Nikolas Roman Herbst

Methods and Benchmarks for Auto-Scaling Mechanisms in Elastic Cloud Environments

Dissertation, Julius-Maximilians-Universität Würzburg
Fakultät für Mathematik und Informatik, 2018

Gutachter:	Prof. Dr. Samuel Kounev, JMU Würzburg,
	Prof. Dr. Alexandru Iosup, VU Amsterdam,
	Prof. Dr. Wilhelm Hasselbring, CAU Kiel

Tag der mündlichen Prüfung: 19. Juli 2018

Abstract

Roughly a decade ago, the first commercial cloud was opened for the general public (Amazon Web Services AWS, in 2006). Meanwhile in 2017, the market for cloud computing offerings reached a size of 247 billion US$, but the peak of the hype period already passed with sustainable market growth rates of 18%. Concerning the next years, Mark Hurd, co-CEO of Oracle Corporation, recently predicted that 80 percent of corporate data centers will disappear by 2025, as production applications increasingly move to the cloud. Especially in the ongoing settling phase of cloud computing, scientific progress and industry growth depend on established principles for measuring and reporting cloud system quality attributes.

A key functionality of cloud systems are automated resource management mechanisms at the infrastructure level. As part of this, elastic scaling of allocated resources is realized by so-called auto-scalers that are supposed to match the current demand in a way that the performance remains stable while resources are efficiently used. The process of rating cloud infrastructure offerings in terms of the quality of their achieved elastic scaling remains undefined. Clear guidance for the selection and configuration of an auto-scaler for a given context is not available. Thus, existing operating solutions are optimized in a highly application specific way and usually kept undisclosed.

The common state of practice is the use of simplistic threshold-based approaches. Due to their reactive nature they incur performance degradation during the minutes of provisioning delays. In the literature, a high-number of auto-scalers has been proposed trying to overcome the limitations of reactive mechanisms by employing proactive prediction methods. The latter can be grouped into approaches from queuing theory, control theory, time series analysis and reinforcement learning. However, the adoption of proactive auto-scalers in production is still very low due to the high risk of relying on a single proactive method based on which scaling decisions are made.

In this thesis, we identify potentials in automated cloud system resource management and its evaluation methodology. Specifically, we make the following contributions:

- We propose a descriptive load profile modeling framework together with automated model extraction from recorded traces to enable reproducible workload generation with realistic load intensity variations. The proposed Descartes Load Intensity Model (DLIM) with its Limbo framework provides key functionality to stress and benchmark resource management approaches in a representative and fair manner.

  Automatically extracted DLIM model instances exhibit an average modeling error of 15.2% over ten different real-world traces that cover between two weeks and seven months. These results underline DLIM's model expressiveness. In terms of accuracy and processing speed, our proposed extraction methods based on the descriptive models deliver better or similar results compared to existing non-descriptive time series decomposition methods.

- We propose a set of intuitive metrics for quantifying timing, stability and accuracy aspects of elasticity. Based on these metrics, which were endorsed by the SPEG Research Group[1], we propose a novel approach for benchmarking the elasticity of Infrastructure-as-a-Service (IaaS) cloud platforms independent of the performance exhibited by the provisioned underlying resources. The proposed Bungee elasticity benchmarking framework leverages DLIM modeling capabilities to generate realistic load intensity profiles.

  The Bungee measurement methodology achieves reproducible results in controlled environments. Based on this property, elastic systems can be compared as each of the proposed metrics provides consistent ranking on an ordinal scale. Finally, we present an extensive case study of real-world complexity demonstrating that the proposed approach is applicable in realistic scenarios and can cope with different levels of resource performance.

- We tackle the challenge of reducing the risk of relying on a single proactive auto-scaler by proposing a new self-aware auto-scaling mechanism, called Chameleon, combining multiple different proactive methods coupled with a reactive fallback mechanism. Chameleon employs on-demand, automated time series-based forecasting methods to predict the arriving load intensity in combination with run-time service demand estimation techniques to calculate the required resource consumption per work unit without the need for a detailed application instrumentation. It can also leverage application knowledge by solving product-form queueing networks used to derive optimized scaling actions. The Chameleon approach is first in resolving conflicts between reactive and proactive scaling decisions in an intelligent way.

  Chameleon is benchmarked against four different state-of-the-art proactive auto-scalers and a standard threshold-based one in three different cloud environments: (i) a private CloudStack-based cloud, (ii) the public AWS EC2 cloud, as well as (iii) an OpenNebula-based shared IaaS cloud. We generate five different representative load profiles each taken from different real-world system traces leveraging Limbo and Bungee to drive a CPU-intensive benchmark workload. Overall, Chameleon achieves the best scaling behavior based on user and elasticity performance metrics, analyzing the results from 400 hours of aggregated experiment time.

- As a side-contribution of this thesis, we propose a self-aware forecasting prototype called Telescope. It incorporates multiple individual forecasting methods by decomposing the univariate time series into the components trend, season, and remainder. First, the frequency is estimated, anomalies are detected and removed, and then the type of decomposition is determined. After the decomposition, ARIMA (auto-regressive integrated moving averages) without seasonality is applied on the trend pattern, whereas the seasonality is simply continued. Moreover, the single periods are clustered in order to learn categorical information. The cluster labels are forecast

[1]SPEC Research Group: `https://research.spec.org`

by applying artificial neural networks. Lastly, eXtreme Gradient Boosting (XGBoost) is used to learn the dependency between them and to combine the forecasts of the individual components.

The preliminary evaluation indicates based on two different traces that an early implementation of the Telescope approach outperforms the best competitor in terms of accuracy. It improves the time-to-result by up to a factor of 19 compared to the three most efficient forecasting methods. In a case study, we demonstrate that Telescope is capable of furter improving Chameleon's auto-scaling performance compared to the formerly used state-of-the-art forecasting method tBATS or seasonal ARIMA.

We are confident that the contributions of this thesis will have a long-term impact on the way cloud resource management approaches are assessed. While this could result in an improved quality of autonomic management algorithms, we see and discuss arising challenges for future research in cloud resource management and its assessment methods: The adoption of containerization on top of virtual machine instances introduces another level of indirection. As a result, the nesting of virtual resources increases resource fragmentation and causes unreliable provisioning delays. Furthermore, virtualized compute resources tend to become more and more inhomogeneous associated with various priorities and trade-offs. Due to DevOps practices, cloud hosted service updates are released with a higher frequency which impacts the dynamics in user behavior.

Zusammenfassung

Vor etwas mehr als einem Jahrzehnt stellte Amazon Web Services (AWS) als erster kommerzieller Anbieter Cloud-Dienstleistungen der allgemeinen Öffentlichkeit bereit. In den folgenden Jahren fand ein ausgeprägtes Wachstum von Cloud-Dienstleistungsangeboten statt, welches sich im Jahr 2017 in einer Marktgröße von 247 Milliarden US$ widerspiegelte. Nunmehr gehört diese Phase ausgeprägten Wachstums der Vergangenheit an. Dennoch sagt Marc Hurd, Co-CEO des Oracle Konzerns, vorher, dass 80 Prozent der klassischen Rechenzentren von Konzernen bis zum Jahre 2025 abgelöst sein werden, da produktive Anwendungen mehr und mehr in Cloud-Umgebungen betrieben würden. Insbesondere in der aktuell anhaltenden Stabilisierungsphase des Cloud-Computing hängt der wissenschaftliche Fortschritt und das weitere Wirtschaftswachstum von etablierten Messverfahren und standardisierten Berichtsformen zur Erfassung der Qualitätsmerkmale von Cloud-Angeboten ab.

Eine Schlüsselfunktionalität von Cloud-Systemen sind automatisierte Mechanismen zur Ressourcenverwaltung auf Infrastrukturebene. Als Teil hiervon wird das elastische Skalieren der allokierten Ressourcen durch eigene Mechanismen realisiert. Diese sind dafür verantwortlich, dass die dynamische Ressourcenzuteilung die aktuelle Nachfrage in einem Maße trifft, welches die Performance stabil hält und gleichzeitig Ressourcen effizient auslastet. Prozesse, welche die Bewertung der Qualität von elastischem Skalierungsverhalten in der Realität ermöglichen, sind derzeit nicht umfassend definiert. Folglich fehlt es an Leitfäden und Entscheidungskriterien bei der Auswahl und Konfiguration automatisch skalierender Mechanismen. In der Praxis zum Einsatz kommende Lösungen sind auf ihr Anwendungsszenario optimiert und werden in fast allen Fällen unter Verschluss gehalten.

Mehrheitlich werden einfache, schwellenwertbasierte Regelungsansätze eingesetzt. Diese nehmen aufgrund ihres inhärent reaktiven Charakters verschlechterte Performance während der Bereitstellungsverzögerung im Minutenbereich in Kauf. In der Literatur wird eine große Anzahl an Mechanismen zur automatischen Skalierung vorgeschlagen, welche versuchen, diese Einschränkung durch Einsatz von Schätzverfahren zu umgehen. Diese können in Ansätze aus der Warteschlangentheorie, der Kontrolltheorie, der Zeitreihenanalyse und des maschinellen Lernens eingeteilt werden. Jedoch erfreuen sich prädiktive Mechanismen zum automatischen Skalieren aufgrund des damit verknüpften hohen Risikos, sich auf einzelne Schätzverfahren zu verlassen, bislang keines breiten Praxiseinsatzes.

Diese Dissertation identifiziert Potenziale im automatisierten Ressourcenmanagement von Cloud-Umgebungen und deren Bewertungsverfahren. Die Beiträge liegen konkret in den folgenden Punkten:

- Es wird eine Sprache zur deskriptiven Modellierung von Lastintensitätsprofilen und deren automatischer Extraktion aus Aufzeichnungen entwickelt, um eine wiederholbare Generierung von realistischen und in ihrer Intensität variierenden Arbeitslasten

zu ermöglichen. Das vorgeschlagene Descartes Lastintensitätsmodell (DLIM) zusammen mit dem Limbo Software-Werkzeug stellt hierbei Schlüsselfunktionalitäten zur repräsentativen Arbeitslastgenerierung und fairen Bewertung von Ressourcenmanagementansätzen zur Verfügung.

Über zehn verschiedene Aufzeichnungen aus der Realität, welche zwischen zwei Wochen und sieben Monate abdecken, weisen automatisch extrahierte DLIM Modellinstanzen im Durchschnitt einen Modellierungsfehler von 15,2% auf. Diese Ergebnisse betonen die Ausdrucksmächtigkeit des DLIM Modells. Hinsichtlich Genauigkeit und Verarbeitungsgeschwindigkeit liefern die hier vorgeschlagenen Methoden zur automatischen Extraktion der deskriptiven Modelle im Vergleich zu existierenden, nicht deskriptiven Methoden der Zeitreihenzerlegung, bessere oder vergleichbar gute Resultate.

- Es wird eine Gruppe intuitiver Metriken zur Quantifizierung der zeit-, genauigkeits- und stabilitätsbezogenen Qualitätsaspekte elastischen Verhaltens vorgeschlagen. Basierend auf diesen zwischenzeitlich von der Forschungsabteilung der Standard Performance Evaluation Corporation (SPEC) befürworteten Metriken, wird ein neuartiges Elastizitätsmessverfahren zur fairen Bewertung von Infrastruktur-Cloud-Dienstleistungen, unabhängig von der Leistungsfähigkeit der zugrunde liegenden Ressourcen, entwickelt.

  Das Elastizitätsverfahren Bungee erzielt wiederholbare Resultate in kontrollierten Umgebungen. Darauf aufbauend bietet jede der vorgeschlagenen Metriken eine konsistente Ordnung elastischer Systeme auf einer Ordinalskala. Schlussendlich wird eine umfassende Fallstudie von realer Komplexität präsentiert und so gezeigt, dass der vorgeschlagene Messansatz in der Lage ist, Szenarien des automatischen Skalierens aus der Realität zu bewerten.

- Durch die Entwicklung eines neuartigen, hybriden Ansatzes zum automatischen Skalieren, genannt Chameleon, wird das Risiko reduziert, welches sich aus dem Einsatz einzelner proaktiver Methoden automatischen Skalierens ergibt. Chameleon kombiniert mehrere verschiedene proaktive Methoden und ist mit einer reaktiven Rückfallebene gekoppelt. Dazu verwendet Chameleon bei Bedarf automatische Zeitreihenvorhersagen, um ankommende Arbeitslasten abzuschätzen. Ergänzend dazu kommen Techniken der Serviceanforderungsabschätzung zur Systemlaufzeit zum Einsatz, um den Ressourcenverbrauch einzelner Arbeitspakete in etwa zu bestimmen, ohne dass eine feingranulare Instrumentierung der Anwendung erforderlich ist. Abgesehen davon nutzt Chameleon anwendungsbezogenes Wissen, um Warteschlangennetze in Produktform zu lösen und optimale Skalierungsaktionen abzuleiten. Der Chameleon-Ansatz ist der erste seiner Art, welcher Konflikte zwischen reaktiven und proaktiven Skalierungsaktionen in intelligenter Art und Weise aufzulösen vermag.

  Chameleon wird messbasiert gegen fünf verschiedene, proaktive Mechanismen automatischen Skalierens sowie gegen das gängige schwellenwertbasierte Verfahren verglichen. Die Messungen werden in drei verschiedenen Umgebungen durchgeführt: (i) Einer privaten CloudStack-basierten Cloud-Umgebung, (ii) der öffentlichen

AWS Elastic Compute Cloud EC2, (iii) einer OpenNebula-basierten, zu Forschungszwecken geteilten Infrastruktur-Cloud. Indem wir die Funktionalitäten von Limbo's Arbeitslastprofilen und dem Bungee Messverfahren ausnutzen, erzeugen wir fünf verschiedene, repräsentative Arbeitslastprofile, welche jeweils von realen Aufzeichnungen abstammen und ein prozessorlastiges Referenzprogramm unter Last setzen. Zusammenfassend erreicht Chameleon, basierend auf den Ergebnissen von insgesamt 400 Experimentstunden, hinsichtlich der Elastizitätsmetriken sowie sekundären Benutzermetriken das beste und zuverlässigste Skalierungsverhalten.

- Als Nebenbeitrag dieser Dissertation wird ein hybrider, prototypischer Mechanismus zur Vorhersage von Zeitreihen, genannt Telescope, vorgeschlagen. Er integriert mehrere eigenständige Vorhersagemechanismen, indem die Eingabe einer univariaten Zeitreihe in ihre Bestandteile an Saisonalität, Trend und Rest zerlegt wird. Zuerst jedoch werden die Frequenz abgeschätzt, Anomalien erkannt und entfernt, sowie auf die Art der Zerlegung getestet. Nach der Zerlegung wird der Trendanteil durch das Verfahren der autoregressiven, integrierten, gleitenden Durchschnitte (ARIMA) vorhergesagt, während der saisonale Anteil direkt fortgesetzt wird. Zusätzlich werden die einzelnen Perioden in Cluster zusammengefasst, um deren kategorische Informationen zu erlernen. Die Cluster-Bezeichnungen werden durch Anwendung künstlicher, neuronaler Netze (ANNs) vorhergesagt. Letztendlich wird ein Verfahren der extremen Gradientenverstärkung (XGBoost) verwendet, um die Abhängigkeiten zwischen den einzelnen Anteilen und den Clusterbezeichnungen zu lernen und die Vorhersageergebnisse zu kombinieren.

  In einer vorläufigen Evaluation, welche auf zwei verschiedenen Datensätzen und einer prototypischen Implementierung basiert, schlägt der Telescope-Ansatz die Konkurrenz im Hinblick auf die Vorhersagegenauigkeit. Zudem wird die Zeit bis zum Ergebnis im Vergleich zu den drei genauesten Vorhersagemethoden um einen Faktor von bis zu 19 verringert. In einer Fallstudie wird demonstriert, dass Telescope in der Lage ist, das Skalierungsverhalten von Chameleon im Vergleich zu den bisher angewandten, aktuellen Vorhersagemethoden weiter zu verbessern.

Zusammenfassend kann gesagt werden, dass die Beiträge dieser Dissertation auf lange Sicht die Art und Weise beeinflussen dürften, in welcher Ressourcenmanagementansätze in Cloudumgebungen bewertet werden. Ein Ergebnis wäre unter anderem eine verbesserte Qualität der Algorithmen für ein automatisches Ressourcenmanagement. Als Grundlage für zukünftige Forschungsarbeiten werden aufkommende Herausforderungen identifiziert und diskutiert: Die Einführung der Containerisierung innerhalb von virtuellen Maschineninstanzen bewirkt eine weitere Ebene der Indirektion. Als Folge dieser Verschachtelung der virtuellen Ressourcen wird die Fragmentierung erhöht und unzuverlässige Bereitstellungsverzögerungen verursacht. Außerdem tendieren die virtualisierten Rechenressourcen aufgrund von Priorisierung und Zielkonflikten mehr und mehr zu inhomogenen Systemlandschaften. Aufgrund von DevOps-Praktiken werden Softwareupdates von Diensten in Cloudumgebungen mit einer höheren Frequenz durchgeführt, welche sich auf das Benutzungsverhalten dynamisierend auswirken kann.

Acknowledgments

The thesis would have been impossible without the aid and support of many people. First of all, I would like to thank my advisor Prof. Dr. Samuel Kounev. I first met him in 2010 in the middle of my diploma studies attending his advanced lecture on Performance Engineering, and since then he always supported me with advice and encouragement on my journey in the academic world. He was a constant source of inspiration and motivation in all these years guiding my work on this thesis, on research papers, and on grant proposals.

From the Descartes research group (at the Chair of Software Engineering) at the University of Würzburg, I want to thank my current and former colleagues and administrative staff, with whom I had the pleasure to work with on many projects: Dr. Nikolaus Huber, Dr. Fabian Brosig, Dr. Rouven Krebs, Jóakim von Kistowski, Dr. Simon Spinner, André Bauer, Veronika Lesch, Marwin Zuefle, Dr. Piotr Rygielski, Jürgen Walter, Johannes Grohmann, Stefan Herrnleben, Fritz Kleemann, and Susanne Stenglin.

I would also like to thank Prof. Dr. Ralf Reussner for hosting the Descartes research group at SDQ and providing an enjoyable working environment at KIT until the move in Spring 2014. Many thanks goes also to my former colleagues from the SDQ group at the KIT and the Forschungszentrum Informatik (FZI), especially Dr. Henning Groenda, Dr. Klaus Krogmann, Dr. Qais Noorshams, Dr. Zoya Durdik and more.

My research was influenced by numerous discussions and joint publications under the umbrella of the Cloud working group of SPEC Research: Here, I would like to thank Prof. Dr. Alexandru Iosup, Alexey Ilyushkin, Ahmed Ali-Eldin, Erwin van Eyk, Giorgos Oikonomou, Prof. Tim Brecht, Prof. Cristina Abad, Dr. Rouven Krebs, Dr. Kai Sachs, Alessandro Papadopoulos and more.

I would like to thank Dr. Valentin Curtef (MaxCon Data-Science, Würzburg) for sharing his experience in data analytics and Erich Amrehn (IBM Distinguished Engineer at IBM Research and Development Lab, Böblingen), who supported me since my diploma studies.

I would also like to thank IBM for supporting my doctoral studies with an IBM PhD fellowship. Also, my thanks go to the FZI in Karlsruhe for supporting me with a doctoral fellowship during the starting phase of my PhD.

My special thanks go the my former Master's thesis students who started their academic career after thesis submission and later became colleagues: Jóakim von Kistowski, André Bauer, Veronika Lesch, Marwin Zuefle, and Johannes Grohmann. Furthermore, I would like to thank Andreas Weber, Marcus Wilhem, Benno Heilmann, Torsten Krauss, Joshua Smolka, and Frederik König for supporting me as students as part of their Bachelor's or Master's theses.

Finally, I would like to thank my wife Jasmin for her continuous support and encouragement throughout the years, which made it all possible.

Publication List

Peer Reviewed Journal Articles

[HBK$^+$18] Nikolas Herbst, André Bauer, Samuel Kounev, Giorgos Oikonomou, Erwin van Eyk, George Kousiouris, Athanasia Evangelinou, Rouven Krebs, Tim Brecht, Cristina L. Abad, and Alexandru Iosup. Quantifying Cloud Performance and Dependability: Taxonomy, Metric Design, and Emerging Challenges. ACM Transactions on Modeling and Performance Evaluation of Computing Systems (ToMPECS). To appear.

[IAEH$^+$18] Alexey Ilyushkin, Ahmed Ali-Eldin, Nikolas Herbst, André Bauer, Alessandro V. Papadopoulos, Dick Epema, and Alexandru Iosup. An Experimental Performance Evaluation of Autoscalers for Complex Workflows. ACM Transactions on Modeling and Performance Evaluation of Computing Systems (ToMPECS), 3(2):8:1-8:32, April 2018, ACM, New York, NY, USA.

[vKHK$^+$17] Jóakim von Kistowski, Nikolas Herbst, Samuel Kounev, Henning Groenda, Christian Stier, and Sebastian Lehrig. Modeling and Extracting Load Intensity Profiles. ACM Transactions on Autonomous and Adaptive Systems (TAAS), 11(4):23:1–23:28, January 2017, ACM, New York, NY, USA.

[HHKA14] Nikolas Herbst, Nikolaus Huber, Samuel Kounev, and Erich Amrehn. Self-Adaptive Workload Classification and Forecasting for Proactive Resource Provisioning. Concurrency and Computation - Practice and Experience, John Wiley and Sons, Ltd., 26(12):2053–2078, March 2014. **2nd most cited CCPE article** (according to Google Scholar).

Journal Articles Under Review or Resubmission

[PVB$^+$18] Alessandro Vittorio Papadopoulos, Laurens Versluis, André Bauer, Nikolas Herbst, Jóakim von Kistowski, Ahmed Ali-Eldin, Cristina Abad, J. Nelson Amaral, Petr Tůma, and Alexandru Iosup. Methodological Principles for Reproducible Performance Evaluation in Cloud Computing. Under Resubmission to IEEE Transactions on Cloud Computing (TCC)

[BHS$^+$18] André Bauer, Nikolas Herbst, Simon Spinner, Samuel Kounev, and Ahmed Ali-Eldin. Chameleon: A Hybrid, Proactive Auto-Scaling Mechanism on a Level-Playing Field. IEEE Transactions on Parallel and Distributed Systems (TPDS). Minor Revision July 2018.

Peer-Reviewed International Full Conference Papers

[LBHK18] Veronika Lesch, André Bauer, Nikolas Herbst, and Samuel Kounev. FOX: Cost-Awareness for Autonomic Resource Management in Public Clouds. In Proceedings of the 9th ACM/SPEC International Conference on Performance Engineering (ICPE 2018), New York, NY, USA, April 11-13, 2018. ACM. **Full paper acceptance rate: 23.7%**

[BGHK18] André Bauer, Johannes Grohmann, Nikolas Herbst, and Samuel Kounev. On the Value of Service Demand Estimation for Auto-Scaling. In 9th International GI/ITG Conference on Measurement, Modelling and Evaluation of Computing Systems (MMB 2018). Springer, February 2018.

[IAEH+17] Alexey Ilyushkin, Ahmed Ali-Eldin, Nikolas Herbst, Alessandro V. Papadopoulos, Bogdan Ghit, Dick Epema, and Alexandru Iosup. An Experimental Performance Evaluation of Autoscaling Policies for Complex Workflows. In Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE 2017), l'Aquila, Italy, April 22–26, 2017. ACM, New York, NY, USA. April 2017. **Best Paper Candidate 1/4**.

[SHK+15] Simon Spinner, Nikolas Herbst, Samuel Kounev, Xiaoyun Zhu, Lei Lu, Mustafa Uysal, and Rean Griffith. Proactive Memory Scaling of Virtualized Applications. In Proceedings of the 2015 IEEE 8th International Conference on Cloud Computing (IEEE CLOUD 2015), New York, NY, USA, June 27, 2015, pages 277–284. IEEE. June 2015, **Acceptance Rate: 15%**.

[vKHZ+15] Jóakim von Kistowski, Nikolas Herbst, Daniel Zoller, Samuel Kounev, and Andreas Hotho. Modeling and Extracting Load Intensity Profiles. In Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2015), Firenze, Italy, May 18–19, 2015. Acceptance rate: 29%.

[HKWG15a] Nikolas Herbst, Samuel Kounev, Andreas Weber, and Henning Groenda. BUNGEE: An Elasticity Benchmark for Self-Adaptive IaaS Cloud Environments. In Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2015), Firenze, Italy, May 18–19, 2015. Acceptance rate: 29%.

[HHKA13] Nikolas Herbst, Nikolaus Huber, Samuel Kounev, and Erich Amrehn. Self-Adaptive Workload Classification and Forecasting for Proactive Resource Provisioning. In Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE 2013), Prague, Czech Republic, April 21–24, 2013, pages 187–198. ACM, New York, NY, USA. April 2013. **Among TOP 3 most cited ICPE papers** (according to h5-index on Google Scholar).

Peer-Reviewed International Short Conference Papers

[GHSK18] Johannes Grohmann, Nikolas Herbst, Simon Spinner, and Samuel Kounev. Using Machine Learning for Recommending Service Demand Estimation Approaches (Position Paper). In Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018), Funchal, Madeira, Portugal, March 19-21, 2018. SciTePress. March 2018.

[ZBH+17] Marwin Züfle, André Bauer, Nikolas Herbst, Valentin Curtef, and Samuel Kounev. Telescope: A Hybrid Forecast Method for Univariate Time Series. In Proceedings of the International Work-Conference on Time Series (ITISE 2017), Granada, Spain, September 2017.

[GHSK17] Johannes Grohmann, Nikolas Herbst, Simon Spinner, and Samuel Kounev. Self-Tuning Resource Demand Estimation. In Proceedings of the 14th IEEE International Conference on Autonomic Computing (ICAC 2017), Columbus, OH, July 17–21, 2017.

[HKR13] Nikolas Herbst, Samuel Kounev, and Ralf Reussner. 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, June 24–28, 2013. USENIX. June 2013. **TOP 1 most cited ICAC paper** (according to h5-index on Google Scholar).

Peer-Reviewed Book Chapters

[HBK+17] Nikolas Herbst, Steffen Becker, Samuel Kounev, Heiko Koziolek, Martina Maggio, Aleksandar Milenkoski, and Evgenia Smirni. Metrics and Benchmarks for Self-Aware Computing Systems. In Self-Aware Computing Systems, Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu, editors. Springer Verlag, Berlin Heidelberg, Germany, 2017.

[HAA+17] Nikolas Herbst, Ayman Amin, Artur Andrzejak, Lars Grunske, Samuel Kounev, Ole J. Mengshoel, and Priya Sundararajan. Online Workload Forecasting. In Self-Aware Computing Systems, Samuel Kounev, Jeffrey O. Kephart, Xiaoyun Zhu, and Aleksandar Milenkoski, editors. Springer Verlag, Berlin Heidelberg, Germany, 2017.

Peer-Reviewed Workshop, Tutorial, Poster, and Demonstration Papers

[BHIP17] Gunnar Brataas, Nikolas Herbst, Simon Ivansek, and Jure Polutnik. Scalability Analysis of Cloud Software Services. In Companion Proceedings of the 14th IEEE International Conference on Autonomic Computing (ICAC 2017), Self Organizing Self Managing Clouds Workshop (SOSeMC 2017), Columbus, Ohio, July 17, 2017. IEEE. July 2017.

[BHK17] André Bauer, Nikolas Herbst, and Samuel Kounev. Design and Evaluation of a Proactive, Application-Aware Auto-Scaler (Tutorial Paper). In Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE 2017), L'Aquilla, Italy, April 22, 2017.

[vKHK14d] Jóakim von Kistowski, Nikolas Herbst, and Samuel Kounev. Using and Extending LIMBO for the Descriptive Modeling of Arrival Behaviors (Short Paper & Poster). In Proceedings of the Symposium on Software Performance 2014, Stuttgart, Germany, November 2014, pages 131–140. University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology. November 2014, **Best Poster Award**.

[WHGK14] Andreas Weber, Nikolas Herbst, Henning Groenda, and Samuel Kounev. Towards a Resource Elasticity Benchmark for Cloud Environments. In Proceedings of the 2nd International Workshop on Hot Topics in Cloud Service Scalability (HotTopiCS 2014), co-located with the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014), Dublin, Ireland, March 22, 2014, HotTopiCS '14, pages 5:1–5:8. ACM, New York, NY, USA. March 2014.

[KSH14] Rouven Krebs, Philipp Schneider, and Nikolas Herbst. Optimization Method for Request Admission Control to Guarantee Performance Isolation. In Proceedings of the 2nd International Workshop on Hot Topics in Cloud Service Scalability (HotTopiCS 2014), co-located with the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014), Dublin, Ireland, March 22, 2014. ACM. March 2014.

[vKHK14c] Jóakim von Kistowski, Nikolas Herbst, and Samuel Kounev. Modeling Variations in Load Intensity over Time. In Proceedings of the 3rd International Workshop on Large-Scale Testing (LT 2014), co-located with the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014), Dublin, Ireland, March 22, 2014, pages 1–4. ACM, New York, NY, USA. March 2014.

[vKHK14a] Jóakim von Kistowski, Nikolas Herbst, and Samuel Kounev. LIMBO: A Tool For Modeling Variable Load Intensities (Demo Paper). In Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014), Dublin, Ireland, March 22–26, 2014, ICPE '14, pages 225–226. ACM, New York, NY, USA. March 2014.

Peer-reviewed Software Artifacts

[vKHK14b] Jóakim von Kistowski, Nikolas Herbst, and Samuel Kounev. LIMBO Load Intensity Modeling Tool. Research Group of the Standard Performance Evaluation Corporation (SPEC), Peer-reviewed Tools Repository, 2014.

Technical Reports

[HKO+16] Nikolas Herbst, Rouven Krebs, Giorgos Oikonomou, George Kousiouris, Athanasia Evangelinou, Alexandru Iosup, and Samuel Kounev. Ready for Rain? A View from SPEC Research on the Future of Cloud Metrics. Technical Report SPEC-RG-2016-01, SPEC Research Group — Cloud Working Group, Standard Performance Evaluation Corporation (SPEC), 2016, CoRR, arXiv:1604.03470.

[BvHW+15] Andreas Brunnert, Andre van Hoorn, Felix Willnecker, Alexandru Danciu, Wilhelm Hasselbring, Christoph Heger, Nikolas Herbst, Pooyan Jamshidi, Reiner Jung, Joakim von Kistowski, Anne Koziolek, Johannes Kroß, Simon Spinner, Christian Vögele, Jürgen Walter, and Alexander Wert. Performance-oriented DevOps: A research agenda. Technical Report SPEC-RG-2015-01, SPEC Research Group — DevOps Performance Working Group, Standard Performance Evaluation Corporation (SPEC), August 2015.

[KHvKR11] Michael Kuperberg, Nikolas Herbst, Jóakim Gunnarsson von Kistowski, and Ralf Reussner. Defining and Quantifying Elasticity of Resources in Cloud Computing and Scalable Platforms. Technical report, Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, 76131 Karlsruhe, Germany, 2011.

Theses

[Her12] Nikolas Herbst. Workload Classification and Forecasting. Diploma Thesis, Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, 76131 Karlsruhe, Germany, 2012. **Forschungszentrum Informatik (FZI) Prize "Best Diploma Thesis"**.

[Her11] Nikolas Herbst. Quantifying the Impact of Configuration Space for Elasticity Benchmarking. Study Thesis, Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, 76131 Karlsruhe, Germany, 2011.

Contents

Chapter 1

Introduction

Several goals are pursued in this introductory chapter. First, we motivate the topic and set the context for the work presented in this thesis. Second, we summarize the current state-of-the-art and formulate the problem statement. Section 1.3 highlights two guiding goals of the thesis accompanied by a compact set of research questions to clarify the points of contribution. Third, we outline the four individual contributions of the thesis highlighting the way they build on top of each other and giving first insights on the design and results of the evaluation. Last, by referring to the previously defined chain of research questions, we motivate the structure of the thesis.

1.1 Motivation and Context

Starting roughly a decade ago in the year 2006 with the opening of the first commercial cloud service offerings by Amazon Web Services (AWS)[1], cloud computing became a hype topic in both academia and industry. Two years later, competition in the field of cloud computing started after Microsoft, Google and IBM entered the market. Now, these three players are estimated to hold approximately one third of the market, while AWS is holding another third as the market leader. The cloud computing market experienced mind-blowing growth rates and reached according to a recent Gartner report[2] a market size of US$ 247 billion in 2017. In the research community during the last decade, countless publications accompanied by a number of newly established flagship conferences (e.g., IEEE Cloud, ACM Symposium on Cloud Computing SoCC) and journals (e.g., IEEE Transactions on Cloud Computing).

Cloud computing is now developing towards a settling and maturation phase with growth rates falling below 18% as predicted by Gartner. Nevertheless, cloud computing is continuing to transform center operations. Co-Chief-Executive-Officer Marc Hurd[3] of the Oracle Corporation predicts that by 2025, 80% of the corporate data centers will disappear, as applications in production are increasingly moving to cloud environments. Especially in the ongoing settling and maturation phase of cloud computing offerings, the scientific progress and industry growth depend on established principles for measuring and reporting cloud system quality attributes, as highlighted in a recent Gigaom analyst report[4].

[1]The History of AWS: `https://www.computerworlduk.com/galleries/cloud-computing/aws-12-defining-moments-for-the-cloud-giant-3636947/`

[2]Gartner Cloud Report 2017: `https://www.gartner.com/newsroom/id/3616417`

[3]Oracle CEO Marc Hurd: `https://markhurd.com/about-mark-hurd/`

[4]Gigaom Analyst Report: The Importance of Benchmarking Clouds:

According to a Gartner report from 2009[5], the major selling point of cloud computing offerings is their pay-per-use paradigm with no need for long term investments and operation costs. In combinations with the enabling technology of hardware virtualization, the pay-per-use service model offers the ability to elastically adapt the allocated computing resources to the current demand. Cloud operators can manage their physical resources - at least in theory - in a way to optimize efficiency. This sometimes involves selling more virtual resources than physically available - also known as overbooking - while still keeping the operation risk for the customer at a tailored minimum by giving them the opportunity to define resource priorities.

