵ	**1.93**	1.21	1.13	0.85	1.35	1.27
	σ [%]	**39.81**	43.31	44.93	47.83	59.22	54.80
	SLOs [%]	**5.04**	21.56	16.60	22.17	54.36	49.30
	Apdex [%]	**95.38**	79.20	83.88	78.54	46.42	51.70
FIFA MMEE	ϵ	**1.92**	1.53	1.30	1.00	1.15	1.18
	σ [%]	**46.53**	47.61	50.02	54.38	53.51	50.10
	SLOs [%]	**20.95**	37.31	26.96	43.61	47.51	41.49
	Apdex [%]	**79.34**	63.23	73.34	58.48	53.30	59.26

Chameleon, Hist, T-Hold, Adapt, ConPaaS, and Reg. To summarize, Chameleon exhibits for all traces, platforms, and measures the best average rank.

11.3.5 Summary of the Results and Threats to Validity

Comparing the scaling behavior across the different experiments (i.e., different platforms and workload traces), our contribution Chameleon performs best based on the average auto-scaling deviation, elastic speed-up score, SLOs violations, and user satisfaction. Among the competing state-of-the-art auto-scalers,

no mechanism outperforms the others in all scenarios. In terms of scaling, Chameleon scales the system reliably with a slight overprovisioning. Adapt succeeds in precisely following the demand with a relatively high number of adjustments. Hist and T-Hold both tend to overprovision the system while allocating more VMs than the other auto-scalers. T-Hold heavily depends on accurate measurements of the CPU utilization. Therefore, T-Hold shows a reduced performance in the AWS EC2 scenario, where overbooking of virtual resources can lead to significant interference with the background load. In contrast to the other auto-scalers, ConPaaS and Reg exhibit unstable/unpredictable behavior (i.e., oscillations) during large parts of the experiments. In summary, Chameleon scales an application reliably regardless of the environment and workload and thus outperforms the competing auto-scalers in all scenarios.

Repeatability of performance-related experiments in public cloud environments is constrained by the lack of control over placement and the co-location of VMs with other workloads stressing the cloud. As a result, performance variability can be significant (Iosup et al., 2011). To mitigate this problem, we performed most experiments in our private environment under controlled conditions (CSPC). We also included experiments performed in two different public cloud deployments (MMEE and AWS EC2) with varying levels of background load. Furthermore, the results may not be generalizable to other types of applications (e.g., applications that are not interactive or CPU intensive), although our experimental analysis covers a wide range of different scenarios. For the evaluated competing auto-scalers, we observed a similar behavior as in related studies on auto-scaling evaluation (Papadopoulos et al., 2016; Ilyushkin et al., 2018). That is, Adapt following closely the resource demand, platoons for Hist, and unstable behavior of ConPaaS and Reg.

11.4 Evaluation of the Fox Approach

In this section, we evaluate our contribution Fox. First, we describe the setup of the experiments in Section 11.4.1. Then, we explain how to interpret the results in Section 11.4.2. In Section 11.4.3 and 11.4.4 we discuss the results based on a hourly and two-phase pricing scheme, respectively. Finally, we sum up the results and discuss threats to validity in Section 11.4.5.

11.4.1 Experimental Description

To investigate how Fox revises the scaling decision to be cost-efficient, we compare the auto-scalers (i) Adapt, (ii) Hist, and (iii) React with and without

Fox revising their decisions. As use case, we used the Verification application deployed on c.small VMs in the CSPC environment (see Section 11.1.3). For stressing the application, we applied the IBM and Wiki trace. Further, we accelerated the traces so that each day in the workload is equal to 3.2 hours of experiment time. In other words, every 2 minutes, VM specific information (such as the amount of running VMs and the average CPU utilization) and application-specific information (such as request arrival rates) were gathered by Fox. In terms of cost-aware scaling, we applied an hourly and a two-phase pricing scheme (see Section 9.4.1). Note that for the following experiments, Fox was used as a stand-alone tool. That is, Fox was decoupled from Chamulteon and acted as a mediator between the three auto-scalers. To this end, we implemented sNaïve (see Section 3.1.1) as forecasting method for the Analyze phase of Fox.

11.4.2 Introduction to the Results

Similar to Section 11.3.2, we first introduce how to interpret the experiment format before discussing the details of the results. Figure 11.8 shows the BibSonomy trace scaled by React. Each sub-figure shows the scaling for a service, where the horizontal axis shows the experiment time in minutes and the vertical axis the number of concurrently running instances. The black curves represent the resource demand (determined by BUNGEE, see Section 4.2.2) and the blue curves the amount of supplied VMs by the auto-scaler. If the supply curve falls below the demand curve, there are too few VMs provisioned. In case the supply curve exceeds the demand curve, too many VMs are instantiated. In terms of scaling, React performs many adjustments to satisfy the current resource demand. Due to React's reactive scaling policy, instances are added too late in some cases, resulting in an underprovisioning of the system. However, most of the time, React is able to supply enough VMs to match the current resource demand.

Figure 11.9, which has the same structure as Figure 11.8, illustrates the BibSonomy trace scaled by React while Fox revised the scaling decisions based on an hourly pricing scheme. For each of the services, the unstable behavior of React is smoothed when using Fox. Now, React tends to overprovision the system. In other words, the supply curve remains above the demand curve most of the time. However, some scaling actions are performed to reduce the amount of unused VMs, if this is in accordance with the charging scheme. This resulting scaling behavior occurs because Fox only performs down-scaling if the instances that should be released will not be used in the future.

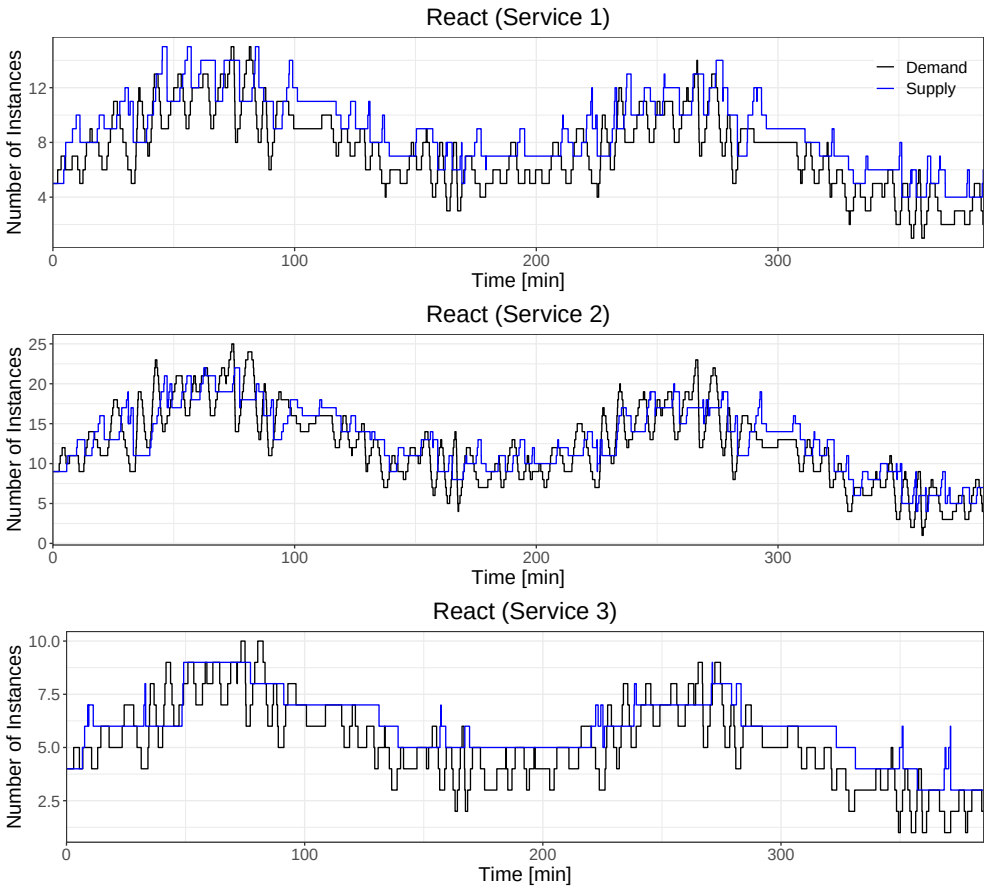

Figure 11.8: Scaling behavior of React without Fox on the BibSonomy trace.

To assess the influence of Fox on the scaling decisions, we investigate elasticity, user-oriented, and cost-saving measures (see Section 11.1.5) listed in Table 11.11. The columns show React with and without Fox, and each row corresponds to a measure, where the best values are highlighted in bold. While using Fox, React exhibits for each measure concerning underprovisioning a better value. Due to the smoothing of Fox, also υ is reduced for all services. However, the measures concerning overprovisioning have deteriorated as down-scaling of an instance takes only place if this instance will not be used the next time. Moreover, React with Fox (60.89%) has a worse ς than stand-alone React (55.71%). However, Fox can reduce the SLO violations from 12.05% to 3.12% and increase Apdex from 88.56% to 97.46%. Further, React with Fox uses almost a quarter of the

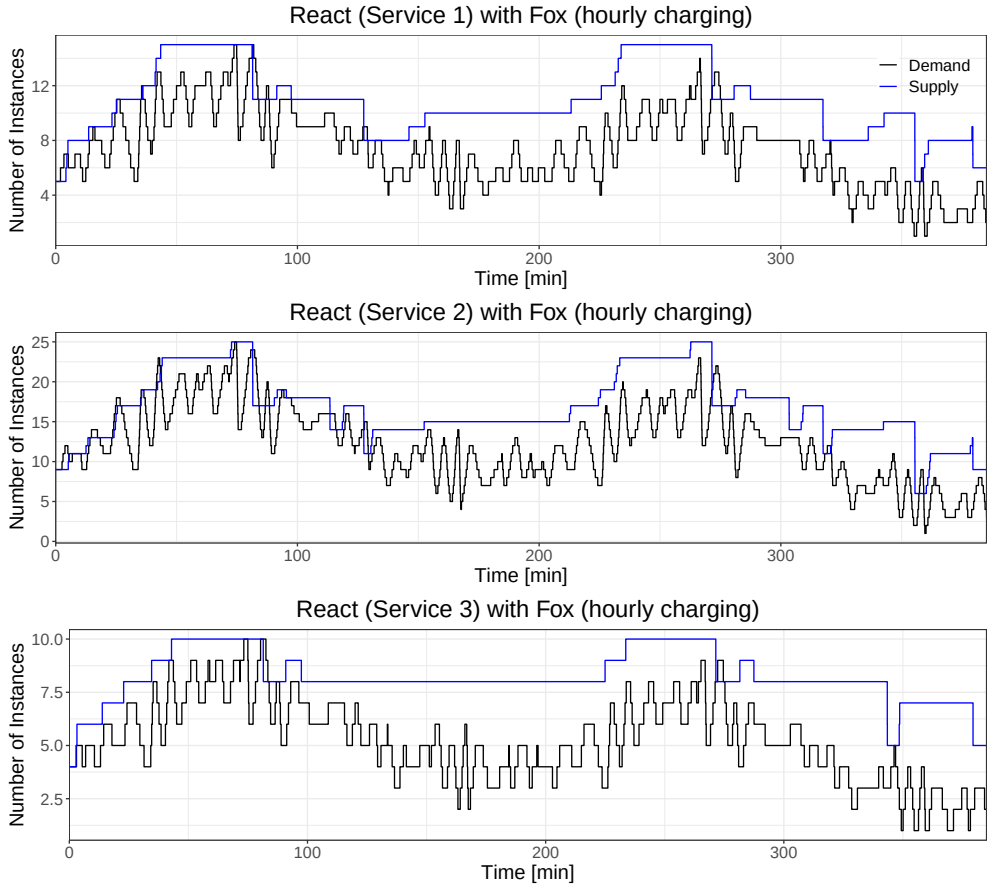

Figure 11.9: Scaling behavior of React with Fox on the BibSonomy trace with hourly charging scheme.

original adjustments. In terms of cost-savings based on hourly charging, React saves 44.52% of the accounted instance time compared to the naïve approach (i.e., using all available resources throughout the experiment) while React with Fox saves only 28.31%. In other words, both approaches use fewer instances than the naïve approach, but due to the fact that Fox only scales down instances if it is appropriate, Fox keeps instances longer than React. However, due to this strategy, Fox saves 26.29% of the charged cost compared to the naïve approach while React only saves 5.14%. In summary, compared to stand-alone React, React with Fox is able to reduce SLO violations and save more money while increasing the auto-scaling worst-case deviation a little.

Table 11.11: Comparison of React on the BibSonomy trace with and without Fox based on hourly charging.

Measure	React	Fox
θ_{U,S_1} [%]	2.65	**0.45**
θ_{O,S_1} [%]	**33.29**	66.08
τ_{U,S_1} [%]	16.48	**3.04**
τ_{O,S_1} [%]	**68.05**	93.57
υ_{S_1} [%]	20.35	**15.56**
θ_{U,S_2} [%]	6.10	**0.99**
θ_{O,S_2} [%]	**20.80**	54.79
τ_{U,S_2} [%]	35.34	**6.67**
τ_{O,S_2} [%]	**52.18**	88.54
υ_{S_2} [%]	30.03	**25.37**
θ_{U,S_3} [%]	2.22	**0.15**
θ_{O,S_3} [%]	**27.93**	82.97
τ_{U,S_3} [%]	13.45	**1.12**
τ_{O,S_3} [%]	**57.97**	96.03
υ_{S_3} [%]	11.15	**10.07**
ς [%]	**55.71**	60.89
SLOs [%]	12.05	**3.12**
Apdex	89.56	**97.46**
#Adaptations	509	136
Avg. #Inst.	28.63	34.08
Π_a [%]	**-44.52**	-28.31
Π_c [%]	-5.14	**-26.29**

11.4.3 Fox with Hourly Charging Scheme

In this section, we compare the different auto-scalers with and without Fox based on hourly charging. Table 11.12 shows the elasticity, user-oriented, and cost-saving (based on hourly charging) measures for the BibSonomy (React, Adapt, and Reg) and the IBM (React) trace. Each row shows a measure and every two columns an auto-scaler without and with Fox. For all underprovisioning related measures and υ, the respective scaling aspect is improved for each auto-scaler while Fox is deployed. Moreover, the SLO violations are significantly reduced, and consequently, the Apdex is increased. In terms of resource management, Fox also reduced the number of adaptations. For instance,

Adapt induces 745 scaling actions, and while using Fox, only 44 are triggered. This reduction is also reflected by the cost-saving rate for accounted instance times. Here, Fox allocates more instances than the stand-alone auto-scaler but less than the naïve approach. However, in all scenarios, Fox is able to save more costs than the auto-scalers without Fox. Especially, in the case of Adapt, Fox can reduce the cost for accounted instance time by 44.85% as Adapt is 39.71% more expensive than running all instances over the entire measurement due to the 745 adaptations.

Table 11.12: Comparison of auto-scalers with and without Fox based on hourly charging.

Measures	BibSonomy						IBM	
	React	w/ Fox	Adapt	w/ Fox	Reg	w/ Fox	React	w/ Fox
θ_{U,S_1} [%]	2.65	0.45	11.50	0.17	9.70	0.49	1.93	1.11
θ_{O,S_1} [%]	33.29	66.08	19.38	143.20	13.70	52.01	86.31	90.91
τ_{U,S_1} [%]	16.48	3.04	45.69	0.99	36.11	4.29	8.33	5.59
τ_{O,S_1} [%]	68.05	93.57	36.71	98.48	40.26	87.85	85.87	90.11
υ_{S_1} [%]	20.35	15.56	23.03	14.65	19.32	15.34	17.19	15.89
θ_{U,S_2} [%]	6.10	0.99	17.78	0.21	14.00	1.69	4.76	2.07
θ_{O,S_2} [%]	20.80	54.79	11.85	133.39	9.44	43.65	66.11	112.59
τ_{U,S_2} [%]	35.34	6.67	63.47	1.50	55.57	12.92	16.43	8.29
τ_{O,S_2} [%]	52.18	88.54	24.93	95.44	32.09	80.75	78.67	88.62
υ_{S_2} [%]	30.03	25.37	30.12	23.68	29.21	24.24	26.71	25.11
θ_{U,S_3} [%]	2.22	0.15	25.69	0.02	16.49	1.34	5.82	0.97
θ_{O,S_3} [%]	27.93	82.97	8.63	121.90	7.82	41.42	96.04	105.07
τ_{U,S_3} [%]	13.45	1.12	64.00	0.11	55.46	9.34	13.61	3.97
τ_{O,S_3} [%]	57.97	96.03	17.19	98.00	17.53	75.93	79.49	93.62
υ_{S_3} [%]	11.15	10.07	15.34	9.68	11.32	10.07	11.00	10.52
ς [%]	55.71	60.89	55.01	79.73	51.72	54.46	65.81	69.56
SLOs [%]	12.05	3.12	28.39	12.19	21.76	4.07	11.73	3.73
Apdex	89.56	97.46	71.72	87.83	78.58	96.47	88.81	96.69
#Adaptations	509	136	745	44	421	87	253	143
Avg. #Inst.	28.63	34.08	22.57	37.43	22.43	34.27	28.95	28.83
Π_a [%]	-44.52	-28.31	-57.30	-5.25	-53.87	-35.48	-58.86	-50.75
Π_c [%]	-5.14	-26.29	39.71	-5.14	-5.14	-35.43	-42.29	-49.71

11.4.4 Fox with Two-Phase Charging Scheme

Analogous to Section 11.4.4, we compare the auto-scalers with and without Fox based on two-phase charging. Table 11.13 shows the elasticity, user-oriented, and cost-saving (based on a two-phase charging) measures for the BibSonomy

(React, Adapt, and Reg) and the IBM (React) trace. Each row shows a measure and every two columns an auto-scaler without and with Fox. While applying Fox, all auto-scalers improve measures concerning underprovisioning (except τ_{U,S_1} for React on IBM). As Fox smooths the scaling, also υ for all auto-scalers has a better value. Like the hourly charging scenario, Fox also reduces the SLO violations and increases Apdex for all auto-scalers. Moreover, while applying Fox, the auto-scalers perform fewer adaptations. For both cost-saving rates, the stand-alone auto-scalers save more costs than applying Fox. This can be explained by the pricing scheme, as every minute is charged separately and there is no rounding to the next full hour as done for the hourly charging.

Table 11.13: Comparison of auto-scalers with and without Fox based on two-phase charging.

Measures	BibSonomy						IBM	
	React	w/ Fox	Adapt	w/ Fox	Reg	w/ Fox	React	w/ Fox
θ_{U,S_1} [%]	2.65	0.62	11.50	0.08	9.70	1.10	1.93	1.10
θ_{O,S_1} [%]	33.29	57.91	19.38	144.25	13.70	38.98	86.31	38.98
τ_{U,S_1} [%]	16.48	3.92	45.69	0.51	36.11	8.42	8.33	8.42
τ_{O,S_1} [%]	68.05	91.01	36.71	98.96	40.26	76.14	85.87	76.14
υ_{S_1} [%]	20.35	15.95	23.03	14.74	19.32	15.73	17.19	15.73
θ_{U,S_2} [%]	6.10	1.53	17.78	1.97	14.00	2.55	4.76	2.55
θ_{O,S_2} [%]	20.80	47.25	11.85	120.05	9.44	36.32	66.11	36.32
τ_{U,S_2} [%]	35.34	8.17	63.47	5.62	55.57	19.86	16.43	19.86
τ_{O,S_2} [%]	52.18	85.45	24.93	88.85	32.09	71.26	78.67	71.26
υ_{S_2} [%]	30.03	25.24	30.12	24.94	29.21	24.81	26.71	24.81
θ_{U,S_3} [%]	2.22	0.47	25.69	10.08	16.49	1.67	5.82	1.67
θ_{O,S_3} [%]	27.93	62.66	8.63	59.69	7.82	30.97	96.04	30.97
τ_{U,S_3} [%]	13.45	2.65	64.00	16.22	55.46	11.02	13.61	11.02
τ_{O,S_3} [%]	57.97	91.71	17.19	79.17	17.53	67.23	79.49	67.23
υ_{S_3} [%]	11.15	10.85	15.34	10.24	11.32	10.20	11.00	10.20
ς [%]	55.71	55.79	55.01	87.33	51.72	51.27	65.81	51.27
SLOs [%]	12.05	3.33	28.39	20.04	21.76	11.02	11.73	11.02
Apdex	89.56	96.72	71.72	80.71	78.58	89.34	88.81	89.34
#Adaptations	509	183	745	103	421	136	253	136
Avg. #Inst.	28.63	31.60	22.57	36.43	22.43	31.26	28.95	31.26
Π_a [%]	-44.52	-31.74	-57.30	-11.48	-53.87	-38.91	-58.86	-52.50
Π_c [%]	-44.06	-31.56	-56.56	-11.25	-52.81	-38.75	-58.75	-52.62

11.4.5 Summary of the Results and Threats to Validity

While evaluating our contribution Fox on different auto-scalers and traces, Fox is able to improve the elasticity aspects (i) underprovisioning accuracy, (ii) underprovisioning timeshare, and (iii) instability. This improvement leads to lower SLO violations and higher user satisfaction. However, the price for this optimization was an increase in overprovisioning reflected in higher auto-scaling worst-case deviation. In terms of cost-saving, Fox significantly reduces the allocated costs and, at the same time, increases the allocated instance time in the hourly charging scenario. In the two-phase charging, however, Fox increases both the accounted and charged instance time that can be explained by the nature of this underlying pricing scheme.

As Fox is a mediator between the auto-scaler and the system, the scaling behavior is highly dependent on the used mechanism. Therefore, we used and compared three state-of-the-art auto-scalers. Nevertheless, several other auto-scaler exists, but the experiments did not focus on optimal auto-scaling decisions. Moreover, the revision of the scaling decisions depends on the forecast accuracy of each forecast. As sNaïve (see Section 3.1.1) is a simple forecasting method, better methods could be used to reduce variations in the results. To sum up, due to the choice of the auto-scalers, traces, forecasting method, and application, the results may not be generalized to other auto-scalers or applications. However, the results have shown that Fox behaves as desired and can reduce the charged costs while increasing the accounted instance time and reducing the SLO violation rate.

11.5 Benchmarking of the Chamulteon Approach

In this section, we benchmark Chamulteon against state-of-the-art auto-scalers. First, we describe the experimental setup in Section 11.5.1. Then, we explain how to interpret the results of the experiments in Section 11.5.2. To investigate how the auto-scalers behave on different setups, we compare their scaling on Docker containers and VMs in Section 11.5.3. Afterwards, the scalability of the auto-scalers is investigated in Section 11.5.4. Finally, we sum up the results and discuss threats to validity in Section 11.5.5.

11.5.1 Experimental Description

To benchmark Chamulteon in authentic circumstances, Chamulteon competed with the auto-scalers (i) Adapt, (ii) Hist, (iii) React, and (iv) Reg. Each auto-scaler scaled the Verification application, which was either deployed on VMs or

as Docker containers. We used c.large VMs in the CSCP environment (see Section 11.1.3). In the Docker scenario, a Kubernetes (Rancher[14] v2.1.0 + kubectl v1.11.3) cluster was deployed on the VMs. Within the Kubernetes cluster, we run Docker (v17.03.2-ce) containers. For stressing the application, the BibSonomy and Wiki traces (see Section 11.1.1) were used. In the VM scenario, the traces were accelerated so that one day in the workload corresponds to six hours of experiment time. In the Docker setting, each day was speedup to last one hour experiment time. The scaling for the VM setting took place every 2 minutes and for the Docker setting every minute due to the faster provision time of containers. For the scaling, the auto-scaler got VM specific information (such as the amount of running VMs and the average CPU utilization) and application-specific information (such as request arrival rates) as input. Note that in the following experiments, the cost-awareness component was deactivated. On the one hand, we want to investigate and compare the scaling performance of Chamulteon. On the other hand, the component was evaluated separately in Section 11.4.

11.5.2 Introduction to the Results

Similar to Section 11.3.2, we first introduce the experiment format before discussing the results. Figure 11.10 shows the Wiki track scaled by Reg. Each sub-figure shows the scaling behavior for a service. In each graph, the horizontal axis shows the time of measurement in minutes, and the vertical axis shows the number of concurrently running instances (i.e., VMs or containers). The black curves represent the resource demand (determined by BUNGEE, see Section 4.2.2) and the blue curves the amount of supplied VMs by the auto-scaler. If the supply curve falls below the demand curve, there are too few instances provisioned. In case the supply curve exceeds the demand curve, too many instances are supplied.