1.2 State-of-the-Art and Problem Statement

Elastic scaling of allocated resources is realized by so-called auto-scaling mechanisms that are based on monitored performance measures supposed to match the current demand in a way that the performance remains stable while resources are utilized efficiently. The common state of practice is the use of simplistic threshold-based mechanisms that due to their reactive nature incur performance degradation during the periods of provisioning delays. In contrast, the responsibilities of cloud infrastructure operators are highly complex and dynamic trade-offs to optimize workload-dependent resource placement, co-location, load-balancing, sizing and routing questions. As a result of these complex and interwoven challenges that are usually addressed in a best-effort manner, the cloud customers experience a high level of performance variability that still hinders mission critical applications to leverage cloud solutions [IYE11].

In the course of the last decade, a high-number of auto-scalers has been proposed in the literature trying to overcome the limitations of reactive mechanisms by employing proactive prediction methods as systematically surveyed by Lorido-Botran et al. [LBMAL14]. Lorido-Botran et al. propose to group auto-scalers into approaches from queueing theory, control theory, time series analysis and reinforcement learning.

Proactive auto-scaling methods based on time-series analysis estimate resource utilization, response times or system load using simplistic regression techniques [IDCJ11], histogram analysis [USC$^+$08] or based on black-box methods like auto-regressive integrated moving averages (ARIMA) [FPK14]. The latter have known deficits with respect to time-to-result and accuracy in scenarios with complex seasonal patterns and metric resolutions in finer than half-hourly averages [HHKA14]. Other approaches, e.g., the closed-source auto-scalers [NSG$^+$13, SSGW11] with involvement from Google, leverage signal processing methods to characterize the frequency spectrum via Fourier or wavelet transformations without supporting the capability to capture trends. Auto-scaling mechanisms that leverage queueing theory are in most cases combined with one of the other approaches. Control theory approaches share the limitation of short prediction horizons, whereas reinforcement learning approaches rely on training phases and (in case of system changes) on calibration periods in a way that might be unfeasible in production environments.

https://gigaom.com/report/the-importance-of-benchmarking-clouds/

[5]Gartner Highlights Five Attributes of Cloud Computing: https://www.gartner.com/newsroom/id/1035013

Furthermore, the majority of proposed auto-scaling mechanisms share the assumption of linear and endless scalability of the cloud hosted applications. In practice, this assumption is not realistic due to communication overheads, overbooking practices and stateful application services.

With few exceptions, existing proactive auto-scaling mechanisms proposed in the literature remain without shared code artifacts. This fact highly reduces the reproducibility of experiment results and comparability between alternatives. As a result, the adoption of proactive auto-scalers in production is still very low due to the high risk of relying on a single proactive method based on which scaling decisions are made.

According to our recent literature survey [PVB$^+$18], approximately 40% of cloud research publications are evaluated using simulation. As this survey also covers auto-scaling mechanisms, we can say that it is common practice to evaluate auto-scaling approaches in simulative frameworks. Experimental auto-scaler evaluations are usually conducted in a case-study like manner by showing that the proposed method is capable to improve the service level compliance compared to an arbitrary static resource assignment in a small number of scenarios. Those evaluation scenarios are often driven by synthetic load intensity profiles such as sinus signals or saw tooth patterns that lack in representativeness, especially as they are easy to predict by a proactive auto-scaling method. Real-world load profiles exhibit a mixture of trend, seasonal, burst and noise components and are thus complex to capture, share, modify and replay at scale.

The process of rating cloud infrastructure offerings in terms of the quality of their elastic scaling remains undefined due to the lack of precisely defined meaningful metrics that capture the quality of elastic resource adaptations achieved in practice together with an established set of measurement run rules. Besides cost/efficiency-oriented measures or experienced end-user performance that both only allow for indirect characterization of elastic adaptations, only low-level metrics like the technical provisioning time to bring up a certain resource on average have been proposed and used in practice.

Thus, clear guidance for the selection and configuration of an auto-scaler for a given context is not available. The few existing proactive auto-scalers are optimized in a highly application specific way and usually kept undisclosed (e.g., Netflix's Predictive Auto-Scaler Scryer[6]) while in contrast many other artifacts of cloud software stacks are open-sourced (e.g., Netflix Open Source Platform[7] with currently over 130 public project repositories).

In summary, the problem addressed in this thesis can be formulated as follows: The risk of using a proactive auto-scaling algorithm in a production environment is still high with the result of a low adoption. Existing solutions are either undisclosed, tailored solutions or approaches described in the literature are not tested with the help of a standardized benchmark that would allow for fair comparisons.

[6]Netflix's Predictive Auto-Scaler Scryer: `https://medium.com/netflix-techblog/scryer-netflixs-predictive-auto-scaling-engine-a3f8fc922270`

[7]Netflix Open Source Platform: `https://github.com/Netflix`

1.3 Guiding Goals and Research Questions

Having identified a number of deficits in the current state-of-the-art of auto-scaler design and assessment, we now formulate two overarching guiding goals, each accompanied by four research questions (RQ) to be addressed and answered in the course of this thesis. The thesis is structured in three parts. Part I presents the background and foundations needed for understanding the contributions of this thesis, as well as and an extensive summary of the state-of-the-art. Part II is focused on the first guiding goal with its set of RQs. Finally, Part III addresses the second guiding goal with its four RQs.

Goal A: Establish a benchmark for state-of-the-art auto-scalers to increase trust in novel proactive mechanisms fostering a broader adoption of auto-scalers in production.

To approach this guiding goal, we split the respective challenges into several subgoals. First, we formulate two research questions expressing the need to define, modify and generate representative load intensity profiles in a flexible manner to trigger an realistic amount of resource adaptations. Second, two additional research questions capture the need for a sound definition of metrics and measurement methodology as building blocks for an elasticity benchmark.

RQ A.1: How to model load intensity profiles from real-world traces in a descriptive, compact, flexible and intuitive manner?

RQ A.2: How to automatically extract load intensity profile models from existing traces with a reasonable accuracy and computation time?

RQ A.3: What are meaningful intuitive metrics to quantify accuracy, timing and stability as quality aspects of elastic resource adaptations?

RQ A.4: How can the proposed elasticity metrics be measured in a reliable and repeatable way to enable fair comparisons and consistent rankings across systems with different performance?

Goal B: Reduce the risk of using novel auto-scalers in operation by leveraging multiple different proactive mechanisms applied in combination with a conventional reactive mechanism.

We approach the challenge of this goal, first, by proposing a novel hybrid auto-scaling mechanism and then evaluating its performance in detail against existing state-of-the-art auto-scalers. The comprehensive auto-scaler evaluation and comparison is enabled via results from Goal 1. Second, we address deficits of current time-series forecasting methods by proposing a hybrid forecasting mechanism and showcasing its benefits in an auto-scaling usage context.

RQ B.1: How can conflicting auto-scaling decisions from independent reactive and proactive decision loops be combined to improve the overall quality of auto-scaling decisions?

RQ B.2: How well does the proposed hybrid auto-scaling approach perform compared to state-of-the-art mechanisms in realistic deployments and application scenarios?

RQ B.3: How can a hybrid forecast approach based on decomposition be designed to be capable of providing accurate and fast multi-step-ahead forecasts of complex seasonal time-series[8] within seconds?

RQ B.4: Is such a hybrid forecast approach capable to improve the performance and reliability of auto-scaling mechanisms?

1.4 Contribution and Evaluation Summary

After defining the set of two guiding goals and research questions, we now summarize four core contributions of this thesis. Each contribution addresses two of the research questions and thus a part of either Goal A or B. The contributions have in common that they build on top of each other and as a results are highly integrated.

Contribution I: Approaching Goal A and answering RQ A.1 and RQ A.2, we propose a descriptive load profile modeling framework together with automated model extraction from recorded traces to enable reproducible workload generation with realistic load intensity variations. The model design follows the approach of decomposing recorded data into its deterministic components of piece-wise defined trends and recurring or overarching seasonal patterns, while also allowing to model stochastic noise distributions and explicit bursts. The components are in principle piece-wise defined mathematical functions that can be nested and combined with mathematical operations in a tree of functions. Automated extraction processes detect frequencies and decompose recorded traces based on an efficient heuristics. We name the proposed model Descartes Load Intensity Model (DLIM) with its Limbo framework that provides key functionality to stress and benchmark resource management approaches in a representative and fair manner.

We evaluate the DLIM model expressiveness by applying and comparing alternative configurations of the extraction processes to ten different real-world traces that cover between two weeks and seven months of data. Automatically extracted DLIM model instances exhibit an average modeling error of 15.2%. In terms of accuracy and processing speed, our proposed extraction methods based on descriptive models deliver better or similar results compared to existing non-descriptive time series decomposition methods. In contrast to DLIM models, classical time-series decomposition approaches deliver three rows of data points as output as opposed to a compact and flexible descriptive model. This contribution resulted in an ACM Transactions on Autonomous and Adaptive Systems (TAAS) journal article published in 2017 [vKHK+17].

[8]With a complex seasonal time-series we refer to data points over time with resolution in the order of minutes, a comprehensive history and a mixture of non-trivial recurring patterns.

Contribution II: To address RQ A.3 and RQ A.4, we first provide a definition of the term elasticity in cloud computing[9] , describing the core aspects of elasticity and distinguishing it from related terms like efficiency and scalability. Furthermore, we propose a set of intuitive metrics for quantifying timing, stability and accuracy aspects of elasticity. Based on these metrics, which were endorsed by the SPEC Research Group [HKO$^+$16], we propose a novel approach for benchmarking the elasticity of auto-scalers deployed in Infrastructure-as-a-Service (IaaS) cloud platforms independent of the performance exhibited by the provisioned underlying resources. The proposed Bungee elasticity benchmarking framework leverages DLIM's modeling capabilities to generate realistic load intensity profiles.

In the corresponding evaluation, we show for each of the proposed metrics that they provide a consistent ranking of elastic systems on an ordinal scale. The Bungee measurement methodology can achieve reproducible results in a controlled environment as well as results with an acceptable variation in uncontrolled environments like public clouds. Finally, we present an extensive case study of real-world complexity demonstrating that the proposed approach is applicable in realistic scenarios and can cope with different performance levels of the underlying resources. This contribution resulted in a full research paper [HKWG15b], and a follow up journal article published in ACM Transactions on Modeling and Performance Evaluation of Computing Systems (ToMPECS) [IAEH$^+$18] based on invitation as best paper candidate at the International Conference on Performance Engineering (ICPE) [IAEH$^+$17]. The elasticity metric definitions became a substantial part of an ACM ToMPECS article [HBK$^+$18]. Furthermore, the Bungee measurement methodology and results became a significant contribution to an article [PVB$^+$18] (under resubmission) on defining and assessing principles for repeatable experimentation in cloud environments.

Contribution III: We tackle the challenge of reducing the risk of relying on a single proactive auto-scaler by answering RQ B.1 and RQ B.2 and proposing a new hybrid auto-scaling mechanism called Chameleon. We combine multiple different proactive methods coupled with a reactive fallback mechanism. Chameleon employs on-demand, automated time series-based forecasting methods to predict the arriving load intensity in combination with run-time service demand estimation techniques to calculate the required resource consumption per work unit without the need for application instrumentation. It can also leverage application knowledge by solving product-form queueing networks used to derive optimized scaling actions. The Chameleon approach is first in resolving conflicts between reactive and proactive scaling decisions in an intelligent way. Chameleon leverages as building blocks core developments of the Descartes Research Group like the Descartes Modeling Language (DML) [HBS$^+$17] to capture data center application deployments and the Library for Resource Demand Estimation (LibReDE) [SCBK15]. The benefits of using self-tuning service demand

[9]Our definition of elasticity in cloud computing [HKR13] from 2013 became highly adopted: The short paper is according to h5-index on Google scholar Top 1 of most cited papers presented at the International Conference on Autonomic Computing (c.f. `https://scholar.google.de/citations?hl=de&view_op=list_hcore&venue=eAgro8m_j_AJ.2017`) and has been picked by Wikipedia.org in a respective encyclopedic article (c.f. `https://en.wikipedia.org/wiki/Elasticity_(cloud_computing)`).

estimation techniques [GHSK17] based on wide-spread utilization measurements as input for auto-scaling mechanisms were published in a full research paper [BGHK18].

We conduct a comprehensive auto-scaler competition enabled by leveraging the results from Contributions I and II: We benchmark Chameleon against four different state-of-the-art proactive auto-scalers and a standard threshold-based one in three different cloud environments: (i) a private CloudStack-based cloud, (ii) the public AWS EC2 cloud, as well as (iii) an OpenNebula-based shared IaaS cloud. We generate five different representative load profiles each taken from different real-world system traces leveraging the functionality of Limbo (Contribution I) and Bungee (Contribution II) to drive a CPU-intensive benchmark workload. The workload is taken from the SPEC Server Efficiency Rating (SERT) tool and computes matrix decompositions. Overall, Chameleon achieves the best and most stable scaling behavior based on user and elasticity performance metrics, analyzing the results from 400 hours of aggregated experiment time. We demonstrate that by combining scaling decisions from reactive and proactive cycles based on our proposed conflict resolution heuristic, the auto-scaling performance of Chameleon is improved. This contribution resulted in a submitted article to the IEEE Transactions on Parallel and Distributed Systems (TPDS) [BHS+18][10].

Contribution IV: As a side-contribution of this thesis, we address RQ B.3 and RQ B.4 by proposing a self-aware forecasting prototype called Telescope. It incorporates multiple individual forecasting methods by decomposing the univariate time series into the components trend, season, and remainder. First, the frequency is determined, anomalies are detected and removed, and the type of decomposition is estimated based on a majority vote of tailored tests for different multiplicative or additive decompositions. After decomposing, the ARIMA (autoregressive integrated moving averages) method without seasonality is applied on the trend pattern, whereas the detected seasonality is simply continued. Moreover, the single periods are clustered in order to learn categorical information. The cluster labels are forecast by applying artificial neural networks. This helps to automatically distinguish between different type of days. Lastly, eXtreme Gradient Boosting (XGBoost), a novel and promising method published in 2016 [CG16] is used to learn the dependency between all previously extracted covariates and to combine the forecasts of the individual components.

The preliminary evaluation indicates based on two different traces that an early implementation of the Telescope approach outperforms the best of six state-of-the-art competitors in terms of accuracy. Telescope improves the time-to-result by up to a factor of 19 compared to the three most competitive forecasting methods. In a case study, we demonstrate that Telescope is capable of further improving Chameleon's auto-scaling performance compared to the formerly used state-of-the-art forecasting method tBATS or seasonal ARIMA. We note that Contribution IV is included in this thesis as a side-contribution due to its initial stage of research and current and planned future works based on the Telescope prototype are ongoing. The idea together with a preliminary evaluation of Telescope is published in an extended abstract [ZBH+17]

[10] The respective IEEE TPDS submission is under review since October 2017.

with more extensive publications planned.

We are confident that the four core contributions of this thesis have the potential to change the way cloud resource management approaches are assessed leading to improving the quality of autonomic management algorithms as a result. To support such a development, we published code artifacts of all four contributions of this thesis as open-source tools actively maintained and accompanied by user and developer guides [BHK17][11].

Beyond the known limitations and assumptions explicitly stated in the individual chapters, we see a number of arising challenges for future research in cloud resource management and its assessment methods: (i) The adoption of containerization on top of virtual machine instances introduces another level of indirection. As a result, the nesting of virtual resources increases resource fragmentation and causes unreliable provisioning delays. (ii) Furthermore, virtualized compute resources tend to become more and more inhomogeneous associated with various priorities and trade-offs. (iii) Due to DevOps practices, cloud hosted service updates are released with a higher frequency, which impacts the dynamics in user behavior. Auto-Scalers are increasingly required to self-adapt to changing service demands and arrival patterns.

1.5 Thesis Outline

This thesis is structured into five parts featuring twelve main chapters, including this introductory chapter (Chapter 1), plus one appendix chapter (Chapter 12.2.3).

Part I starts with a foundational chapter (Chapter 2) to enable a common understanding, before reviewing the various domains of related work including an elaboration of the deficits of existing approaches and the differences of our work to the state-of-the-art (Chapter 3).

Part II approaches Goal A and its four RQs A.1-4 in two separate chapters. Chapter 4 defines the DLIM Load Intensity Model together with automated extraction processes. Chapter 5 features the definition of a set of elasticity metrics together with metric aggregation methods before defining the Bungee measurement methodology.

In Part III, we address Goal B and its four RQs B.1-4 in, two separate chapters. Chapter 6 introduces the hybrid auto-scaling mechanism Chameleon with an intelligent scaling decision conflict resolution. Chapter 7 outlines as a side-contribution of this thesis, the approach we follow in the design of the hybrid forecasting method Telescope for complex seasonal time-series.

Part IV consists of four evaluation chapters (Chapter 8 to Chapter 11) that answer and underline what is postulated in the research questions as addressed in Contributions I to IV, namely: (i) a DLIM model accuracy assessment, (ii) sets of experiments evaluating the proposed elasticity metrics, including an analysis of repeatability and a comprehensive case-study, (iii) a broad auto-scaler competition underlining the superior performance of the proposed Chameleon auto-scaler and, finally, (iv) a preliminary evaluation of the early Telescope forecast prototype applied to two selected time series comparing against state-of-the-art approaches and a case-study to highlight the potential of Telescope to improve auto-scaler performance.

[11] Descartes Tools: `https://descartes.tools`

The remaining Part V summarizes the results of the thesis also motivating the challenges for future research partially enabled by the contributions of this thesis. An appendix chapter includes the detailed results of the conducted auto-scaler competition in the evaluation chapter on the Chameleon auto-scaler (Chapter10).

Part I

Foundations and State-of-the-Art

Chapter 2

Foundations

In this chapter, we aim to lay the ground for a common understanding of the contributed approaches and interpretation of the evaluation results.

We assume that the reader is familiar with basics of queueing theory [HB13] and the operational laws of performance engineering [MDA04], as well as aware of virtualization technologies and the notion of cloud computing including its settled abstraction levels: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service(PaaS), and Software-as-a-Service (SaaS) [AFG$^+$10].

First, we introduce and discuss our meanwhile established definition of elasticity in cloud computing, before we cover the definition of self-aware computing systems together with arguments why we think our proposed auto-scaling and forecasting approaches fulfill this definition and can be considered "self-aware". Then, we define our notion of load intensity profiles stepping into the field of time series analysis: We give an overview on error metrics, and summarize two approaches for time series decomposition (STL and BFAST). We discuss the distinction of seasonal vs. cyclic time series and summarize methods for frequency estimation using auto-correlation and a Fourier-transformation-based periodigram. Towards the end of this chapter, we cover relevant methods for time series forecasting from the fields of machine learning (ANN, XGBoost) and statistical modeling frameworks (ARIMA, ETS and tBATS).

2.1 Elasticity in Cloud Computing

Elasticity has originally been defined in physics as a material property capturing the capability of returning to its original state after a deformation. In economical theory, informally, elasticity denotes the sensitivity of a dependent variable to changes in one or more other variables [CW09]. In both cases, elasticity is an intuitive concept and can be precisely described using mathematical formulas.

The concept of elasticity has been transferred to the context of cloud computing and is commonly considered as one of the central attributes of the cloud paradigm [PSB$^+$09]. For marketing purposes, the term elasticity is heavily used in cloud providers' advertisements and even in the naming of specific products or services. Even though tremendous efforts are invested to enable cloud systems to behave in an elastic manner, no common and precise understanding of this term in the context of cloud computing has been established so far, and no ways have been proposed to quantify and compare elastic behavior. To underline this observation, we cite five definitions of elasticity demonstrating the inconsistent use and understanding of the term:

1. ODCA, Compute Infrastructure-as-a-Service [OCD12]
   ”[...] defines elasticity as the expandability of the solution [...] Centrally, it is the ability to scale up and scale down capacity based on subscriber workload.”

2. NIST Definition of Cloud Computing [MG11]
   ”Rapid elasticity: Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.”

3. IBM, Thoughts on Cloud, Edwin Schouten, 2012 [Sch12]
   ”Elasticity is basically a ’rename’ of scalability [...]” and ”removes any manual labor needed to increase or reduce capacity.”

4. Rich Wolski, CTO, Eucalyptus, 2011 [Wol11]
   ”Elasticity measures the ability of the cloud to map a single user request to different resources.”

5. Reuven Cohen, 2009 [Coh09]
   Elasticity is ”the quantifiable ability to manage, measure, predict and adapt responsiveness of an application based on real time demands placed on an infrastructure using a combination of local and remote computing resources.”

Definitions (1), (2), and (3) in common describe elasticity as the scaling of system resources to increase or decrease capacity, whereby definitions (1), (2) and (5) specifically state that the amount of provisioned resources is somehow connected to the recent demand or workload. In these two points there appears to be some consent. Definitions (4) and (5) try to capture elasticity in a generic way as a ’quantifiable’ system ability to handle requests using different resources. Both of these definitions, however, neither give concrete details on the core aspects of elasticity, nor provide any hints on how elasticity can be measured. Definition (3) assumes that no manual work at all is needed, whereas in the NIST definition (2), the processes enabling elasticity do not need to be fully automatic. In addition, the NIST definition adds the adjective ’rapid’ to elasticity and draws the idealistic picture of ’perfect’ elasticity where endless resources are available with an appropriate provisioning at any point in time, in a way that the end-user does not experience any performance variability.

We argue that existing definitions of elasticity fail to capture the core aspects of this term in a clear and unambiguous manner and are even contradictory in some parts. To address this issue, we propose a new refined definition of elasticity considering in detail its core aspects and the prerequisites of elastic system behavior (Section 2.1). Thereby, we clearly differentiate elasticity from its related terms scalability and efficiency.

We first describe some important prerequisites in order to be able to speak of elasticity, present a new refined and comprehensive definition, and then analyze its core aspects and dimensions. Finally, we differentiate between elasticity and its related terms scalability and efficiency.

2.1.1 Prerequisites

The scalability of a system including all hardware, virtualization, and software layers within its boundaries is a prerequisite for speaking of elasticity. Scalability is the ability of a system to sustain increasing workloads with adequate performance, provided that hardware resources are added. In the context of distributed systems, it has been defined by Jogalekar and Woodside [JW00], as well as in the works of Duboc et al. [DRW07], where also a measurement methodology is proposed.

Given that elasticity is related to the ability of a system to adapt to changes in workloads and demanded resource units, the existence of at least one adaptation process is typically assumed. The process is normally automated, but it could contain manual steps. Without a defined adaptation process, a scalable system cannot scale in an elastic manner, as scalability on its own does not include temporal aspects.

When evaluating elasticity, the following points need to be checked beforehand:

- Automated Scaling:
  What adaptation process is used for automated scaling?
- Elasticity Dimensions:
  What is the set of resource types scaled as part of the adaptation process?
- Resource Scaling Units:
  For each resource type, in what unit is the amount of allocated resources varied?
- Scalability Bounds:
  For each resource type, what is the upper bound on the amount of resources that can be allocated?

2.1.2 Definition of Elasticity in Cloud Computing

> Elasticity 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.

Our meanwhile established definition of elasticity in cloud computing has been adopted by the Wikipedia.org encyclopedia[1].

2.1.3 Elasticity Dimensions and Core Aspects

Any given adaptation process is defined in the context of at least one or possibly multiple types of resources that can be scaled up or down as part of the adaptation. Each resource type can be seen as a separate dimension of the adaptation process with its own elasticity properties. If a resource type consists of other resources types, like in the case of a virtual

[1] Elasticity in Cloud Computing on Wikipedia.org: `https://en.wikipedia.org/wiki/Elasticity_(cloud_computing)`

machine having assigned CPU cores and memory, elasticity can be considered at multiple levels. Normally, resources of a given resource type can only be provisioned in discrete units like CPU cores, VMs, or physical nodes. For each dimension of the adaptation process with respect to a specific resource type, elasticity captures the following core aspects of the adaptation:

Timing The timing aspect is captured by the time shares a system is in an under-provisioned, over-provisioned or perfect state and by the stability or oscillations of adaptations.

Accuracy The accuracy of scaling is defined as the relative deviation of the current amount of allocated resources from the actual resource demand on average.

A direct comparison between two systems in terms of elasticity is only possible if the same resource types (measured in identical units) are scaled. To evaluate the actual elasticity in a given scenario, one must define the criterion through which the amount of provisioned resources is considered to match the actual demand needed to satisfy the system's given performance requirements. Based on such a matching criterion, specific metrics that quantify the above mentioned core aspects, as discussed in more detail in Section 5.2, can be defined to quantify the practically achieved elasticity in comparison to the hypothetical optimal elasticity. The latter corresponds to the hypothetical case where the system is scalable with respect to all considered elasticity dimensions without any upper bounds on the amount of resources that can be provisioned and where resources are provisioned and de-provisioned immediately as they are needed exactly matching the actual demand at any point in time. Optimal elasticity, as defined here, would only be limited by the granularity of resource scaling units.

2.1.4 Differentiation to Scalability and Efficiency

This paragraph discusses the conceptual differences between elasticity and the related terms scalability and efficiency.

Scalability is a prerequisite for elasticity, but it does not consider temporal aspects of how fast, how often, and at what granularity scaling actions can be performed. Scalability is the ability of the system to sustain increasing workloads by making use of additional resources, and therefore, in contrast to elasticity, it is not directly related to how well the actual resource demands are matched by the provisioned resources at any point in time.

Efficiency expresses the amount of resources consumed for processing a given amount of work. In contrast to elasticity, efficiency is directly linked to resource types that are scaled as part of the system's adaptation mechanisms. Normally, better elasticity results in higher efficiency. This implication does not apply in the other direction, as efficiency can be influenced by other factors (e.g., different implementations of the same operation).

2.2 Self-Aware Computing Systems

In 2017, Kounev et al. define self-aware computing in their book [KKMZ17] in its first chapter [KLB$^+$17, p. 5] as follows:

> Self-aware computing systems are computing systems that:
>
> 1. learn models capturing knowledge about themselves and their environment (such as their structure, design, state, possible actions, and runtime behavior)
> 2. reason using the models (e.g., predict, analyze, consider, and plan) enabling them to act based on their knowledge and reasoning (e.g., explore, explain, report, suggest, self-adapt, or impact their environment)
>
> in accordance with higher-level goals, which may also be subject to change.

A self-aware computing system may be „built by an entity with [...] higher-level goals in mind. This entity may be a human [...] or a set of humans [...] but it does not necessarily have to be“ [KLB$^+$17, p. 5]. Kounev et al. define two major distinctive characteristics of a self-aware computing system: „The capability to learn models on ongoing basis, capturing knowledge relevant to the purpose“ [KLB$^+$17, p. 5]. And the system „must be able to use the models to reason about this knowledge“ [KLB$^+$17, p. 5]. Both characteristics are driven by higher-level goals. This means that the goals „are at a higher level of abstraction than the system and [...] are not under its direct control“ [KLB$^+$17, p. 5]. Kounev et al. describe five types of reasoning [KLB$^+$17, p. 7]: (i) „predict the load of an IT system [...] in a future time horizon“. (ii) „predict the system performance (e.g., response time) for a given workload and resource allocation“. (iii) „predict the expected impact of a given system adaptation action [...] on the end-to-end system performance“. (iv) „determine how much resources need to be added to ensure that performance requirements are satisfied under an increasing system workload“. (v) „estimate the system's energy consumption at runtime and compare it to other system configurations in order to select an optimal configuration“.

The self-aware learning and reasoning loop (LRA-M) is a concept „capturing the main activities in a self-aware computing system“ [KLB$^+$17, p. 13]. Figure 2.1 illustrates the self and its interfaces, as well as the activities within the self. These activities are driven by goals and its observations represented as empirical observations. The observations are used for the ongoing process of learning models, which are then used as basis for the system's reasoning process. The reasoning process may trigger actions affecting the behavior of the system and possibly impacting the environment [KLB$^+$17].

Our proposed hybrid auto-scaling mechanism Chameleon (c.f. Chapter 6) can be classified as self-aware as it learns and updates models based on observing the cloud and applications to scale, the arrival and the service processes. With the help of these models it reasons about the future load and the required resource amount to serve all requests. This reasoning results in scaling decisions to adapt the system to varying load trying to fulfill as possible the higher goals of efficient resource usage and end-user satisfaction via stable performance. Our forecasting approach Telescope learns and updates its internal models

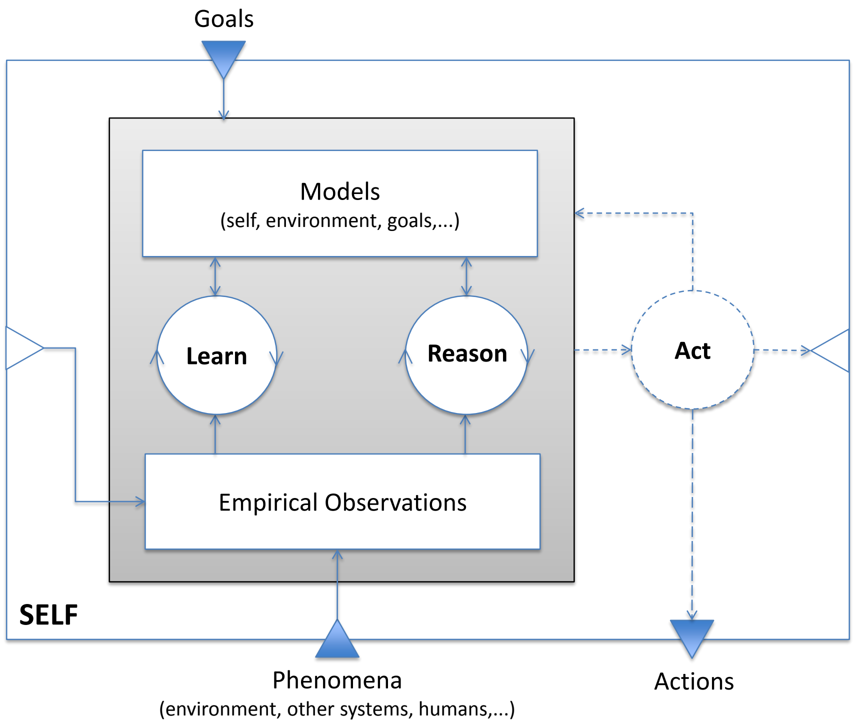

Goals
Models
(self, environment, goals,...)
Learn
Reason
Act
Empirical Observations
SELF
Phenomena
(environment, other systems, humans,...)
Actions

Figure 2.1: Self-aware learning and reasoning loop: LRA-M loop [KLB$^+$17].

capturing structural knowledge about observed time series values in terms of classification of patterns over time and composition of values at instances of time. It predicts future values for further reasoning according to the higher goals of high accuracy, low variance and a short time-to-result.

2.3 Time Series Analysis

In this section, we first define the terms time series and load intensity profiles, before we summarize error measures for time series models (internal errors) and forecast errors. We summarize two decomposition approaches for time series - namely STL and BFAST. Before we discuss two ways to estimate domination frequencies of time series, we highlight the differences between cyclic and seasonal time series. At the end of this section, we present in a compact way two approaches for time series forecasting from machine learning (ANN, XGBoost) as well as two statical times series modeling frameworks for predictions (seasonal ARIMA and tBATS).

2.3.1 Time Series and Load Intensity Profiles

Our proposed modeling framework DLIM has been designed to capture variations of load intensity in the form of user, job, or request arrival rates over time. The models employ an open workload view.

Open workloads are defined as by Schröder, Wierman and Harchol-Balter [SWHB06] as workloads in which new jobs arrive independently of job completion. We use the term load to denote user, job or request arrival rates containing the actual work units. Open workloads are typical for cloud environments as users are usually unaware of other users or the current workload.

In this thesis, load intensity denotes the arrival rate of abstract workload units (e.g., users, jobs, sessions, or requests) at a given point in time. The function $r(t)$ describes the load intensity at that point in time (t) as follows:

$$r(t) = R'(t) \quad with \quad R(t) = |\{u_{t_0} | t_0 \leq t\}| \tag{2.1}$$

where $R(t)$ is the amount of work units u_{t_0} (set cardinality of the set containing all u_{t_0}) with their respective arrival time t_0 which precedes or equals our time of observation t. This means that all work units u_{t_0} have arrived up until time t. $r(t) = R'(t)$ is the derivative of $R(t)$.

A load intensity profile is the function that describes the load intensity over time. Real world examples of such profiles are shown in several figures throughout this paper, including Figure 8.1.

In general, a load intensity profile can be seen as a time series. A time series is a sequence of data points ordered by equidistant time steps. In other words, let A be a countable set. A time series is a function

$$Y : A \to \mathbb{R}.$$

As time series emerge when observing phenomena from reality, A is usually finite. Also, it is useful to think of A as a subset of $\mathbb{Q}$, because we deal with a discrete set, but still we might encounter sampling frequencies so high that our time series would start to look continuous. Thus, we have no access to derivatives of a function, at least not in the sense of calculus.

2.3.2 Error Measures for Forecasting Results

In order to evaluate the accuracy of a forecasting result, several error measures can be used. Each method has its use cases and thus, most measures have to be used very carefully. For the calculation of the following error measures, the required basic notation is listed in Table 2.1.

According to Hyndman and Koehler, all error measures of forecasting accuracy can be grouped into four categories, i.e., scale-dependent measures, measures based on percentage errors, measures based on relative errors, and scaled error measures [HK06].

Table 2.1: Notations for error measures.

Symbol	Meaning
Y_t	observation at time t
F_t	forecast at time t
$e_t = Y_t - F_t$	forecast error at time t
$e_t^* = Y_t - F_t^*$	forecast error at time t of a standard forecasting method
$p_t = 100 \cdot \frac{e_t}{Y_t}$	percentage error at time t
$r_t = \frac{e_t}{e_t^*}$	relative error at time t
$q_t = \frac{e_t}{\frac{1}{n-1} \cdot \sum_{i=2}^{n} \lvert Y_i - Y_{i-1} \rvert}$	scaled error at time t

Scale-Dependent Measures

The value of scale-dependent measures depends highly on the scale of the input data. Thus, scale-dependent measures are not feasible for comparisons across different data sets containing time series with varying scales. However, scale-dependent measures are useful for comparing the errors of different forecasting methods on the same data set. A set of scale-dependent error measures is presented in Table 2.2.

Table 2.2: Scale-dependent measures with formulas.