We show this figure as it is a good example of bottleneck shifting (see Chapter 9). While the first service is scaled after one minute to satisfy the demand, the second service is scaled one minute later, and the last service is scaled two minutes afterward. This behavior can be explained as follows: During the first minute, the first service can handle a lower number of requests, so the following services also receive fewer requests. After scaling the first service, the second service cannot process all incoming requests, so the last service receives a lower number of requests again. This effect can be observed for up to 15 minutes, as each service's resource supply increases more slowly than that of the previous

[14]Rancher: `https://rancher.com/`

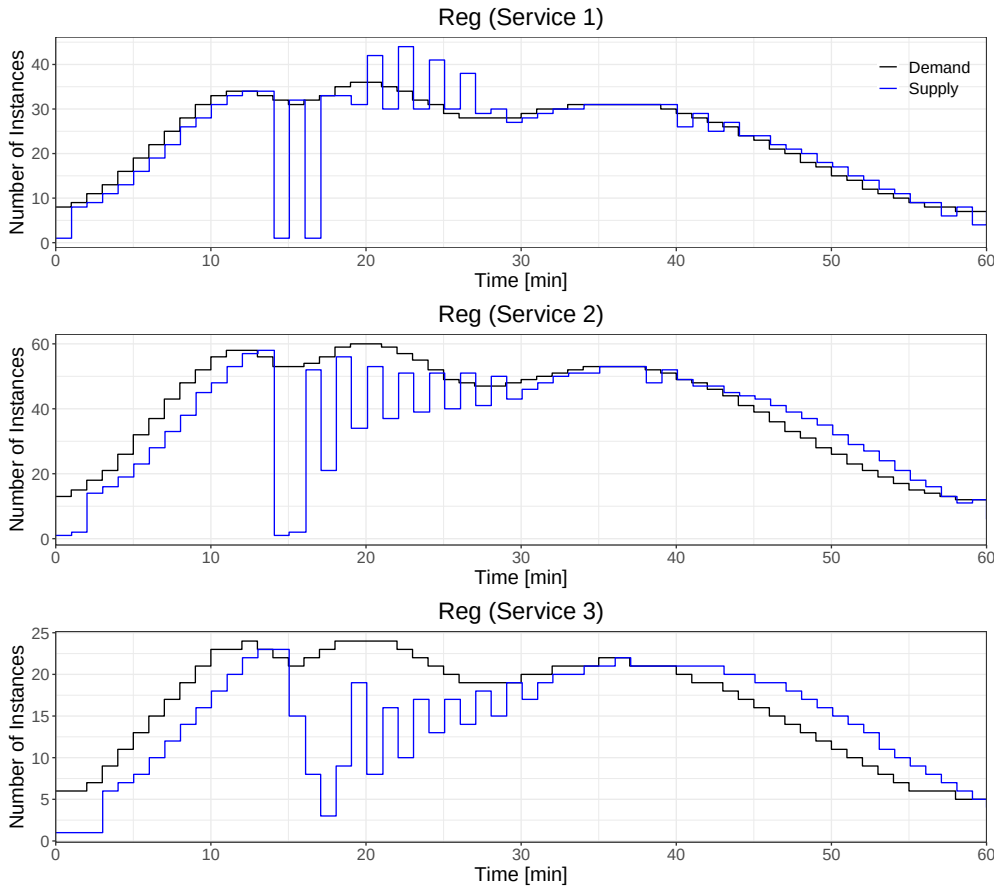

Figure 11.10: Scaling behavior of Reg on the Wiki trace.

services. Similar to the observations in the articles (Papadopoulos et al., 2016; Ilyushkin et al., 2018) and Section 11.3.2, Reg exhibits a high rate of oscillations (between minute 16 and 27) that cannot be explained. After minute 33, Reg gets stable and tends to overprovision. Figure 11.11, which is structured exactly like Figure 11.10, shows the scaling behavior of Chamulteon. In contrast, to Reg, the scaling behavior of Chamulteon exhibits neither bottleneck shifting nor oscillations. Due to the configuration of Chamulteon, the system is always slightly overprovisioned so that almost all requests can be served within the SLOs.

To assess the scaling behavior of Chamulteon, Reg, and the other auto-scalers, we investigate the respective elasticity and user-oriented measures shown in

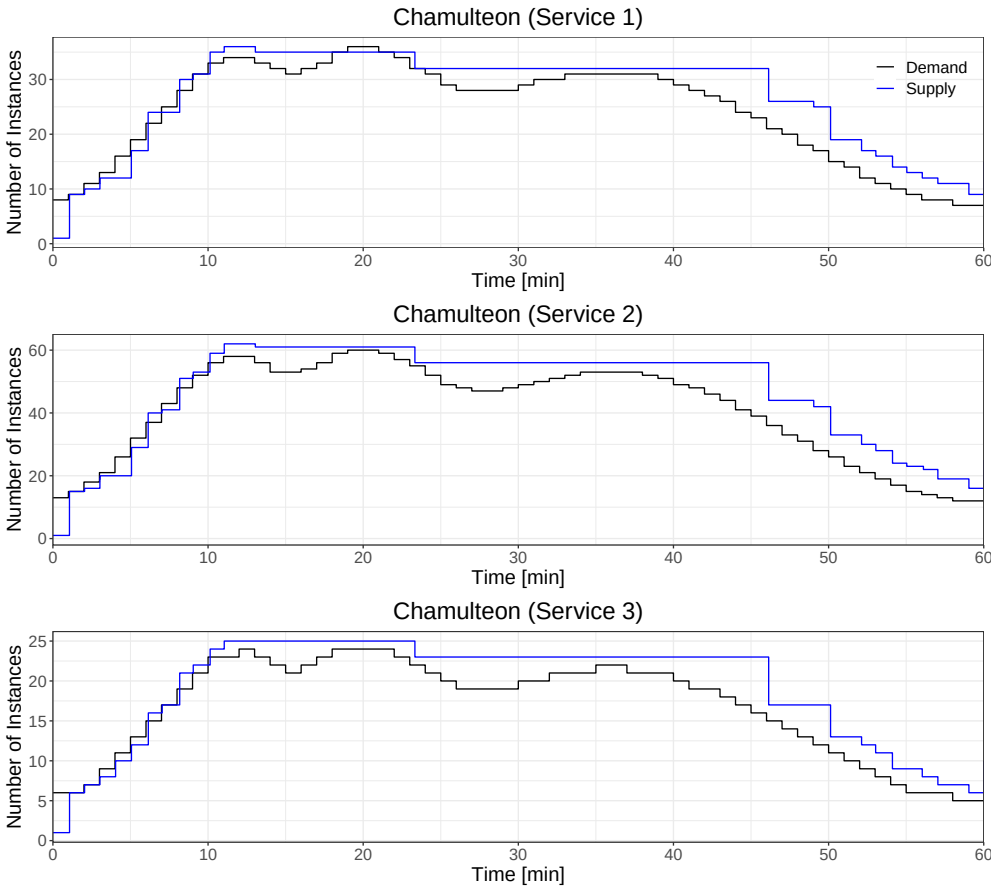

Figure 11.11: Scaling behavior of Chamulteon on the Wikipedia trace.

Table 11.14. Each row represents a measure and each column an auto-scaler. A short description of each measure can be found in Section 11.1.5. For instance, Reg exhibits for all services the worst values for θ_U and τ_U. Consequently, Reg also has the highest SLO violations (37.32%) and the lowest Apdex (63.15%). In contrast, Chamulteon has for all services the best values for θ_U and τ_U, also leading to the best values for SLOs (6.17%) as well as Apdex (93.05%).

11.5.3 Docker vs. VM Scaling

In this section, we investigate how well Chamulteon scales in different setups compared to the other auto-scalers. In the first scenario, the application was deployed on Docker containers. In the second scenario, the application was

Table 11.14: Comparison of the auto-scalers on the Wiki trace (Docker).

Measure	Chamulteon	Adapt	Hist	Reg	React
θ_{U,S_1} [%]	**3.83**	8.14	4.75	10.84	4.04
θ_{O,S_1} [%]	30.00	13.17	36.60	**6.94**	13.38
τ_{U,S_1} [%]	**15.43**	29.81	21.62	46.34	22.99
τ_{O,S_1} [%]	84.38	56.71	75.00	**44.62**	71.83
υ_{S_1} [%]	**18.55**	25.28	18.89	26.39	23.06
θ_{U,S_2} [%]	**3.87**	12.44	6.96	15.77	6.06
θ_{O,S_2} [%]	31.16	9.30	32.70	**7.50**	10.09
τ_{U,S_2} [%]	**15.43**	37.99	23.48	53.76	26.33
τ_{O,S_2} [%]	84.46	58.53	71.79	**40.87**	65.39
υ_{S_2} [%]	20.16	28.61	**19.72**	29.44	26.94
θ_{U,S_3} [%]	**3.32**	17.33	9.40	19.39	5.77
θ_{O,S_3} [%]	26.68	**8.06**	26.86	11.84	15.69
τ_{U,S_3} [%]	**13.94**	36.33	31.71	56.52	21.54
τ_{O,S_3} [%]	84.48	49.55	61.53	**38.01**	71.85
υ_{S_3} [%]	17.20	22.78	**15.00**	24.44	20.83
ς [%]	**51.71**	51.85	55.58	54.13	51.82
SLOs [%]	**6.17**	24.20	12.53	37.32	11.19
Apdex	**93.05**	76.84	87.56	63.15	88.99
#Adaptations	75	139	55	154	117
Avg. #Inst.	71.06	77.21	94.70	73.54	85.70

hosted on VMs. In both scenarios, the application is stressed with the Wiki trace and requires a maximum of 20 containers or VMs. The results are listed in Table 11.14 for the Docker and in Table 11.15 for the VM setup. The rows show the measures (user-oriented and elasticity) and the columns the auto-scalers in each table. The best values are highlighted in bold. In the Docker scenario, Chamulteon has the best values for 7 of 15 (three services with every five measures) single elasticity measures. As focusing only on the individual measures may lead to ambiguous results, we take ς into account for ranking the auto-scalers. The lowest value for ς is achieved by Chamulteon (51.71%) closely followed by React (51.82%) and Adapt (51.58%). Besides the lowest ς, Chamulteon also has the lowest SLO violations (6.17%) and the highest value for Apdex (93.05%).

In contrast to the Docker scenario, React has the best values for 9 of 15 individual elasticity measures (see Table 11.15). However, React overprovisions

the system almost the whole time, as indicated by the high θ_O values. To this end, React exhibits the worst value for ς (52.56). The best value for ς is achieved by Reg (33.21%) followed by Adapt (33.25%), Hist (34.61%), and Chamulteon (35.92%). In terms of SLOs and Apdex, React has the best value for both measures closely followed by Chamulteon, while Reg exhibits the second-worst values for both measures.

In summary, Chamulteon and React achieved the best user-oriented measures in both scenarios. In terms of the auto-scaling worst-case deviation, Chamulteon has the best value in the Docker scenario and shows, in the second scenario, only a slight deviation from the winner. In contrast, React has by far the worst value for this measure in the second scenario. Although both mechanisms tend to overprovision, Chamulteon delivers more robust performance in both scenarios compared with React.

Table 11.15: Comparison of the auto-scalers on the Wiki trace (VM).

Measure	Chamulteon	Adapt	Hist	Reg	React
θ_{U,S_1} [%]	1.11	3.22	2.52	3.47	**0.13**
θ_{O,S_1} [%]	16.25	**6.58**	21.77	10.04	51.53
τ_{U,S_1} [%]	3.82	10.60	10.33	9.33	**0.63**
τ_{O,S_1} [%]	67.51	**18.58**	35.58	24.07	95.17
υ_{S_1} [%]	2.33	4.19	1.99	2.33	**1.64**
θ_{U,S_2} [%]	0.41	9.19	3.34	7.38	**0.27**
θ_{O,S_2} [%]	12.07	**3.46**	24.86	7.59	28.29
τ_{U,S_2} [%]	2.32	35.98	16.61	25.54	**1.73**
τ_{O,S_2} [%]	63.94	**19.17**	46.70	28.07	91.72
υ_{S_2} [%]	3.37	5.88	2.94	4.36	**2.46**
θ_{U,S_3} [%]	1.08	16.58	7.70	11.16	**0.05**
θ_{O,S_3} [%]	18.55	**7.83**	24.91	12.94	62.53
τ_{U,S_3} [%]	2.75	46.30	20.01	37.23	**0.09**
τ_{O,S_3} [%]	50.38	**9.47**	33.84	19.79	95.30
υ_{S_3} [%]	1.64	4.67	1.82	1.51	**1.47**
ς [%]	35.92	33.35	34.61	**33.21**	52.56
SLOs [%]	1.98	19.09	5.12	12.55	**1.00**
Apdex	98.54	81.60	95.64	88.20	**99.05**
#Adaptations	110	299	102	132	66
Avg. #Inst.	13.98	14.00	15.13	13.47	16.45

11.5.4 Scalability

In this experiment, we investigate the scalability of the auto-scalers. In other words, we used and scaled the BibSonomy trace to stress the application so that the first scenario requires a maximum of 60 containers (small) and the second scenario requires a maximum of 120 containers (large). The results of the small setup are listed in Table 11.16. Each row shows a measure (user-oriented and elasticity) and each column an auto-scaler. The best values are highlighted in bold. In terms of the individual elasticity measures, Reg (6 of 15) has most of the best values followed by Chamulteon (4 of 15). However, Chamulteon exhibits the best value for σ (47.54%), SLOs (7.28%), and Apdex (93.84%). In the large scenario (see Table 11.17), Chamulteon outperforms the other auto-scalers. More precisely, Chamulteon has the best values for 9 of 15 single elasticity measures, the best value for ς (50.52%), the lowest SLO violations (9.58%), and the highest Apdex (91.74%).

To quantify the scalability, we calculated for each single elasticity measure the relative deviation between the two scenarios. Then, we averaged the deviations and used this value for assessing the scalability. The lowest scalability has Chamulteon (14.07%) followed by Hist (17.47%), Reg (22.58%), Adapt (30.04%), and React (51.81%).

11.5.5 Summary of the Results and Threats to Validity

We conducted four different experiments to evaluate Chamulteon. More precisely, we varied the deployment (Docker vs. VM), the scale (20, 60, and 120 instances), and the workload trace (Wiki vs. BibSonomy). In three of these four experiments, Chamulteon exhibits the best user-oriented and elasticity measures. In the remaining scenario, Chamulteon has the second-best user-oriented measures. In the scalability experiment, Chamulteon also achieves the lowest deviation of 14.07%. In contrast to the other auto-scalers, the bottleneck shifting effect was not observed when Chamulteon scaled the application. That is, the coordinated scaling of Chamulteon is able to counter the bottleneck shifting. Although React is a reactive auto-scaler, it achieves the best user-oriented measures in one scenario and the second-best user-oriented measures in two scenarios. However, since React tends to overprovision the system, it also exhibits the worst auto-scaling worst-case deviation in two scenarios. Also, React's scalability differs by 51.81% between the small and large scenarios. In one experiment, Hist has the second-best user-oriented measure due to its tendency to overprovision. In terms of scalability, Hist has the second-lowest deviation. In contrast to the other auto-scalers, Reg and Adapt tend to underprovision the

Table 11.16: Comparison of the auto-scalers on the BibSonomy trace (small setup, i.e., 60 containers).

Measure	Chamulteon	Adapt	Hist	Reg	React
θ_{U,S_1} [%]	2.27	7.00	3.43	6.44	**2.03**
θ_{O,S_1} [%]	17.99	13.60	20.43	**7.13**	16.02
τ_{U,S_1} [%]	8.47	30.14	18.20	25.21	**5.20**
τ_{O,S_1} [%]	76.53	48.38	66.77	**40.33**	70.08
υ_{S_1} [%]	19.72	24.44	**15.56**	21.94	22.78
θ_{U,S_2} [%]	**2.10**	9.79	5.68	13.90	4.86
θ_{O,S_2} [%]	21.07	5.96	20.62	**3.89**	11.15
τ_{U,S_2} [%]	**8.53**	48.43	26.46	54.78	29.95
τ_{O,S_2} [%]	83.31	36.85	63.57	**31.87**	55.47
υ_{S_2} [%]	20.83	26.39	**17.50**	26.11	26.94
θ_{U,S_3} [%]	**1.75**	12.16	7.18	12.74	3.72
θ_{O,S_3} [%]	18.18	8.38	15.70	**3.76**	17.39
τ_{U,S_3} [%]	**5.12**	43.07	26.62	48.23	8.32
τ_{O,S_3} [%]	76.63	36.87	53.34	**24.78**	80.08
υ_{S_3} [%]	17.78	21.11	**12.78**	17.50	14.72
ς [%]	**47.54**	51.16	47.90	50.20	57.23
SLOs [%]	**7.28**	17.77	11.87	23.41	10.54
Apdex	**93.84**	85.26	89.85	79.73	91.35
#Adaptations	77	122	25	99	94
Avg. #Inst.	44.57	39.67	45.13	37.49	43.82

system. Consequently, both mechanisms exhibit the worst user-oriented measures. In summary, Chamulteon exhibits the best auto-scaling performance and reliability compared to the competing methods regardless of the deployment and workload.

Since Chamulteon is designed to focus on user satisfaction, the experiments show a slight overprovisioning. This behavior seems to contradict the cost-efficiency concept, but the cost awareness component was deactivated in the experiments presented in this section. Furthermore, in our view, cost-efficiency does not imply providing as few instances as possible but trying to use the accounted instances as efficiently as possible. For example, in the case of hourly charging, resources would not be released when they are paid and could be required again in a few minutes to avoid duplicate costs for the same resources. Although our experimental analysis covered different scenarios and

Table 11.17: Comparison of the auto-scalers on the BibSonomy trace (large setup, i.e., 120 containers).

Measure	Chamulteon	Adapt	Hist	Reg	React
θ_{U,S_1} [%]	**1.91**	10.92	3.74	8.81	3.86
θ_{O,S_1} [%]	19.34	10.65	26.67	**7.31**	11.65
τ_{U,S_1} [%]	**6.91**	40.49	23.29	38.14	26.51
τ_{O,S_1} [%]	88.14	**50.18**	73.39	50.29	60.28
υ_{S_1} [%]	**13.61**	30.56	18.89	27.78	28.61
θ_{U,S_2} [%]	**2.04**	19.85	5.71	17.16	6.73
θ_{O,S_2} [%]	20.44	6.18	24.97	**3.75**	7.37
τ_{U,S_2} [%]	**8.59**	56.64	28.44	64.70	44.68
τ_{O,S_2} [%]	89.74	36.84	66.92	**33.34**	48.38
υ_{S_2} [%]	**13.33**	30.56	20.00	29.44	29.44
θ_{U,S_3} [%]	**3.20**	21.70	8.33	20.12	6.16
θ_{O,S_3} [%]	18.76	6.37	22.23	**2.62**	9.04
τ_{U,S_3} [%]	**5.14**	55.19	33.12	63.39	26.69
τ_{O,S_3} [%]	91.34	29.72	56.73	**24.50**	56.54
υ_{S_3} [%]	**13.42**	26.94	16.94	24.72	18.89
ς [%]	**50.52**	56.99	54.78	60.20	55.49
SLOs [%]	**9.58**	33.17	12.85	36.27	15.25
Apdex	**91.74**	67.34	87.48	63.86	85.29
#Adaptations	76	164	48	143	125
Avg. #Inst.	89.07	69.82	91.67	69.90	80.66

we benchmarked different auto-scalers, the results may not be generalizable to other types of applications or closed-source auto-scalers. However, for the evaluated competing auto-scalers, a similar behavior was observed in studies on auto-scaler evaluation (Papadopoulos et al., 2016; Ilyushkin et al., 2018). Also, we used the same auto-scaler for each service of the application. In fact, it is possible to use different mechanisms for each service. However, the choice of auto-scalers and the order in which they are deployed is a crucial challenge. In theory, it is possible to achieve better results by experimenting with different combinations of auto-scalers in different orders. In practice, however, it is challenging to find an optimal configuration. Moreover, there is no guarantee that this configuration would remain static as the system and workload evolve.

11.6 Concluding Remarks

In this chapter, we benchmarked Chamulteon and its components in scenarios covering five different workloads, four different applications, and three different cloud environments. In the first experiments, we investigated the impact of service demand as scaling indicator. We showed that a service demand-based auto-scaling approach is not based on knowledge of the bottleneck resource and can be configured independently of the application. In other words, the service demand is an independent and reliable input for auto-scalers. Then, in a broad competition, we benchmarked the Chameleon approach in different scenarios against state-of-the-art auto-scalers. In these experiments, the competing auto-scalers behave similarly to former studies (Papadopoulos et al., 2016; Ilyushkin et al., 2018), and no method outperforms the others. In contrast, Chameleon exhibits in all experiments the best scaling behavior. Afterward, we evaluated the cost-awareness component of Chamulteon called Fox to investigate RQ 7 "*How can scaling decisions be adjusted so that the charged costs in a public cloud environment are minimized?*". These experiments show that Fox can significantly reduce the charged costs while increasing the allocated instance time for an hourly pricing scheme. This improvement leads to a significant reduction of the SLO violations in exchange for a slightly worse auto-scaling performance. Lastly, we benchmark Chamulteon on different setups against state-of-the-art auto-scalers to investigate RQ 8 "*How to enable coordinated scaling of applications comprising multiple services?*". In three of these four scenarios, Chamulteon exhibits the best user-oriented and elasticity measures, and in the remaining scenario, Chamulteon has the second-best user-oriented measures. Chamulteon also has the least variation between the different setup sizes. The experiments with Chameleon and Chamulteon show that the usage of reactive and proactive decisions in conjunction with decision resolution management improves the auto-scaling behavior (RQ 6 "*What is a meaningful combination of proactive and reactive scaling techniques to minimize the risk of auto-scaling in operation?*").

Part IV

Conclusion

Chapter 12

Thesis Summary

Nowadays, we are living in a fast-paced world, and thus many domains are subject to trends and varying requirements. For instance, cloud environments have to cope with load fluctuations and respective rapid and unexpected changes in the computing resource demands. Since reacting to changes once they are observed introduces an inherent delay, the future resource demand must be "foreseen" to identify necessary steps in advance. A useful and established technique in this context is time series forecasting, which is also applied in many other domains. Although time series forecasting enables the proactive auto-scaling of the required resources in cloud environments, business-critical applications are still run with highly overprovisioned resources to guarantee a stable and reliable service operation. This strategy is pursued mainly due to two major problems of existing work: First, no fully automated and generic forecasting approach exists that can effectively combine existing forecasting methods in a way to leverage their strengths and avoid their weaknesses, providing accurate forecasts with a reliable time-to-result. Second, existing cloud auto-scalers are distrusted to provide reliable and cost-effective autonomic resource management for modern cloud environments due to the concern that inaccurate or delayed adaptations may result in financial losses. To approach both problems, we defined three goals that were addressed within this thesis:

Goal I: *Provide a forecasting benchmark to establish a level playing field for evaluating and comparing the performance of forecasting methods in a broad setting covering a diverse set of evaluation scenarios.*

Contribution I: *Forecasting Benchmark*
In Chapter 7, we propose a novel benchmark that automatically evaluates and ranks forecasting methods based on their performance in a diverse set of evaluation scenarios. The benchmark comprises four different use cases, each covering 100 heterogeneous time series taken from differ-

ent domains. The data set was assembled from publicly available time series and was designed to exhibit much higher diversity than existing forecasting competitions. Besides proposing a new data set, we introduce two new measures that describe different aspects of a forecast. We applied the developed benchmark to evaluate Telescope.

Goal II: *Provide a fully automated and generic hybrid forecasting method that automatically extracts relevant information from a given time series and uses it to combine existing methods in a way to provide high forecast accuracy coupled with a low time-to-result variance.*

Contribution II: *Telescope*
In Chapter 8, we introduce a novel machine learning-based forecasting approach that automatically retrieves relevant information from a given time series. More precisely, Telescope automatically extracts intrinsic time series features and then decomposes the time series into components, building a forecasting model for each of them. Each component is forecast by applying a different method and then the final forecast is assembled from the forecast components by employing a regression-based machine learning algorithm. In more than 1300 hours of experiments benchmarking 15 competing methods (including approaches from Uber and Facebook) on 400 time series, Telescope outperformed all methods, exhibiting the best forecast accuracy coupled with a low and reliable time-to-result. Compared to the competing methods that exhibited, on average, a forecast error (more precisely, the symmetric mean absolute forecast error) of 29%, Telescope exhibited an error of 20% while being 2556 times faster. In particular, the methods from Uber and Facebook exhibited an error of 48% and 36%, and were 7334 and 19 times slower than Telescope, respectively.

Goal III: *Develop a hybrid auto-scaler enabling the coordinated scaling of applications comprising multiple services by combining proactive scaling (based on the developed forecasting method) with*

reactive scaling as a fallback mechanism in order to provide maximum reliability of resource adaptations.