Error Measure	Formula
Mean Square Error (MSE)	$MSE = mean(e_t^2)$
Root Mean Square Error (RMSE)	$RMSE = \sqrt{MSE}$
Mean Absolute Error (MAE)	$MAE = mean(\lvert e_t \rvert)$
Median Absolute Error (MdAE)	$MdAE = median(\lvert e_t \rvert)$

Measures based on Percentage Errors

In contrast to scale-dependent error measures, measures based on percentage errors are independent of the scale of the input. That is, these measures are very useful for comparing forecast accuracy across data sets with arbitrary scale. A set of measures based on percentage errors is listed in Table 2.3. However, measures based on percentage errors also have drawbacks. Firstly, these error measures assume a meaningful zero. If any observation in scope is zero, these measures are infinite or even undefined. On top of that, measures based on percentage errors are having issues with very small values close to zero. They show an extremely skewed distribution if any observation in scope is close to zero. Thus, these measures are not feasible for data sets with many small values. Lastly, the MAPE and MdAPE punish positive errors harder than negative errors. This can be explained by the following example. Assume the forecast and original time series to have the same algebraic sign. Now, if the forecast is always smaller than the original time series, the MAPE will never exceed 100% since it is normalized by the observation values. However, positive errors, i.e., overestimating forecasts, can exceed a MAPE value of 100%. Therefore, sMAPE and sMdAPE are introduced to solve this problem. Yet, they do not completely succeed since lower forecasts are punished harder than higher forecasts. The factor 200 is set by definition of Armstrong [Arm85]. In practice, sMAPE and sMdAPE are often used with factor 100, so that the result lies between 0% and 100%.

Table 2.3: Measures based on percentage errors with formulas.

Error Measure	**Formula**
Mean Absolute Percentage Error (MAPE)	$MAPE = mean(\|p_t\|)$
Median Absolute Percentage Error (MdAPE)	$MdAPE = median(\|p_t\|)$
Root Mean Square Percentage Error (RMSPE)	$RMSPE = \sqrt{mean(p_t^2)}$
Root Median Square Percentage Error (RMdSPE)	$RMdSPE = \sqrt{median(p_t^2)}$
Symmetric MAPE (sMAPE)	$sMAPE = mean(200 \cdot \frac{\|Y_t - F_t\|}{Y_t + F_t})$
Symmetric MdAPE (sMdAPE)	$sMdAPE = median(200 \cdot \frac{\|Y_t - F_t\|}{Y_t + F_t})$

Measures based on Relative Errors

The approach followed by measures based on relative errors differs significantly from the first two approaches as they compare the forecasting error with the error of a standard forecasting method. Typically, naïve forecasting is chosen as standard of comparison. Thus, measures based on relative errors are easy to interpret. However, they also have deficits. Firstly, if e_t^*, i.e., the error of the standard forecasting method, is very small, the measure has a very high value. Furthermore, e_t^* has a positive probability density at zero and thus, the variance of the relative error is infinite. Table 2.4 shows a list of measures based on relative errors.

Table 2.4: Measures based on relative errors with formulas.

Error Measure	**Formula**
Mean Relative Absolute Error (MRAE)	$MRAE = mean(\|r_t\|)$
Median Relative Absolute Error (MdRAE)	$MdRAE = median(\|r_t\|)$
Geometric Mean Relative Absolute Error (GMRAE)	$GMRAE = gmean(\|r_t\|)$

Scaled Error Measures

The fourth category are scaled error measures that consider the developing of the original time series. That is, the forecasting error is normalized by the mean of the derivation of the original time series. Scaled error measures have several advantages. Firstly, they are independent of the input data scale. For this reason, scaled error measures can be used for comparisons across data sets with arbitrary scales. Furthermore, these error measures are suitable for almost all situations. There is only one exception, i.e., if and only if all observations in the horizon have the same value. In this case, the denominator of q_t is 0 and thus, the scaled error measures are infinite or undefined. Lastly, the error is scaled on the in-sample MAE from the naïve forecast. Thus, the examined forecasting method is better than the reference forecast if the scaled error is less than 1 and worse if the scaled error is greater than 1. The definitions of the Mean Absolute Scaled Error and the Median Absolute Scaled Error are shown in Table 2.5.

Table 2.5: Scaled error measures with formulas.

Error Measure	**Formula**
Mean Absolute Scaled Error (MASE)	$MASE = mean(\lvert q_t \rvert)$
Median Absolute Scaled Error (MdASE)	$MdASE = median(\lvert q_t \rvert)$

We prefer to use for forecast accuracy evaluations the Mean Absolute Scaled Error (MASE) because it is independent of the data scales, can deal with data values close to 0, and can handle negative values. However, a single error measure is often not significant enough and thus, the Mean Absolute Percentage Error (MAPE) is used in addition since it is one of the mostly used error measures in time series forecasting.

2.3.3 Time Series Decomposition

Decomposition is the act of splitting a time series in a trend, seasonal and remainder component. There are two types of decompositions: Additive and multiplicative that are represented as $Y_t = T_t + S_t + R_t$ and$Y_t = T_t \cdot S_t \cdot R_t$, respectively. In Table 2.6, we clarify the meaning of the variables used in the decompositions.

Table 2.6: Decomposition variables definition

Y_t	The t-th value of the time series
T_t	The t-th value of the trend component
S_t	The t-th value of the seasonal component
R_t	The t-th value of the remainder component

Next we will introduce two time series decomposition methods: The season and trend decomposition based on Loess (STL) method and the breaks for additive season and trend (BAST) method.

STL

This section is based on the paper *"STL: A Seasonal-Trend Decomposition Procedure based on Loess"* by Cleveland et al. [CCMT90].

STL provides a decomposition into season, trend and remainder component for a time series. It uses Loess smoothing. The decomposition of the time series Y is of the form:

$$Y_v = T_v + S_v + R_v$$

for $v \in \{1, ..., N\}$ with N as length of the trace.

In Figure 2.2, we see a STL decomposition consisting of four different plots. The first plot is the original time series, while the second, third and last plot are trend, seasonal and remainder component, respectively.

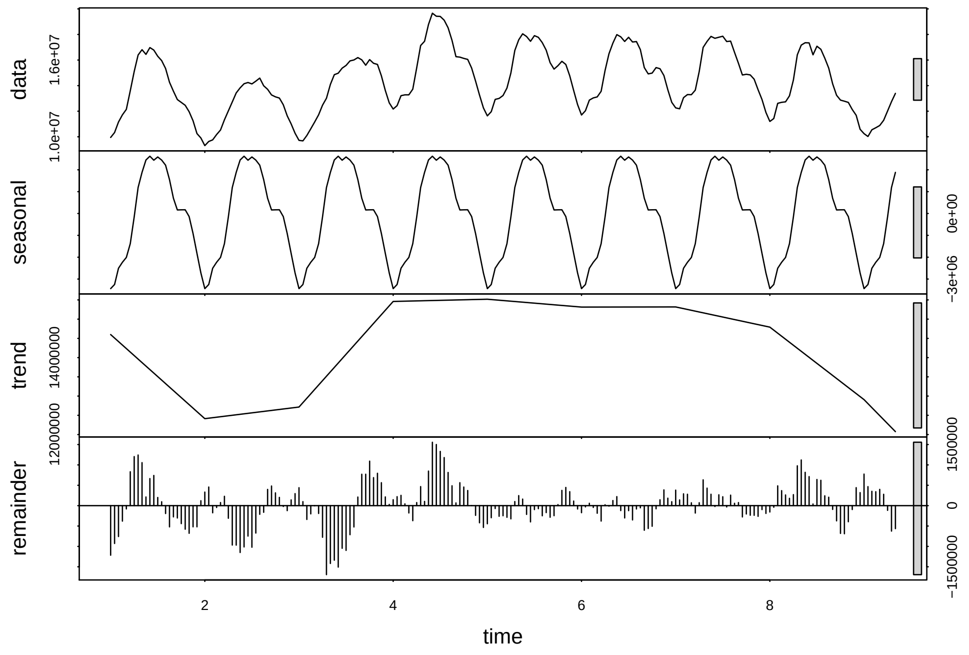


Figure 2.2: STL decomposition example: (From top to bottom) original time series, trend component, seasonal component and remainder component.

Loess: Let x_i and y_i be variables ($i = 1, ..., n$). $\hat{g}(x)$ is the smoothing Loess curve. We choose $q \in \mathbb{N}, 1 \leq q \leq n$. Those q nearest to x, that belong to x_i, are given a weight depending on the distance. We define $\lambda_1(x)$ as the distance from the closest x_i to x. Then, $\lambda_2(x)$ and $\lambda_3(x)$ are distances from the second and third closest x_i to x, respectively. $\lambda_q(x)$ is the distance of

q-th closest x_i to x. For $0 \le u < 1$

$$W(u) = (1-u^3)^3$$

If $u \ge 1$, then $W(u) = 0$. The weight function v is defined as

$$v_i(x) = W\left(\frac{|x_i - x|}{\lambda_q(x)}\right)$$

A polynom of degree d has to interpolate (x_i, y_i) with weight $v_i(x)$. For $q > n$

$$\lambda_q(x) = \lambda_n(x)\frac{q}{n}.$$

Before Loess is applied, d and q have to be set.

The overall structure of STL is given by two nested loops. Inside the inner loop trend and seasonal components are updated within $n_{(i)}$ repetitions. $n_{(i)}$ counts the number of data points within a period of the trace. Afterwards the weights are calculated within a run of the outer loop. These updated weights are used again in the next pass of the inner loop.

a) **Inner Loop**: The k-th time, we run through the inner loop, the trend component $T_v^{(k)}$ and the seasonal component $S_v^{(k)}$ are smoothed ($v = 1, ..., N$). Given $T_v^{(k)}$ and $S_v^{(k)}$, $S_v^{(k+1)}$ and $T_v^{(k+1)}$ ($T_v^{(0)} \equiv 0$) are calculated with the following steps:
Step 1: Calculate $Y_v - T_v^{(k)}$.
Step 2: Period smoothing with Loess for $q = n_{(s)}$ and $d = 1$. We get a temporary seasonal trace C_v^{k+1} with length $N + 2n_{(p)}$ ($v = -n_{(p)} + 1, ..., N + n_{(p)}$)
Step 3: Next, we use a low-pass filter on C_v^{k+1}. The filter uses a moving average with size $n_{(p)}$. Then, another moving average filter with length 3 is used. Finally, a Loess smoothing with $d = 1$ and $q = n_{(l)}$ is applied. We receive L_v^{k+1} ($v = 1, ..., N$). $n_{(p)}$ data points vanish this way.
Step 4: $S_v^{k+1} = C_v^{k+1} - L_v^{k+1}$ ($v = 1, ..., N$)
Step 5: $Y_v - S_v^{(k+1)}$ is calculated.
Step 6: Once again a Loess smoothing with $q = n_{(s)}$ and $d = 1$ is applied on $Y_v - S_v^{(k+1)}$.

b) **Outer Loop**: From the inner loop we get T_v and S_v. Hence, the remainder R_v can be written as $R_v = Y_v - T_v - S_v$. We define $h = 6 \cdot median(|R_v|)$ and $B_u = (1-u^2)^2$ for $0 \le u < 1$ and $B_u = 0$ for $u > 1$. Thus,

$$\rho_v = B\left(\frac{|R_v|}{h}\right).$$

The outer loop is executed $n_{(0)}$ times.

BFAST

Verbesselt, Hyndman, Newnham and Culvenor introduced BFAST (Breaks For Additive Seasonal and Trend) in their paper *"Detecting trend and seasonal changes in satellite image time series"* [VHNC10].

On the one hand, the *R* package `BFAST` can decompose any time series into a trend, seasonal and remainder part. On the other hand, it uses methods to detect significant change. For example it contains the `bfast()` function to estimate a decomposition. Figure 2.3 displays a BFAST plot. Similar to the STL plot in Figure 2.2, we have the original trace, seasonal, trend and remainder component in line one, two, three and four, respectively.

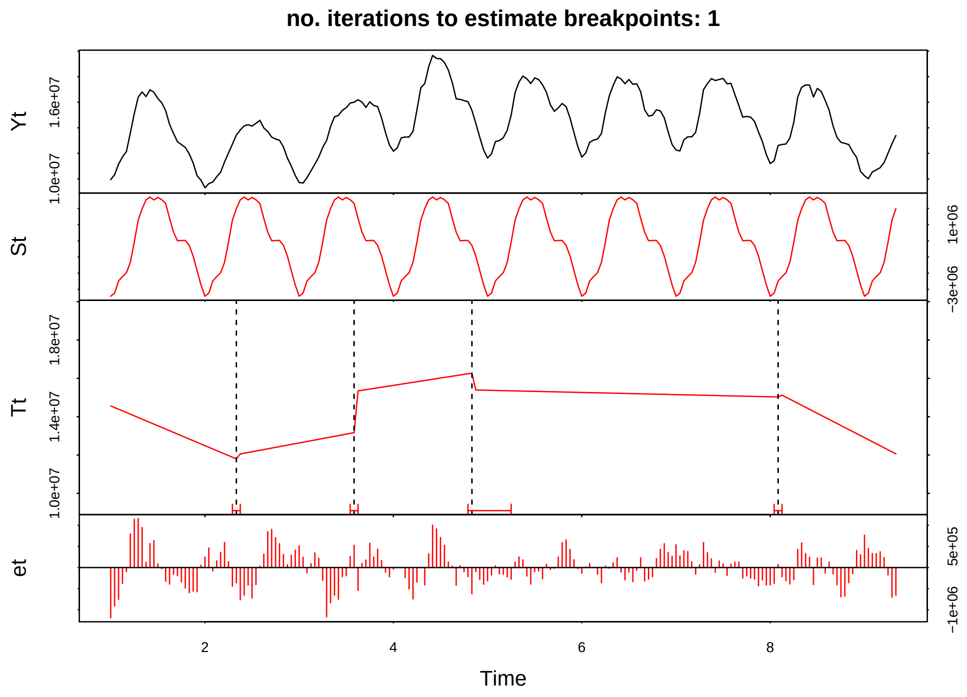


Figure 2.3: An example for a *BFAST* decomposition with detected trend breaks.

BFAST uses the ordinary least squares (OLS) residuals-based MOving SUM (MOSUM) tests to test for the existence of breakpoints. Furthermore, it uses the `breakpoints()` function of the `strucchange` package of R. The `breakpoints()` function uses the Bayes Information Criterion (BIC) to estimate a suitable number of breakpoints for the time series. First of all, an initial Season $\hat{S}_t$ is estimated with the `breakpoints()` function. After that BFAST iteratively loops through the following four steps until the number of breakpoints stays the same.

Step 1: Before the breakpoints $(t_1^*, ..., t_m^*)$ are estimated, OSL-MOSUM tests for the existence of any breakpoints on $Y_t - \hat{S}_t$.

Step 2: The coefficients α_j and β_j $(j = 1, ..., m)$ are estimated with regression of $T_t = \alpha_j + \beta_j t$ So, the temporary trend is set to $\hat{T}_t = \hat{\alpha}_j + \hat{\beta}_j t$.

Step 3: The OSL-MOSUM test also looks for breakpoints of the seasonal component. If there are seasonal breakpoints $(t_1^\#, ..., t_m^\#)$, these are estimated from $Y_t - \hat{T}_t$.

Step 4: Once again, for the seasonal components $\gamma_{i,j}$ $(j = 1, ..., m)$ $(i = 1, ..., s-1)$ we use regression of

$$S_t = \sum_{i=1}^{s-1} \gamma_{i,j}(d_{t,i} - d_{t,0})$$

for a temporal estimation

$$\hat{S}_t = \sum_{i=1}^{s-1} \hat{\gamma}_{i,j}(d_{t,i} - d_{t,0}).$$

2.3.4 Seasonal vs. Cyclic Time Series

The differentiation between seasonal time series and cyclic time series is a very important part of time series forecasting. However, these two terms are often getting mixed up [Hyn11]. Both types of time series are briefly explained and the differences are highlighted by examples. As seasonal time series show fluctuations with constant frequency, seasonal time series are also called periodic. This periodic pattern is often caused by seasonal influences. A typical influence factor is for instance the day-and-night behavior. Indeed, a seasonal time series can have multiple seasonal patterns of different frequencies, e.g., a weekdays-and-weekends behavior additionally to the day-and-night pattern. Figure 2.4 shows a seasonal time series.

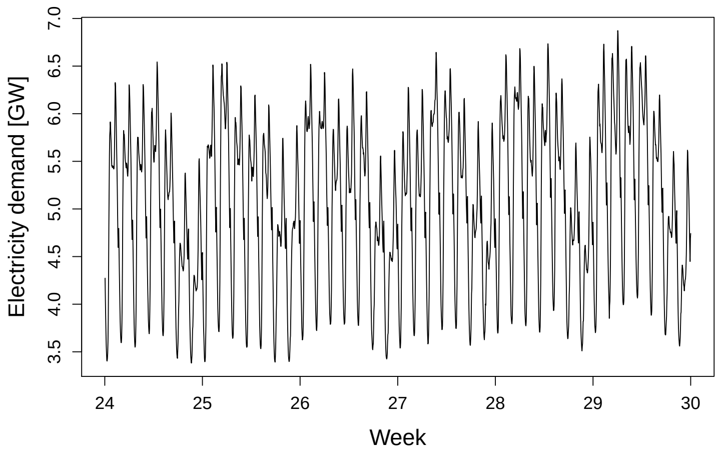


Figure 2.4: The half-hourly electricity demand of Victoria, Australia in GW in 2014.

This time series represents the electricity demand of Victoria, Australia in Gigawatt from the beginning of week 25 until the end of week 30 of the year 2014. Each value in the trace represents the electricity demand of the past 30 minutes. The electricity demand of Victoria is affected by two seasonal influence factors. At first, the time series contains a daily (day-and-night) period. The electricity demand rises until midday and afterwards, the electricity demand drops and reaches a low point in the middle of the night. This behavior is repeated with a fixed frequency. On top of that, a weekly (weekdays-and-weekends) period can be seen. Besides the weekends, the electricity demand process is similar for each day. On the weekends, the peeks are much lower compared to the weekdays. Also, this pattern repeats with a fixed frequency.

Cyclic time series also show recurring fluctuations with rises and falls, but these fluctuations do not occur with a fixed frequency. Thus, this non-periodic behavior is called cyclic. Another indication for cyclic time series is a highly varying amplitude of the cycles. In terms of seasonal time series, the amplitude of each period should be around the same except if there is another overlapping seasonal pattern.

Figure 2.5 shows an example for a cyclic time series that represents the amount of lynx trappings per year for Canada from 1821 to 1934. Although the trace seems to be seasonal because of its relatively repeating cycles, the time series is cyclic as the cycles do not occur with a fixed frequency. Some of the cycles last 8 or 9 years and others 10 years or even more. Additionally, the amplitudes of the cycles vary highly.

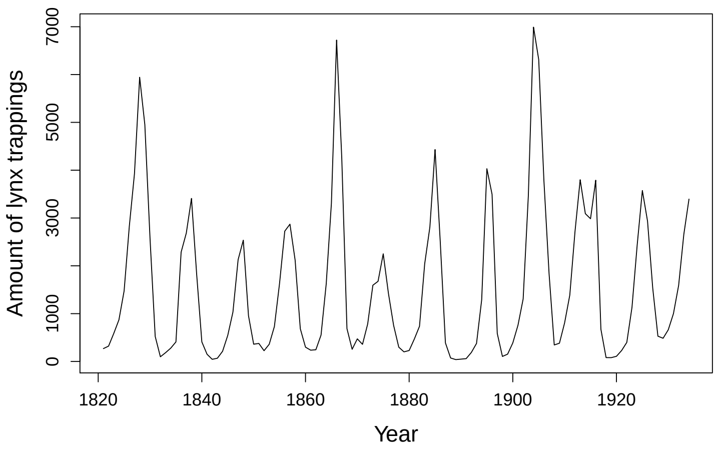


Figure 2.5: The annual amount of lynx trappings for Canada in the years 1821 to 1934.

As mentioned before, the differentiation between seasonal time series and cyclic time series is a very important task since some forecasting methods cannot handle seasonal time series and others exhibit significantly better performances for seasonal time series. Thus, this time series characteristics is also very important for automated forecast method recommendation. Moreover, the differentiation between seasonal time series and cyclic time series is also very important for frequency estimation. In terms of cyclic time series, typical frequency estimation tools will also find a dominant frequency. However, applying this frequency on cyclic time series will achieve poor results in the forecasting result since the time series does not have a constant periodic pattern as explained above.

2.3.5 Frequency Estimations using Periodogram

An important field of time series analysis is the spectral analysis. In terms of spectral analysis, a time series is seen as a sum of cosines and sines with different frequencies and amplitudes. The periodogram is a mathematical tool for estimation of spectral density that uses this composition assumption. Therefore, the periodogram determines the spectrum of several frequencies in a time series to find a dominant frequency and thus, a seasonal pattern [Pla11]. In order to determine the most dominant frequency, periodograms use a brute force algorithm in which the spectrum is calculated for many different possible frequencies by using Fourier transformation. More precisely, the aim is to split a time series consisting of the observations $x_1, x_2, \ldots, x_n$ with length n into multiple cosines and sines with periods of length $n/(n/2) = 2, \ldots, n/2, n$. The sum of the found cosines and sines together with the mean of the time series observations needs to result in the original time series. Now, discrete Fourier transformation is used for frequencies $v_k = \frac{k}{n}$ with $k = 1, \ldots, \frac{n}{2}$. For better readability and comprehension, the discrete Fourier transformation is represented by real and imaginary parts for this purpose [Bar10]:

$$\begin{aligned} X(v_j) &= \frac{1}{\sqrt{n}} \sum_{t=1}^{n} e^{-2\pi i t v_j} x_t \\ &= \frac{1}{\sqrt{n}} \sum_{t=1}^{n} \cos(2\pi t v_j) x_t - i \frac{1}{\sqrt{n}} \sum_{t=1}^{n} \sin(2\pi t v_j) x_t \\ &= X_c(v_j) - i X_s(v_j) \end{aligned} \tag{2.2}$$

Then, the periodogram calculates the squared modus of the discrete Fourier transformation as $I(v_j)$ [Bar10]:

$$\begin{aligned} I(v_j) &= |X(v_j)|^2 \\ &= \frac{1}{n} \left| \sum_{t=1}^{n} e^{-2\pi i t v_j} x_t \right|^2 \\ &= X_c^2(v_j) + X_s^2(v_j) \end{aligned} \tag{2.3}$$

$$X_c(v_j) = \frac{1}{\sqrt{n}} \sum_{t=1}^{n} \cos(2\pi t v_j) x_t \tag{2.4}$$

$$X_s(v_j) = \frac{1}{\sqrt{n}} \sum_{t=1}^{n} \sin(2\pi t v_j) x_t \tag{2.5}$$

However, the interpretation of such a periodogram is a crucial part on its own. In general, three types of time series can be identified by interpreting periodograms, i.e., time series with strong sinusoidal patterns, non-sinusoidal signals, and random time series. In case of random time series, all possible frequencies should have about the same spectrum since random signals do not have a seasonal pattern. Time series with strong sinusoidal patterns will show a peak in the periodogram at the dominant frequency. If the periodogram shows even more peaks at multiples of that frequency, this is a strong indication for time series with non-sinusoidal patterns [New99].

Nevertheless, the periodogram will find a most dominant frequency for seasonal time series as well as for cyclic time series. As illustrated in Figure 2.5 in Section 2.3.4, cyclic time series can also show sinusoidal patterns and thus, the automated differentiation between seasonal time series and cyclic time series is a very hard and inaccurate task. So, the differentiation between seasonal time series and cyclic time series has to be done before applying a periodogram method since a meaningful frequency should only be calculated for seasonal time series and not for cyclic time series. Otherwise, the forecasting accuracy can be highly diminished.

Frequency Estimation via Autocorrelation

The Autocorrelation compares a shifted version of a time series with its original version. If shifted and original version are very similar, the autocorrelation reaches values close to its maximum value 1. This is a signal that the length of the shift was a period of the time series. For example, if the sine function is shifted 2π to the right the resulting function is $\sin(t+2\pi), t \in \mathbb{R}$. But $\sin(t) = \sin(t+2\pi)$. Therefore 2π is a period of sine. Let $X \in \mathbb{R}^n$ be a time series of length n. Then we call

$$Y_t = X_{(t+k) \bmod n}$$

the k-shifted time series of X. For instance, if $X_t = (2,-3,1)^T$, then $X_{t+1} = (-3,1,2)^T$ and $X_{t+5} = (1,2,-3)^T$. The autocorrelation $r_{XY} \in [-1,1]$ measures the linear relationship between these two time series. If $r_{XY} \approx 1$, we assume that there is a seasonal component with the period k and validate this by checking if this shift results in a local maximum. Let $X, Y \in \mathbb{R}^n$ be two time series. Then

$$r_{XY} = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2 \cdot \sum_{i=1}^{n}(Y_i - \bar{Y})^2}} \in [-1,1]$$

is called correlation between X and Y.

2.3.6 Machine Learning Aproaches for Time Series Forecasting

eXtreme Gradient Boosting

XGBoost is an implementation of gradient boosted decision trees designed for execution speed and model performance [CG16]. The implementation allows multiple system features: 1) parallelization of tree construction by using all CPU cores in training 2) distributed computing: using a cluster of machines for training very large models 3) out-of-core computing: very large datasets that do not fit in memory 4) cache optimization: make the best use of the hardware. The process of boosting is an ensemble technique itself. Thus, new models are added in order to correct errors in existing models, e.g., models can be added sequentially until no further improvements can be made. Gradient boosting is a variation of the typical boosting task. The main steps of the gradient boosting decision tree algorithm are as follows. Firstly, new models are created that predict the residuals, or errors, of prior models. Afterwards, additional models are added to make the final prediction. When adding new models, a gradient descent algorithm is applied in order to minimize the loss. Contrary to other machine learning approaches like random forests or feed-forward neural networks, gradient boosted decision trees are not prone to overfitting, e.g., by leaving some portion of the available data for internal model cross-validation. Besides, XGBoost is said to be the go-to algorithm for Kaggle competition winners [Kag15a, Kag15b, Kag15c].

Feed-forward Neural Network

Feed-forward Neural Network is a machine learning technique trying to recognize regularities in the data, learn from experience, and provide generalized results based on current knowledge. Their main advantage is the capability of capturing non-linear patterns in time series, without any presumption about the statistical distribution of the observations. On top of that, neural networks can deal with fuzzy, incomplete, or erroneous input data. Neural networks are data-driven and self-adaptive. They use universal functional approximators, i.e., parallel processing of the data to approximate a large class of functions with a high accuracy. The Multi-Layer Perceptron (MLP) is characterized by a single hidden layer Feed-forward Neural Network. A simplified architecture of a Multi-Layer Perceptron with n input neurons, m hidden neurons, and l output neurons is shown in Figure 2.6.

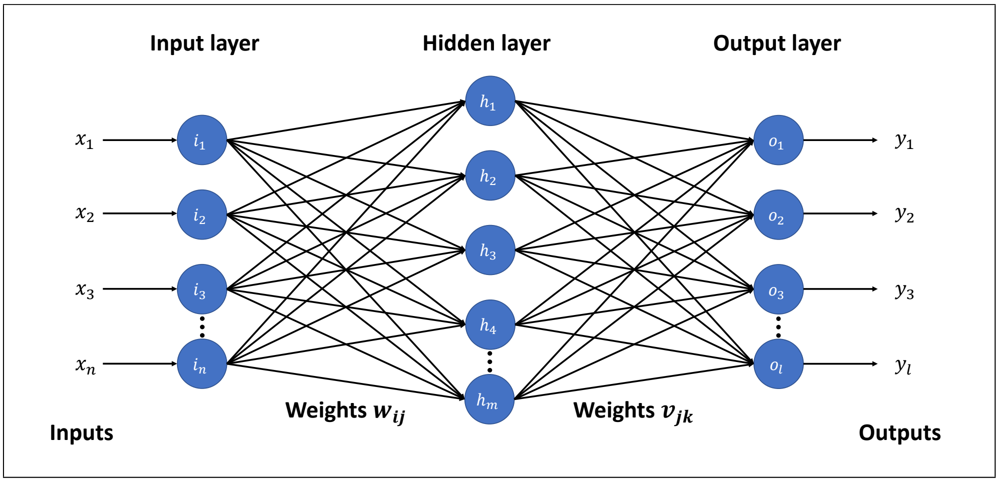


Figure 2.6: A simplified architecture of a Multi-Layer Perceptron.

A MLP gets as input the input values $x_1, x_2, \ldots, x_n$ that are passed to the input layer with input neurons $i_1, i_2, \ldots, i_n$. Then, the hidden layer neurons $h_1, h_2, \ldots, h_m$ are calculated by using the values of each input neuron and the weights w_{ij}. The value of the j-th hidden neuron is determined by applying the following formulas [WSMH09]:

$$net_j^h = \sum_{i=1}^{n} w_{ij} \times x_i \tag{2.6}$$

$$h_j = f(net_j^h) \tag{2.7}$$

Afterwards, these hidden neuron values are used to calculate the output neuron values as follows [WSMH09]:

$$net_k^o = \sum_{j=1}^{m} v_{jk} \times h_i \tag{2.8}$$

$$o_k = f(net_k^o) \tag{2.9}$$

The sigmoidal function is used in both formulas to calculate the values
of the neurons [WSMH09]:

$$f(net) = \frac{1}{1 + e^{-\lambda \times net}} \tag{2.10}$$

In this formula, e is the Euler's constant and $\lambda \in (0,1)$ is a parameter for controlling the gradient of the function [WSMH09]. However, since neural networks learn the weights w_{ij} and v_{jk} between the layers, a learning method is needed. Most neural networks apply the back-propagation learning rule because it allows to learn dependencies between any set of input combinations. Thus, the weights are updated in each iteration according to the average squared error until a certain predefined tolerance level is reached or a maximum amount of iterations is exceeded. There are two common variations of the Feed-forward Neural Network: 1) the Time Lagged Neural Network (TLNN) and 2) the Seasonal Artificial Neural Network (SANN) for seasonal data.

2.3.7 Statistical Modeling Frameworks for Time Series Forecasting

Autoregressive Integrated Moving-Average

ARIMA is a stochastic forecasting method. In fact, ARIMA is a generalization of an autoregressive moving-average (ARMA) model and based on the Box-Jenkins principle [BC64]. ARMA models are a combination of an autoregressive $AR(p)$ model and a moving-average $MA(q)$model. On the one hand, the autoregressive part predicts the future as linear combination of p past observations, also called order p, their weights φ_i, also called AR-coefficients, a random error ϵ_t, and a constant term c:

$$\text{AR(p)}: y_t = c + \sum_{i=1}^{p} \varphi_i \cdot y_{t-i} + \epsilon_t \tag{2.11}$$

On the other hand, the moving-average part of ARMA uses past errors ϵ_{t-j} and their weights Θ_j, also called MA-coefficients, for predicting the new value:

$$\text{MA(q)}: y_t = \mu + \sum_{j=1}^{q} \Theta_j \cdot \epsilon_{t-j} + \epsilon_t \tag{2.12}$$

The random error is assumed to be independent and identically distributed (i.i.d.) and following the standard normal distribution. The complete ARMA model can be formulated as:

$$\text{ARMA(p,q)}: \; y_t = c + \epsilon_t + \sum_{i=1}^{p} \varphi_i \cdot y_{t-i} + \sum_{j=1}^{q} \Theta_j \cdot \epsilon_{t-j} \tag{2.13}$$

As ARMA models require a stationary time series, the usage of ARMA model is restricted. To overcome this issue, ARIMA is developed as a generalization of ARMA. ARIMA makes a non-stationary time series stationary. The parameters of an ARIMA model are p, d, $q \in N_0$, where p is the order of the AR model and q the order of the MA model. The parameter d is for the integrated part of the model and thus, controls the degree of differencing. If $d = 0$, the ARIMA$(p, 0, q)$ model is equal to an ARMA(p, q) model. However, ARIMA makes two important assumptions. Firstly, the considered time series is assumed to be linear and secondly, it is assumed that the time series is following a particular known statistical distribution. The first assumption is very strong since real world data is rarely linear. Thus, the main disadvantage of ARIMA models is the assumption of the linear form of the model [Zha03]. Despite the disadvantage, there are several variations of the ARIMA model, e.g., the Seasonal ARIMA (SARIMA) for seasonal time series.

Extended Exponential Smoothing

Simple Exponential Smoothing (SES) was introduced by Brown in 1956 [Bro56]. SES is based on the moving-average method at which the past observations are weighted with an exponential function. Thus, the newest observations get the highest weight. The typical form for SES is:

$$y_t = \alpha \cdot x_t + (1 - \alpha) \cdot y_{t-1} \tag{2.14}$$

In this equation, $0 \leq \alpha \leq 1$ denotes the smoothing factor. The Exponential Smoothing State Space Model is based on the innovation state space approach. It distinguishes between the three components error, trend, and season. These components can be combined either additive or multiplicative, whereas their weightings are adjustable. Exponential Smoothing State Space Model exhibits good performance in detecting sinus-like seasonal patterns, but is rather bad in detecting complex seasonal patterns. According to Hyndman, ETS exhibits good performance on the M3-competition data set [HKSG02] that is dominated by short time series with below 100 observation and no significant seasonal patterns.

Trigonometric Box-Cox Transformation ARMA Errors Trend and Seasonality Model

TBATS was introduced by De Livera [DLHS11] in 2011 as an extension of the Exponential Smoothing State Space Model. It was developed to overcome the bad performance in detecting complex seasonal patterns that is for instance achieved by ETS. Therefore, TBATS uses the Fourier transformation, Box-Cox transformation and ARMA models. On top of that, it uses a method to reduce computational cost of maximum likelihood estimation.

Chapter 3

State-of-the-Art

We structure the discussion of the state-of-the-art and related research according to the contributions of this thesis into four parts:

- In Section 3.1, we approach the broad field of workload modeling from different angles to position and delineate our proposed approach for automated load profile modeling (Descartes Load Intensity Model (DLIM), c.f. Chapter 4).
- We split the Section 3.2 into two distinct parts to cover earlier proposed metrics for quantifying aspects of elastic platforms in the first place. Secondly, we describe existing measurement approaches to elastic properties in a way to highlight significant differences compared to our proposed measurement methodology Bungee (c.f. Chapter 5) for rating the observable elasticity achieved by an auto-scaler in a fair manner also across platforms of different performance characteristics.
- Section 3.3 aims at giving an overview of existing methods for automated scaling of systems by summarizing the findings from three surveys [GB12, JS15, LBMAL14]. The problem of auto-scaling is often approached leveraging techniques from different research domains in isolation with the resulting methods often tailored for certain specific application types. Our hybrid approach to auto-scaling Chameleon (c.f. Chapter 6) combines proactive decisions based on an integrated view of different models and reactive decisions based on direct feedback from monitoring in an intelligent way to improve scaling accuracy while lowering the risk of relying on a single method for making scaling decisions.
- Proactive auto-scaling mechanisms rely on methods for forecasting the arriving load efficiently in time and accurate. In Section 3.4, we approach the established domain of time series analysis and forecasting by statistical modeling frameworks and machine learning approaches as summarized in our book chapter [HAA$^+$17] here with a focus on the modern type of hybrid approaches. As a side-contribution of this thesis, the hybrid forecast prototype approach Telescope (c.f. Chapter 7) builds upon the findings we made analyzing related work.

3.1 Approaches to Workload Modeling

Describing real-world workload characteristics using various abstractions and modeling approaches has been a prominent activity in performance engineering for decades. The well-known study on workload characterization of web servers published by Arlitt et al. [AW96] in 1996 can be seen as an initiator for modern server workload models. Based on our analysis of related research on workload characterization, we propose a grouping into the following four categories:

3.1.1 User Behavior Models

User behavior models usually traverse graphs, where nodes represent actions that users may perform on a system. Workload traces are analyzed to identify similarities between users. The identified clusters with high similarity in between system users are then used to form at groups of users that can be mapped to request classes or probabilities of certain user actions. Zakay and Feitelson [ZF13] propose to partition workloads according to the user types, and then sample workload traces for each user type to capture the user behavior.

The Descartes Modeling Language (DML) [KHBZ16, HBS$^+$17] and the Palladio Component Model (PCM) [BKR09] with their simulation frameworks both offer a meta model to capture the probabilistic interactions between groups of users and the modeled system. An automated extraction of such user behavior models is presented as the WESSBAS approach in the work of van Hoorn et al. [vHVS$^+$14]. The Markov4JMeter approach [vHRH08] and the RAIN workload generator [BLY$^+$10] propose workload models for generating the behavior and tasks triggered by different types of users.

Approaches in this group have in common that they lay their focus on the behavior of users instead of how the users' load arrives at the system. We define the arriving load intensity at the system boundary. Models like the above can in principle be combined with our load profile modeling approach DLIM to further characterize the user behavior after a request has arrived at the system and a client session is started.

3.1.2 Service Demand Focused Workload Modeling

Approaches in this category focus on characterizing the units of work processed estimating their demand for resources (a.k.a. service demand). A service demands are the average time durations a unit of work (e.g., request or transaction) spends obtaining service from a resource (e.g., CPU or hard disk) in a system over all visits excluding any waiting times [MDA04] and are a key parameter of performance models. Requests of the same workload class (or submitted by the same group of users) usually exhibit similar resource demands and thus can be seen as characterizing parameters of the effects a user interaction has on the system. Over the years, a number of approaches to service demand estimation have been proposed using different statistical estimation techniques (e.g., linear regression [RV95, BKK09] or Kalman filters [Wm12, KTZ09]) and based on different laws from queueing theory. The state-of-the in service demand estimation is surveyed and analyzed in the work of Spinner et. al [SCBK15]. The Library for Resource Demand Estimation (LibReDE)[1] is, to the best of our

[1]LibReDE: Available at `https://descartes.tools/librede`.

knowledge, the only publicly available tool providing different ready-to-use approaches to service demand estimation. The latter currently supports implementations of the following estimation approaches:

- Service Demand Law [BKK09]
- Least squares (LSQ) regression using queue-lengths and response times (Response Time Regression) [KPSCD09]
- Least squares (LSQ) regression using utilization law (Utilization Regression) [RV95]
- Approximation with response times (Response Time Approximation) [BKK09]
- Recursive optimization using response times (Menascé Optimization) [Men08]
- Recursive optimization using response times and utilization (Liu Optimization) [LWXZ06]
- Kalman filters using utilization law (Wang Kalman Filter) [Wm12]
- Kalman filters using response times and utilization (Kumar Kalman Filter) [KTZ09]

Service demand estimation represents an important portion of research in workload modeling, We argue that this is complementary to modeling the arriving load profiles, as we do in Chapter 4. We rely on the service demand estimation approaches when combining them with predicted load profiles within the hybrid proactive auto-scaling mechanism Chameleon (c.f. Chapter 6).

3.1.3 Regression Techniques

M5 trees [$Q^{+}92$], multivariate adaptive regression splines (MARS) [Fri91], or cubic forests [KWKC12] are advanced regression techniques capable of calibrating mathematical functions to fit a measured load intensity trace. Similarly to our proposed load profile modeling approach DLIM, these functions describe the load intensity over time with functions. Yet, in contrast to the descriptive DLIM models, they do not convey the additional information of the types and composition of load intensity components.

Both, the regression techniques and service demand estimation approaches can be configured with various parameters. The Stepwise Sampling Search (S3) or also called Iterative Parameter Optimizer (IPO) [NBKR13] has been developed in the context of regression model optimization [Noo15]. We adapted this algorithm for self-tuning the parameters of resource demand estimation techniques [GHSK17] Please note, that details on this are not included as contribution in this thesis.

3.1.4 Statistical Inter-Arrival Models

Statistical inter-arrival models capture the workload intensity by fitting of statistical distributions. Feitelson et al. [Fei02] create a statistical model for parallel job schedulers. Li et al. [Li10] model batch workloads for eScience grids, Casale et al. [CKKR12] focus on modeling bursty workloads, whereas Barford and Corvella [BC98] focus on file distribution and popularity. Menasce et al. [MAR+03] as well as Reyes et al. [RLGPC+99] analyze workloads at multiple levels, such as request and session level. These approaches differ from our load modeling approach as they use independent random variables to capture inter-arrival rate distributions. Model input is usually the type of workload or certain workload characteristics, which may include arrival rate, but also other characteristics, such as burst-size, whereas the output is a statistical distribution. In our case, we use the current time as the primary input (usually seconds since a pre-defined point in time t_0, e.g. measurement start), with the model returning the concrete load intensity at that point in time.

Our approach for automated, descriptive modeling of load profiles DLIM is either complementary to the two mentioned groups mentioned first, and different to the third and fourth group of workload modeling techniques as our approach does not model the arriving load via distribution fitting or with the help of one regression function. We explicitly decompose an given trace into its deterministic parts as trends and seasonal patterns, and estimate only optionally the remainder with a Gaussian distribution.

3.2 Related Elasticity Metrics and Measurement Approaches

In this section, we group existing elasticity metrics and benchmarking approaches according to their perspective and discuss their shortcomings.

3.2.1 Related Elasticity Metrics

Several metrics for elasticity have been proposed so in literature:

The “scaling latency” metrics [LYKZ10, LOZC12] or the “provisioning interval” [CCB+12] capture the pure time to bring up or drop a resource. This duration is a technical property of an elastic environment independent of the timeliness and accuracy of demand changes, thus independent the elasticity mechanism itself that decides when to trigger a reconfiguration. We consider these metrics as insufficient to fully characterize the elasticity of a platform.

The “reaction time” metric we proposed at an early stage of our research already in 2011 in the report [KHvKR11] can only be computed if a unique mapping between resource demand changes and supply changes exists. This assumption does not hold especially for proactive elasticity mechanisms or for mechanisms that have unstable (alternating) states.

The “elastic speedup” metric proposed by SPEC OSG in their report [CCB+12] relates the processing capability of a system at different scaling levels. This metric - contrary to intuition - does not capture the dynamic aspects of elasticity and is regarded as scalability metric.

The integral-based “agility” metric also proposed by SPEC OSG in their report [CCB+12] compares the demand and supply over time normalized by the average demand. They state that the metric becomes invalid in cases where service level objectives (SLOs) are not met. This “agility” metric has not been included as part of the SPEC Cloud IaaS 2016 benchmark[2]. This metric resembles an early version of our proposed “precision” metric [HKR13]. In this thesis, we propose a refined version normalized by time (see Section 5.2) to capture the accuracy aspect of elastic adaptations, considering also situations when SLOs are not met.

[2]SPEC Cloud IaaS 2016: `https://www.spec.org/cloud_iaas2016/`

The approaches in the work of Binning et al., Cooper et al., Almeida et al. and Dory et al. [BKKL09, CST$^+$10, ASLM13, DMRT11] characterize elasticity indirectly by analyzing response times for significant changes or for SLO compliance. In theory, perfect elasticity would result in constant response times for varying arrival rates. In practice, detailed reasoning about the quality of platform adaptations based on response times alone is hampered due to the lack of relevant information about the platform behavior, e.g., the information about the amount of provisioned surplus resources.

Becker et al. [BLB15] introduced in 2015 the "mean-time-to-repair" in the context of elasticity as the time the systems needs on average to step out of an imperfectly provisioned state. This "mean-time-to-repair" is estimated indirectly based on analyzing for how long the request response times violate a given SLO or a systems runs below a target utilization - in other words, without knowing the exact demand for resources. The notion of "mean-time-to-repair" from 2015 is basically equivalent to our previously proposed wrong-provisioning timeshare metric (c.f. Section 5.2.2) as published in 2013 [HKR13].

Numerous cost-based elasticity metrics have been proposed so far [ILFL12, FAS$^+$12, Sul12, Wei11, TTP14]. They quantify the impact of elasticity either by comparing the resulting provisioning costs to the costs for a peak-load static resource assignment or the costs of a hypothetical perfect elastic platform. In both cases, the resulting metrics strongly depend on the underlying cost model, as well as on the assumed penalty for under-provisioning, and thus they do not support fair cross-platform comparisons.

We propose a set of elasticity metrics explicitly covering the timeliness, accuracy and stability aspects of an deployed elasticity mechanism (a.k.a auto-scaler) together with three different way to aggregate in a possibly weighted manner. This set of core elasticity metrics can of course be complemented by user-oriented measures like response-time distributions, percentage of SLO violations, a cost measure by applying a cost model, or accounted and charged resource instance times.

3.2.2 Elasticity Measurement Approaches

As a high number of auto-scaling mechanisms has been proposed in literature over the course of the last decades, many of them come with individual and tailored evaluation approaches. We observe that in most cases an experimentally comparison against other state-of-the-art mechanisms is omitted and the mechanisms effectiveness is justified by show-cases of improvements in service level compliance. The field of resource management algorithms has also been flooded by approaches only evaluated in simulation frameworks. A broader simulative analysis of auto-scalers can be found in the work of Papadopoulos et al. [PAEÅ$^+$16].

A number of approaches for elasticity evaluation - in some cases only on an abstract level - can be found in literature [FAS$^+$12, Sul12, Wei11, SA12, ILFL12, DMRT11, ASLM13, TTP14, CST$^+$10, Bel16]:

In the work of Folkerts et al. [FAS$^+$12], the requirements for a cloud benchmark are described on an abstract level. The work of Suleiman [Sul12], Weinman [Wei11], Shawky and Ali [SA12] as well as the work of Islam et al. [ILFL12] have in common their end-user perspective on elasticity and quantify costs or service level compliance. These are important end-user metrics, but we argue that good results in these metrics are indirectly affected by elasticity mechanisms. Timeliness, accuracy and stability of resource adaptations are not considered although those are the core aspect of an elastic behavior. Furthermore, these approaches account neither for differences in the performance of the underlying physical resources, nor for the possibly non-linear scalability of the evaluated platform. As a consequence, elasticity evaluation is not performed in isolation from these related platform attributes, as we highlight in Section 2.1. In contrast, the approach proposed in this thesis uses the results from an initial systematic scalability and performance analysis to adapt the generated load profile for evaluating elasticity, in such a way that differences in the platform performance and scalability are factored out (c.f. Section 5.5.2).

Another major limitation of existing elasticity evaluation approaches is that systems are subjected to load intensity variations that are not representative for real-life workload scenarios. For example, in the work of Dory et al. [DMRT11] and of Shawky and Ali [SA12], the direction of scaling the workload downwards is completely omitted. In the work of Islam et al. [ILFL12], sinus like load profiles with plateaus are employed. Real-world load profiles exhibit a mixture of seasonal patterns, trends, bursts and noise. We account for the generic benchmark requirement “representativeness" [Hup09] by employing the load profile modeling formalism DLIM as presented in Chapter 4 and the corresponding article [vKHK$^+$17].

One recent example of a paper on cloud elasticity, elasticity metrics and an elasticity benchmark called BeCloud is the work of Beltran [Bel16]. Here a new elasticity metric is proposed, based on the finding that current metrics do not reflect scaling accuracy well enough. This does obviously not apply as we have proposed over-/under-provisioning accuracy already in 2013 [HKR13] and experimentally applied as published in 2015 [HKWG15b]. The proposed elasticity metric remains fuzzy and without an intuitive interpretation: It is derived as the product of ‘scalability’ (Ψ), an indicator for the accuracy of the scaling (Δ), and an indicator for the delay of the scaling (R):

$$E = \Psi \cdot g(\Delta) \cdot (R).$$

The indicators for accuracy and delay (reconfiguration time) are calculated as the fraction of how good the system could possibly be and how good it actually is, e.g.

$$g(\text{actual scaling inaccuracy}) = \frac{\text{best possible scaling inaccuracy}}{\text{actual scaling inaccuracy}}$$

or 1 if the actual scaling inaccuracy is 0. The reason for that is that in contrast to our measurement approach BUNGEE (c.f. Section 5.5), where a perfect scaling can happen and the demand for resources of time is defined and know ahead of an experiment, Beltran [Bel16] also considers systems that offer software as a service (SaaS) where the best possible supply may not match the demand. As it is also noted in the paper this has the consequence that a system a with worse accuracy can have a better result for the elasticity metric than a system b with better accuracy, if a is closer to its best possible accuracy than b. For the calculation of the scalability Ψ, the ratio of performance and cost between configurations is compared. This is very different to our proposed approach BUNGEE measurement approach, where scalability only indirectly affects elasticity. (Systems that can not be scaled well can not possibly exhibit good elastic behavior.) Also as cost is a factor that influences the elasticity metric, the problems discussed above related to the dependence on a cost model apply.

Another important difference to our measurement approach BUNGEE is the way in which accuracy/inaccuracy of adaptations is evaluated. Beltran [Bel16] measures elasticity is not with an elaborate measurement techniques, but estimates as the amount of VM minutes during which the system is not running the user's application. We suppose that this can not be as accurate as measuring the accuracy directly. In summary, the paper proposes a way of measuring elasticity that is on the one hand similar to the one in this thesis in the way that it also looks at the accuracy and timing of the scaling. On the other hand, the proposed metric also considers cost and elasticity as important factors and in summary offers a method that favors less measurement overhead instead of accurate and reproducible measurements.

Towards the end of this discussion on related elasticity measurement approaches, it is worth mentioning the SPEC Cloud IaaS 2016 Benchmark[3] that can be seen as the flagship in industry-standard performance of infrastructure cloud measurement frameworks in terms of representativity and repeatability of results. The elastic behavior of clouds with an active auto-scaling mechanism is in the current release 1.1 not covered. The workload is scaled out controlled by the benchmark harness. As mentioned above the "elastic speedup" quantifies how good the cloud system actually scales.

3.3 On the State of the Art in Auto-Scaling

The topic of auto-scaling has been a popular research topic in the resource management community over the past decade. There have been multiple recent efforts to survey the state-of-the-art in auto-scaling, for example: (i) Galante and de Bona [GdB12], (ii) Jennings and Stadler [JS15], and (iii) Lorido-Botran et al. [LBMAL14]. These three surveys provide an overview on the current state of research on auto-scaling. The survey by Lorido-Botran et. al [LBMAL14] proposes a classification of auto-scalers into five groups:

3.3.1 Threshold-based Rule Auto-Scalers

These are auto-scalers that react to load changes based on pre-defined threshold rules for scaling a system. As the scaling decisions are triggered by performance metrics and predefined thresholds, they are easy to deploy. Thus, they are popular with commercial cloud providers and clients. Common threshold-based auto-scalers are designed for instance by Han et al. [HGGG12] or Maurer et al. [MBS11]. We use an standard reactive auto-scaler as competitor in our broad auto-scaler evaluation in Section 10.2.1.

3.3.2 Auto-Scalers based on Queueing Theory

Queueing theory is usually used to determine the relationship between incoming and outgoing jobs in a system. Proactive auto-scalers in this category rely on a model of the system for predicting the future resource demand. A simple approach is to model each VM as a queue of requests. However, these auto-scalers have a major drawback as they are inflexible. That is, the models have to be updated and re-calibrated after each change in the workload or in the application. State-of-the-art auto-scalers are proposed, e.g., by

[3]SPEC Cloud IaaS 2016: `https://www.spec.org/cloud_iaas2016/`

Urgaonkar et al. [USC$^+$08] (Hist) or Zhang et al. [ZCS07]. We select Hist as competitor in our auto-scaler evaluation representing a queueing theory based auto-scalers. More details on Hist can be found in Section 10.2.3.

In addition, we consider as part of this group, resource management approaches that leverage descriptive performance modeling formalisms and simulate the effects of deployment changes via transformations to queueing networks and discrete event simulation. We take ideas from the SLAstic.SIM [vMvHH11] approach that builds upon on the Palladio Component Model and the S/T/A adaptation framework [HHK$^+$14]. Both similar approaches implement abstract strategies, tactics and actions to resolve resource shortages and optimize deployments of simulated descriptive performance model instances. Due to their overheads and complexity, these approaches have not yet been applied to control real infrastructure cloud environments.

3.3.3 Auto-Scalers based on Control Theory

Similarly to queueing theory, auto-scalers that use control theory also employ a model of the application. Hence, the performance of such controllers depends on the application model and the controller itself. A new approach in recent research is to combine this type auto-scalers with queueing theory. For example, popular hybrid auto-scalers of this group are from Padala et al. [P$^+$09] or Ali-Eldin et al. [AETE12] (Adapt). As we were able to establish a collaboration and exchange with Ali-Eldin, we obtained the source for Adapt to participate in our auto-scaler evaluation (c.f. Section10.2.2).

3.3.4 Auto-Scalers based on Reinforcement Learning

Instead of having explicit knowledge or a model of the application, approaches in this category aim to learn the best action for a specific state. The learning is based on a trial-and-error approach converging towards an optimal policy. This allows the controller to be proactive. However, finding the best option for a particular state can take a long time. Tesauro et al. [TJDB06] or Rao et al. [RBX$^+$09], for example, propose such auto-scalers. Auto-scalers based on reinforcement learning are highly dependent on the initial extensive training phase and therefore we do not consider them in our evaluation. In addition, our efforts to obtain reinforcement learning auto-scaler code was not successful.

3.3.5 Auto-Scalers leveraging Time Series Analysis

Auto-scalers that use time series analysis are the most common representatives of proactive scaling. They allow to predict the near future behavior of sequences of job arrivals. A variety of forecasting methods based on time series analysis exist in the literature. The choice of forecasting technique and its associated parameters influences the forecasting accuracy. This type of auto-scalers may be highly optimized for a specific scenario like Netflix Scryer[4] that bases on Fourier frequency analysis to predict alternations in load. Common examples of this type of auto-scalers are proposed by Pierre and Stratan [PS12] (ConPaaS) or Iqbal

[4]Scryer: Netflix's Predictive Auto Scaling Engine at `https://medium.com/netflix-techblog/scryer-netflixs-predictive-auto-scaling-engine-a3f8fc922270`

et al. [IDCJ11] (Reg). We include these to well-cited examples as representatives for this group in our evaluation with more details to be found in Section10.2.4 and Section 10.2.5.

Existing auto-scalers try to leverage both reactive and proactive methods [USC+08,AETE12, IDCJ11]. For example, Reg [IDCJ11] limits up-scaling to reactive and down-scaling to proactive decisions. Adapt [AETE12] detects the envelope of the workload and is thus limited to short-term predictions. In contrast to the state-of-the-art, our proposed Chameleon approach is a hybrid, proactive auto-scaler that combines long-term predictions from cutting-edge time series analysis methods that are only triggered on demand with online solved queueing net models and an integrated reactive fall-back mechanism. Chameleon consists of two explicit reactive and proactive cycles for generating combined and optimized scaling decisions covering current and future auto-scaling intervals. To the best of our knowledge, we have not found this unique combination of methods smartly integrated as in our proposed auto-scaling mechanism. We systematically show in our evaluation Chapter 10 that this yields in an superior auto-scaling performance with a significantly lower risk of wrong and costly resource adaptations.

While not common, we identify two cases of the previous work comparing multiple and different, proactive auto-scaling policies. The work of Padadopoulos et al. [PAEÅ+16] establishes theoretical bounds on the worst case performance in simulation. A related experimental evaluation to which we contributed, in the work of Ilyushkin et al. [IAEH+17], compares auto-scaler performance for the different type of workflow applications in one deployment. To the best of our knowledge, currently only the above two works contain a comparably broad set of auto-scalers in their evaluations. Thus, this thesis contains the first broad experimental evaluation covering different deployments and traces with a representative set of auto-scaling algorithms applied to a generic web application worklet.

3.4 Related Work on Hybrid Forecasting

In this thesis, we identify the potential of improving the accuracy, reliability and the time-to-result of automated forecasting mechanisms to further improve auto-scaling mechanisms and include an prototypical hybrid forecasting approach Telescope as a side-contribution of this thesis. This section reviews three different ways for hybridizing forecast approaches.

In 1997, Wolpert and Macready presented the "No Free Lunch Theorem" for optimization algorithms [WM97]. It claims, that there is not a single algorithm that performs best for all scenarios since improving the performance of one aspect normally leads to a degradation in performance for some other aspect. This theorem can also be applied to forecasting methods as there is no single method that outperforms the others for all types of data. To address this issue, many hybrid forecasting approaches have been developed. A hybrid forecasting method makes use of at least two forecasting methods to compensate the limitations of individual forecasting methods. We categorize hybrid forecast methods into three groups of approaches each sharing the same basic concept:

3.4.1 Ensemble Forecasting

The first and historically oldest group is the concept of ensemble forecasting and is the technique of using at least two forecasting methods. While assigning a weight to each method, the forecast result is the weighted sum of each forecast method. This approach was introduced by Bates and Granger in 1969 [BG69]. The concept of this approach is rather simple, however, the assignment of weights is a crucial part. Thus, many methods for weight estimation have been investigated [Cle89, DMBT00, KKZ+00].

3.4.2 Forecaster Recommendation

The second group of forecasting methods is based on the concept of forecast recommendation, where the goal is to build a rule set to guess the assumed best forecasting method based on certain time series features. There are two common ways to generate the rule set. One method is using an expert system. Collopy and Armstrong used this approach to create a rule set by hand in 1992 [CA92]. The other method is using machine learning techniques to automatically generate a rule set. In 2009, Wang et al. proposed clustering and algorithms for rule set learning based on a large variety of time series features [WSMH09]. As part of our exploitative research in time series characteristics and forecast method selection based on a heuristic for forecast method selection [HHKA14], we also reimplemented and systematically tuned the rule learning approach of Wang (not included as contribution in this thesis). Wang postulates that the found rules for method selection are generally applicable. This contradicts with our findings where the rules are strongly tuned (over-fitted) for the respective data set.

3.4.3 Feature Engineering based on Decomposition

The third group of forecasting methods is based on decomposition of the time series with the goal to leverage the advantages of each method to compensate for the drawbacks of the others. In literature, there are common approaches. The first approach is to apply a single forecasting method on the time series and then apply another method on the residuals of the first one [Zha03, PL05]. The second forecasting method is intentionally chosen to have different characteristics than the first one, that is, it should have antithetic advantages and drawbacks compared to the first one. An alternative approach is to split the time series into its components trend, seasonality, and noise, applying a different forecasting method on each of them. Liu et al. introduced an approach like this targetted for short-term load forecasting of micro-grids [LTZ+14]. They used empirical mode decomposition to split the time series and extended Kalman filter, extreme learning machine with kernel, and particle swarm optimization for the forecast.

Our hybrid forecasting approach Telescope can be clearly classified into the third group of forecasts using decomposition in a automated way for feature engineering. In contrast to Zhang or Pain and Lin [Zha03, PL05], we use explicit time series decomposition, that is, forecast methods are applied to the individual components of the time series as opposed to the residuals of previous forecasts. Liu et al. introduced a short-term hybrid forecasting method based on an intrinsic mode function. In our approach, the time series is split into trend, seasonality, and noise components. Additionally, completely different forecasting methods are used as a basis. Furthermore, our newly proposed hybrid approach is designed to perform multi-step-ahead forecasting with low overhead and short runtime as this are key requirements for an online usage in the context of auto-scaling.

Part II

Benchmarking Elasticity of Auto-Scaling Mechanisms

Chapter 4

Descriptive Load Intensity Profiles: Modeling Language and Automatic Extraction

Today's system developers and operators face the challenge of creating software systems that make efficient use of dynamically allocated resources under highly variable and dynamic load profiles, while at the same time delivering reliable performance. Autonomic controllers, e.g., an advanced auto-scaling mechanism in a cloud computing context, can benefit from an abstracted load model as knowledge to reconfigure on time and precisely.

Existing workload characterization approaches have limited support to capture variations in the inter-arrival times of incoming work units over time (i.e., a variable load profile). For example, industrial and scientific benchmarks support constant or stepwise increasing load, or inter-arrival times defined by statistical distributions or recorded traces. These options show shortcomings either in representative character of load variation patterns or in abstraction and flexibility of their format.

In this chapter, we answer the first two RQs A.1 and A.2 of Goal A: To address RQ A.1: „How to model load intensity profiles from real-world traces in a descriptive, compact, flexible and intuitive manner?“, we present the Descartes Load Intensity Model (DLIM) approach. DLIM provides a modeling formalism for describing load intensity variations over time. A DLIM instance is a compact formal description of a load intensity trace. DLIM-based tools provide features for benchmarking, performance and recorded load intensity trace analysis.

As manually obtaining and maintaining DLIM instances becomes time consuming, and to address RQ A.2: „How to automatically extract instances of load intensity profiles from existing traces with a reasonable accuracy and computation time?“, we contribute three automated extraction methods and devised metrics for comparison and method selection. We discuss how these features are used to enhance system management approaches for adaptations during run-time, and how they are integrated into simulation contexts and enable benchmarking of elastic or adaptive behavior.

In the evaluation in Chapter 8, we show that automatically extracted DLIM instances exhibit an average modeling error of 15.2% over ten different real-world traces that cover between two weeks and seven months. These results underline DLIM model expressiveness. In terms of accuracy and processing speed, our proposed extraction methods for the descriptive models deliver better or comparable results compared to existing non-descriptive time series decomposition methods.

4.1 Introduction

Today's cloud and web-based IT services need to handle large numbers of concurrent users under highly variable and dynamic load intensities. Customers access services independently of each other and expect a stable Quality-of-Service (QoS). In this context, any knowledge about a service's load intensity profile and their variations becomes a crucial information for managing the underlying IT resource landscape. Human behavior patterns due to human habits, trends, calendar effects, and events heavily influence load profiles. An autonomic controller, e.g., an advanced auto-scaling mechanism deployed in a cloud computing context, may implement the MAPE-K control loop [KC03]. When such a controller is constantly provided with abstracted knowledge about the observed and expected load profile, it could trigger the majority of adaptations more precisely and on time, as envisioned by the Models@Run-Time community [BFB09]

Also, performance evaluation of systems under dynamic load conditions poses new challenges. Benchmarking frameworks such as Faban [Fab06], Rain [BLY$^+$10], JMeter [Hal08] allow request injection rates to be configured either to constant values, or to stepwise increasing rates (e.g., for stress tests). The feature of generating variable rates based on a recorded or synthetic load trace is not fully supported by the mentioned frameworks. They usually work with a fixed number of load generating threads, whereas one of those corresponds to one virtual user with its routine and think time, as in a closed workload. From this, we see that open workloads with a possibly unlimited number of users and a representative load intensity variation independent of the actual system's performance are supported only to a limited extent by existing benchmarking frameworks.

In this chapter, we introduce the Descartes Load Intensity Model (DLIM) and the corresponding tools. DLIM describes load profiles by combining piece-wise mathematical functions in a tree-like manner. Manual construction and maintenance of DLIM model instances becomes infeasible in complex scenarios or at run-time usage. We address this by proposing the high-level Descartes Load Intensity Model (hl-DLIM) to support the description of load variations using a small set of parameters to characterize seasonal patterns, trends, as well as bursts and noise.

The DLIM modeling language can be used to define an arbitrary load intensity profile which can than be leveraged for benchmarking purposes to evaluate the behavior of a system under different dynamic workload scenarios (e.g., bursty workloads with seasonal patterns). This is useful in several use-cases, e.g., for both on-line and off-line evaluation of the quality of system adaptation mechanisms such as elastic resource provisioning techniques in modern cloud environments. In contrast to pure regression approaches, DLIM offers the advantage of classifying load intensity variations by type, as they are fitted to certain model elements. As a result, models include additional information on types, which is useful when analyzing or modifying load intensity variations.

A load intensity profile, represented as a DLIM model instance, can be created either manually by the user or it can be extracted from a request arrival trace obtained by monitoring a real-life production system. To support this, we provide generic trace analysis algorithms, define quality metrics for these algorithms, and integrate them into the DLIM tools called

LIMBO[1]. The tools allow users to use our approach in a variety of use-cases. As a result, the trace is represented as a compact DLIM model instance carrying a tree of mathematical functions that are combined over the modeled time. The model instance captures the major properties of the trace (e.g., burstiness, seasonality, patterns and trends) and can be used at any time to automatically generate comparable traces, i.e., the trace exhibits the same properties. Furthermore, the extracted model instance can be easily modified to reflect a target dynamic load scenario, e.g., by changing the frequency of bursts or adding a given trend behavior.

Furthermore, we introduce three automated DLIM model extraction methods: s-DLIM, p-DLIM, and hl-DLIM. s-DLIM's workflow is inspired by the time series decomposition approach STL [CCMT90]. p-DLIM focuses more on periodic patterns strengthening extrapolation capabilities, and the hl-DLIM extraction method works on a higher abstraction level for more compact model instances.

In the DLIM evaluation in Chapter 8, we compare the DLIM model accuracy achieved by automatically extracting model instances from real-world arrival rate traces against non-descriptive, decomposition-based approaches for time-series modeling. We highlight as major benefits of this work the new capabilities to accurately and automatically extract load intensity models with 15.2% median modeling error on average from a representative set of ten different real-world traces. Each extraction completes in less than 0.2 seconds on common consumer hardware. These results demonstrate and validate the capability of DLIM to capture realistic load intensity profiles.

DLIM-based applications and developments in the fields of benchmarking and system resource planning both at design-time and run-time are enabled by providing the automatic model extraction processes. Since DLIM has been released in 2014, it has been adopted by several third party researchers. It has been integrated into the Palladio Component Model (PCM) and its simulator as presented in the paper [LB14]. DLIM models became solution elements in two EU FP7 projects, namely CloudScale [BSL+13] and CACTOS [Om14]. The software artifacts of the LIMBO tool chain has been assessed by the Research Group of the Standard Performance Evaluation Corporation (SPEC) and included in the repository for peer-reviewed tools [2].

The remainder of this chapter is structured as follows: Section 4.2 describes the DLIM model and the hl-DLIM model. The extraction methods are presented in detail in Section 4.3. Section 4.4 offers a delimitation from the state-of-the-art in workload modeling, as presented earlier in Section 3.1.

4.2 The Descartes Load Intensity Models DLIM and hl-DLIM

In this section, we first explain in detail the more fine-grained DLIM meta-model, before we reduce the modeling capability to a higher abstraction level in the High-level DLIM with the target to increase usability for human users.

[1]LIMBO (DLIM tools): Primary page `http://descartes.tools/limbo`

[2]LIMBO (DLIM tools:): Also hosted at SPEC RG peer-reviewed tools repository `https://research.spec.org/tools/overview/limbo.html`

4.2.1 DLIM Model

The Descartes Load Intensity Model (DLIM) describes load intensity over time. Specifically, the model is aimed at describing the variations of work unit arrival rates by capturing characteristic load intensity behaviors.

The basic idea for DLIM load intensity models is defining and combining piece-wise mathematical functions to approximate variable arrival rates over time. This idea is sketched in Figure 4.1. This idea for DLIM models supports load intensity profiles with periodicity and offers flexibility to adapt and incorporate unplanned events. It also allows for nested composition of model instances. This nesting results in DLIM instances always having a tree structure, where different nodes and leaves within the tree are added or multiplied onto their parents.

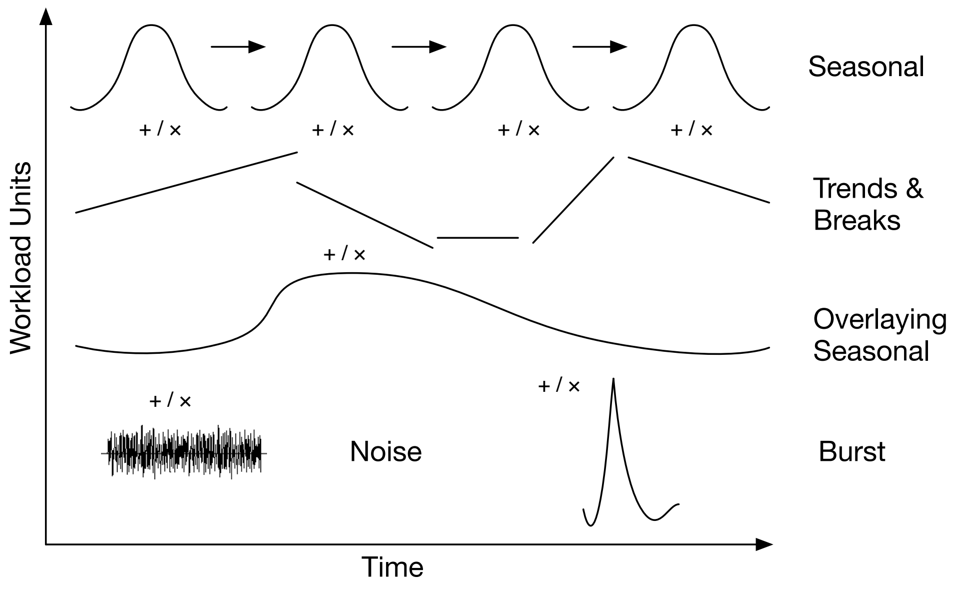


Figure 4.1: Abstract idea for the DLIM meta-model.