Contribution III: *Chamulteon*
In Chapter 9, we present a hybrid auto-scaler that combines proactive and reactive techniques to scale distributed cloud applications comprising multiple services in a coordinated and cost-effective manner. More precisely, proactive adaptations are planned based on forecasts of Telescope, while reactive adaptations are triggered based on actual observations of the monitored load intensity. To solve occurring conflicts between reactive and proactive adaptations, a complex conflict resolution algorithm is implemented. Moreover, when deployed in public cloud environments, Chamulteon reviews adaptations with respect to the cloud provider's pricing scheme in order to minimize the charged costs. In more than 400 hours of experiments evaluating five competing auto-scaling mechanisms in scenarios covering five different workloads, four different applications, and three different cloud environments, Chamulteon exhibited the best auto-scaling performance and reliability while at the same time reducing the charged costs. The competing methods provided insufficient resources for (on average) 31% of the experimental time; in contrast, Chamulteon cut this time to 8% and the SLO (service level objective) violations from 18% to 6% while using up to 15% less resources and reducing the charged costs by up to 45%.

The contributions of this thesis can be seen as major milestones in the domain of time series forecasting and cloud resource management. (i) This thesis is the first to present a forecasting benchmark that covers a variety of different domains with a high diversity between the analyzed time series. Based on the provided data set and the automatic evaluation procedure, the proposed benchmark contributes to enhance the comparability of forecasting methods. The benchmarking results for different forecasting methods enable the selection of the most appropriate forecasting method for a given use case. (ii) Telescope provides the first generic and fully automated time series forecasting approach that delivers both accurate and reliable forecasts while making no assumptions about the analyzed time series. Hence, it eliminates the need for expensive, time-consuming, and error-prone procedures, such as trial-and-error searches

or consulting an expert. This opens up new possibilities especially in time-critical scenarios, where Telescope can provide accurate forecasts with a short and reliable time-to-result.

Although Telescope was applied for this thesis in the field of cloud computing, there is absolutely no limitation regarding the applicability of Telescope in other domains, as demonstrated in the evaluation. Moreover, Telescope, which was made available on GitHub, is already used in a number of interdisciplinary data science projects, for instance, predictive maintenance in an Industry 4.0 context, heart failure prediction in medicine, or as a component of predictive models of beehive development. (iii) In the context of cloud resource management, Chamulteon is a major milestone for increasing the trust in cloud auto-scalers. The complex resolution algorithm enables reliable and accurate scaling behavior that reduces losses caused by excessive resource allocation or SLO violations. In other words, Chamulteon provides reliable online adaptations minimizing charged costs while at the same time maximizing user experience.

Chapter 13

Open Challenges and Outlook

In this thesis, we addressed the two main problems identified: (i) The "No-Free-Lunch Theorem" (Wolpert and Macready, 1997) states that there is no single forecasting method that performs best for all time series, and therefore the choice of an appropriate forecasting method for a given time series is crucial since expert knowledge cannot be fully automated; and (ii) the distrust of auto-scalers as proactive and cost-effective resource management for distributed systems is due to the high operational risk. We are convinced that our approaches, ideas, and solutions presented in this thesis provide a good foundation and offer room for further research. In the following, we identify research topics that may extend our work:

Selecting the best time series transformation automatically
To increase the forecast accuracy, Telescope transforms the time series. More precisely, Telescope applies the Box-Cox transformation (see Section 2.3.2), as this transformation tries to shift the distribution of the data to a normal distribution. However, there are time series where this transformation leads to a deterioration of the forecast. Consequently, different time series require different transformations. Based on the idea of automatic forecasting, the choice of the transformation in Telescope should therefore be performed automatically. Regardless of Telescope, selecting the best transformation should also be done automatically to avoid costly trial-and-error.

Forecasting based on artificial neural networks
In its current version, Telescope performs a forecast based on a single time series. In contrast, the winner (Smyl, 2020) of the M4-Competition trains an artificial neural network on a set of time series. Following this idea, a long short-term memory network can be integrated instead of the current regression-based machine learning methods. Accordingly, Telescope would also have to learn on a set of time series to efficiently use the network. In this case, the time series generator, which is integrated into Telescope, would be beneficial, as it can increase the amount of training data. Another possibility is to use

bagging (Bergmeir et al., 2016) to create a set of similar time series from a single time series for training the network.

Estimating the forecast error
In general, when performing a forecast, the future values are not available. Therefore, the only indicator for the forecast accuracy is the model error. However, this error is not a good estimator, as, for example, overfitting can occur during the training. One approach to enable forecasting confidence is to examine the distribution of the model errors. Based on the distribution, a confidence interval can be assigned to each forecast value. Again, this procedure is not a useful guide, as it is also prone to overfitting. Moreover, the forecast error cannot be inferred from this interval. To sum up, a good estimator for the forecast error poses an ongoing challenge.

Supporting multivariate time series
In this thesis, we focus on univariate time series. Consequently, Telescope supports in its current version only univariate time series. The forecasting of multivariate time series has advantages and disadvantages. On the one hand, there is additional information that can be used to refine the prediction model. On the other hand, each extra piece of information has to be predictable to form the final forecast. For example, the weather helps model the power consumption of a household better, but for the forecast of the power consumption in a month, the weather for this period is also needed.

In production scenarios (e.g., auto-scaling) where forecasts are applied, calendar information is advantageous. As calendar information is available both long in the past and the future, it would be easy to model and find correlations. For example, the Cyber Monday (i.e., the Monday after Thanksgiving) is a special day for e-commerce in the USA. If such information is taken into account, such events can be addressed accordingly.

Supporting non-equidistant time series
By time series, we refer to an ordered collection of values of a quantity obtained over a specific time, whereby the observations are recorded in equidistant time steps. However, in some fields, such as investigating natural disasters, the observations are taken at irregular time intervals. These collections of observations are called unevenly spaced time series. To apply the techniques from time series analysis, these observations are usually converted into a time series, whereby the transformation (e.g., interpolation) may be subject to errors. To avoid the risk of an error-prone transformation, Telescope could be extended to support unevenly distributed time series.

Detecting structural changes in time series
In general, Telescope is prone to poor forecasts when structural changes occur in the time series, as it assumes, for example, that the seasonal pattern does not change over time. Besides the change in the seasonal pattern, further structural changes include level shifts or breakpoints in the trend. To mitigate this problem, structural changes have to be detected automatically. Based on the found changes, the values before the last change can be discarded if there is no regularity in the changes to consider only the time series's current structure for forecasting the time series.

Combining horizontal and vertical scaling
In its current version, Chamulteon supports only horizontal scaling. That is, vertical scaling could be combined with the existing horizontal scaling, where the decision logic must decide which scaling direction is more efficient. For determining which scaling type should take place, the current decision logic of Chamulteon has to be significantly adapted. For example, the complex decision logic could be based on machine learning.

Although there are auto-scalers (see Chapter 6) in the literature that support both types of scaling, the decision between the two options is a challenge, especially concerning nested auto-scaling. For instance, when using Docker container deployed in VMs, the following decision conflicts may occur: Increasing the resources for a VM, increasing the resources of a container running in a VM, adding a container to an existing VM, or adding a container to a new VM.

Considering energy-efficient scaling
Concerning the increasing energy consumption of data centers, Chamulteon could be equipped with another component responsible for optimizing energy consumption. This component should monitor the energy consumption and perform voltage scaling (e.g., tuning the CPU frequency of instances) or the placement of instances so that the underlying hardware runs more energy efficient. The general goal of this component should be the minimization of the energy consumption of the application.

Moving on to new technologies
During the last few years, new computing trends such as micro-services and FaaS have emerged. Any new trend behaves differently from established technologies and generates new requirements. For example, in the case of FaaS (Function as a Service), an auto-scaler may take the warm-up time into account to start a function at an early stage to "warm" the function up. As everything evolves and new technologies will always emerge, auto-scaling should not be considered as solved, as there will always be new challenges.

Considering more prizing schemes
The cost-aware component (see Section 9.4) of Chamulteon supports an hourly charging and two-phase charging. In fact, there are numerous other pricing schemes or other options, such as Amazon's EC2 spot market[1]. Especially the new cloud computing paradigm FaaS offers different options. Here, the price of a function execution depends on the execution time and the provisioned memory size. These pricing options introduce a new complexity compared to the old pricing schemes since the charged costs are no longer dependent on the number of resources (Eismann et al., 2020). In other words, to keep up with the evolution of pricing schemes, Chamulteon should be updated with the latest schemes and should be able to solve the newly emerging conflicts.

[1] AWS EC2 spot market: `https://aws.amazon.com/ec2/spot/`

Back Matter

List of Figures

List of Tables

Bibliography

Adhikari, R. and Agrawal, R. K. (2013). "An Introductory Study on Time Series Modeling and Forecasting". In: *CoRR* abs/1302.6613 (see pages 18, 41, 97).

Adhikari, R., Verma, G., and Khandelwal, I. (2015). "A Model Ranking Based Selective Ensemble Approach for Time Series Forecasting". In: *Procedia Computer Science* 48, pp. 14–21 (see pages 54, 118).

Adya, M., Collopy, F., Armstrong, J. S., and Kennedy, M. (2001). "Automatic Identification of Time Series Features for Rule-Based Forecasting". In: *International Journal of Forecasting* 17.2, pp. 143–157 (see page 56).

Ali-Eldin, A., Tordsson, J., and Elmroth, E. (2012). "An Adaptive Hybrid Elasticity Controller for Cloud Infrastructures". In: *IEEE NOMS 2012*. IEEE, pp. 204–212 (see pages 4, 64, 144, 185).

Arlitt, M. and Jin, T. (2000). "A Workload Characterization Study of the 1998 World Cup Web Site". In: *IEEE Network* 14.3, pp. 30–37 (see page 182).

Assimakopoulos, V. and Nikolopoulos, K. (2000). "The Theta Model: A Decomposition Approach to Forecasting". In: *International journal of forecasting* 16.4, pp. 521–530 (see page 33).

Athanasopoulos, G., Hyndman, R. J., Song, H., and Wu, D. C. (2011). "The Tourism Forecasting Competition". In: *International Journal of Forecasting* 27.3, pp. 822–844 (see pages 62, 86).

Bal, H., Epema, D., Laat, C. de, Nieuwpoort, R. van, Romein, J., Seinstra, F., Snoek, C., and Wijshoff, H. (2016). "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term". In: *IEEE Computer* 49.5, pp. 54–63 (see page 184).

Bates, J. M. and Granger, C. W. (1969). "The Combination of Forecasts". In: *Journal of the Operational Research Society* 20.4, pp. 451–468 (see pages 53, 118).

Bauer, A. (2016). "Design and Evaluation of a Proactive, Application-Aware Elasticity Mechanism". Master Thesis. Am Hubland, Informatikgebäude, 97074 Würzburg, Germany: University of Würzburg (see pages 122, 123).

Bauer, A., Grohmann, J., Herbst, N., and Kounev, S. (2018a). "On the Value of Service Demand Estimation for Auto-Scaling". In: *Proceedings of the 19th International GI/ITG Conference on Measurement, Modelling and Evaluation of Computing Systems* (*MMB*). Springer, pp. 142–156 (see pages 8, 181).

Bauer, A., Herbst, N., Spinner, S., Ali-Eldin, A., and Kounev, S. (2018b). "Chameleon: A Hybrid, Proactive Auto-Scaling Mechanism on a Level-Playing Field". In: *IEEE Transactions on Parallel and Distributed Systems* (*TPDS*) 30.4, pp. 800–813 (see pages 8, 122, 123, 181).

Bauer, A., Lesch, V., Versluis, L., Ilyushkin, A., Herbst, N., and Kounev, S. (2019b). "Chamulteon: Coordinated Auto-Scaling of Micro-Services". In: *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems* (*ICDCS*). IEEE, pp. 2015–2025 (see pages 8, 123, 181).

Bauer, A., Züfle, M., Grohmann, J., Schmitt, N., Herbst, N., and Kounev, S. (2020a). "An Automated Forecasting Framework based on Method Recommendation for Seasonal Time Series". In: *Proceedings of the 11th ACM/SPEC International Conference on Performance Engineering* (*ICPE*). ACM, pp. 48–55 (see pages 7, 92, 149).

Bauer, A., Züfle, M., Herbst, N., Kounev, S., and Curtef, V. (2020b). "Telescope: An Automatic Feature Extraction and Transformation Approach for Time Series Forecasting on a Level-Playing Field". In: *Proceedings of the 36th IEEE International Conference on Data Engineering* (*ICDE*). IEEE, pp. 1902–1905 (see pages 7, 92, 149).