The resulting DLIM meta-model is visualized in Figure 4.2. It uses the Sequence as its central element for the composition of piece-wise mathematical functions. A Sequence carries an ordered set of TimeDependentFunctionContainers, which describe the duration of each time interval. The containers, in turn, contain the actual mathematical functions describing the load intensity during their interval. The Sequence's ordered set of TimeDependentFunctionContainers repeats as many times as indicated by the terminate-AfterLoops attribute. Alternatively, the sequence repeats for the time indicated by the terminateAfterTime attribute. The sequence terminates as soon as at least one of the termination criteria is met. Infinite sequences are not allowed and at least one termination

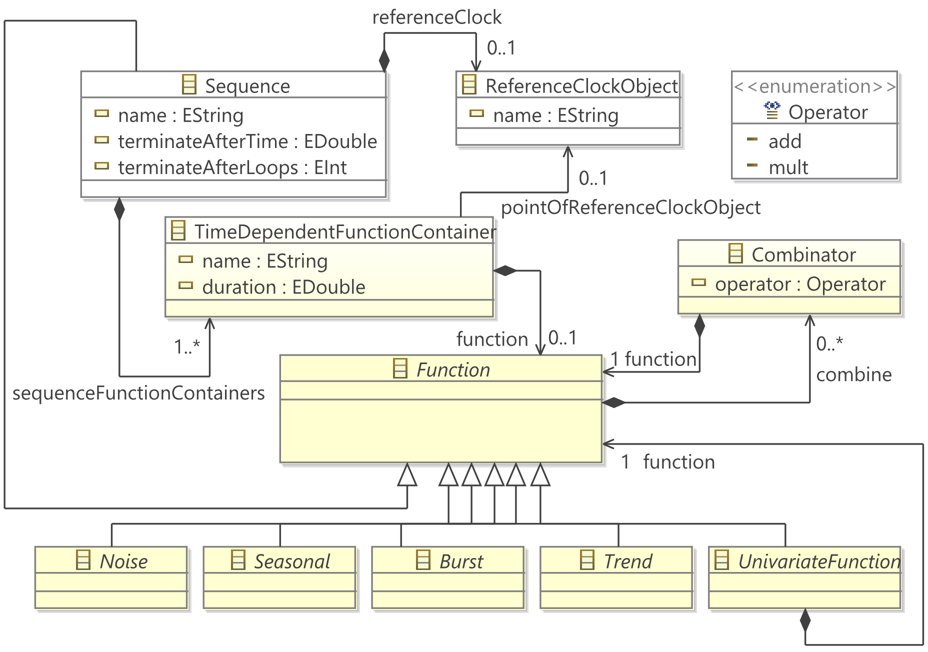


Figure 4.2: Outline of the DLIM meta-model excluding a detailed description of concrete Functions.

criterion must be specified. This ensures a finite time-frame and guarantees that benchmarks or predictions terminate.

Any DLIM Function may use mathematical operators called Combinators to contain additional Functions. A Combinator allows the multiplication or additon of a Function's load intensity with the load intensity as defined by another Function. In the context of the overall load variation over time, any Function contained within a Combinator is valid for the exact same duration as its containing parent Function. This containment results in trees of functions containing zero or more additional functions. All of these functions describe the load intensity behavior during a specific time period defined by the containing TimeDependentFunctionContainer.

The TimeDependentFunctionContainer describes its load intensity for a set duration, after which the next TimeDependentFunctionContainer in the parent Sequence's ordered set becomes valid.

Function is the abstract parent class to all classes denoting mathematical functions. It is contained within a TimeDependentFunctionContainer. A number of concrete children are provided that can be used as Functions. The default set of functions provides Noise, Seasonal, Burst, Trend, and UnivariateFunction specifications. New functions can be provided by inheritance. The most notable concrete Function is the Sequence. As a result, any function can contain a new Sequence using a mathematical operator. The containment hierarchy prevents cycles.

Figure 4.3 shows an example DLIM instance together with a plot of the model example in the lower part. In both model and plot, red areas map to the impact of the additive combinators and yellow areas to the impact of multiplicative combinator. The root Sequence (named "root") of the instance contains a TimeDependentFunctionContainer (TDFC) "season" with a duration of 24 abstract time units. "root" repeats its single TDFC three times before terminating. The container itself contains a sinus function modeling the load intensity. This sinus function repeats itself. As a result, the seasonal duration of 24 is technically speaking unnecessary, but was chosen nonetheless to add additional information to the model. "root" also contains Combinators, which modify its load intensity. The multiplicative Combinator contains a separate Sequence ("trends"), which contains two TDFCs. The first of those models a linear increase in load intensity, up to a factor of 1.8, whereas the latter models a decrease back to the original level. The "trends" Sequence terminates before "root" and stops modifying it after its first iteration. A "burst" Sequence and normal noise are added to "root". Note, that Combinators may contain any DLIM-Function. This is usually a Sequence, as it is the case for trends and bursts in this example model. Contained Sequences terminate on their own or at the end of the containing Sequence if their duration would exceed the parent's duration. All other functions, such as the noise in this example, are valid for the entire duration of their containing Sequence.

4.2.2 High-level DLIM

DLIM offers a convenient way of structuring and ordering functions for the description of load intensity profiles. Its tree of piece-wise mathematical functions already provides human users with better understanding of the load variations. However, abstract knowledge about variations contained in complex models can still be difficult to understand, as a large tree of composite functions may be difficult to grasp. The High-level (hl-)DLIM is a separate model which addresses this issue by providing means to capture load intensity variations described only by a limited number of non-hierarchical parameters instead of a potentially deeply nested tree of mathematical functions. These parameters can then be used to quickly define and characterize a load intensity model.

hl-DLIM separates into a Seasonal and Trend part (inspired by the time-series decomposition approach in BFAST [VHNC10]) and features a Burst and a Noise part. In contrast to DLIM, it is designed to model a subset of the most common load variation profiles in favor of better readability.

The Seasonal part describes the function that repeats after every seasonal duration (e.g., every day in a month long load intensity description). hl-DLIM describes the seasonal part using the following parameters (as shown in Figure 4.4): period, number of peaks, base arrival rate level, first peak arrival rate, last peak arrival rate, and the interval containing peaks. Arrival rates of additional peaks between the first and last peak are derived using linear interpolation. Linear interpolation is chosen because it is the most intuitive for the modeling user. More detailed changes can be performed in (non-hl-)DLIM.

The Trend part describes an overarching function that captures the overall change in the load intensity over multiple seasonal periods. It consists of a list of equi-length Trend

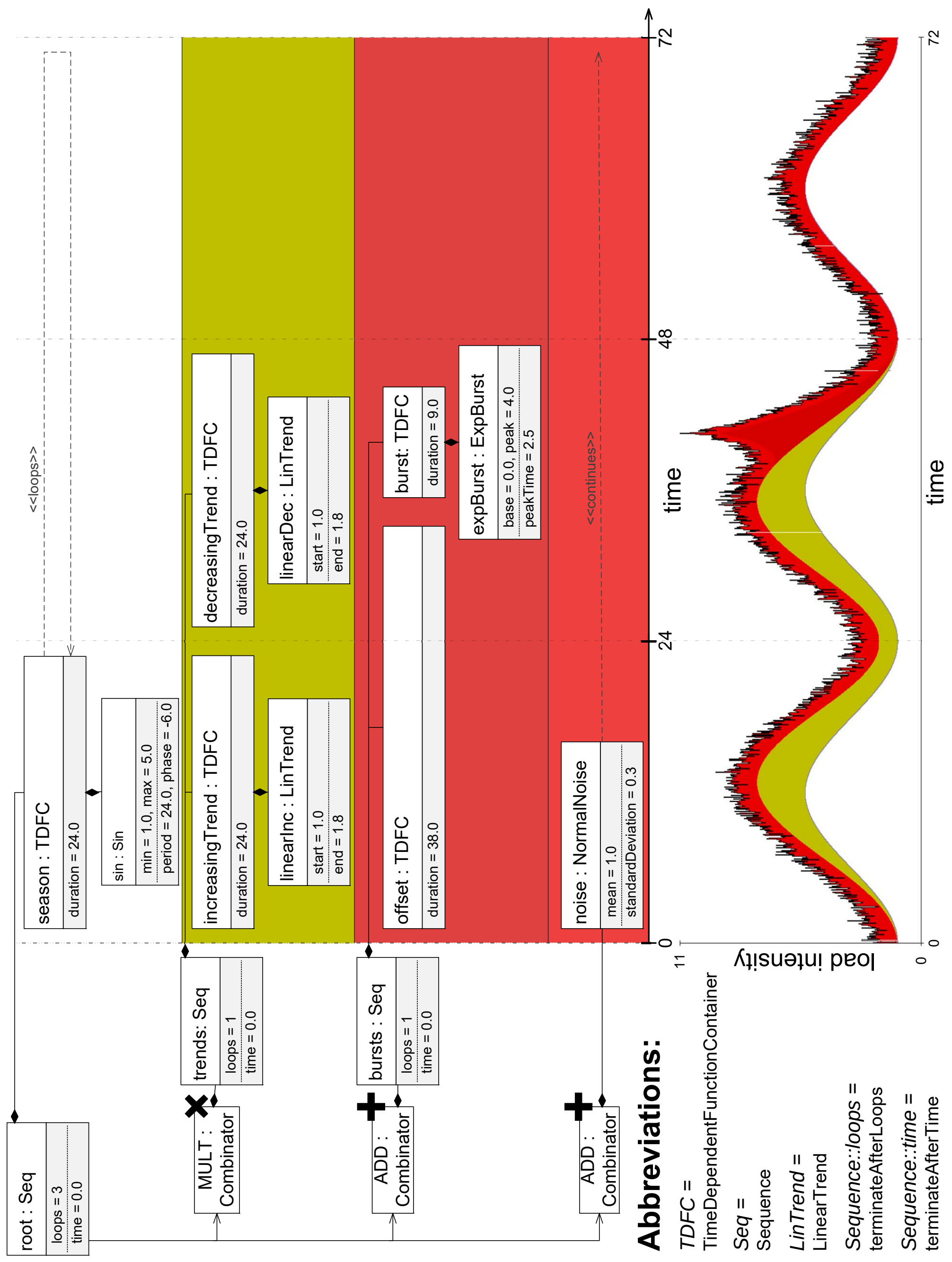


Figure 4.3: Example instance of a DLIM model.

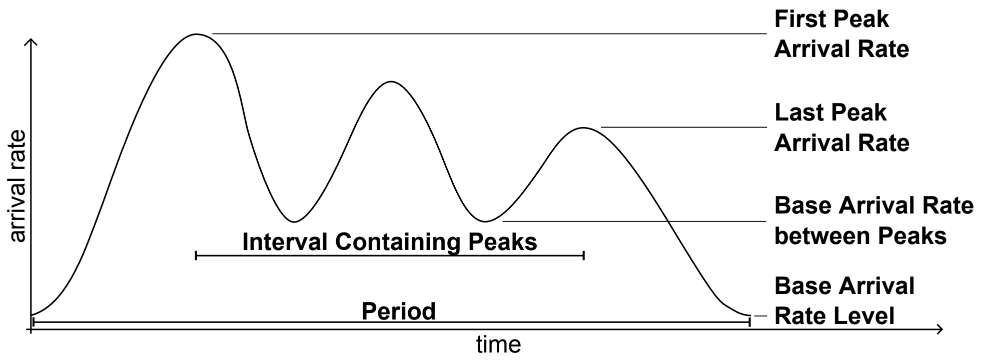


Figure 4.4: hl-DLIM Seasonal part.

segments. hl-DLIM describes the respective arrival rates at the start of each of these segments. The Trend must modify the arrival rate of the Seasonal part's maximum peak in such a way that it matches this specified target arrival rate. The actual trend segments between these specified points can be interpolated using any DLIM Trend function. In contrast to the Trend within BFAST, the hl-DLIM Trend can interact with the Seasonal part either by addition or multiplication. The restrictions posed by hl-DLIM to Trend modeling are intended to speed the human modeling process. (Non-hl-)DLIM allows for more varied Trend specifications. The Trend part is defined using the following parameters: number of seasonal periods within one trend (i.e., the length of a single trend segment), operator (addition or multiplication), and the list of seasonal arrival rate peaks. The latter defines the arrival rate at the beginning and end of the Trend segments. Since the list defines the maximum seasonal peak for seasonal iterations, trend segments can be interpreted as overlying interpolated functions, beginning and ending at the maximum peaks of their respective seasonal iterations. This interpretation is visualized in Figure 4.5.

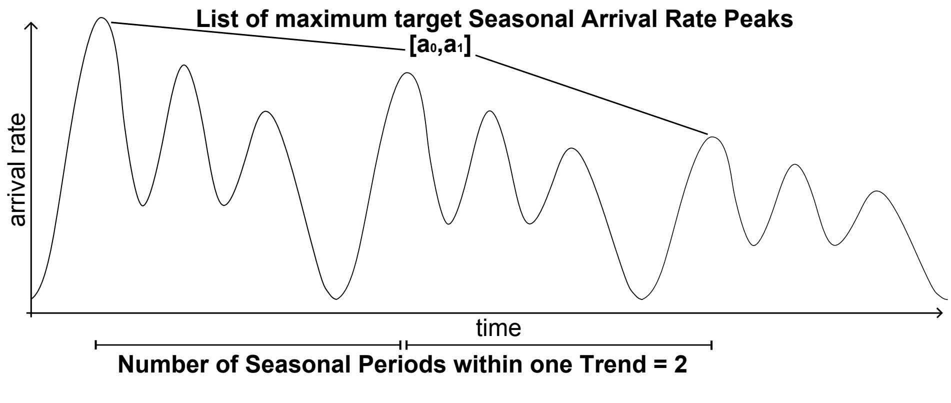

List of maximum target Seasonal Arrival Rate Peaks
[a0,a1]
arrival rate
time
Number of Seasonal Periods within one Trend = 2

Figure 4.5: hl-DLIM Trend part.

The Burst part allows the definition of recurring bursts, which are added onto the existing Seasonal and Trend behavior (in contrast to DLIM, where bursts may also be multiplicative). It is defined using the following parameters: First burst offset, inter-burst period, burst peak arrival rate, and burst width.

The Noise part allows the addition of uniformly distributed white noise. The distribution is defined by its upper and lower bounds, which are named Minimum Noise Rate and Maximum Noise Rate. Uniform noise is used in hl-DLIM, due to its ease and intuitiveness when being specified by a human user. Other Noise distributions can easily be added to DLIM instances, which are obtained from hl-DLIM instances via a model-to-model transformation.

4.3 Model Instance Extraction

In this section, we present three methods for the extraction of DLIM instances from arrival rate traces, consisting of pairs of arrival rates at their respective time stamps. Each model extraction method requires few configuration parameters that we discuss in detail. Given the set of configuration parameters, the extraction runs completely automated.

We define the following three methods:

1. **Simple DLIM Extraction Method** (s-DLIM):
   Extracts a DLIM instance. This process (and its resulting DLIM instance) are inspired by the time-series decomposition approach STL [CCMT90]. s-DLIM extracts a repeating Seasonal Part and a non-repeating Trend Part. The non-repeating Trend Part contains a list of Trend segments of fixed size, that interpolate between their start and end arrival rate value. The Trend list extends throughout the entire time duration for which the extracted model is defined. Additionally, a Burst Part and an optional Noise Part are extracted. s-DLIM is visualized in Figure 4.6.

2. **Periodic DLIM Extraction Method** (p-DLIM):
   This is a variation of the simple extraction process that features multiple repeating trends. Again a DLIM instance is extracted, however, in contrast to s-DLIM, p-DLIM does not feature a single list of equal length Trend segments. Instead it features multiple lists of Trends, each containing a fixed number of Trend segments of (potentially) different lengths.

3. **High-level DLIM Extraction Method** (hl-DLIM):
   Extracts an hl-DLIM instance. This process is based on the simple model extraction process and uses the information extracted by the latter to derive the parameters needed to construct an hl-DLIM instance.

4.3.1 Extracting a s-DLIM and p-DLIM Instance

The following sections describe the extraction of the different model parts by s-DLIM and p-DLIM. These two processes only differ in their approach to the extraction of the Trend Part.

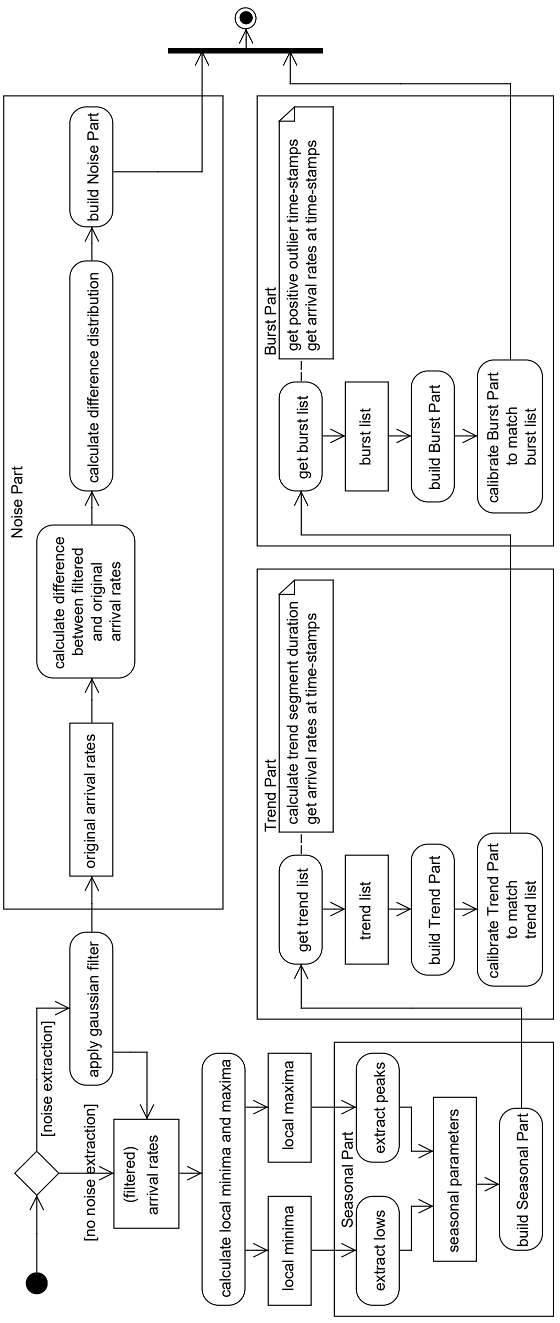

[noise extraction]
[no noise extraction]
apply gaussian filter
(filtered) arrival rates
calculate local minima and maxima
local maxima
local minima
Seasonal Part
extract peaks
extract lows
seasonal parameters
build Seasonal Part
Noise Part
original arrival rates
calculate difference between filtered and original arrival rates
calculate difference distribution
build Noise Part
Trend Part
calculate trend segment duration
get arrival rates at time-stamps
get trend list
trend list
build Trend Part
calibrate Trend Part to match trend list
Burst Part
get positive outlier time-stamps
get arrival rates at time-stamps
get burst list
burst list
build Burst Part
calibrate Burst Part to match burst list

Figure 4.6: Activity diagram of the Simple DLIM Extraction Method (s-DLIM).

Extracting the Seasonal Part

The Seasonal Part of the arrival rate trace is modeled using a Sequence of TimeDependent-FunctionContainers and their Functions. Each Function interpolates the corresponding peaks and lows within each seasonal period. As a result, the following data needs to be derived in order to build the Seasonal Part:

- Duration of the dominating seasonal period: automatically derived or configured.
- Arrival rate peaks and their time-stamps: automatically derived.
- Arrival rate lows and their time-stamps: automatically derived.
- Function type used to interpolate between peaks and lows: (pre-)configured.

The duration of the seasonal period can be set by the user in cases when the sampling interval of the input time series known. A trace that extends for multiple days and contains daily patterns, for example, features a period of 24 hours. In cases, where the seasonal period is not transparent, we provide an implementation of a heuristic to estimate the seasonal period duration. We follow a heuristic described in the work of Wang et al. [WSMH09, p.17] and compute the autocorrelation function for all lags up to one third of the time series length. As preprocessing, an overarching trend is removed from the input data using a regression spline. In the following step, the local maximum with the smallest lag is searched with a sliding window. Finally, we cross-check if multiples of the first local minimum lag correspond to other local minima. The peaks and lows are automatically determined by using a local minimum/maximum search on the arrival rates within the trace. The local arrival rate minima and maxima and their corresponding time-stamps within a seasonal period constitute the peaks and lows. Since the trace usually contains multiple seasonal periods, the respective median arrival rate value is selected for each local maximum and minimum within the Seasonal Part. Selecting the median instead of the mean reduces the impact of outliers on the extracted seasonal values. Outliers can be significant, but more importantly they are usually positive outliers. Negative outliers (disconnects, maintenance breaks) are less common and do not have as much of an impact, as their minimum size is 0. Positive outliers (bursts) are more common and intended to be detected separately using our burst detection. As a result, the derived functions interpolate first between the first median low and the first median peak, then between the first median peak and the second median low, and so on. The last low must be of the same arrival rate as the first low in order for the Seasonal Part to repeat seamlessly. The type of the interpolating function (linear, exponential, logarithmic, sin-flank) can be selected by the user, depending on his needs. According to our experience, the sin interpolation usually results in a good model fit. The Seasonal Part extraction is illustrated in Algorithm 4.1.

Extracting the Trend Part

The Trend Part consist of a series of functions (trend segments) that are either added or multiplied onto the Seasonal Part. Each trend segment begins at the maximum peak of the Seasonal Part and ends at the maximum peak of the Seasonal Part in a later Seasonal

ALGORITHM 4.1: Extracting the Seasonal Part

```
   Data: duration: seasonal period duration,
   LIST: list of tuples $\vec{t} = \begin{pmatrix} arrivalRate \\ timeStamp \end{pmatrix}$,
1  rootSequence: root Sequence of the DLIM instance
2  Function extractSeasonalPart()
3      MIN ← getLocalMinima (LIST)
4      MAX ← getLocalMaxima (LIST)
5      peakNum ← median (number of peaks within each Seasonal iteration)
6      for $i \leftarrow 0$ to peakNum − 1 do
7          peak [i].arrivalRate ← median (arrival rate of all $i_{th}$ $peaks \in$ MAX within each
             seasonal iteration)
8          peak [i].timeStamp ← median (time stamp of all $i_{th}$ $peaks \in$ MAX within
             each seasonal iteration)
           /* In seasonal iterations with more than peakNum
              peaks, the $i_{th}$ peak is selected, so that peaks are
              evenly spaced throughout that seasonal iteration.
              */
9          peak [i] ← $\begin{pmatrix} \text{peak}[i].arrivalRate \\ \text{peak}[i].timeStamp \end{pmatrix}$;
10     for $i \leftarrow 0$ to peakNum − 1 do
11         low [i].arrivalRate ← median (arrival rate of all $i_{th}$ $lows \in$ MIN within each
             seasonal iteration)
12         low [i].timeStamp ← median (time stamp of all $i_{th}$ $lows \in$ MIN within each
             seasonal iteration)
           /* In seasonal iterations with more than peakNum lows,
              the $i_{th}$ low is selected, so that lows are evenly
              spaced throughout that seasonal iteration.        */
13         low [i] ← $\begin{pmatrix} \text{low}[i].arrivalRate \\ \text{low}[i].timeStamp \end{pmatrix}$;
14     for $i \leftarrow 0$ to peakNum − 1 do
15         interpolatingFunction ← DLIM Function starting at low [i], ending at peak [i]
16         rootSequence.append (interpolatingFunction )
```

iteration. This minimizes errors with trend calibration. The trend extraction calibrates the trend in a way that the model output arrival rate at the trend segment's beginning (or end) equals the trace's actual arrival rate at the respective point in time. The shape of the trend function (linear, exponential, logarithmic, sin) is predefined as a sin-shape, but can be changed on demand.

Trend Part for s-DLIM: The simple extraction process features a list of equal-length trend segments. These segments have a user defined duration that is a multiple of the seasonal period. Like the seasonal period it is also selected using meta-knowledge about the trace. These segments are then calibrated at their beginning and end to match the arrival rates in the trace. The s-DLIM Trend Part extraction is displayed in Algorithm 4.2.

ALGORITHM 4.2: Extracting the Trend Part using s-DLIM

```
Data: duration: seasonal period duration,
LIST: list of tuples $\vec{t} = \begin{pmatrix} arrivalRate \\ timeStamp \end{pmatrix}$,
1  MAX: list of local maxima in LIST,
2  trendSequence: root Sequence of all Trend segments
3  Function extractTrendPart()
4      largestPeakOffset ← offset of peak with largest arrival rate within a seasonal
         iteration
5      largestPeakArrivalRate ← arrival rate of peak with largest arrival rate within a
         seasonal iteration
6      iterations ← LIST.$lastTuple.timeStamp$/duration
7      for $i$ ← 0 to iterations do
8          a ← nearestTuple(MAX, $i *$ duration + largestPeakOffset)
9          trendPoint [i] = a/largestPeakArrivalRate
10     trendSequence.append(constant trendPoint [0] with duration
         largestPeakOffset )
11     for $i$ ← 0 to iterations do
12         interpolatingFunction ← DLIM Function starting at trendPoint [i], ending at
             trendPoint [i+1]
13         trendSequence.append(interpolatingFunction with duration duration)
14     trendSequence.append(constant trendPoint [iterations] with duration
         (duration − largestPeakOffset))
15 Function nearestTuple(tuple list L, time)
16     returns the tuple $\vec{t} = \begin{pmatrix} arrivalRate \\ timeStamp \end{pmatrix} \in \mathbf{L}$ with minimal $d \leftarrow |\mathbf{L}.timeStamp - time|$
```

Trend Part for p-DLIM: The periodic extraction process takes into account, that multiple repeating trends may be part of the arrival rate trace. Examples are weekly and monthly trends. Since repeating trends (like the Seasonal Part's dummy function) should end on the same arrival rate as the arrival rate they started on (allowing seamless repetition), each of these repeating trends contains at least two trend segments. These trend segments' duration is a multiple of the seasonal period. Unlike the s-DLIM trend segments they are not required to be of equal length, thus allowing odd multiples of seasonal periods as total trend durations. The user selects lists of at least two trend segment durations for each repeating Trend Part.

Extracting the Burst Part

Extracting bursts is a matter of finding the points in time at which significant outliers from the previously extracted Seasonal and Trend parts are observed in the trace. However, for our extraction, bursts are explicitly not missed seasonal peaks. Extracted bursts are not intended to model the model's remaining noise, only semantic bursts. In the context of load intensity modeling, we define bursts as a significant amount of additional load that exceeds the seasonal pattern and is too short to be modeled as trend. To this end, we require a filter function that smooths the seasonal function to prevent modeling of missed seasonal peaks using bursts. With this filter function bursts can be detected if:

$$r(t) > (f(\sigma(t)) \cdot \tau(t)) \cdot c$$

where $r(t)$ is the original load intensity (arrival rate), f is the filter function, σ and τ are the previously extracted Seasonal and (multiplicative) Trend parts, and c is a constant factor that requires bursts to be a certain factor larger than filtered season and trend. c is set to a default value of 1.2.

Once a burst is found, it is added to the root Sequence and then calibrated to match the arrival rate from the trace. The filtered Seasonal Part used as reference model in the burst recognition activity differs from the actual extracted Seasonal Part. It is filtered so that the Seasonal Part used in the burst recognition activity does not interpolate between the peaks and lows of the original arrival rate trace. Instead it interpolates only between the peaks. This removes false positives due to seasonal periods that are slightly offset in time, however, it also eliminates bursts that do not exceed the current seasonal peak. This trade-off is considered acceptable, since time-wise offset seasonal periods are commonly observed.

Extracting the Noise Part

The Noise Part extraction consists of two steps: Noise reduction and the calculation of the noise distribution. The idea behind our approach is to first reduce the noise observed within the arrival rates contained in the trace, and then reconstruct the reduced noise by calculating the difference between the original trace and the filtered one. Having filtered the noise, the extraction of the Seasonal Part, Trend Part, and Burst Part are then performed on the filtered trace. This has a significant impact on the extraction accuracy of these parts, and thus on the overall accuracy of the extracted model instance, especially when extracting hl-DLIM instances, as will be shown in the model accuracy evaluation (Chapter 8). Depending on the trace, the overall accuracy of the DLIM extraction can be improved by noise elimination. In this case, we recommend applying noise extraction, even if the extracted noise component itself is deleted later on.

Noise Reduction: Noise is reduced via the application of a one-dimensional Gaussian filter on the read arrival rates. A Gaussian filter has a kernel based on the Gaussian distribution, it thus has the following form (as defined in foundational work, e.g. [BZ86]):

$$G(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{x^2}{2\sigma^2}}$$

We choose the kernel width ω depending on the Seasonal period length λ (duration of a single seasonal iteration) and the expected number of peaks n (local maxima) within a Seasonal period:

$$\omega = \frac{\lambda}{n}$$

A Gaussian filter's kernel width is defined as:

$$\omega = 6 \cdot \sigma - 1$$

As a result, the standard deviation is:

$$\sigma = \frac{\frac{\lambda}{n}+1}{6}$$

Calculating the Noise Distribution The Noise Part is modeled as a normally distributed random variable. This variable is added to the DLIM instance's root Sequence. The normal distribution's mean and standard deviation are calculated as the mean and standard deviation of the differences between the filtered arrival rate trace. This is illustrated in Algorithm 4.3.

s-DLIM and p-DLIM both only support the extraction of normally distributed noise. Other noise distributions are not supported. hl-DLIM extraction, however, supports the extraction of uniformly distributed noise.

ALGORITHM 4.3: Calculating the Noise distribution.

Data: LIST: list of read tuples $\vec{t} = \begin{pmatrix} arrivalRate \\ timeStamp \end{pmatrix}$;

```
1 Function calculatNoiseDistribution()
2     FILTERED_LIST ← applyGaussianFilter(LIST)
3     for $i \leftarrow 0$ to $|LIST| - 1$ do
4         difference[i] ← LIST [i].arrivalRate - FILTERED_LIST [i].arrivalRate
5     distribution ← normal distribution with mean(difference) and
        standardDistribution(difference)
```

4.3.2 Extracting an hl-DLIM Instance

The hl-DLIM extraction is similar to s-DLIM extraction. This section only highlights the differences between those two processes.

Seasonal Part

hl-DLIM is restricted to only support peaks with an equal distance from one another. The arrival rates of such peaks are linearly interpolated between the first peak's arrival rate and the last peak's arrival rate. When extracting an hl-DLIM instance from an arrival rate trace, the difference thus lies in the interval containing peaks and in the search for the maximum and minimum peak. The interval containing peaks is calculated as the time difference between the first and the last peak, the first peak's arrival rate is then set either to the minimum or maximum peak (depending on whether the first median peak has a greater or a smaller arrival rate than the last median peak in the trace) and the last peak is set to the corresponding counter-part.

Trend Part

Extracting the Trend Part is done almost identically as in the simple model extraction process, since hl-DLIM defines its Trend Part as a list of arrival rates at the beginning and end of each trend segment, identically to the arrival rate list extracted in s-DLIM. The only difference is the offset before the first trend segment begins. The trend segment always ranges from the maximum peak within one seasonal period to the maximum peak within a following seasonal period. The simple model extraction process allows this maximum peak to be any seasonal peak. hl-DLIM, however, only allows the first or last peak to be the maximum peak. As a result, the time offset for the first trend segment is slightly different.

Burst Part

Bursts are detected and calibrated using the same peak-only Seasonal Part as in s-DLIM. While the other model extraction processes modeled each burst individually, hl-DLIM only supports recurring bursts. Thus, only the first burst offset and the inter burst period are extracted, as well as only a single burst arrival rate. The first burst offset is selected based on its time-stamp, whereas the period between recurring bursts is calculated as the median inter burst period from the independent bursts. The burst arrival rate is also calculated as the median burst arrival rate.

Noise Part

In hl-DLIM, noise is extracted using our previously described filtering approach, thus having the same noise reduction side-effects as in the other model extraction processes. hl-DLIM, however, only supports a uniformly distributed random variable as noise. In order to eliminate outliers but keep the intuitiveness of hl-DLIM, hl-DLIM extraction only approximates the noise distribution of the original trace, eliminating major outliers. For this approximation the minimum and maximum value of the respective uniform distribution are selected as the 10^{th} and 90^{th} percentile of the difference between the filtered and unfiltered arrival rates.

4.4 Delimitation from the State-of-the-Art in Load Profile Modeling

As discussed in related work in Section 3.1, several approaches to describe and generate workloads with variable intensity exist in literature. However, they differ from our approach in the following two key aspects.

First, a set of approaches that work purely statistically using independent random variables and therefore do not offer models describing deterministic load intensity changes over time.

Second, approaches for workload or user behavior modeling that capture the structure of the actual units of work they dispatch or emphasize the behavior of users after their arrival on the system.

In contrast, DLIM models focus on the description of request or user arrivals, and not on user behavior and its impact after arrival. We combine both deterministic and statistical approaches. This enables a mapping between load profile variations and their respective time-stamps. DLIM uses a composition of piece-wise defined mathematical functions. In contrast to statistical approaches, which model distributions, DLIM models a concrete load intensity for each distinct point in time.

DLIM also supports random noise including spikes of various shapes and therefore represent noise, random events and individual outliers as well. To the best of our knowledge, a directly comparable combined deterministic and statistical descriptive modeling approach for load intensity profiles is not published in literature.

4.5 Concluding Remarks

This chapter presents the Descartes Load Intensity Model (DLIM) for capturing load profiles in a structured combination of piecewise-defined mathematical functions. The design of the DLIM meta-model is our answer to RQ A.1 („How to model load intensity profiles from real-world traces in a descriptive, compact, flexible and intuitive manner?“). Second, to answer RQ A.2 („How to automatically extract instances of load intensity profiles from existing traces with a reasonable accuracy and computation time?“), we introduce three heuristic methods enabling the automated extraction of DLIM instances from existing arrival rate traces: (i) Simple DLIM Extraction (s-DLIM), (ii) Periodic DLIM Extraction (p-DLIM), and (iii) High-Level DLIM Extraction (hl-DLIM).

Model extraction, editing, accuracy assessment and exporting is part of the Limbo tool chain for DLIM models. We leverage DLIM extraction and modeling capabilities to generate representative load profiles when benchmarking and comparing approaches for auto-scaling. We thoroughly evaluate DLIM model expressiveness by computing the accuracy of automatically extracted model instances from a set of real-world traces in Chapter 8.

Chapter 5

Quantifying the Observable Elasticity of Auto-Scaling Mechanisms: Metrics and BUNGEE Measurement Methodology

Today's infrastructure clouds provide resource elasticity (i.e. auto-scaling) mechanisms enabling autonomic resource provisioning to reflect variations in the load intensity over time. These mechanisms impact on the application performance, however, their effect in specific situations is hard to quantify and compare. To evaluate the quality of elasticity mechanisms provided by different systems and configurations, respective metrics and benchmarks are required. Existing metrics for elasticity only consider the time required to provision and de-provision resources or the costs impact of adaptations. Existing benchmarks lack the capability to handle open workloads with realistic load intensity profiles and do not explicitly distinguish between the performance exhibited by the provisioned underlying resources, on the one hand, and the quality of the elasticity mechanisms themselves, on the other hand.

In this chapter, we answer RQ A.3 („What are meaningful intuitive metrics to quantify accuracy, timing and stability as quality aspects of elastic resource adaptations?“). After having defined a set of elasticity metrics, we address RQ A.4 („How can the proposed elasticity metrics be measured in a reliable and repeatable way to enable fair comparisons and consistent rankings even across systems with different performance of the underlying scaled resources?“). To this end, we propose a novel approach for benchmarking the elasticity of Infrastructure-as-a-Service (IaaS) cloud platforms independent of the performance exhibited by the provisioned underlying resources.

In the evaluation in Chapter 9, we show that each of the proposed metrics provides a consistent ranking of elastic systems on an ordinal scale concerning their elasticity aspect. Finally, we present an extensive case study of real-world complexity demonstrating that the proposed approach is applicable in realistic scenarios and can cope with different levels of resource performance.

5.1 Introduction

Infrastructure-as-a-Service (IaaS) cloud environments provide the benefit of Utility Computing [AFG+10]. Accordingly, many providers offer tools that allow customers to configure automated adaptation processes and thus benefit from the increased flexibility and the ability to react on variations in the load intensity. Predicting and managing the performance impact of such adaptation processes and comparing the elasticity of different approaches is still in its infancy. However, customers require that performance related service level objectives (SLOs) for their applications are continuously met.

Elasticity itself is influenced by these adaptation processes as well as by other factors such as the underlying hardware, the virtualization technology, or the cloud management software. These factors vary across providers and often remain unknown to the cloud customer. Even if they are known, the effect of specific configurations on the performance of an application is hard to quantify and compare. Furthermore, the available adaptation processes are quite different in their methods and complexity as shown in the surveys by Lorido-Botran et al. [LBMAL14], Galante et al. [GB12] and of Jennings and Stadler [JS15].

Previous works on elasticity metrics and benchmarks evaluate this quality attribute only indirectly and to a limited extent: The focus of existing metrics lays either on the technical provisioning time [LYKZ10, LOZC12, CCB+12], on the response time variability [BKKL09, DMRT11, ASLM13], or on the impact on business costs [ILFL12, FAS+12, Sul12, Wei11, TTP14]. Existing approaches do not account for differences in the efficiency of the underlying physical resources and employ load profiles that are rarely representative of modern real-life workloads with variable load intensities over time. However, the quality of a mechanism in maintaining SLOs depends on the scenario and workload.

In this chapter, we propose: (i) a set of intuitively understandable and hardware-independent metrics for characterizing the elasticity of a self-adaptive system including ways to aggregate them and (ii) a novel benchmarking methodology, called Bungee, for evaluating the elasticity of IaaS cloud platforms using the proposed metrics.

The metrics, we propose here, support evaluating both the accuracy and the timing aspects of elastic behavior. We discuss how the metrics can be aggregated and used to compare the elasticity of cloud platforms. The metrics are designed to support human interpretation and to ease decision making by comparing resource supply and demand curves. The proposed elasticity benchmarking approach Bungee[1] supports the use of characteristic open workload intensity profiles tailored to the intended application domain. It leverages the Limbo[2] toolkit [vKHK14a] with its underlying modeling formalism DLIM for describing load profiles (see Chapter 4). Different levels of system performance are accounted for by performing an automated calibration phase, the results of which are used to adjust the load profile executed on each system from the considered systems under test. In combination with the proposed metrics this allows an independent and quantitative evaluation of the actual achieved system elasticity.

In the evaluation in Chapter 9, we demonstrate that the proposed metrics provide a consistent ranking of elastic system and adaptation process configurations on an ordinal scale. The Bungee benchmarking approach is evaluated by applying it to an extensive real-world

[1]Bungee Cloud Elasticity Benchmark: `https://descartes.tools/bungee`
[2]Limbo Load Intensity Modeling: `https://descartes.tools/limbo`

workload scenario considering deployments on both a private cloud (based on Apache CloudStack) and the public Amazon Web Services (AWS) EC2 cloud. The evaluation scenario employs a realistic load profile, consisting of several millions of request submissions, and is conducted using different types of virtual machine instances that differ in terms of their exhibited performance. We demonstrate that the proposed approach is applicable in realistic scenarios and can cope with different levels of resource performance.

The remainder of this chapter is structured as follows: Section 5.2 proposes a set of metrics for the quantification of elasticity followed by three different metric aggregation approaches in Section 5.3 and a discussion of metric properties in Section 5.4. Section 5.5 explains the measurement methodology implemented in the benchmarking framework Bungee in detail.

5.2 Elasticity Metrics

In order to compare and quantify the performance of different auto-scalers in practice and across deployments, we use a set of both system- and user-oriented metrics. Since initially proposed in a definition paper [HKR13] in 2013, the system-oriented elasticity metrics have been iteratively refined based on countless discussions, own experience from experiments and reviewer feedback. The set of metrics was proposed to and has been endorsed by the Research Group of the Standard Performance Evaluation Corporation (SPEC) as documented in a corresponding technical report [HKO+16]. This report has been extended as a SPEC RG group effort and resulted in a journal article [HBK+18] [currently under review]. The set of elasticity metrics has been adopted by a number of other researchers applied in several papers, e.g., [IAEH+17, IAEH+18, PAEÅ+16].

We propose and distinguish between three different types of elasticity metrics:

- provisioning accuracy metrics that explicitly distinguish between over- and underprovisioning states and quantify the (relative) amount of resources supplied on average in excess or to less.
- wrong provisioning timeshare metrics that again explicitly distinguish between over- and underprovisioning states and quantify the percentage of time spend in each state.
- instability and jitter metrics to quantify the degree of convergence of demand and supply. Instability denotes the percentage of time in which demand and supply change in opposite directions. Jitter accounts for the relative number of superfluous (positive) or missed (negative) adaptations in the supply.

Since under-provisioning results in violating SLOs, a customer might want to use a system that minimizes under-provisioning ensuring that enough resources are provided at any point in time, but at the same time minimizing the amount of over-provisioned resources. The defined separate accuracy and timeshare metrics for over- and under-provisioning allow providers to better communicate their auto-scaling capabilities and customers to select an auto-scaler that best matches their needs.

In addition to the elasticity oriented metrics, we suggest to compute user-oriented metrics: we consider the amount of adaptations, the average amount of VMs, the accounted instance minutes, and the average and median response time in combination with the percentage of

SLO violations as important indicators. However in this thesis, we do not take cost-based metrics into account, as they would directly depend on the applied cost model of a provider and thus would be inherently biased.

When using only individual metrics for judging the performance of auto-scalers, the results can be ambiguous. Hence, we use three different methods for deriving an overall result from the individual metrics: (i) We calculate the auto-scaling deviation of each auto-scaler from the theoretically optimal auto-scaler. (ii) We perform pairwise comparisons among the auto-scalers. (iii) We compute the gain of using an auto-scaler via an elastic speedup metric. Each elasticity and aggregate metric is explained in the remainder of this section. For the following equations, we define:

- T as the experiment duration with time $t \in [0, T]$
- s_t as the resource supply at time t
- d_t as the resource demand at time t
- Δt as the time interval between the last and current change in demand d or supply s.

The resource demand d_t is the minimal amount of resources required to meet a predefined SLO under the load at time t. The demand curve is derived based on load measurements by Bungee (see Section 5.5). The resource supply s_t is the monitored number of running VMs at time t.

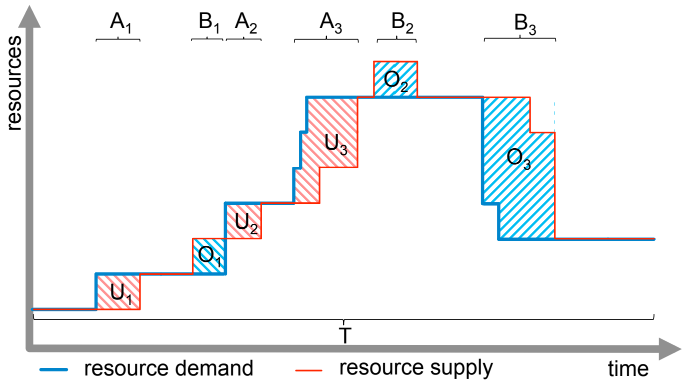


Figure 5.1: Illustrating example for accuracy (U, O) and timing (A, B) metrics - red/blue areas indicate under-/over-provisioning.

Figure 5.1 illustrates the time spans in under-/overprovisioned states as A_i, B_i and areas U_i, O_i that can be derived by combining the supply s_t and demand d_t curves.

5.2.1 Provisioning Accuracy

The provisioning accuracy metrics θ_U and θ_O describe the (relative) amount of resources that are underprovisioned, respectively, over-provisioned during the measurement interval, i.e., the under-provisioning accuracy θ_U is the amount of missing resources required to meet the SLO in relation to the current demand normalized by the experiment time. Similarly, the over-provisioning accuracy θ_O is the amount of resources that the auto-scaler supplies in excess of the current demand normalized by the experiment time. Values of this metric lie in the interval $[0,\infty)$, where 0 is the best value and indicates that there is no under-provisioning or over-provisioning during the entire measurement interval.

$$\theta_U[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} \frac{max(d_t - s_t, 0)}{\max(d_t, \varepsilon)} \Delta t \tag{5.1}$$

$$\theta_O[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} \frac{max(s_t - d_t, 0)}{\max(d_t, \varepsilon)} \Delta t \tag{5.2}$$

with $\varepsilon > 0$; we selected $\varepsilon = 1$.

These normalized accuracy metrics are particularly useful when the resource demand has a large variance over time, and it can assume both large and small values. In fact, under-provisioning of one resource unit when two resource units are requested is much more harmful than under-provisioning one resource unit when hundred resource units are requested. Therefore, this type of normalization allows a more fair evaluation of the obtainable performance.

For an intuitive interpretation when comparing result in experiments with a low variation in the resource demand, we define the unscaled provisioning accuracy metrics a_U and a_O as the average amount of resource units by which the demand exceeds the supply for a_U, and analogously the average amount of superfluous resources during overprovisioned periods for a_O :

$$a_U[\#res] := \frac{1}{T} \cdot \sum_{t=1}^{T} max(d_t - s_t, 0)\Delta t \tag{5.3}$$

$$a_O[\#res] := \frac{1}{T} \cdot \sum_{t=1}^{T} max(s_t - d_t, 0)\Delta t \tag{5.4}$$

Figure 5.1 shows the intuitive the meaning of the accuracy metrics. Under-provisioning accuracy a_U is equivalent to summing the areas U where the resource demand exceeds the supply normalized by the duration of the measurement period T. Similarly, the over-provisioning accuracy metric a_O is based on the sum of areas O where the resource supply exceeds the demand.

5.2.2 Wrong Provisioning Time Share

The wrong provisioning time share metrics τ_U and τ_O capture the portion of time in percentage, in which the system is under-provisioned, respectively over-provisioned, during the experiment, i.e., the under-provisioning time share τ_U is the time relative to the measurement duration, in which the system has insufficient resources. Similarly, the over-provisioning time share τ_O is the time relative to the measurement duration, in which the system has more resources than required. Values of this metric lie in the interval $[0, 100]$. The best value

0 is achieved when no under-provisioning, respectively no over-provisioning, is detected within a measurement.

$$\tau_U[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} max(sgn(d_t - s_t), 0)\Delta t \tag{5.5}$$

$$\tau_O[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} max(sgn(s_t - d_t), 0)\Delta t \tag{5.6}$$

5.2.3 Instability

The accuracy and timeshare metrics quantify the core aspects of elasticity. Still, systems can behave differently while producing the same metric values for accuracy and timeshare metrics. An example of such a situation is shown in Figures 5.2 and 5.3: Systems A and B exhibit the same accuracy and spend the same amount of time in the under-provisioned and over-provisioned states. However, B triggers three unnecessary resource supply adaptations whereas A triggers seven. This results in a different fraction of time in which the system is in stable phases. We propose to capture this with a further metric called instability.

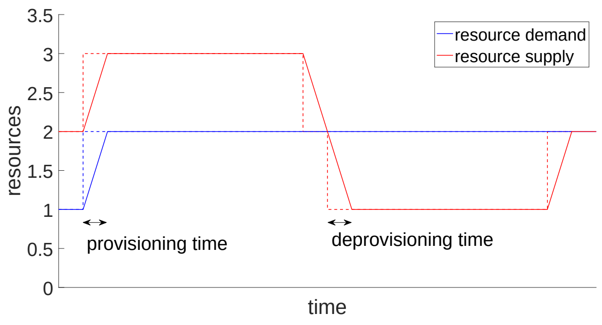


Figure 5.2: System A: Systems with different elastic behaviors that produce equal results for *accuracy* and *timeshare* metrics

We define the instability metric v as the fraction of time in which the sign of the changes in the curves of supplied and demanded resource units are not equal. As a requirement for the calculation of this metric, the average provisioning and deprovisioning time per resource unit has to be determined experimentally before or during the measurement. The step-functions of demanded and supplied resource units are transformed to ramps based on the average deprovisioning and provisioning time as depicted in Figure 5.2. Without this transformation, the resulting value would become either zero or depend on the sampling granularity of the demand and supply curves d_t and s_t. In summary, v measures the fraction of time in which the demanded resource units and the supplied units change into different directions:

$$v[\%] = \frac{100}{T-1} \cdot \sum_{t=2}^{T} min(|sgn(\Delta s_t) - sgn(\Delta d_t)|, 1)\Delta t \tag{5.7}$$

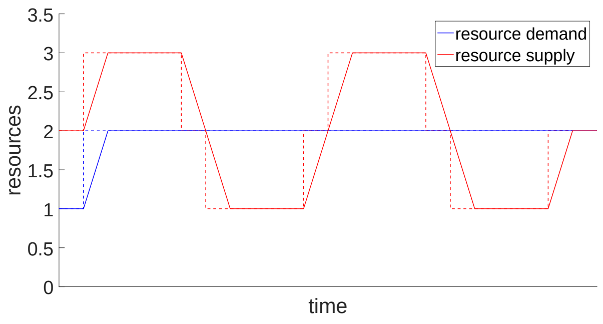


Figure 5.3: System B: Systems with different elastic behaviors that produce equal results for $accuracy$ and $timeshare$ metrics

An instability value close to zero indicates that the system adapts closely to changes in demand. A relatively high value indicates that the system oscillates heavily and does not converge to the demand. In contrast to the accuracy and timeshare metrics, a υ value of zero is a necessary but not sufficient requirement for a perfect elastic system, e.g., continuously to few resources in parallel to the demand would result in a value of zero. The instability is a useful indicator for reasoning about the cost-overhead of instance-time-based pricing models as well as for the operators view on estimating resource adaptation overheads.

Calculating Instability The demand d and supply s are discrete stair functions. The changes of the discrete stairs have no duration as the curves simply skip to another y-value. Theoretically, at every point in the duration of the experiment, demand and supply curves are parallel. Therefore, the instability metric as defined above cannot be calculated straight forward because it would always result in the value zero or depend the sampling granularity of demand and supply curves d_t and s_t.

To calculate the instability metric in a well-defined and fair way, we introduce the average measured provisioning and a deprovisioning time per resource unit right before an impulse happens, to artificially generate increasing and decreasing intervals where demand and supply change their amount. Then, the instability can be calculated as defined. Figure 5.4 shows the process of inserting provisioning times before an impulse. First, the demand and supply curves are given. Afterwards, at each impulse is replaced by a diagonal line. This diagonal is created using the provisioning time for upscaling and the deprovisioning time for downscaling. For upscaling, the upper end of the impulse is used as the endpoint of the diagonal. The provisioning time is used as distance to move to the left for finding the x-value of the start point of the diagonal, i.e. the start of the diagonal represents the time when a virtual machine is requested and the end of the diagonal the time when the virtual machine is available. The y-value is defined as the y-value of the lower end of the impulse. This way, a triangle is created and the diagonal is inserted. For downscaling the calculation works analogously. The lower end point of the impulse is used as end point of the diagonal. Then, the deprovisioning time is used to find the x-value of the start point and the upper end of the impulse is used to define the y-value. This is done for all impulses in demand and supply curves. In case the inserted diagonals intersect each other as can be seen in the center of

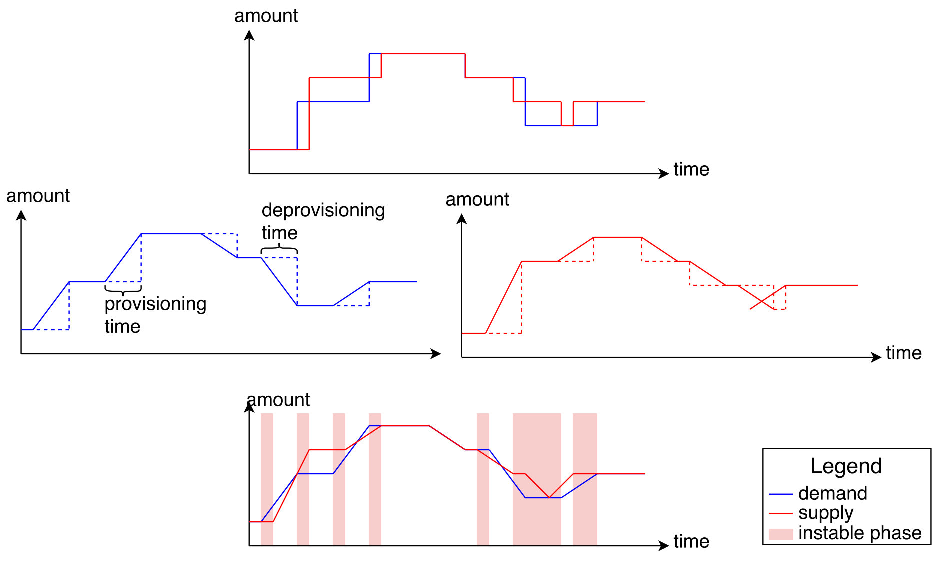


Figure 5.4: Preparations for instability computation.

Figure 5.4 on the right, there has to be additional handling that is described in the following. After both are processed, they can be merged and the instability can be calculated. At the bottom of Figure 5.4, the red areas show the unstable phases, where demand and supply do not change in the same direction, that are considered in the metric. Here, it does not matter how large the gradient is. Instead, only the sign of the gradient is important, whether the function increases or decreases.

The additional handling of intersection diagonals is described in the following. In general, the intersection point has to be calculated and the y-value is set to the y-value of the lower end points of the diagonal as shown at the bottom in Figure 5.5. Therefore, the mathematical equation of a line is used. This equation of a line can be defined using the first derivation of it. Therefore, the y-intersect is needed, that can be calculated by using the negative gradient that times the x-value. Finally, the derivation of the equation of a line is the gradient.

$$
\begin{aligned}
f(x) &= m \cdot x + c \\
f(x) &= f'(x) \cdot x + f(0) \\
f(0) &= (-m) \cdot (x) \\
f'(x) &= m
\end{aligned}
\tag{5.8}
$$

Four points are given to define two diagonals. These are used to find the intersection point. To calculate the intersection point, two points per diagonal are given. Figure 5.5 visualizes the given points, the diagonals, the intersection and the resulting point after calculation. The points a and b are the original points.

The points a' and b' are the newly inserted points for creating the diagonal.

$$\begin{aligned} f(a_x) &= a_y \\ f(a'_x) &= a'_y \\ f(b_x) &= b_y \\ f(b'_x) &= b'_y \end{aligned} \tag{5.9}$$

These given points can now be used to define the two equations of a line for the diagonals.

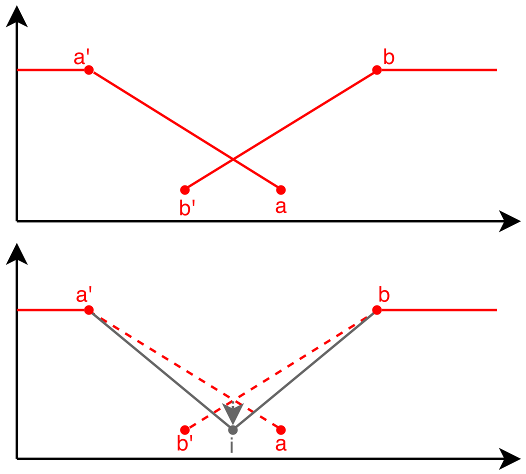


Figure 5.5: Preparations for Instability - intersection resolution.

Therefore, the gradient of both diagonals is calculated using the y-values of the points. Afterwards, the values can be inserted into the equation of a line, that results in two equations of a line for the diagonals.

$$\begin{aligned} m_a &= a_y - a'_y \\ m_b &= b_y - b'_y \\ f_a(x) &= m_a \cdot x + a_x \cdot (-m_a) \\ f_b(x) &= m_b \cdot x + b_x \cdot (-m_b \end{aligned} \tag{5.10}$$

Afterwards, the equations for the diagonals are set equal and the resulting equation is solved to get the x-value of the intersection point i_x. The y-value of the point is set to be the y-value of one of the surrounding points (a or b'). After finding this intersection point and inserting it into the data, the surrounding points, a and b' are removed. Now, after conducting this transformation, the instability can be calculated as described in the section above.

$$\begin{aligned}
f_a(x) &= f_b(x)\\
m_a \cdot x + a_x \cdot (-m_a) &= m_b \cdot x + b_x \cdot (-m_b)\\
m_a \cdot x - m_b \cdot x &= b_x \cdot (-m_b) - a_x \cdot (-m_a)\\
i_x &= \frac{b_x \cdot (-m_b) - a_x \cdot (-m_a)}{m_a - m_b}
\end{aligned} \tag{5.11}$$

5.2.4 Alternative Stability Measure Jitter

Originally, we proposed the jitter metric j to quantify the convergence and inertia of elasticity mechanisms. The newer instability metric v has the strong benefit of a finite positive value range $[0, 100]$. This is an important aspect for integration into an aggregated metric.

The jitter metric j compares the amount of adaptations in the supply curve E_S with the number of adaptations in the demand curve E_D. The difference is normalized by the length of the measurement period T: If a system de-/allocates more than one resource unit at a time, the adaptations are counted individually per resource unit.

$$j\left[\frac{\#}{t}\right] = \frac{E_S - E_D}{T} \tag{5.12}$$

A negative j value indicates that the system adapts rather sluggish to changes in the demand. A positive j value means that the system tends to oscillate like System A (little) and B (heavily) as in Figures 5.2 and 5.3. High absolute values of j value in general indicate that the system is not able to react on demand changes appropriately. In other word, the jitter metric denotes the average amount of missed (negative) or superfluous (positive) adaptations per time unit. In contrast to the accuracy and timeshare metrics, and as for instability v, a jitter j value of zero is a necessary, but not sufficient requirement for a perfect elastic system. The jitter metric j is easier to compute compared to the instability metric and also comes with an intuitive interpretation. It has an theoretically unbounded value range, but is capable to distinguish between oscillating/instable elasticity and elasticity with inertia to adapt timely to the workload changes.

5.3 Aggregating Elasticity Metrics

In this section, we introduce three different ways to aggregate the proposed elasticity metrics to a unique value for building a consistent ranking given a weight vector. The Auto-Scaling Deviation is based on the Minkowski distance and quantifies directly the distance to the theoretical optimal auto-scaling. The pairwise competition requires a fixed set of experiments run, whereas the elastic speedup just needs a defined reference.

5.3.1 Auto-Scaling Deviation

In order to quantify the overall performance of an auto-scaler and rank them, we propose to calculate the auto-scaling deviation σ of a given auto-scaler compared to the theoretically optimal auto-scaler. For the calculation of the deviation σ between two auto-scalers, we use the Minkowski distance d_p:

$$\text{Let x, y} \in \mathbb{R}^n \text{ and } 1 \leq p \leq \infty:$$

$$d_p(x,y) = \|x-y\|_p := \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}} \tag{5.13}$$

Here, the vectors consist of a subset of the aforementioned system- and user-oriented evaluation metrics. We take the provisioning accuracy θ, the wrong provisioning time share τ, and the instability υ into account. The metrics are specified as percentages. The closer the value of a metric is to zero, the better the auto-scaler performs with respect to the aspect characterized by the respective metric, i.e., the closer the auto-scaling deviation is to zero, the closer the behavior of the auto-scaler to the theoretically optimal auto-scaler.

The first step is to calculate the elasticity metrics. Then, we calculate the overall normalized provisioning accuracy θ and the overall wrong provisioning time share τ. Hereby, we use a weighted sum for both metrics consisting of both components and a penalty factor $0 < \gamma < 1$. This penalty can be set individually to reflect custom (in our experiments, we always set γ to 0.5 to reflect indifference), with $\gamma > 0.5$ indicating that under-provisioning is worse than over-provisioning, $\gamma = 0.5$ indicating that under- and over-provisioning are equally bad, and $\gamma < 0.5$ indicating that over-provisioning is worse than under-provisioning. This can be mathematically expressed as follows:

$$\theta[\%] := \gamma \cdot \theta_U + (1-\gamma) \cdot \theta_O \tag{5.14}$$

$$\tau[\%] := \gamma \cdot \tau_U + (1-\gamma) \cdot \tau_O \tag{5.15}$$

In the last step, the Minkowski distance d_p between the auto-scaler and the theoretically optimal auto-scaler is calculated. As the theoretically optimal auto-scaler is assumed to know when and how much the demanded resources change, the values for provisioning accuracy θ, wrong provisioning time share τ, and instability υ are equal to zero. In other words, if an auto-scaler is compared to the theoretically optimal auto-scaler, the L_p-norm can be used as $\|x-0\|_p = \|x\|_p$ with $x = (\theta, \tau, \upsilon)$, i.e., in our case the auto-scaling deviation σ between an auto-scaler and the theoretically optimal auto-scaler is defined as follows:

$$\sigma[\%] := \|x\|_3 = \left(\theta^3 + \tau^3 + \upsilon^3\right)^{\frac{1}{3}} \tag{5.16}$$

In summary, the smaller the value of the auto-scaling deviation σ, the better rated is the auto-scaler in the given context.

5.3.2 Pairwise Competition

Another approach for ranking - given a fixed set of auto-scaler experiments - is to use the pairwise comparison method [Dav87]. Here, for each auto-scaler the value of each metric is pairwise compared with the value of the same metric for all other auto-scalers. As values closer to zero are better, the auto-scaler with the lowest value gets one point. If a metric for both auto-scalers is equal, both auto-scalers get each half a point. In addition, we divide the reached score of each auto-scaler by the maximum achievable score. In other words, the pairwise competition κ shows the fraction of the achievable points each auto-scaler collected. For the comparison, we take the metrics $x = (\theta, \tau, \upsilon)$ into account. Mathematically, κ for an auto-scaler $a \in [1; n]$ with n as the number of auto-scalers can be expressed as:

$$\kappa_a[\%] := \frac{1}{(n-1)\cdot|x|} \cdot \sum_{i=1;i\neq a}^{n} \sum_{j=1}^{|x|} \omega(i,j) \qquad \text{where} \qquad \omega(i,j) := \begin{cases} 0, & x_a(j) > x_i(j) \\ 0.5, & x_a(j) = x_i(j) \\ 1, & x_a(j) < x_i(j) \end{cases} \tag{5.17}$$

In summary, the closer the value of the pairwise competition κ is to 1, the better rated is the auto-scaler in the given context and the other auto-scalers in competition.

5.3.3 Elastic Speedup

The elastic speedup score ϵ is computed similar to the aggregation and ranking of results in established benchmarks, e.g., SPEC CPU2006[3]. Here, the use of the geometric mean to aggregate speedups in relation to a defined baseline scenario is a common approach.

The geometric mean produces consistent rankings and is suitable for normalized measurements [FW86]. The resulting elastic speedup score allows to compare elasticity performance without having to compare each elasticity metric separately, and a later point in time add a new result to the ranking (in contrast to a closed set in, e.g., a pair-wise competition).

A drawback of the elastic speedup score is its high sensitivity to values close to zero and becoming undefined if one or more of the metrics are zero. To minimize the probability of zero-valued metrics, we aggregate as in Section 5.3.1 the normalized accuracy and timeshare metrics into a overall accuracy θ and a overall wrong provisioning timeshare metric τ, respectively. This way, θ and τ become zero only for the theoretical optimal auto-scaler. Thus, we compute the elastic speedup score ϵ based on the accuracy, timeshare and instability metrics for an elasticity measurement k - having the values of a shared baseline scenario *base* given - as follows:

$$\epsilon_k = \left(\frac{\theta_{base}}{\theta_k}\right)^{w_\theta} \cdot \left(\frac{\tau_{base}}{\tau_k}\right)^{w_\tau} \cdot \left(\frac{\upsilon_{base}}{\upsilon_k}\right)^{w_\upsilon} \qquad \text{with } w_\theta,\ w_\tau,\ w_\upsilon \in [0,1], w_\theta + w_\tau + w_\upsilon = 1 \tag{5.18}$$

The weights can be used to implement user-defined preferences, e.g. to increase the impact of accuracy and timeshare aspects compared to the instability aspect if desired. We assume here that a baseline measurement is available with the same application and workload profile executed within the same predefined range of resource units. In summary, the higher the value of the elastic speedup score ϵ_k, the better rated is the auto-scaler in the given context compared to a reference.

[3]SPEC CPU2006: `http://www.spec.org/cpu2006/`

5.4 Metric Discussion

In the previous sections, we defined a number of elasticity metrics and three different ways to aggregate them. In a mathematical sense, one would call these metrics more precisely measures as their computation is based on raw measurements and they do not fulfill all the requirements of a distance function (namely: non-negativity, identity of indiscernibles, symmetry, triangle inequality). In computer science, metrics are understood in a broader sense as a mean to quantify a certain property. Speaking general, it is impossible to prove correctness of a metric; it is more a common agreement on how to quantify the given property. One can discuss to what degree metrics fulfill characteristics of a good, well-defined and intuitive metric and additionally demonstrate their usefulness by meaningful results in rows of experiments (r.t. Section 9.3). Let us go in the following step by step over a list of metric characteristics:

Definition. A metric should come along with a precise, clear mathematical (symbolic) expression and a defined unit of measure, to assure consistent application and interpretation. We are compliant with this requirement, as all of our proposed metrics come with a mathematical expression, a unit or are simply time-based ratios.

Interpretation. A metric should be intuitively understandable. We address this by keeping the metrics simplistic and describe the meaning of each in a compact sentence. Furthermore, it is important to specify (i) if a metric has a physical unit or is unite-free, (ii) if it is normalized and if yes how, (iii) if it is directly time-dependent or can only be computed ex-post after a completed measurement and (iv) clear information on the value range and the optimal point. Aggregate metrics should keep generality and fairness, combined with a way to customize by agreement on a weight vector.

Measureability. A transparently defined and consistently applied measurement procedure is important for reliable measurements of a metric. For example, it is important to state where the sensors need to be placed (to have an unambiguous view on the respective resource), the frequency of sampling idle/busy counters and the intervals for reporting averaged percentages. The easier a metric is to measure, the more likely it is that it will be used in practice and that its value will be correctly determined. In the following sections, we define a measurement methodology (a.k.a benchmarking approach) for the proposed elasticity metrics and demonstrate their applicability in Section 9.3.

Repeatability. Repeatability implies that if the metric is measured multiple times using the same procedure, the same value is measured. In practice, small differences are usually acceptable, however, ideally, a metric should be deterministic when measured multiple times. We demonstrate that repeatability is possible in a controlled experiment environment and to a certain degree in the public cloud domain.

Reliability. A metric is considered reliable if it ranks experiment results consistently with respect to the property that is subject of evaluation. In other words, if System A performs better than System B with respect to the property under evaluation, then the values of the metric for the two systems should consistently indicate this (e.g., higher value meaning better score). For our proposed metrics, we demonstrate consistent ranking for each individual metric in a row of experiments, and for the aggregate metrics in a case study.

Linearity. A metric is considered linear if its value is linearly proportional to the degree to which the system under test exhibits the property under evaluation. For example, if a performance metric is linear, then twice as high value of the metric would indicate twice as good performance. Linear metrics are intuitively appealing since humans typically tend to think in linear terms. For our proposed elasticity metrics, we claim linearity as they are all based on sums or counts of raw measures. For the aggregate metrics, linearity might not be given. In the long run, the elasticity metric's distributions in reality should be analyzed to improve the reliability of their aggregation.

Independence. A metric is independent if its definition and behavior cannot be influenced by proprietary interests of different vendors or manufacturers aiming to gain competitive advantage by defining the metric in a way that favors their products or services. As our proposed metrics come with a mathematical definition and a measurement methodology, we claim that it should be possible to verify that an elasticity measurement was conducted according to given run-rules. Still, there could be ways to manipulate the results in a way we are not aware of yet.

5.5 Bungee Elasticity Benchmarking Framework

This section presents our benchmarking approach that addresses generic and cloud specific benchmark requirements as stated in Huppler [Hup09, Hup12] and Folkerts et. al [FAS+12]. A general overview of the benchmark components and of the benchmarking workflow is given in this section. The conceptual ideas about the essential benchmark components are discussed in individual subsections. We provide an implementation of the benchmark concept named Bungee[4].

Figure 5.6 illustrates an elastic cloud platform architecture as a blueprint for an infrastructure cloud "system under test" (SUT) together with the benchmark controller, which runs the benchmark. The individual benchmark components automate the process for benchmarking resource elasticity in four sequential steps as illustrated in Figure 5.7:

1. **System Analysis:** The benchmark analyzes the SUT with respect to the performance of its underlying resources and its scaling behavior.

2. **Benchmark Calibration:** The results of the analysis are used to adjust the load intensity profile injected on the SUT in a way that it induces the same resource demand on all compared systems.