Bauer, A., Züfle, M., Herbst, N., Zehe, A., Hotho, A., and Kounev, S. (2020c). "Time Series Forecasting for Self-Aware Systems". In: *Proceedings of the IEEE* 108.7, pp. 1068–1093 (see pages 4, 7, 74, 92, 149).

Bell, W. R. and Hillmer, S. C. (1984). "Issues Involved with the Seasonal Adjustment of Economic Time Series". In: *Journal of Business & Economic Statistics* 2.4, pp. 291–320 (see page 23).

Beltrán, M. (2015). "Automatic Provisioning of Multi-Tier Applications in Cloud Computing Environments". In: *The Journal of Supercomputing* 71.6, pp. 2221–2250 (see pages 66, 122, 144, 186).

Benifa, J. B. and Dejey, D. (2019). "Rlpas: Reinforcement Learning-Based Proactive Auto-Scaler for Resource Provisioning in Cloud Environment". In: *Mobile Networks and Applications* 24.4, pp. 1348–1363 (see pages 67, 144).

Benz, D., Hotho, A., Jäschke, R., Krause, B., Mitzlaff, F., Schmitz, C., and Stumme, G. (2010). "The Social Bookmark and Publication Management System Bibsonomy". In: *The International Journal on Very Large Data Bases* 19.6, pp. 849–875 (see page 182).

Bergmeir, C., Hyndman, R. J., and Benítez, J. M. (2016). "Bagging Exponential Smoothing Methods Using STL Decomposition and Box–Cox Transformation". In: *International journal of forecasting* 32.2, pp. 303–312 (see pages 3, 59, 116, 118, 119, 230).

Bolch, G., Greiner, S., De Meer, H., and Trivedi, K. S. (2006). *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. John Wiley & Sons (see pages 43, 189).

Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). "A Training Algorithm for Optimal Margin Classifiers". In: *Proceedings of the fifth annual workshop on Computational learning theory*. ACM, pp. 144–152 (see page 37).

Boulegane, D., Bifet, A., and Madhusudan, G. (2019). "Arbitrated Dynamic Ensemble with Abstaining for Time-Series Forecasting on Data Streams". In: *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, pp. 1040–1045 (see pages 55, 118).

Box, G. and Jenkins, G. (1970). *Time Series Analysis: Forecasting and Control*. Holden-Day (see pages 2, 33).

Box, G. E. and Cox, D. R. (1964). "An Analysis of Transformations". In: *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 211–252 (see page 25).

Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1984). "Classification and Regression Trees". In: (see page 35).

Breiman, L. (2001). "Random Forests". In: *Machine learning* 45.1, pp. 5–32 (see page 36).

Brockwell, P. J. and Davis, R. A. (2016). *Introduction to Time Series and Forecasting*. springer (see page 97).

Brown, R. G. (1956). "Exponential Smoothing for Predicting Demand". In: *Operations Research*. Vol. 5. 1, pp. 145–145 (see page 32).

Bunch, J. R. and Hopcroft, J. E. (1974). "Triangular Factorization and Inversion by Fast Matrix Multiplication". In: *Mathematics of Computation* 28.125, pp. 231–236 (see page 183).

Cerqueira, V., Torgo, L., Pinto, F., and Soares, C. (2017). "Arbitrated Ensemble for Time Series Forecasting". In: *Joint European conference on machine learning and knowledge discovery in databases*. Springer, pp. 478–494 (see pages 3, 54, 55, 118).

Chen, T. and Guestrin, C. (2016). "Xgboost: A Scalable Tree Boosting System". In: *ACM SIGKDD 2016*. ACM, pp. 785–794 (see page 37).

Chieu, T. C., Mohindra, A., Karve, A. A., and Segal, A. (2009). "Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment". In: *E-Business Engineering, 2009. ICEBE'09. IEEE International Conference on*. IEEE, pp. 281–286 (see pages 67, 144, 186).

Cleveland, R. B., Cleveland, W. S., McRae, J. E., and Terpenning, I. (1990). "STL: A Seasonal-Trend Decomposition Procedure based on Loess". In: *Journal of Official Statistics* 6.1, pp. 3–73 (see page 23).

Collopy, F. and Armstrong, J. S. (1992). "Rule-Based Forecasting: Development and Validation of an Expert Systems Approach to Combining Time Series Extrapolations". In: *Management Science* 38.10, pp. 1394–1414 (see pages 56, 118).

Crone, S. F., Hibon, M., and Nikolopoulos, K. (2011). "Advances in Forecasting with Neural Networks? Empirical Evidence from the NN3 Competition on Time Series Prediction". In: *International Journal of forecasting* 27.3, pp. 635–660 (see pages 62, 86).

Dagum, E. B. and Bianconcini, S. (2016). *Seasonal Adjustment Methods and Real Time Trend-Cycle Estimation*. Springer (see page 23).

Dezhabad, N. and Sharifian, S. (2018). "Learning-Based Dynamic Scalable Load-Balanced Firewall as a Service in Network Function-Virtualized Cloud Computing Environments". In: *The Journal of Supercomputing* 74.7, pp. 3329–3358 (see pages 66, 144).

Dietterich, T. G. (2002). "Machine learning for Sequential Data: A Review". In: *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*. Springer, pp. 15–30 (see page 99).

Domingos, P. M. (2012). "A Few Useful Things to Know about Machine Learning". In: *Commun. acm* 55.10, pp. 78–87 (see page 22).

Dos Santos, P. M., Ludermir, T. B., and Prudencio, R. B. C. (2004). "Selection of Time Series Forecasting Models Based on Performance Information". In: *Fourth International Conference on Hybrid Intelligent Systems (HIS'04)*. IEEE, pp. 366–371 (see page 89).

Drucker, H., Burges, C. J., Kaufman, L., Smola, A. J., and Vapnik, V. (1997). "Support Vector Regression Machines". In: *Advances in neural information processing systems*, pp. 155–161 (see page 37).

Durbin, J. and Watson, G. S. (1950). "Testing for Serial Correlation in Least Squares Regression: I". In: *Biometrika* 37.3/4, pp. 409–428 (see page 106).

Efron, B. (1992). "Bootstrap Methods: Another Look at the Jackknife". In: *Breakthroughs in statistics*. Springer, pp. 569–593 (see page 36).

Eismann, S., Grohmann, J., Eyk, E. van, Herbst, N., and Kounev, S. (2020). "Predicting the Costs of Serverless Workflows". In: *Proceedings of the 2020 ACM/SPEC International Conference on Performance Engineering (ICPE)* (see page 232).

Eismann, S., Walter, J., Kistowski, J. von, and Kounev, S. (2018b). "Modeling of Parametric Dependencies for Performance Prediction of Component-based Software Systems at Run-time". In: *2018 IEEE International Conference on Software Architecture (ICSA)*. Seattle, USA, pp. 135–144 (see page 127).

Fernandez, H., Pierre, G., and Kielmann, T. (2014). "Autoscaling Web Applications in Heterogeneous Cloud Infrastructures". In: *2014 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, pp. 195–204 (see pages 4, 68, 144, 145, 186).

Folkerts, E., Alexandrov, A., Sachs, K., Iosup, A., Markl, V., and Tosun, C. (2012). "Benchmarking in the Cloud: What it Should, Can, and Cannot Be". In: *Technology Conference on Performance Evaluation and Benchmarking*. Springer, pp. 173–188 (see page 50).

Fontes, X. and Castro Silva, D. (2020). "Hybrid Approaches for Time Series Prediction". In: *Hybrid Intelligent Systems*. Cham: Springer International Publishing, pp. 146–155 (see page 91).

Fourier, J. B. J. (1822). *Théorie Analytique de la Chaleur*. F. Didot (see page 20).

Friedman, M. (1937). "The Use of Ranks to Avoid the Assumption of Normality Implicit in the Analysis of Variance". In: *Journal of the american statistical association* 32.200, pp. 675–701 (see pages 162, 168, 178).

Fulcher, B. D., Little, M. A., and Jones, N. S. (2013). "Highly Comparative Time-Series Analysis: The Empirical Structure of Time Series and their Methods". In: *Journal of the Royal Society Interface* 10.83, p. 20130048 (see pages 28, 86, 87, 168).

Galante, G. and Bona, L. de (2012). "A Survey on Cloud Computing Elasticity". In: *IEEE UCC 2012*. IEEE, pp. 263–270 (see page 63).

Gardner, J. R., Pleiss, G., Bindel, D., Weinberger, K. Q., and Wilson, A. G. (2018). "GPyTorch: Blackbox Matrix-Matrix Gaussian Process Inference with GPU Acceleration". In: *Advances in Neural Information Processing Systems* (see page 150).

Gershenfeld, N. A. and Weigend, A. S. (1993). *The Future of Time Series*. Tech. rep. Xerox Corporation, Palo Alto Research Center (see page 62).

Grohmann, J., Herbst, N., Spinner, S., and Kounev, S. (2017). "Self-Tuning Resource Demand Estimation". In: *IEEE ICAC 2017*. Columbus, OH (see page 189).

Grubinger, T., Zeileis, A., and Pfeiffer, K.-P. (2011). *evtree: Evolutionary Learning of Globally Optimal Classification and Regression Trees in R*. Tech. rep. Working Papers in Economics and Statistics (see page 35).

Guerrero, V. M. (1993). "Time-Series Analysis Supported by Power Transformations". In: *Journal of Forecasting* 12.1, pp. 37–48 (see pages 26, 94).

Han, R., Guo, L., Ghanem, M. M., and Guo, Y. (2012). "Lightweight Resource Scaling for Cloud Applications". In: *IEEE/ACM CCGrid 2012*. IEEE, pp. 644–651 (see pages 67, 144).

Haslett, J. and Raftery, A. E. (1989). "Space-Time Modelling with Long-Memory Dependence: Assessing Ireland's Wind Power Resource". In: *Journal of the Royal Statistical Society: Series C* (*Applied Statistics*) 38.1, pp. 1–21 (see page 29).

Herbst, N. (2018). "Methods and Benchmarks for Auto-Scaling Mechanisms in Elastic Cloud Environments". PhD thesis. University of Würzburg, Germany (see pages 8, 122, 123, 137, 198).

Herbst, N. R., Kounev, S., and Reussner, R. (2013). "Elasticity in Cloud Computing: What it is, and What it is Not". In: *Proceedings of the 10th International Conference on Autonomic Computing* (*ICAC 2013*). San Jose, CA: USENIX (see page 47).

Herbst, N., Kounev, S., Weber, A., and Groenda, H. (2015). "BUNGEE: An Elasticity Benchmark for Self-Adaptive IaaS Cloud Environments". In: *SEAMS 2015*. IEEE Press, pp. 46–56 (see page 50).

Herbst, N., Krebs, R., Oikonomou, G., Kousiouris, G., Evangelinou, A., Iosup, A., and Kounev, S. (2016). "Ready for Rain? A View from SPEC Research on the Future of Cloud Metrics". In: *CoRR* abs/1604.03470 (see page 48).

Herbst, N. et al. (2018). "Quantifying Cloud Performance and Dependability: Taxonomy, Metric Design, and Emerging Challenges". In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* (*TOMPECS*) 3.4, p. 19 (see pages 8, 123).

Ho, T. K. (1995). "Random Decision Forests". In: *Proceedings of 3rd international conference on document analysis and recognition*. Vol. 1. IEEE, pp. 278–282 (see page 36).

Holt, C. (1957). "Forecasting Trends and Seasonal by Exponentially Weighted Moving Averages". In: *ONR Memorandum* 52 (see page 32).

Hong, T., Pinson, P., and Fan, S. (2014). "Global Energy Forecasting Competition 2012". In: *International Journal of Forecasting* 30.2, pp. 357–363 (see page 62).

Hong, T., Pinson, P., Fan, S., Zareipour, H., Troccoli, A., Hyndman, R. J., et al. (2016). "Probabilistic Energy Forecasting: Global Energy Forecasting Competition 2014 and Beyond". In: *International Journal of Forecasting* 32.3, pp. 896–913 (see page 62).

Huber, N., Brosig, F., Spinner, S., Kounev, S., and Bähr, M. (2017). "Model-Based Self-Aware Performance and Resource Management Using the Descartes Modeling Language". In: *IEEE Transactions on Software Engineering* (*TSE*) 43.5 (see pages 124, 127).

Huppler, K. (2011). "Benchmarking with your Head in the Cloud". In: *Technology Conference on Performance Evaluation and Benchmarking*. Springer, pp. 97–110 (see page 50).