[4] Bungee Cloud Elasticity Benchmark: `http://descartes.tools/bungee`

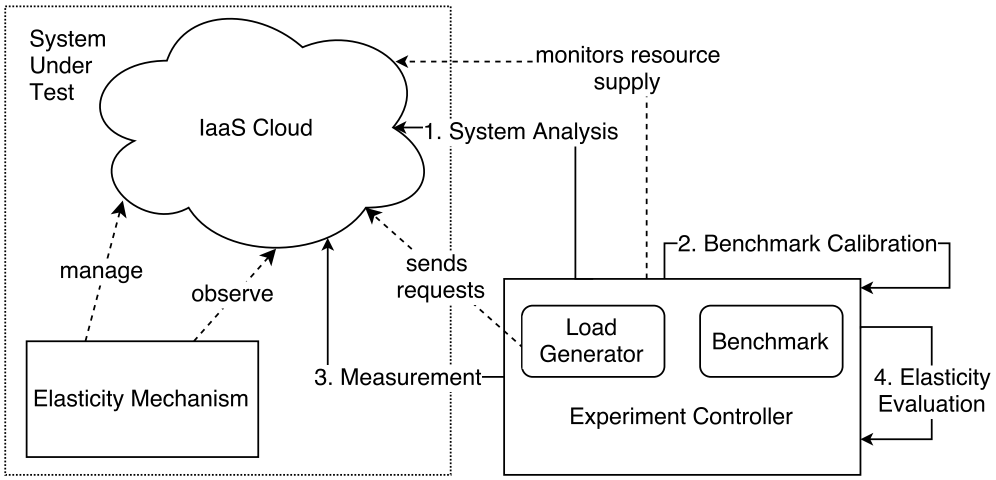


Figure 5.6: Blueprint for the SUT and the Bungee benchmark controller

3. **Measurement:** The load generator exposes the SUT to a varying workload according to the adjusted load profile. The benchmark extracts the actual induced resource demand and monitors resource supply changes on the SUT.

4. **Elasticity Evaluation:** The elasticity metrics are computed and used to compare the resource demand and resource supply curves with respect to different elasticity aspects.

The remainder of this section explains the benchmark components according to the following structure: Section 5.5.1 explains how workloads are modeled and executed. Section 5.5.2 explains why analyzing the evaluated system and calibrating the benchmark accordingly is required and how it is realized. Finally, Section 5.5.3 explains how the resource demand curve and the resource supply curve can be extracted during the measurement.

5.5.1 Load Modeling and Generation

This section covers modeling and executing workloads suitable for elasticity benchmarking.

Load Profile

A benchmark should stress the SUT in a representative way. Classical performance benchmarks achieve this by executing a representative mix of different programs. An elasticity benchmark measures how a system reacts when the demand for specific resources changes. Thus, an elasticity benchmark is required to induce a representative profile of demand changes. Changes in demand and accordingly elastic adaptation of the system are mainly caused by a varying load intensity. Hence, for elasticity benchmarking, it is important that the variability of the load intensity is directly controlled by generating precisely timed requests based on an open workload profile. Workloads are commonly modeled either as

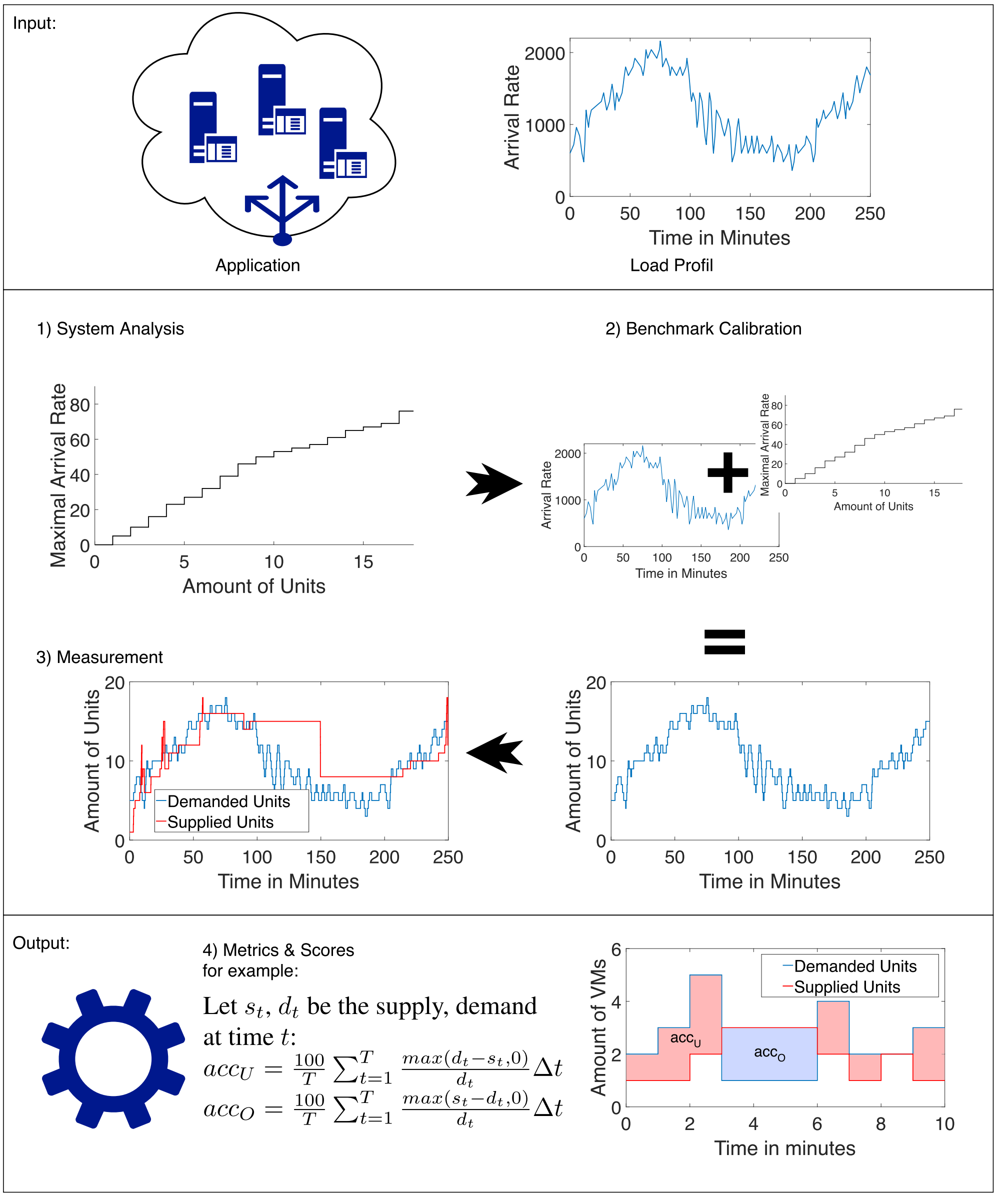


Figure 5.7: Bungee experiment workflow

closed workloads or as open workloads [SWHB06]. In closed workloads new job arrivals are triggered by job completions, whereas arrivals in open workloads are independent of job completions. Real-world load profiles are typically composed of trends, seasonal patterns, bursts and noise elements. To address these requirements, Bungee employs realistic load intensity profiles to stress the SUT in a representative manner.

In Chapter 4, we provide the DLIM formalism that enables us to define load intensity profiles in a flexible manner. The corresponding LIMBO toolkit, described in a tool paper [vKHK14a], facilitates the creation of new load profiles that are either automatically extracted from existing load traces or modeled from scratch with desired properties like seasonal patterns or bursts. The usage of this toolkit and the underlying formalism within Bungee allows the creation of realistic load profiles that remain configurable and compact.

Load Generation

In order to stress an elastic system in a reproducible manner, it is necessary to inject accurately timed requests into the SUT that mainly stress the resources that are elastically scaled. Depending on the request response time, the handling of consecutive requests (sending of a request and waiting for the corresponding response) overlaps and therefore requires a high level of concurrency. In its load generator, Bungee employs the thread pool pattern (also known as replicated worker pattern [FHA99]) with dynamic assignments of requests to threads.

When the maximum response time is known - for example when a timeout is defined for the requests - the amount of threads required to handle all requests without delays due to a lack of parallelism can be computed directly.

These threads can either be located on one node or be split up across several nodes. After each benchmark execution, Bungee evaluates the request submission accuracy by checking whether the 95^{th} percentile of the deviation between the defined load profile and the actual request submission times remained below a certain threshold.

Request Transmission Accuracy Checking

Sending requests in time is a crucial aspect to ensure that resource demands are induced correctly and repeatable over different runs. Therefore, we evaluate the accuracy of the request submission times, to assure valid benchmark runs. We achieve this, by matching the planned submission times with the timestamps of the actual request submission using unique identifiers. In the next step, statistical metrics like standard deviation and median on the differences are approved to be below defined thresholds for a valid run in a specific benchmarking scenario.

A simple approach for this evaluation bases on comparing the planned submission times of the requests with the real ones. Therefore, real submission times have to be logged by the load driver which executes the request. Unfortunately, the order of the logged timestamps is not necessarily equal to the order of planned timestamps. In particular, the logged timestamps can be disordered when the record for the request is written only after the response has been received.

Thus, it is necessary to match the logged real submission times with the correct planned submission times. This can be achieved by assigning a unique identificator (UID) to every timestamp. When this UID is logged together with the submission time, it is easy to match the real and planned submission times.

If planned and real time of submission are known for all requests, simple statistical measures such as sample mean $\overline{X}$ and sample standard deviation S can be calculated for difference of planned and real submission times. These measures allow insights into different aspects: $\overline{X}$ specifies how much requests are delayed on average. S specifies if the observed delay is rather constant or is varying. For accurately timed requests both measures should be close to zero. Since especially $\overline{X}$ is highly affected by outliers another option for measuring the submission accuracy is evaluating the p percentile, e.g., the 95^{th} percentile, of the difference between planned and real submission times.

5.5.2 Analysis and Calibration

The resource demand of a system for a fixed load intensity depends on two factors: The performance of a single underlying resource unit and the overhead caused by combining multiple resource units. Both aspects can vary from system to system and define distinct properties namely performance and scalability. Elasticity is a different property and should be measured separately. We achieve this by analyzing the load processing capabilities of a system before evaluating its elasticity. After such an analysis, it is known how many resource units the system needs for every constant load intensity level. The resource demand can then be expressed as a function of the load intensity: $d(i)$. With the help of this mapping function, which is specific for every system, it is possible to compare the resource demand to the amount of the actually allocated resources. In the following section, we explain how the mapping function can be derived.

(a) System A - resources with lower performance **(b)** System B - resources with higher performance

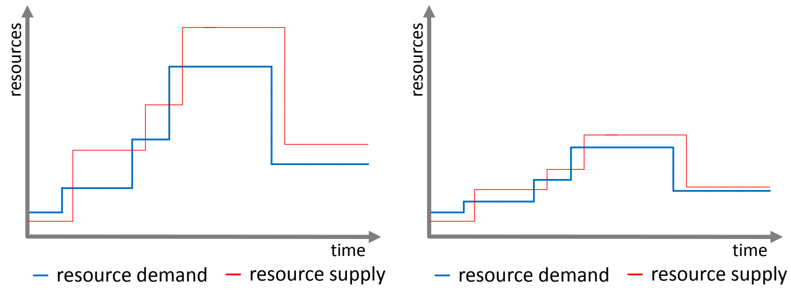


Figure 5.8: Different resource demand and supply curves for the same load profile

Figure 5.8 illustrates the effect of the same workload on two exemplary systems. Since the resources of System B are faster than those of System A, System B can handle the load with less resources than System A. For both systems, there exist some intervals in time where the system over-provisions and other intervals in time where they under-provision. As the resource demands of the systems are different, a direct comparison of their elasticity is not possible, an indirect comparison cloud be based on cost models.

In contrast to related approaches, the idea of our approach introduces an additional benchmark calibration step that adjusts the load profile injected on each tested system in a way that the induced resource demand changes are identical on all systems. By doing so, the possibly different levels of performance of the underlying resources as well as different scaling behaviors are compensated. With an identical injected resource demands it is now possible to directly compare the quality of the adaptation process and thus evaluate elasticity in a fair way. The realization consists of two steps, explained in the following two subsections.

System Analysis

The goal of the System Analysis is to derive a function that maps a given load intensity to the corresponding resource demand - in other words to specify the system's scalability. Hereby, the resource demand is the minimum resource amount that is necessary to handle the load intensity without violating a set of given SLOs. Therefore, the analysis assumes a predefined set of SLOs. They have to be chosen according to the targeted domain.

Since this System Analysis is not intended to evaluate elastic behavior, scaling is controlled manually by the framework during the analysis phase. The results of three hypothetical analyses are shown in Figure 5.9. The upper end of the line marks the upper scaling bound and thus the maximal load intensity the system can sustain without violating SLOs. This upper bound is either caused by a limited amount of available resources or by limited scalability due to other reasons like limited bandwidth or increased overhead. In the latter case, additional resources are available, but even after adding resources the SLOs cannot be satisfied. Figure 5.9 shows that the derived mapping function $d(i)$ is a step function. The function is characterized by the intensity levels at which the resource demand increases. Thus, analyzing the system means finding these levels.

The analysis starts by configuring the SUT to use one resource instance. This includes configuring the load balancer to forward all requests to this instance. Now, the maximum load intensity that the system can sustain without violating the SLOs is determined as the level at which the resource demand increases. The maximum load intensity is also referred to as the load processing capability. Several load picking algorithms that are able to find the load processing capability are discussed in [SMC$^+$08]. We apply a binary search algorithm to realize a searchMaxIntensity-function. Binary search consumes more time than a model guided search but, in contrast to the latter, it is guaranteed to converge and it is still effective compared to a simple linear search. Since the upper and lower search bounds are not known at the beginning, the binary search is preceded by an exponential increase/decrease of the load intensity to find those bounds. As soon as both bounds are known, a regular binary search is applied. Once the load processing capability for one resource is known, the system is reconfigured to use an additional resource unit and the load balancer is reconfigured accordingly. After the reconfiguration, the system should be able to comply with the SLOs again. The load processing capability is again searched with a binary search algorithm. This process is repeated until either there are no additional resources that can be added to the system, or a state is reached such that adding further resources does not increase the load processing capability. In both cases, the upper scaling bound is reached and the analysis is finished. .

Benchmark Calibration

The goal of the Benchmark Calibration activity is to induce the same demand changes at the same points in time on every system. To achieve this, Bungee adapts the load intensity curve for every system to compensate for different levels of performance of the underlying resources and different scaling behaviors. The mapping function $d(i)$ derived in the previous System Analysis step, contains information about both. Figure 5.9 illustrates what impact different levels of performance and different scaling behaviors may have on the mapping function. Compared to System C, the underlying resources of System D are more efficient. Hence, the steps of the mapping function are longer for System D than for System C. For both system, there is no overhead when adding further resources. The resource demand is increasing linearly with the load intensity and the length of the steps for one system is therefore independent of the load intensity. For System E, in contrast, the overhead increases when additional resources are used. As a result of this non-linear increasing resource demand, the length of the steps of the mapping function decreases for increasing load intensity.

As illustrated in Figure 5.10, the resource demand variations on Systems C-E are different when they are exposed to the same load profile, although they offer the same amount of resources. Different scaling behaviors and different levels of performance of the underlying resources cause this difference. As a basis for the transformation of load profiles, one system is selected as baseline to serve as reference for demand changes. The induced resource demands on the compared system are the same as for the baseline system since the respective transformed load profiles are used.

(a) System C **(b)** System D **(c)** System D

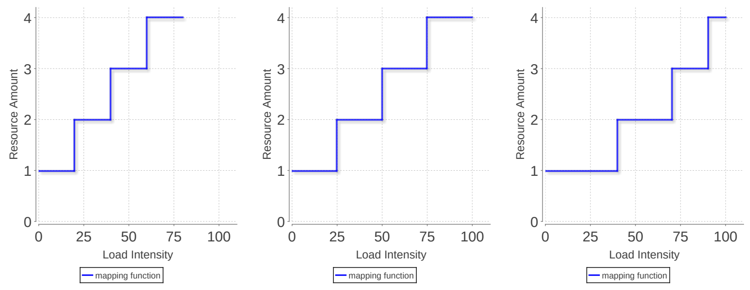


Figure 5.9: Different mapping functions

(a) System C **(b)** System D **(c)** System E

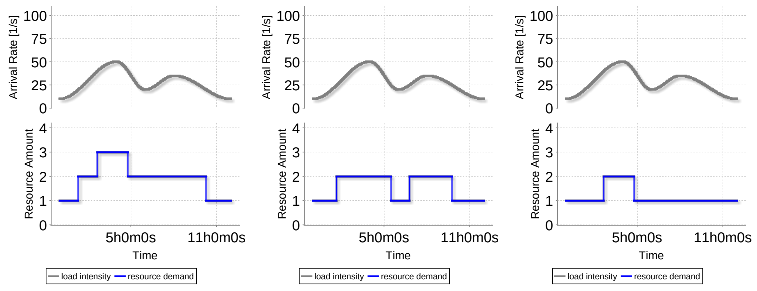


Figure 5.10: Resource demand for the same load profile

(a) System C **(b)** System D **(c)** System E

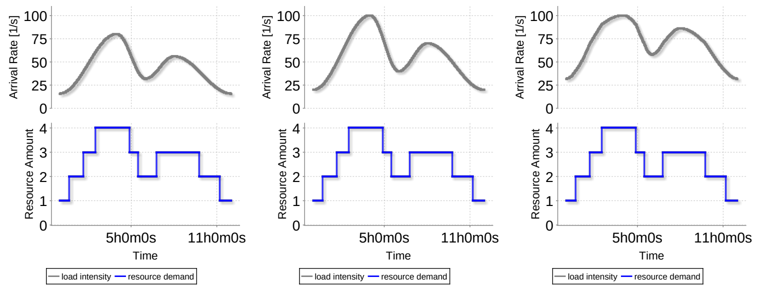


Figure 5.11: Resource demands for system specific adjusted load profiles

The resource demand of the baseline system is assumed to increase linearly with load intensity. Thus, the steps of the mapping function are equal in length. Using this assumption, the mapping function $d_{base}(i)$ of the baseline system can be characterized by two parameters: The number of steps n_{base}, which is equal to the assumed amount of available resources, and the maximum load intensity $max_{base}(i)$ that the base system can sustain using all resources. The first parameter is chosen as the maximum amount of resources that all system support. The second parameter should be greater than or equal to the maximum load intensity that occurs in the load profile. Having defined the mapping function $d_{base}(i)$, for every benchmarked system k, the load profile adjustment step finds an adjustment function $a_k(i)$ such that the following applies:

$$\begin{aligned} &\forall i \in [0, max_{base}(i)] : \\ &d_{base}(i) = d_k(a_k(i)) \end{aligned}$$

The adjustment function $a_k(i)$ maps the steps from the $d_{base}(i)$ function to steps of the $d_k(i)$ function. The result is a piecewise linear function, whereby every linear section represents the mapping of one step in the $d_{base}(i)$ to the same step in the $d_k(i)$ function. The parameters m_j and b_j that define the linear function $y_{k_j}(x) = m_j * x + b_j$ mapping the intensity values belonging to the j^{th} step of the $d_{base}(i)$ function to the j^{th} step of the $d_k(i)$ function can be calculated as follows:

$$\begin{aligned} s_{k_j} &= \max\{i|d_k(i) < j\} \text{ as start intensity,} \\ e_{k_i} &= \max\{i|d_k(i) = j\} \text{ as end intensity and} \\ s_{k_j} &= e_{k_j} - s_{k_j} \text{ as step length} \\ m_{k_j} &= \frac{s_{k_j}}{\frac{max_{base}(i)}{n_{base}}} \\ b_{k_j} &= s_{k_j} - s_{k_j} \cdot (j-1) \end{aligned}$$

and thus the function $a_k(i)$ can be expressed as:

$$a_k(x) = \begin{cases} y_{k_1}(x) & \text{when } s_{k_1} < x \le e_{k_1} \\ \vdots & \\ y_{k_n}(x) & \text{when } s_{k_n} < x \le e_{k_n} \end{cases} \tag{5.19}$$

Having calculated the adjustment function $a_k(i)$, this function can be applied to the original load profile $i(t) = p(t)$ in order to retrieve a system specific adjusted load profile $p_k(t)$. This adjusted load profile can then be used in the actual benchmark run.

$$lp_k(t) = a(p(t))$$

Figure 5.11 shows the induced load demand for Systems C, D and E using the adjusted load profiles. Although the systems have underlying resources with different levels of performance and different scaling behaviors, the induced resource demand variations are now equal for all compared systems.

5.5.3 Measurement and Metric Calculation

After having conducted the System Analysis and Calibration steps, Bungee executes the measurements with a reasonable warm-up at the beginning. Afterwards, the measurement run is validated using the sanity check explained at the end of Section 5.5.1. For the computation of the accuracy, provisioning timeshare and jitter metrics, the resource demand and supply changes are required as input: The demand changes can be directly derived from the adapted load profile using the system specific mapping function. The supply changes need to be extracted either from cloud management logs or by polling during the measurement. In both cases, a resource is defined to be supplied during the time spans when it is ready to process or actively processing requests. The time spans a resource is billed may differ significantly.

Resource Supply

The resource supply is the amount of resources which is available in a usable state. Unfortunately, the resource supply is not necessarily the resource amount that a cloud provider bills its customer. There are two reasons for the discrepancy. The main reason is that cloud providers usually charge based on time intervals which are coarse grained compared to the provisioning time. For example, for a resource that is used for ten minutes only a provider may bills the same costs as for a resource used for one hour. The second reason is that a resource may not be ready to use immediately after the allocation. This is especially true for container resources like VMs. After allocation, a VM needs time to boot and start required applications. Cloud providers may even bill the VM when its creation was scheduled, although the customer cannot use it at this point. This approach does not evaluate the business model of cloud providers. Therefore, the benchmark compares the amount usable resources, the resource supply, with the amount of necessary resources, the resource demand, independently of what the customer is billed for. The benchmark user should be aware of this fact when implementing the resource supply monitoring.

For the presented benchmark, the amount of allocated or the resource supply refers to the amount of resource that are currently available for use.

Elastic cloud systems are usually managed by a cloud computing software which also provides monitoring tools. These monitoring tools can be used to retrieve the resource supply. Hereby can, depending on the capabilities of the monitoring tools, two different monitoring techniques be applied: Active Monitoring based on polling and passive monitoring based on log parsing.

Active Monitoring - Polling the State This method polls the information about the resource supply periodically. The granularity of this method depends on the frequency of the polling requests. The disadvantage of this method is the creation of unnecessary network traffic and CPU load on the benchmark and on the management node.

Passive Monitoring - Log Parsing If the monitoring tool offers access to logs which contain information about when resources have been de-/allocated, this log can be parsed to obtain the desired information. Log parsing usually only offers information about relative

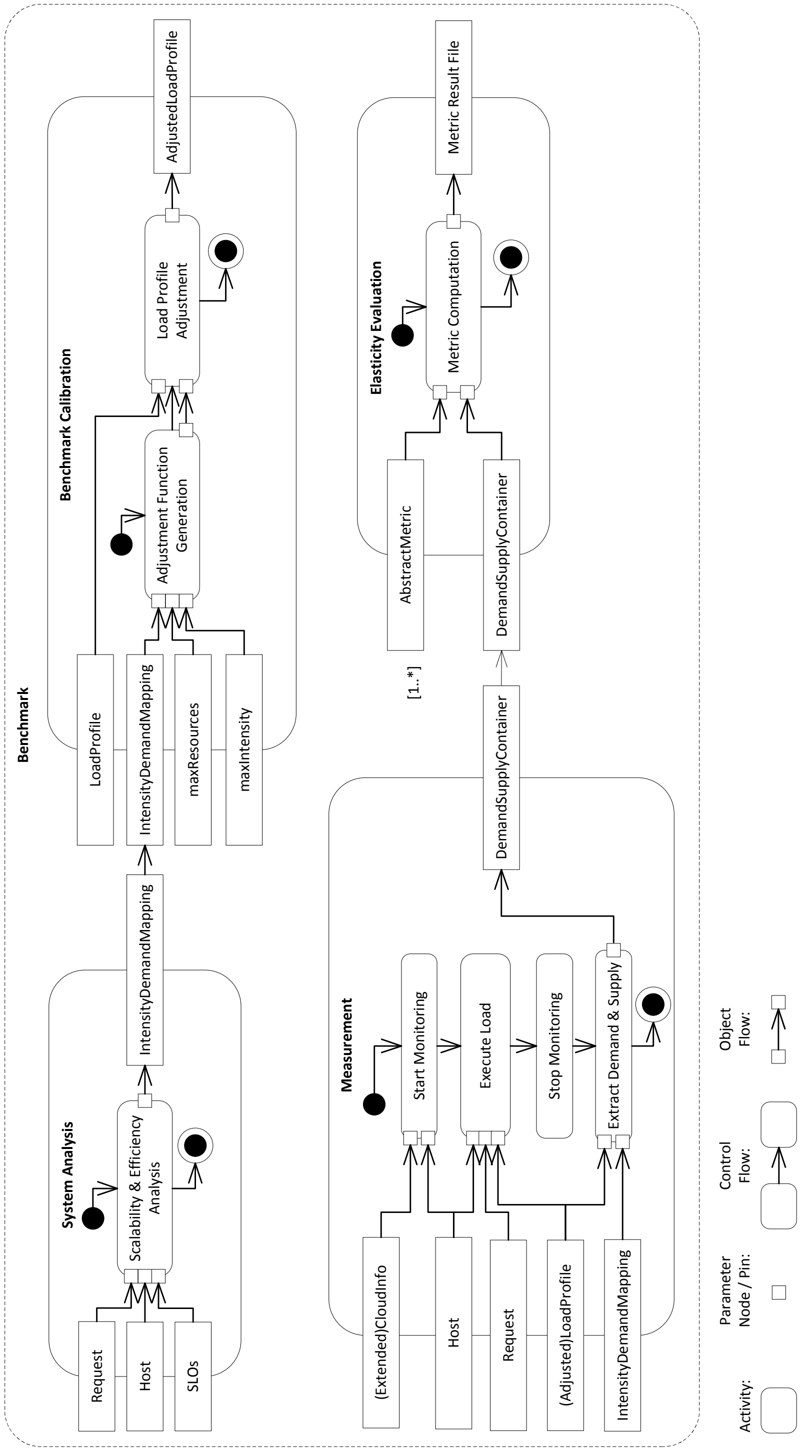


Figure 5.12: Activity diagram of the Bungee experiment workflow.

changes of the resource supply not about its absolute level. Thus, it is necessary to poll the absolute number of used resources additionally, once. Using this information about the absolute level of the resource supply, the information from the parsed log can be used to compute the resource supply for other points in time. If this method is feasible, it should be preferred over the active monitoring, since it requires less overhead.

The activity diagram in Figure 5.12 covers in detail all steps of the measurement process as discussed in the previous sections.

5.5.4 Limitations of Scope

Our approach adopts a technical perspective on resource elasticity. Therefore, the developed benchmark targets researchers, cloud providers and customers interested in comparing elastic systems from a technical, not a business value perspective. As a result, this approach does not take into account the business model of a provider or the concrete financial implications of choosing between different cloud providers, elasticity strategies or strategy configurations.

Since this approach evaluates resource elasticity from a technical perspective, a strict black-box view of the SUT is not sufficient. The evaluation bases on comparing the induced resource demand with the actual amount of used resources. To monitor the latter, access to the SUT is required. Furthermore, the calibration requires to manually scale the amount of allocated resources. Since cloud providers usually offer APIs that allow resource monitoring and manual resource scaling, this limitation does not restrict the applicability of the benchmark.

Cloud services offer their customers different abstraction layers. These layers are commonly referred to as IaaS, PaaS and SaaS. The measurement methodology is mainly targeted at systems that provide compute infrastructure like IaaS providers. In the SaaS context resources are not visible to the user. The user pays per usage quota instead of paying for allocated resources. SaaS platforms are therefore not within scope of this work. Although this approach is not explicitly targeted at PaaS, the approached benchmark should also be applicable in the PaaS context as long as the underlying resources are transparent.

The workloads used for this approach are realistic with respect to load intensity. They are modeled as open workloads with uniform requests. The work units are designed to specifically stress the scaled resources. Workloads that use a mixture of work unit sizes, stress a several (scalable) resources types at the same time or workloads modeled as closed workloads remain future work. The application that uses the resources is assumed to be stateless. Thus, a requests always consumes the same amount of resources. Furthermore, selecting appropriate load profiles is not the scope of this research. However, we demonstrate how complex realistic load profiles can be modeled and adjusted in a platform specific manner in order to allow fair comparisons of platforms with different levels of performance of underlying resources and different scaling behaviors.

The range of different resources that a resource elastic platform can scale is broad. As a first step, we focus on processing resources such as CPU but can also be applied to other physical resources. The evaluation will showcase a simple IaaS scenario where VMs are bound to processing resources. Thus, scaling the virtual machines corresponds to scaling the processing resources. In early works, we have analyses the elasticity of virtual compute resources on a higher level of abstraction, like thread pools [KHvKR11]. Elastic platforms can scale resources horizontally, vertically or even combine both methods to match the resource demand. For this thesis, we focus on comparing platforms that scale homogeneous resource containers like VMs in one dimension either vertical or horizontal.

5.6 Concluding Remarks

In this chapter, we answer RQ A.3 („What are meaningful intuitive metrics to quantify accuracy, timing and stability as quality aspects of elastic resource adaptations?“) by presenting a set of metrics for capturing the accuracy, timing and stability aspects of elastic systems. We provide metric aggregation methods and showed how they can be adapted to personal preferences. Next, we address RQ A.4 („How can the proposed elasticity metrics be measured in a reliable and repeatable way to enable fair comparisons and consistent rankings across systems with different performance?“) by defining a novel elasticity benchmarking methodology that is implemented as part of the Bungee benchmarking framework. In the evaluation in Chapter 9, we show that the proposed metrics and benchmarking methodology are able to rank elastic platforms and configurations on an ordinal scale according to their exhibited degree of elasticity for a given domain-specific and realistic load profile.

Part III

Methods for Reliable Auto-Scaling

Chapter 6

Integrating Proactive and Reactive Deployment Decisions: A Reliable Hybrid Auto-Scaling Mechanism

Auto-scalers for clouds promise stable service quality at low costs when facing a changing workload intensity. For example, Amazon Web Services (AWS), with their Elastic Compute Cloud (EC2), provide a trigger-based auto-scaler with configurable thresholds. However, trigger-based auto-scaling has a long reaction time, which lays in the order of minutes, depending on the boot-up time of the scaled resources and the deployed software. Novel auto-scalers proposed in literature try to overcome the limitations of reactive mechanisms by employing proactive prediction methods. However, the adoption of proactive auto-scalers in production is still very low due to the high risk of relying on a single proactive method based on which scaling decisions are made.

This chapter tackles Goal B of reducing this risk by proposing a new hybrid auto-scaling mechanism, called Chameleon, combining multiple different proactive methods coupled with a reactive fallback mechanism. Chameleon employs on-demand, automated time series-based forecasting methods to predict the arriving load intensity in combination with run-time resource demand estimation techniques to calculate the required resource consumption per work unit without the need for application instrumentation. It can also leverage application knowledge by solving product-form queueing networks used to derive optimized scaling actions. Chameleon is first in resolving conflicts between reactive and proactive scaling events in an intelligent way improving both quality and reliability of scaling actions.

In the evaluation in Chapter 10, we benchmark Chameleon against four different state-of-the-art proactive auto-scalers and a threshold-based one in three different cloud environments: (i) a private CloudStack-based cloud, (ii) the public AWS EC2 cloud, as well as (iii) an OpenNebula-based shared IaaS cloud. We generate five different representative workload each taken from different real-world system traces. The variable workload profiles extracted from the five different traces drive a CPU-intensive benchmark worklet. Overall, Chameleon achieves the best scaling behavior based on user and elasticity performance metrics, analyzing the results from 400 hours aggregated experiment time.

6.1 Introduction

Over the past decade, the cloud computing paradigm gained significant importance in the ICT domain as it addresses manageability and efficiency of modern Internet and computing services at scale. Cloud computing provides on-demand access to data center resources (e.g., networks, servers, storage and applications). Infrastructure-as-a-Service (IaaS) cloud providers promise stable service quality by leveraging trigger-based auto-scaling mechanisms to deal with variable workloads. However, depending on the type of resources and deployed software stack, scaling actions may take several minutes to be effective. In practice, business-critical applications in clouds are usually deployed with over-provisioned resources to avoid becoming dependent on an auto-scaling mechanism with its possibly wrong or badly-timed scaling decisions.

Sophisticated, state-of-the-art auto-scaling mechanisms from the research community focus on proactive scaling, aiming to predict and provision required resources in advance of when they are needed. An extensive survey [LBMAL14] groups auto-scalers into five classes according to the prediction mechanisms they use: (i) threshold-based rules, (ii) queueing theory, (iii) control theory, (iv) reinforcement learning, and (v) time series analysis. With few exceptions, like at Netflix[1] , proactive auto-scalers are not yet broadly used in production. This might be a result of the need for application-specific fine-tuning, and the lack of knowledge about the performance of an auto-scaler in different contexts. Auto-scalers employed in production systems are responsible to dynamically trade-off user-experienced performance and costs in an autonomic manner. Thus, they carry a high operational risk.

For this contribution, we pose ourselves the following research questions: RQ B.1 „How can conflicting auto-scaling decisions from independent reactive and proactive decision loops be combined to improve the overall quality of auto-scaling decisions while increasing reliability?." And concerning the evaluation: RQ B.2 „How well does the proposed Chameleon approach perform compared to state-of-the-art auto-scaling mechanisms in realistic deployment and application scenarios?"