– (2009). "The Art of Building a Good Benchmark". In: *Technology Conference on Performance Evaluation and Benchmarking*. Springer, pp. 18–30 (see page 50).

Hyndman, R. J. and Athanasopoulos, G. (2017). *Forecasting: Principles and Practice*. Melbourne, Australia: OTexts (see pages 3, 13, 14, 23, 25, 29, 31, 32, 39, 74, 81, 91, 92, 94, 97, 150).

Hyndman, R. J. and Khandakar, Y. (2008). "Automatic Time Series Forecasting: The Forecast Package for R". In: *Journal of Statistical Software* 26.3, pp. 1–22 (see page 101).

Hyndman, R. J. and Koehler, A. B. (2006). "Another Look at Measures of Forecast Accuracy". In: *International journal of forecasting* 22.4, pp. 679–688 (see page 41).

Hyndman, R. J., Koehler, A. B., Snyder, R. D., and Grose, S. (2002). "A State Space Framework for Automatic Forecasting Using Exponential Smoothing Methods". In: *International Journal of Forecasting* 18.3, pp. 439–454 (see page 32).

Hyndman, R. J., Wang, E., and Laptev, N. (2015). "Large-Scale Unusual Time Series Detection". In: *2015 IEEE international conference on data mining workshop (ICDMW)*. IEEE, pp. 1616–1619 (see pages 28, 86, 87, 168).

Hyndman, R., Athanasopoulos, G., Bergmeir, C., Caceres, G., Chhay, L., O'Hara-Wild, M., Petropoulos, F., Razbash, S., Wang, E., and Yasmeen, F. (2018). *forecast: Forecasting Functions for Time Series and Linear Models*. R package version 8.4 (see page 38).

Ilyushkin, A., Ali-Eldin, A., Herbst, N., Bauer, A., Papadopoulos, A. V., Epema, D., and Iosup, A. (2018). "An Experimental Performance Evaluation of Autoscalers for Complex Workflows". In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 3.2, pp. 1–32 (see pages 145, 206, 215, 221, 222).

Iosup, A., Yigitbasi, N., and Epema, D. (2011). "On the Performance Variability of Production Cloud Services". In: *CCGrid 2011*, pp. 104–113 (see pages 197, 206).

Iqbal, W., Dailey, M. N., Carrera, D., and Janecek, P. (2011). "Adaptive Resource Provisioning for Read Intensive Multi-Tier Applications in the Cloud". In: *Future Generation Computer Systems* 27.6, pp. 871–879 (see pages 4, 68, 144, 186).

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). *An Introduction to Statistical Learning*. Vol. 112. Springer (see page 31).

Jennings, B. and Stadler, R. (2015). "Resource Management in Clouds: Survey and Research Challenges". In: *Journal of Network and Systems Management* 23.3, pp. 567–619 (see page 63).

Jiang, J., Lu, J., Zhang, G., and Long, G. (2013). "Optimal Cloud Resource Auto-Scaling for Web Applications". In: *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE, pp. 58–65 (see pages 4, 65, 144, 145).

Kalyvianaki, E., Charalambous, T., and Hand, S. (2009). "Self-Adaptive and Self-Configured CPU Resource Provisioning for Virtualized Servers Using Kalman Filters". In: *Proceedings of the 6th international conference on Autonomic computing*, pp. 117–126 (see pages 64, 144).

Kang, Y., Hyndman, R. J., and Smith-Miles, K. (2017). "Visualising Forecasting Algorithm Performance Using Time Series Instance Spaces". In: *International Journal of Forecasting* 33.2, pp. 345–358 (see pages 28, 86, 87, 168).

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T.-Y. (2017). "Lightgbm: A Highly Efficient Gradient Boosting Decision Tree". In: *Advances in Neural Information Processing Systems*, pp. 3146–3154 (see page 98).

Kendall, D. G. (1953). "Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain". In: *The Annals of Mathematical Statistics*, pp. 338–354 (see page 45).

Kephart, J. O. and Chess, D. M. (2003). "The Vision of Autonomic Computing". In: *Computer* 36.1, pp. 41–50 (see page 134).

Khandelwal, I., Adhikari, R., and Verma, G. (2015). "Time Series Forecasting Using Hybrid ARIMA and ANN Models Based on DWT Decomposition". In: *Procedia Computer Science* 48.1, pp. 173–179 (see pages 59, 118).

Khorsand, R., Ghobaei-Arani, M., and Ramezanpour, M. (2018). "FAHP Approach for Autonomic Resource Provisioning of Multitier Applications in Cloud Computing Environments". In: *Wiley Software: Practice and Experience* 48.12, pp. 2147–2173 (see pages 69, 144).

Kistowski, J. v., Arnold, J. A., Huppler, K., Lange, K.-D., Henning, J. L., and Cao, P. (2015). "How to Build a Benchmark". In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. Austin, Texas, USA: ACM, pp. 333–336 (see page 73).

Kistowski, J. v., Herbst, N. R., and Kounev, S. (2014). "Modeling Variations in Load Intensity over Time". In: *Proceedings of the Third International Workshop on Large Scale Testing*. Dublin, Ireland: ACM, pp. 1–4 (see page 52).

Kounev, S., Huber, N., Brosig, F., and Zhu, X. (2016). "A Model-Based Approach to Designing Self-Aware IT Systems and Infrastructures". In: *IEEE Computer* 49.7, pp. 53–61 (see page 124).

Kounev, S., Lange, K.-D., and Kistowski, J. von (2020). *Systems Benchmarking: For Scientists and Engineers* (see pages 43, 44).

Kück, M., Crone, S. F., and Freitag, M. (2016). "Meta-Learning with Neural Networks and Landmarking for Forecasting Model Selection an Empirical Evaluation of Different Feature Sets Applied to Industry Data". In: *2016 International Joint Conference on Neural Networks (IJCNN)*. IEEE, pp. 1499–1506 (see pages 57, 118, 119).

Kuhn, M. and Johnson, K. (2013). *Applied Predictive Modeling*. Vol. 26. Springer (see page 31).

Kwiatkowski, D., Phillips, P. C., Schmidt, P., Shin, Y., et al. (1992). "Testing the Null Hypothesis of Stationarity against the Alternative of a Unit Root". In: *Journal of econometrics* 54.1-3, pp. 159–178 (see page 29).

Lakew, E. B., Papadopoulos, A. V., Maggio, M., Klein, C., and Elmroth, E. (2017). "KPI-Agnostic Control for Fine-Grained Vertical Elasticity". In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, pp. 589–598 (see pages 65, 144).

Lazowska, E. D., Zahorjan, J., Graham, G. S., and Sevcik, K. C. (1984). *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc. (see page 46).

Lemke, C. and Gabrys, B. (2010a). "Meta-learning for Time Series Forecasting and Forecast Combination". In: *Neurocomputing* 73.10-12, pp. 2006–2016 (see page 105).

– (2010b). "Meta-Learning for Time Series Forecasting in the NN GC1 Competition". In: *International Conference on Fuzzy Systems*. IEEE, pp. 1–5 (see pages 27, 56, 118, 119).

Lesch, V., Bauer, A., Herbst, N., and Kounev, S. (2018). "FOX: Cost-Awareness for Autonomic Resource Management in Public Clouds". In: *Proceedings of the 9th ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, pp. 4–15 (see pages 8, 123, 181).

Liu, N., Tang, Q., Zhang, J., Fan, W., and Liu, J. (2014). "A Hybrid Forecasting Model with Parameter Optimization for Short-Term Load Forecasting of Micro-Grids". In: *Applied Energy* 129, pp. 336–345 (see pages 58, 118).

Livera, A. M. D., Hyndman, R. J., and Snyder, R. D. (2011). "Forecasting Time Series With Complex Seasonal Patterns Using Exponential Smoothing". In: *Journal of the American Statistical Association* 106.496, pp. 1513–1527 (see page 34).

Lorido-Botran, T., Miguel-Alonso, J., and Lozano, J. A. (2014). "A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments". In: *Journal of Grid Computing* 12.4, pp. 559–592 (see page 63).

Makridakis, S. (1976). "A Survey of Time Series". In: *International Statistical Review/Revue Internationale de Statistique*, pp. 29–70 (see pages 16, 17).

Makridakis, S., Andersen, A., Carbone, R., Fildes, R., Hibon, M., Lewandowski, R., Newton, J., Parzen, E., and Winkler, R. (1982). "The Accuracy of Extrapolation (Time Series) Methods: Results of a Forecasting Competition". In: *Journal of forecasting* 1.2, pp. 111–153 (see pages 61, 86).

Makridakis, S., Chatfield, C., Hibon, M., Lawrence, M., Mills, T., Ord, K., and Simmons, L. F. (1993). "The M2-Competition: A Real-Time Judgmentally Based Forecasting Study". In: *International Journal of Forecasting* 9.1, pp. 5–22 (see page 61).

Makridakis, S. and Hibon, M. (1979). "Accuracy of Forecasting: An Empirical Investigation". In: *Journal of the Royal Statistical Society: Series A (General)* 142.2, pp. 97–125 (see page 61).

– (2000). "The M3-Competition: Results, Conclusions and Implications". In: *International journal of forecasting* 16.4, pp. 451–476 (see pages 62, 86, 152).

Makridakis, S., Spiliotis, E., and Assimakopoulos, V. (2018a). "Statistical and Machine Learning Forecasting Methods: Concerns and Ways Forward". In: *PloS one* 13.3, e0194889 (see pages 93, 94, 99).

– (2018b). "The M4 Competition: Results, Findings, Conclusion and Way Forward". In: *International Journal of Forecasting* 34.4, pp. 802–808 (see pages 53, 62, 86, 161, 179).

Maurer, M., Brandic, I., and Sakellariou, R. (2011). "Enacting Slas in Clouds Using Rules". In: *Euro-Par 2011*. Springer, pp. 455–466 (see pages 67, 144).

Mell, P. and Grance, T. (2011). "The NIST Definition of Cloud Computing". In: (see page 47).

Menascé, D. A., Dowdy, L. W., and Almeida, V. A. F. (2004). *Performance by Design: Computer Capacity Planning By Example*. Upper Saddle River, NJ, USA: Prentice Hall PTR (see pages 46, 126).

Montero-Manso, P., Athanasopoulos, G., Hyndman, R. J., and Talagala, T. S. (2020). "FFORMA: Feature-Based Forecast Model Averaging". In: *International Journal of Forecasting* 36.1, pp. 86–92 (see pages 3, 55, 118).

Naskos, A., Gounaris, A., and Katsaros, P. (2017). "Cost-Aware Horizontal Scaling of NoSQL Databases Using Probabilistic Model Checking". In: *Cluster Computing* 20.3, pp. 2687–2701 (see pages 4, 68, 144, 145).

Nguyen, H., Shen, Z., Gu, X., Subbiah, S., and Wilkes, J. (2013). "AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service." In: *10th USENIX International Conference on Autonomic Computing (ICAC 13)*. San Jose, CA: USENIX, pp. 69–82 (see pages 68, 122, 144, 186).

Nuttall, A. H. and Carter, G. C. (1982). "Spectral Estimation Using Combined Time and Lag Weighting". In: *Proceedings of the IEEE* 70.9, pp. 1115–1125 (see page 28).

Padala, P., Hou, K.-Y., Shin, K. G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., and Merchant, A. (2009). "Automated Control of Multiple Virtualized Resources". In: *Proceedings of the 4th ACM European conference on Computer systems*, pp. 13–26 (see pages 64, 144).

Pai, P.-F. and Lin, C.-S. (2005). "A Hybrid ARIMA and Support Vector Machines Model in Stock Price Forecasting". In: *Omega* 33.6, pp. 497–505 (see pages 58, 118).

Panigrahi, S. and Behera, H. S. (2017). "A Hybrid ETS–ANN Model for Time Series Forecasting". In: *Engineering Applications of Artificial Intelligence* 66, pp. 49–59 (see pages 59, 118).

Papadopoulos, A. V., Ali-Eldin, A., Arzén, K.-E., Tordsson, J., and Elmroth, E. (2016). "PEAS: A Performance Evaluation Framework for Auto-Scaling Strategies in Cloud Applications". In: *ACM ToMPECS* 1.4, pp. 1–31 (see pages 145, 186, 206, 215, 221, 222).

Papadopoulos, A. V., Versluis, L., Bauer, A., Herbst, N., Kistowski, J. von, Ali-Eldin, A., Abad, C., Amaral, J. N., Tuma, P., and Iosup, A. (2019b). "Methodological Principles for Reproducible Performance Evaluation in Cloud Computing". In: *IEEE Transactions on Software Engineering* (*TSE*) (see page 145).

Phillips, P. C. and Perron, P. (1988). "Testing for a Unit Root in Time Series Regression". In: *Biometrika* 75.2, pp. 335–346 (see pages 18, 29).

Pincus, S. M., Gladstone, I. M., and Ehrenkranz, R. A. (1991). "A Regularity Statistic for Medical Data Analysis". In: *Journal of Clinical Monitoring* 7.4, pp. 335–345 (see page 105).

Plummer, D. C., Smith, D., Bittman, T., Cear-Ley, D. W., Cappuccio, D., Scott, D., Kumar, R., and Robertson, B. (2009). *Study: Five Refining Attributes of Public and Private Cloud Computing*. Tech. rep. Gartner (see page 47).

Qu, C., Calheiros, R. N., and Buyya, R. (2018). "Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey". In: *ACM Comput. Surv.* 51.4, pp. 1–33 (see pages 4, 63, 144).

Quinlan, J. R. (1993). "Combining Instance-Based and Model-Based Learning". In: *Proceedings of the tenth international conference on machine learning*, pp. 236–243 (see page 36).

Quinlan, J. R. et al. (1992). "Learning with Continuous Classes". In: *5th Australian joint conference on artificial intelligence*. Vol. 92. World Scientific, pp. 343–348 (see page 36).

Rao, J., Bu, X., Xu, C.-Z., Wang, L., and Yin, G. (2009). "VCONF: A Reinforcement Learning Approach to Virtual Machines Auto-Configuration". In: *Proceedings of the 6th international conference on Autonomic computing*, pp. 137–146 (see pages 66, 144).

Rasmussen, C. E. and Williams, C. K. (2006). *Gaussian Processes for Machine Learning*. Vol. 2. 3. MIT press Cambridge, MA (see page 152).

Rice, J. R. (1976). "The Algorithm Selection Problem". In: *Advances in computers*. Vol. 15. Elsevier, pp. 65–118 (see page 104).

RightScale (2019). *RightScale 2019 State of the Cloud Report from Flexera*. Tech. rep. Flexera (see pages 2, 121).

Rose, O. (1996). "Estimation of the Hurst Parameter of Long-Range Dependent Time Series". In: *University of Wurzburg, Institute of Computer Science Research Report Series.-February* (see page 29).

Rosenbush, S. and Loten, A. (2017). *Oracle CEO Hurd Says 80% of Corporate Data Centers Gone by 2025*. `https://blogs.wsj.com/cio/2017/01/17/oracle-co-ceo-mark-hurd-says-80-of-corporate-data-centers-gone-by-2025/`. Accessed: 2020-09-23 (see page 1).

Saâdaoui, F. and Rabbouch, H. (2019). "A Wavelet-Based Hybrid Neural Network for Short-Term Electricity Prices Forecasting". In: *Artificial Intelligence Review* 52.1, pp. 649–669 (see pages 60, 118).

Saâdaoui, F., Saadaoui, H., and Rabbouch, H. (2019). "Hybrid Feedforward ANN with NLS-Based Regression Curve Fitting for US Air Traffic Forecasting". In: *Neural Computing and Applications*, pp. 1–13 (see pages 60, 118).

Schouten, E. (2012). *Rapid Elasticity and the Cloud*. `https://www.ibm.com/blogs/cloud-computing/2012/09/12/rapid-elasticity-and-the-cloud/`. Accessed: 2020-06-19 (see page 47).

Schuster, A. (1899). "The Periodgram of Magnetic Declination as Obtained from the Records of the Greenwich Observatory during the Years 1871-1895". In: *Transactions of the Cambridge Philosophical Society* 18, pp. 107–135 (see page 21).

Sevcik, P. (2005). "Defining the Application Performance Index". In: *Business Communications Review* 20 (see page 187).

Sharma, U., Shenoy, P., and Towsley, D. F. (2012). "Provisioning Multi-Tier Cloud Applications Using Statistical Bounds on Sojourn Time". In: *9th ACM International Conference on Autonomic Computing (ICAC) 2010*. ACM, pp. 43–52 (see pages 65, 144).

Shcherbakov, M. V., Brebels, A., Shcherbakova, N. L., Tyukov, A. P., Janovsky, T. A., and Kamaev, V. A. (2013). "A Survey of Forecast Error Measures". In: *World Applied Sciences Journal* 24.24, pp. 171–176 (see page 41).

Shen, Z., Subbiah, S., Gu, X., and Wilkes, J. (2011). "Cloudscale: Elastic Resource Scaling for Multi-Tenant Cloud Systems". In: *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pp. 1–14 (see pages 122, 186).

Shumway, R. H. and Stoffer, D. S. (2000). "Time Series Analysis and its Applications". In: *Studies In Informatics And Control* 9.4, pp. 375–376 (see pages 13, 17, 18).

Singh, P., Gupta, P., Jyoti, K., and Nayyar, A. (2019). "Research on Auto-Scaling of Web Applications in Cloud: Survey, Trends and Future Directions". In: *Scalable Computing: Practice and Experience* 20.2, pp. 399–432 (see page 63).

Smith-Miles, K. A. (2009). "Cross-Disciplinary Perspectives on Meta-Learning for Algorithm Selection". In: *ACM Computing Surveys (CSUR)* 41.1, pp. 1–25 (see page 104).

Smyl, S. (2020). "A Hybrid Method of Exponential Smoothing and Recurrent Neural Networks for Time Series Forecasting". In: *International Journal of Forecasting* 36.1, pp. 75–85 (see pages 3, 61, 118, 229).

Sommer, M., Stein, A., and Hähner, J. (2016). "Local Ensemble Weighting in the Context of Time Series Forecasting Using XCSF". In: *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, pp. 1–8 (see pages 54, 118).

Spinner, S., Casale, G., Brosig, F., and Kounev, S. (2015). "Evaluating Approaches to Resource Demand Estimation". In: *Perform. Evaluation* 92, pp. 51–71 (see pages 46, 125).

Spinner, S., Casale, G., Zhu, X., and Kounev, S. (2014). "LibReDE: A library for Resource Demand Estimation". In: *ACM/SPEC ICPE 2014*. ACM, pp. 227–228 (see pages 125, 126, 184).

Spinner, S., Walter, J., and Kounev, S. (2016). "A Reference Architecture for Online Performance Model Extraction in Virtualized Environments". In: *ACM/SPEC ICPE 2017*. ACM, pp. 57–62 (see page 124).

Sugiyama, M. and Kawanabe, M. (2012). *Machine Learning in Non-Stationary Environments: Introduction to Covariate Shift Adaptation*. The MIT Press (see pages 91, 94).

Talagala, T. S., Hyndman, R. J., and Athanasopoulos, G. (2018). *Meta-Learning How to Forecast Time Series*. Tech. rep. Monash University, Department of Econometrics and Business Statistics (see pages 3, 28, 57, 118, 119).

Taylor, S. J. and Letham, B. (2018). "Forecasting at Scale". In: *The American Statistician* 72.1, pp. 37–45 (see pages 3, 25, 60, 118).

Teräsvirta, T., Lin, C.-F., and Granger, C. W. (1993). "Power of the Neural Network Linearity Test". In: *Journal of Time Series Analysis* 14.2, pp. 209–220 (see page 29).

Tesauro, G., Jong, N. K., Das, R., and Bennani, M. N. (2006). "A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation". In: *IEEE ICAC 2006*. IEEE, pp. 65–73 (see pages 66, 144).

Urgaonkar, B., Shenoy, P., Chandra, A., and Goyal, P. (2005). "Dynamic Provisioning of Multi-Tier Internet Applications". In: *Second International Conference on Autonomic Computing (ICAC'05)*. IEEE, pp. 217–228 (see pages 8, 122).

Urgaonkar, B., Shenoy, P., Chandra, A., Goyal, P., and Wood, T. (2008). "Agile Dynamic Provisioning of Multi-tier Internet Applications". In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 3.1, pp. 1–39 (see pages 4, 65, 144, 186).

Vapnik, V. (1995). *The Nature of Statistical Learning* (see pages 37, 38).

Wang, X., Smith-Miles, K., and Hyndman, R. (2009). "Rule Induction for Forecasting Method Selection: Meta-Learning the Characteristics of Univariate Time Series". In: *Neurocomputing* 72.10 - 12, pp. 2581–2594 (see pages 27, 56, 57, 105, 118).

Wang, Z., Koprinska, I., Troncoso, A., and Martínez-Álvarez, F. (2018). "Static and Dynamic Ensembles of Neural Networks for Solar Power Forecasting". In: *2018 International Joint Conference on Neural Networks (IJCNN)*. IEEE, pp. 1–8 (see pages 55, 118).

Widodo, A. and Budi, I. (2013). "Model Selection Using Dimensionality Reduction of Time Series Characteristics". In: *International Symposium on Forecasting, Seoul, South Korea* (see pages 57, 118, 119).

Willinger, W., Paxson, V., and Taqqu, M. S. (1998). "Self-Similarity and Heavy Tails: Structural Modeling of Network Traffic". In: *A Practical Guide to Heavy Tails: Statistical Techniques and Applications* 23, pp. 27–53 (see page 29).

Willnecker, F., Dlugi, M., Brunnert, A., Spinner, S., Kounev, S., Gottesheim, W., and Krcmar, H. (2015). "Comparing the Accuracy of Resource Demand Measurement and Estimation Techniques". In: *EPEW 2015*. Ed. by M. Beltrán, W. Knottenbelt, and J. Bradley. Vol. 9272. Madrid, Spain: Springer, pp. 115–129 (see page 46).

Winters, P. R. (1960). "Forecasting Sales by Exponentially Weighted Moving Averages". In: *Management science* 6.3, pp. 324–342 (see page 32).

Wold, H. (1938). "A Study in the Analysis of Stationary Time Series". PhD thesis. Almqvist & Wiksell (see page 34).

Wolpert, D. H. and Macready, W. G. (1997). "No Free Lunch Theorems for Optimization". In: *IEEE Transactions on Evolutionary Computation* 1.1, pp. 67–82 (see pages 3, 31, 53, 91, 152, 162, 179, 229).

Wu, S., Li, B., Wang, X., and Jin, H. (2016). "HybridScaler: Handling Bursting Workload for Multi-tier Web Applications in Cloud". In: *15th International Symposium on Parallel and Distributed Computing (ISPDC), 2016*, pp. 141–148 (see pages 4, 69, 144, 145).

Xiong, P., Wang, Z., Malkowski, S., Wang, Q., Jayasinghe, D., and Pu, C. (2011). "Economical and Robust Provisioning of n-Tier Cloud Workloads: A Multi-Level Control Approach". In: *2011 31st International Conference on Distributed Computing Systems*. IEEE, pp. 571–580 (see pages 65, 144).

Yule, G. U. (1927). "On a Method of Investigating Periodicities Disturbed Series, with Special Reference to Wolfer's Sunspot Numbers". In: *Philosophical Transactions of the Royal Society of London* 226.636-646, pp. 267–298 (see page 2).

Zhang, D., Chen, S., Liwen, L., and Xia, Q. (2020). "Forecasting Agricultural Commodity Prices Using Model Selection Framework With Time Series Features and Forecast Horizons". In: *IEEE Access* 8, pp. 28197–28209 (see pages 58, 118).

Zhang, G. P. (2003). "Time Series Forecasting Using a Hybrid ARIMA and Neural Network Model". In: *Neurocomputing* 50, pp. 159–175 (see pages 58, 59, 118).

Zhang, J., Wei, Y., Tan, Z.-f., Ke, W., and Tian, W. (2017). "A Hybrid Method for Short-Term Wind Speed Forecasting". In: *Sustainability* 9.4, p. 596 (see pages 59, 118).

Zhu, Q. and Agrawal, G. (2012). "Resource Provisioning with Budget Constraints for Adaptive Applications in Cloud Environments". In: *IEEE Transactions on Services Computing* 5.4, pp. 497–511 (see pages 64, 144).

Zuefle, M., Bauer, A., Lesch, V., Krupitzer, C., Herbst, N., Kounev, S., and Curtef, V. (2019). "Autonomic Forecasting Method Selection: Examination and Ways Ahead". In: *Proceedings of the 16th IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, pp. 167–176 (see page 56).

Züfle, M., Bauer, A., Herbst, N., Curtef, V., and Kounev, S. (2017). "Telescope: A Hybrid Forecast Method for Univariate Time Series". In: *Proceedings of the 4th International Work-Conference on Time Series (ITISE)* (see pages 7, 92).