In this chapter, we propose a new hybrid auto-scaling mechanism called Chameleon combining multiple different proactive methods coupled with a reactive fall-back. Chameleon reconfigures the deployment of an application in a way that the supply of resources matches the current and estimated future demand for resources as closely as possible according to the definition of elasticity [HKWG15a]. It consists of two integrated controllers: (i) a reactive rule-based controller taking as input the current request arrival rate and resource demands estimated in an online fashion, and (ii) a proactive controller that integrates three major building blocks: (a) an automated, dynamic on-demand forecast execution based on an ensemble of the sARIMA[2] [BJ$^+$15] and tBATS[3] [LHS11] stochastic time series modeling frameworks, (b) an optional descriptive software performance model as an instance of the Descartes Modeling Language (DML) [KHBZ16] enabling the controller to leverage structural application knowledge by transformation into product-form queueing networks, and (c) the LibReDE resource demand estimation library [SCBK15] for accurate service-time estimations at run-time without the requirement of detailed application instrumentation. As

[1]Scryer: Predictive auto-scaling at Netflix

[2]sARIMA: seasonal, auto-regressive, integrated moving averages [BJ$^+$15]

[3]tBATS: trigonometric, Box-Cox transformed, ARMA errors using trend and seasonal components [LHS11]

part of Chameleon, we resolve the internal conflicts between reactive and proactive scaling events in a way that increases both reliability and quality of scaling actions. Chameleon is available as open-source project[4] together with experiment data presented in this thesis.

In the evaluation in Chapter 10, we benchmark Chameleon against four different proactive auto-scaling mechanisms covering the mentioned domains, as well as one commonly used reactive auto-scaler based on average CPU utilization thresholds and finally a scenario without the use of an active auto-scaler. We conduct seven sets of extensive and realistic experiments of up to 9.6 hours duration in three different infrastructure cloud environments: (i) in a private CloudStack-based cloud environment, (ii) in the public AWS EC2 IaaS cloud, as well as (iii) in the OpenNebula-based IaaS cloud of the Distributed ASCI Supercomputer 4 (DAS-4) [BEdLm16]. We generate five different representative workload profiles each taken from different real-world system traces: BibSonomy, Wikipedia, Retail Rocket, IBM mainframe transactions and FIFA World Cup 1998. The workload profiles drive a CPU-intensive worklet as benchmark application that is comparable to the LU worklet from SPEC's Server Efficiency Rating Tool SERT™2[5]. As elasticity measurement methodology and experiment controller, we employ the elasticity benchmarking framework Bungee [HKWG15a] enabling extensive and repeatable elasticity measurements. The results are analyzed with our set of SPEC endorsed elasticity metrics [HKO+16] (see Section 5.2) in addition to metrics capturing the user-perspective. For each experiment set, the metric results are used to conduct three competitions: (i) a rating based on the deviation from the theoretically optimal auto-scaling behavior, (ii) a pair-wise competition, and (iii) a score-based ranking of metric speedups aggregated by an unweighted geometric mean. The results of the individual metrics and the three competitions show that Chameleon manages best to match the demand for resources over time in comparison to the other proactive auto-scalers.

We summarize the highlights of this contribution and the corresponding evaluation in Chapter 10 as follows:

- In Sections 6.2 to 6.4, we address RQ B.1 and present the Chameleon approach that integrates reactive and proactive scaling decisions based on time series forecasts, combined with online resource demand estimates used as input to product-form queueing networks.

- In Chapter 10, we address RQ B.2 and apply the Bungee measurement methodology and elasticity metrics as defined in Chapter 5 in a broad auto-scaler evaluation for compute intensive workloads.

- Section 10.3 contains the results of extensive experiments that demonstrate the superior auto-scaling capabilities of Chameleon compared to a set of state-of-the-art auto-scaling mechanisms.

[4]Chameleon - sources and experiment data: `http://descartes.tools/chameleon`
[5]SPEC SERT™2: `https://spec.org/sert2`

6.2 Chameleon Design Overview

The Chameleon mechanism consists of four main components: (i) a controller, (ii) a performance data repository, (iii) a forecast component and (iv) the resource demand estimation component based on LibReDE [SCZK14]. The performance data repository contains a time series storage and an optional descriptive performance model instance of the application to be scaled dynamically in form of the Descartes Modeling Language (DML) [KHBZ16]. The design and the flow of information is depicted in Figure 6.1. The central part of Chameleon is the controller. It communicates with the three remaining components and the managed infrastructure cloud. The functionality of the controller is divided into two parallel sequences: the reactive cycle (red) and the proactive cycle (dashed blue). The two cycles act more the less independently of each other and may also produce conflicting events that are at the end merged based on a conflict resolution heuristic.

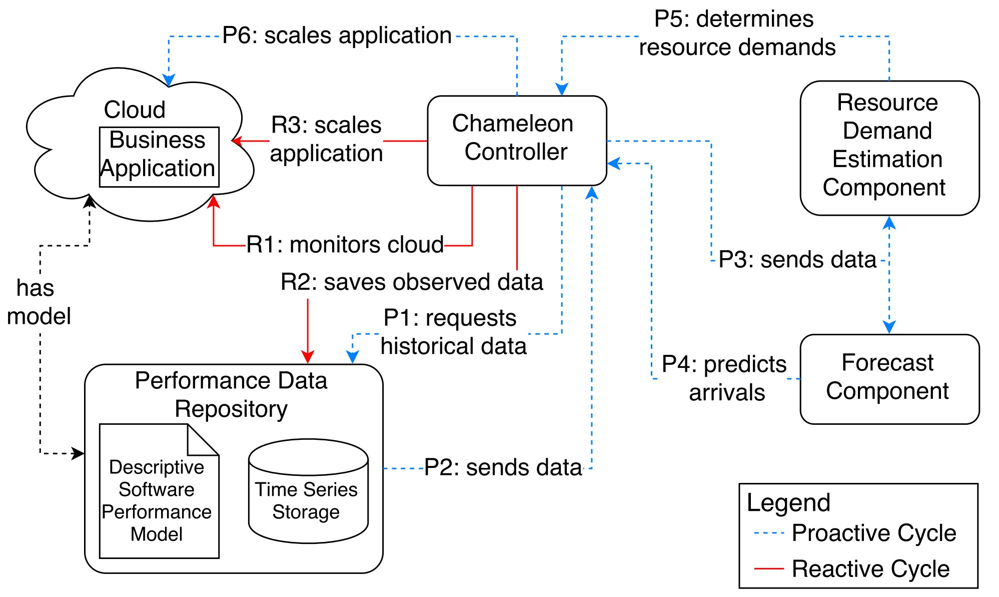


Figure 6.1: Design overview of Chameleon.

During the reactive cycle, the controller has three main tasks: (R1) at first, the controller communicates with the cloud management and periodically polls (e.g., every minute) information on the current state of the application delivered via asynchronous message queues. As part of the application run-time environment, a monitoring agent [SWK16] is deployed to fill the message queues. The collected information includes CPU utilization averages per node and the number of request arrivals. Average residence and response times per request type on a node can be provided by the agents to Chameleon (but are not required to). (R2) Then, the new information is stored in the performance data repository for the current time window. (R3) With this information, the controller decides if the system needs to be scaled, based on the computed average system utilization and a standard threshold-based approach. The average system utilization is derived from the the arrival rate and the estimated resource demand based on the service demand law from queueing theory [Bm06].

The proactive cycle is planned in longer intervals, e.g., 4 minutes, for a set of future scaling intervals. It involves six tasks: (P1) at first, the controller queries the performance data repository for available historical data and checks for updates in the structure of the DML performance model. (P2) Then, the available time series data is sent to the controller. (P3) Afterwards, the time series of request arrival rates is forwarded to the forecast component and data about the CPU utilization and request arrivals per node (plus residence and response times if available) is sent to the resource demand estimation component. (P4) Then, the new available forecast values are sent to the controller. (P5) The LibReDE resource demand estimation component estimates the time a single request needs to be served on the CPU of a node and sends the estimated value to the controller. (P6) Finally, the controller scales the application deployment based on the estimated resource demands, the forecast request arrivals and structural knowledge from the DML descriptive performance model.

After having the flow of the two decoupled cycles described the following two paragraphs now focus on the forecast and resource demand estimation components.

6.2.1 Forecast Component

The forecast component predicts the arrival rates for a configurable number of future reconfiguration intervals. In order to reduce overhead, the forecast component is not called in fixed periods. No new time series forecast is computed if an earlier forecast result still contains predicted values for requested future arrival rates. In case, a drift between the forecast and the recent monitoring data is detected, a new forecast execution is triggered. To detect a drift between monitoring and forecast values, we compare the forecast accuracy with the mean absolute scaled error measure (MASE) [HK06] considering a configurable threshold value, e.g., 0.4. The MASE measure is suitable for almost all situations and the error is based on the in-sample mean absolute error from the random walk forecast. For a multi-step-ahead forecast, the random walk forecast would predict the last value of the history for the entire horizon. Thus, the investigated forecast is better than the random walk forecast if the MASE value is < 1 and worse if the MASE value is > 1. We calculate the MASE values as follows:

$$MASE = \frac{\frac{1}{n}\sum_{i=1}^{n}|e_i|}{\frac{1}{n-1}\cdot\sum_{i=2}^{n}|Y_i - Y_l|} \tag{6.1}$$

Where Y_l is the observation at time l with l being the index of the last observation of the history. Y_i is the observation at time $l+i$. Thus, Y_i is the value of the i-th observation in the forecast horizon. The forecast error at time $l+i$ is calculated as $e_i = Y_i - F_i$ where F_i is the forecast at time $l+i$. For detailed discussion of forecast error measures, please refer to Section 2.3.2.

For the dynamic on-demand forecast executions, we select an ensemble of the following two stochastic time series modeling frameworks as implemented in R forecast package [HK08,Hyn17]: (i) sARIMA seasonal, auto-regressive, integrated moving averages [BJ$^+$15] and (ii) tBATS trigonometric, Box-Cox transformed, ARMA errors using trend and seasonal components [LHS11]. Due to the capability of both methods to capture seasonal patterns as soon as the data contains two full periods (in our auto-scaling context days), they are considered as complex. We consider the more lightweight approaches that can only estimate trend extrapolations (like splines) as insufficient for auto-scaling decisions in the order of minutes and even hours into the near future. We observe that the time overhead for the forecast executions can vary significantly dependent on the data characteristics and that longer running forecast execution tend to have a lower forecast accuracy. For applicability in an auto-scaling context, timely and accurate forecast results with reasonable overhead are required. Thus, we design the forecast component in a way that the two methods are not run in parallel, but the method that is more likely to have the more accurate result is automatically selected before execution. This selection is performed based on a re-implementation of the meta-learning approach for forecast method selection using data characteristics as in [WSMH09]. We are aware that the sARIMA and tBATS methods are not capable to deliver a reliable and efficient time-to-results and the forecast results may lack in accuracy from time to time. Please note that the Telescope approach presented in Chapter 7 has been developed as part of future work of this research approximately one year later for direct integration with the Chameleon approach. The benefit of applying Telescope forecasts instead of the tBATS and sARIMA ensemble is shown separately in Section 11.2.

6.2.2 LibReDE Resource Demand Estimation Component

The LibReDE library offers eight different estimation approaches for service demands on a per request type basis [SCZK14]. Among those eight, there are estimators based on regression, e.g. [KPSCD09], optimization, e.g. [Men08], Kalman filters, e.g. [Wm12] and the service demand law. LibReDE supports to dynamically select an optimal approach via parallel execution and cross-validation of the estimated error. Furthermore, configuration parameters of the estimation approaches can be tuned automatically for a given concrete scenario [GHSK17]. To minimize estimation overheads, the service demand law based estimator is used. As input, the request arrivals per resource and the average monitored utilization are required. Request response and residence times can be provided optionally as they are required by some of the estimators and for an enhanced cross-validation based on utilization and response time errors. For complex service deployments, LibReDE requires structural knowledge about the application deployment to have a defined mapping of what services are deployed on which resources. Chameleon can provide this information from querying a DML performance model instance.

6.3 Decision Management

In the proactive cycle, the controller determines events (i.e., scaling actions) for each forecast. Decisions are created based on the rules in the Decision Logic (see Algorithm 6.1 for a simplified example of proactive decision making). Then, these decisions are improved/optimized and finally, they are added to the *Event Manager*, see Section 6.4. There, the decisions are scheduled according to their target execution time.

ALGORITHM 6.1: Proactive decision logic.

```
Decision Logic at time t in the future
    services = model.getServices() // gets all services
    $\rho_S$ = getArrivalRateForecast() · getAverageResourceDemand(services) // calculates the future system utilization
    $\rho = \rho_S$ / getRunningVMs() // calculates the future average utilization on each VM
    response = calculatesEndtoEndResponseTime($\rho$, services) // calculates the maximum response time of all services
    amount = 0 // the number of VMs for adding or releasing
    if response ≥ up_resp_threshold · slo then // is the response time ≥ a predefined percentage of the slo
        while response ≥ up_resp_threshold · slo or $\rho \geq$ pro_up_util_threshold do
            amount++
            $\rho = \rho_S$ / (getRunningVMs()+amount) // calculates the new average utilization
            response = calculatesEndtoEndResponseTime($\rho$, services)
        end
    end
    else if response ≤ down_resp_threshold · slo then // is the response time ≤ a predefined percentage of the slo
        while response ≤ down_resp_threshold · slo and $\rho \leq$ target_utilization do
            amount−−
            $\rho = \rho_S$ / (getRunningVMs()+amount) // calculates the new average utilization
            response = calculatesEndtoEndResponseTime($\rho$, services)
            if response > up_resp_threshold · slo or $\rho >$ pro_up_util_threshold then
                amount++ // undo as one condition is violated
            end
        end
    end
    decisions.add(amount, t)
end
```

The *Decision Logic* determines for each event the number of instances that need to be removed or added. Algorithm 6.1 shows the step-by-step approach. The associated parameters that a user needs to specify are explained in Table 6.1. In the first line, the algorithm loads the user services. These services form the application's interface to the users. Then, the future system utilization is calculated by multiplying the predicted arrival rate with the average resource demand of all services (line 2). Afterwards, the utilization of a single VM is calculated, the response time of all user services are calculated, and the maximum response time is returned. This is done by adding the residence times of each called service. Hereby, we model each service as an M/M/1/∞ queue [Bm06]. The residence time of each internal service is computed with the formula: residence time $R = \frac{S}{100-\rho}$ where

S is the resource demand and ρ the system utilization. The calculated response time can lie in one of the following intervals:

- (i) $[0, down\_resp\_threshold \cdot slo]$,
- (ii) $(down\_resp\_threshold \cdot slo, up\_resp\_threshold \cdot slo)$ or
- (iii) $[up\_resp\_threshold \cdot slo, \infty)$.

Parameter	**Explanation**
$up\_resp\_threshold$	The upper response time threshold related to the SLO.
$pro\_up\_util\_threshold$	The system utilization threshold for upscaling.
$down\_resp\_threshold$	The lower response time threshold related to the SLO.
$target\_utilization$	The target system utilization for downscaling.
slo	The acceptable resp. time as service level objective (SLO).

Table 6.1: Parameters of the decision logic.

If the response time lies in the second interval, the calculated response time has an acceptable value. The algorithm skips the if-else block and records a NOP (as the amount is zero) as a decision (line 20). If the response time lies within the last interval, i.e., the response time is greater than $up\_resp\_threshold \cdot slo$, the number of VMs are iteratively increased until the new response time drops below $up\_resp\_threshold \cdot slo$ and the new average VM utilization ρ is less than $pro\_up\_util\_threshold$ (line 7-10). The new average VM utilization ρ is calculated by dividing the system utilization ρ_S by the new amount of VMs and the response time is computed based on this utilization. Finally, the algorithm records a new decision with the number of the additional VMs required for this service (line 20). If the response time is less than $down\_resp\_threshold \cdot slo$, the number of VMs are iteratively decreased until the new response time is greater than $down\_resp\_threshold \cdot slo$ or the average VM utilization ρ is greater than $target\_utilization$ (line 14-17). The recalculation of the response time is analogous to line 7 - 10. In the end, a new decision is recorded with the amount of VMs that can be released (line 20).

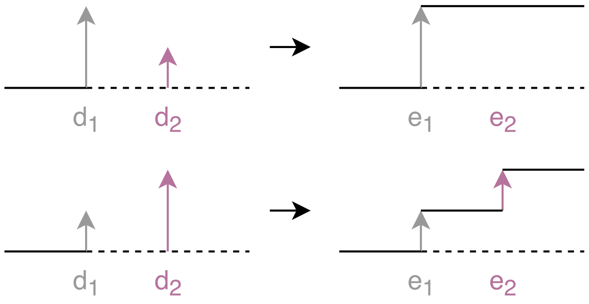
During an interval of the proactive cycle (we use 4 minutes), two proactive decisions are made based on the logic of Algorithm 6.1. The time between the two decisions and the respective scheduled events are equidistant (we use 2 minutes), see Figure 6.3. As proactive decisions are calculated based on the current observations and predictions, some rules are needed to adjust/improve these decisions before adding them as events to the *Event Manager* (see Section 6.4). Note that we improve the pair of events in each interval of the proactive cycle (and not more) as a trade-off between decision stability and reactivity of the approach. Basically, there are three possibilities when combining two decisions. (i) Both decisions want to scale up the system, (ii) want to scale down the system, (iii) or they have contrary scaling decisions. If one of the decisions is a NOP, no combination is required. Hence, six different cases can be distinguished as depicted in Figures 6.2a - 6.2c. The black line represents the current supply, the dashed black line the reference supply, the grey arrow the first decision (d_1)/event (e_1) and the purple one the second decision (d_2)/event (e_2).

The first possibility (both decisions plan to scale up) and the two resulting cases are shown in Figure 6.2a. In the upper case, the first decision wants to scale up n VMs and the second one wants to scale up m instances, where $n \geq m$. The resulting first event allocates n extra VMs. The second event triggers the allocation of 0 new VMs ($\neq$ NOP). In the second case $m > n$ and so, the first event scales up n VMs and second one allocates $m - n$ VMs.

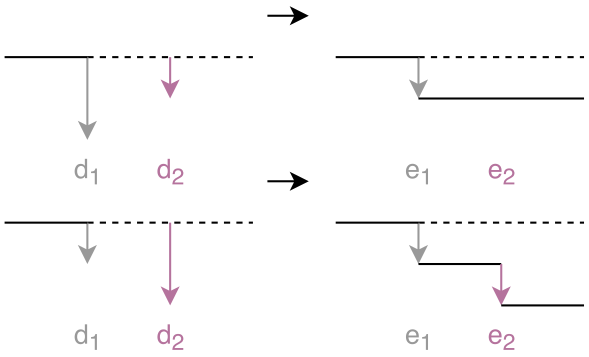
The second case (both decisions want to scale down the system) and the two resulting sub-cases are shown in Figure 6.2b. The upper case shows that the first decision wants to scale down n VMs and the second one wants to scale down m instances, where $n \geq m$. As the down-scaling policy of Chameleon is conservative, the first event releases m VMs and the second one triggers the releasing of 0 VMs ($\neq$ NOP). In the second case, $m > n$ and thus, the first event scales n VMs down and the second one releases $m - n$ VMs.

The last option is that the decisions request opposite scaling actions. The two resulting cases are shown in Figure 6.2c. In the upper case, the first decision wants to release n VMs and the second one wants to allocate m VMs with $n \geq m$ or $m > n$. To handle contrary decisions, Chameleon uses a shock absorption factor $0 < \xi \leq 1$. Thus, the first event scales $\lfloor(\xi \cdot d_1)\rfloor$ VMs down and the second one scales $\lceil(\xi \cdot (d_1 + d_2))\rceil$ VMs up. The second case is complementary to the first case. Hence, the first event allocates $\lceil(\xi \cdot d_1)\rceil$ VMs and the second one releases $\lfloor(\xi \cdot (d_1 + d_2))\rfloor$ VMs. If $\xi = 1$, then 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 the oscillation in a proactive interval. A mathematical formula of the complete combination rules can be written as follows:

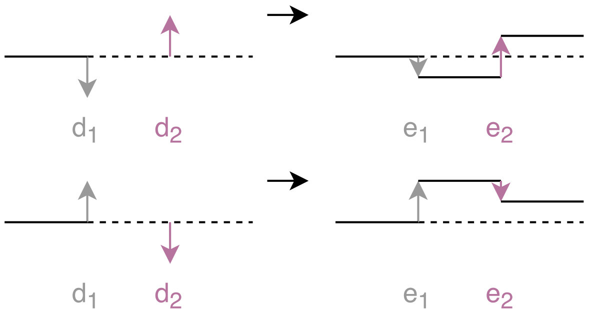
$$\begin{gathered}\text{Let } e_i \text{ be the i-th event, } d_i \text{ be the i-th decision and } \xi \\ \text{the shock absorption with } 0 < \xi \leq 1,\ \mathrm{i} \in \{1,2\}: \\ e_1 = \begin{cases} d_1 & (d_1, d_2) \in \mathrm{UP}^2 \\ min(d_1, d_2) & (d_1, d_2) \in \mathrm{DOWN}^2 \\ \lceil(\xi \cdot d_1)\rceil, & (d_1, d_2) \in \mathrm{UPxDOWN} \\ \lfloor(\xi \cdot d_1)\rfloor, & (d_1, d_2) \in \mathrm{DOWNxUP} \end{cases} \\ e_2 = \begin{cases} max(d_2 - d_1, 0) & (d_1, d_2) \in \{\mathrm{UP}^2\ , \mathrm{DOWN}^2\} \\ \lfloor(\xi \cdot (d_1 + d_2))\rfloor, & (d_1, d_2) \in \mathrm{UPxDOWN} \\ \lceil(\xi \cdot (d_1 + d_2))\rceil, & (d_1, d_2) \in \mathrm{DOWNxUP} \end{cases}\end{gathered} \tag{6.2}$$



(a) Combination Rule I.



(b) Combination Rule II.



(c) Combination Rule III.

Figure 6.2: Combination rules for sets of two planned scaling events.

6.4 Event Manager

As Chameleon consists of a proactive and reactive cycle, the management of events created by the two cycles is required. The event manager of Chameleon has to accept events, resolve conflicts, and schedule events. An event carries information on its type, either proactive or reactive, the amount of VMs to allocate or release, its trustworthiness, and its planned execution time. In contrast to reactive events that are always considered as trustworthy, a proactive event is trustworthy only when the MASE (mean absolute scaled error) [HK06] of the associated forecast is lower than a tolerance value, see Section 6.2. A reactive event should be executed immediately, whereas a proactive event has an execution time in the future. As the proactive and reactive cycle have different interval lengths, their respective events may have different execution times. An overview of how events are planned and scheduled is shown in Figure 6.3. In order to handle all the constrains, the manager has to resolve the following conflicts:

1. SCOPE CONFLICT: Each proactive event has an associated scope, which is a time interval before the event execution in which no other event should occur. That is, the scope has a fixed length (based on the equidistant time intervals between two events) and ends when the associated event is executed. As proactive events are scheduled in longer intervals, reactive events can be triggered during the scope of a proactive event. This leads to a scope conflict with two cases: (i) If the proactive event is trustworthy and the associated action is UP or DOWN, then the reactive events are skipped. Figure 6.3 shows an example of this case. In the scope of the proactive event $p_{5,0}$ for instance, two reactive events r_{10} and r_{11} are triggered. As $p_{5,0}$ is trustworthy and its action is not a NOP, $p_{5,0}$ is scheduled and both reactive cycle events are skipped. (ii) If the proactive event is not trustworthy or contains the action NOP, then skip the proactive event and execute the reactive one. In Figure 6.3, the proactive event $p_{1,0}$ for instance is not trustworthy and hence, it is ignored. That is, the reactive events r_2 and r_3 that are triggered during the scope of $p_{1,0}$ are executed without modification.

2. TIME CONFLICT: Each event can be identified by its execution time. The time conflict describes the problem when two proactive events with the same execution time appear. This conflict occurs since Chameleon plans proactive events throughout the forecast horizon, however, a new forecast is executed as soon as a drift between the forecast and the monitored load is detected. In such a situation, for some intervals there may be proactive events based on the old forecast and respective events based on the newly conducted forecast. Given that the proactive events based on the new forecast have more recent information, the proactive events based on the older forecast are simply skipped. In Figure 6.3, the values of forecast f_1 have a deviation from the measured values greater than the tolerance limit. Therefore, the forecast f_2 is executed although the forecast horizon of f_1 is still active. In this situation, the proactive events $p_{4,1}$ and $p_{2,2}$ are scheduled at the same time. As $p_{2,2}$ has more recent information, e.g., the current resource demand, $p_{2,2}$ is executed and $p_{4,1}$ is skipped accordingly.

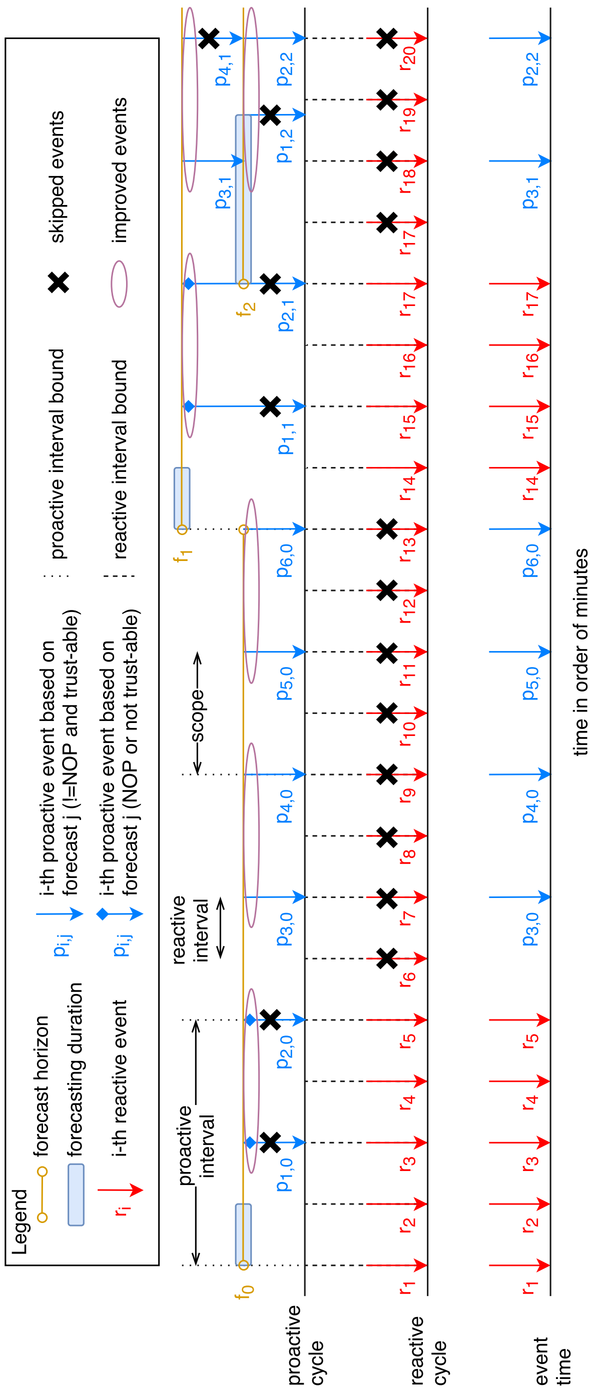


Figure 6.3: Exemplary illustration of Chameleon's working mode.

3. DELAY CONFLICT: The execution time of the forecast component ranges between seconds and minutes. Thus, some forecasts may take longer so that the creation of proactive events is delayed, i.e., the event can not take place in the equidistant time interval. In order to prevent such delays, the proactive event of the previous forecast is used for this execution time. An example of this conflict is depicted in Figure 6.3. Here, through the long computing time, illustrated by the blue box of forecast f_2, the proactive event $p_{1,2}$ is delayed. Thus, the proactive event $p_{3,1}$ of forecast f_1 is executed and $p_{1,2}$ is ignored.

6.5 Assumptions and Limitations

We make the following assumptions explicit: (i) in order to obtain sARIMA or tBATS forecasts with a model of the seasonal pattern, the availability of two days of historical data is required. With less historical data, the forecasts cover only trend and noise components resulting in a decreased accuracy and fewer proactive scaling decisions. (ii) The monitoring values of request arrivals per resource of the application are accurately provided by the monitoring infrastructure, e.g., by polling from a load-balancer or from an instrumented run-time middleware. Chameleon does not rely on utilization measurements. (iii) The service level objective at the application level, which is monitored, is based on the response time of the application. (iv) Chameleon is currently focused on scaling CPU intensive applications as it relies on the respective resource demand estimation models that perform best for CPU-bound workloads. (v) The optional DML descriptive performance model instance can be transformed to a product-form queueing network, whereby each service is modelled as an $M/M/1/\infty$ queue. (vi) As a possible limitation to usability for the Chameleon approach in comparison to existing simplistic reactive auto-scaling mechanisms, we are aware of a setup effort that comes in connection with the forecast and resource demand estimation components. However, Chameleon is designed in a way that it would also work with replaced forecast mechanisms and resource demand estimation components as long as they work compliant to the defined required interfaces. The overhead of running Chameleon is optimized and comparable to the other proactive policies considered in the evaluation. Especially, the compute intense forecast executions are only triggered when the forecast arrivals drift away from the monitored ones.

6.6 Concluding Remarks

In this chapter, we tackle the main Goal B („Reduce the risk of using novel auto-scalers in operation by leveraging multiple different proactive mechanisms applied in combination with a conventional reactive mechanism.“). As part of the hybrid, proactive auto-scaler Chameleon, we answer RQ B.1(„How can conflicting auto-scaling decisions from independent reactive and proactive decision loops be combined to improve the overall quality of auto-scaling decisions?“) by defining intelligent conflict resolution rules for scaling events that are derived from independent reactive and proactive mechanisms. In its proactive part, Chameleon combines forecasting (time series analysis) and resource demand estimation (queueing theory), which can optionally be enriched by application knowledge captured using a descriptive software performance model to increase the timeliness and accuracy of the auto-scaling reconfigurations. The forecast and the resource demand estimation are realized by integrating established open-source tools provided by the research community.

Chapter 7

Forecasting Complex Seasonal Time-Series: A Hybrid Approach based on Decomposition

Forecasting is an important part of the decision-making process and used in many fields like business, economics, finance, science, and engineering. However, according to the No-Free-Lunch Theorem [WM97] from 1997, there is no general forecasting method that performs best for all time series. Instead, expert knowledge is needed to decide which forecasting method to choose for a specific time series with its own characteristics. Expert knowledge is useful, but a time-consuming task that cannot be fully automated. Thus, forecasting methods that deliver stable forecasts for a particular type of time series are desired.

In the field of auto-scaling, time series of request arrival rates typically show seasonal patterns with a high frequency and so, many observations within single periods. Thus, in order to forecast such time series, multi-step-ahead forecasting with a long horizon is required. Yet, most state-of-the-art forecasting methods cannot handle time series with high frequencies when forecasting several hundreds of values.

We start into this research motivated by RQ B.3 („How can a hybrid forecast approach based on decomposition be designed to be capable of providing accurate and fast multi-step-ahead forecasts of complex seasonal time-series within seconds?") and RQ B.4 („Is such a hybrid forecast approach capable to improve the performance and reliability of auto-scaling mechanisms?").

We develop a novel hybrid, multi-step-ahead forecasting approach called Telescope for seasonal, univariate time series based on time series decomposition is outlined in this chapter as a side-contribution of this thesis. Telescope is a self-aware forecasting method incorporating multiple individual forecasting methods. Therefore, the time series is decomposed into the components trend, season, and remainder. However, first of all, the frequency is estimated, anomalies are detected and removed, and the type of decomposition is estimated. After the decomposition, ARIMA without seasonality is applied on the trend pattern, whereas the seasonality is simply continued. Moreover, the single periods are clustered into two clusters in order to learn categorical information. The cluster labels are forecast by applying artificial neural networks. Lastly, the eXtreme Gradient Boosting (XGBoost) is used to learn the dependency between them and to combine the forecasts of the components.

The preliminary evaluation in Chapter 11 show based on two different traces that an early implementation of the Telescope approach can keep up with or beats the best competitor in terms of accuracy. It achieves to improve the time-to-result up to a factor of 19 compared to the three most competitive forecasting methods. In a case study, we answer RQ B.4 by demonstrating that Telescope improves auto-scaling performance further compared to the state-of-the-art method tBATS.

7.1 Telescope Approach

Forecasting allows to predict the future by examining past observations. Classical forecasting methods have their benefits and drawbacks depending on the specific use cases. Thus, there is no globally best forecasting technique [WM97] and further respectively expert knowledge is required for determining the best forecast method. Typically, expert knowledge is needed for two domains, i.e., method selection and feature engineering. The serious problem of expert knowledge is that it can take a long time to deliver results and it cannot be completely automated. In the field of feature engineering, expert knowledge can be replaced by using deep learning [NKK+11, LBH15] or random forests [EHPG+17, CGRGG+17]. To overcome the need of expert knowledge in method selection, a more robust forecasting method compared to the classical forecaster is needed. In this field, robust means that the variance in forecasting results should be reduced, not necessarily improving the forecasting accuracy itself. By reducing the variance of the results, the risk when trusting a bad forecast is lowered. Hybrid forecasting is such a technique since the benefits of multiple forecasting methods can be combined to improve the overall performance. Thus, we introduce a new hybrid, multi-step-ahead forecasting approach for univariate time series. The approach is based on time series decomposition and makes use of existing forecasting methods, i.e., ARIMA, ANN, and XGBoost.

We call the proposed hybrid forecasting approach Telescope according to the analogy with the vision on far-distanced objects. Telescope is developed in R to perform multi-step-ahead forecasting while maintaining a short runtime. To this end, only fast and efficient single forecasting methods are used as components of Telescope. A diagram of the forecasting procedure is shown in Figure 7.1. First, a preprocessing step is executed. The frequency of the time series is estimated using periodograms, i.e., applying the R function `spec.pgram`. This function uses fast Fourier transformation to estimate the spectral density. The estimated frequency is needed to remove anomalies in the time series by applying the `Anomaly Detection` R package [HVK17]. This package uses a modified version of the seasonal and trend decomposition using Loess (STL) [CCMT90]. The `STL` decomposition splits the time series into the three components season, trend, and remainder. After the decomposition, `Anomaly Detection` applies generalized extreme studentized deviate test (ESD) with median instead of mean and median absolute deviation instead of standard deviation on the remainder to identify outliers. Furthermore, we use `STL` for an additive decomposition of the revised time series. If the amplitude of the seasonal pattern increases as the trend increases and vice versa, we assume multiplicative decomposition. Thus, a heuristic testing for such a behavior is implemented. If a multiplicative decomposition is detected, the logarithmised time series is used for the `STL` decomposition and the components are back-transformed after the decomposition. We apply the `STL` package because of its short runtime compared to other R decomposition functions like `bfast`. Afterwards, the season

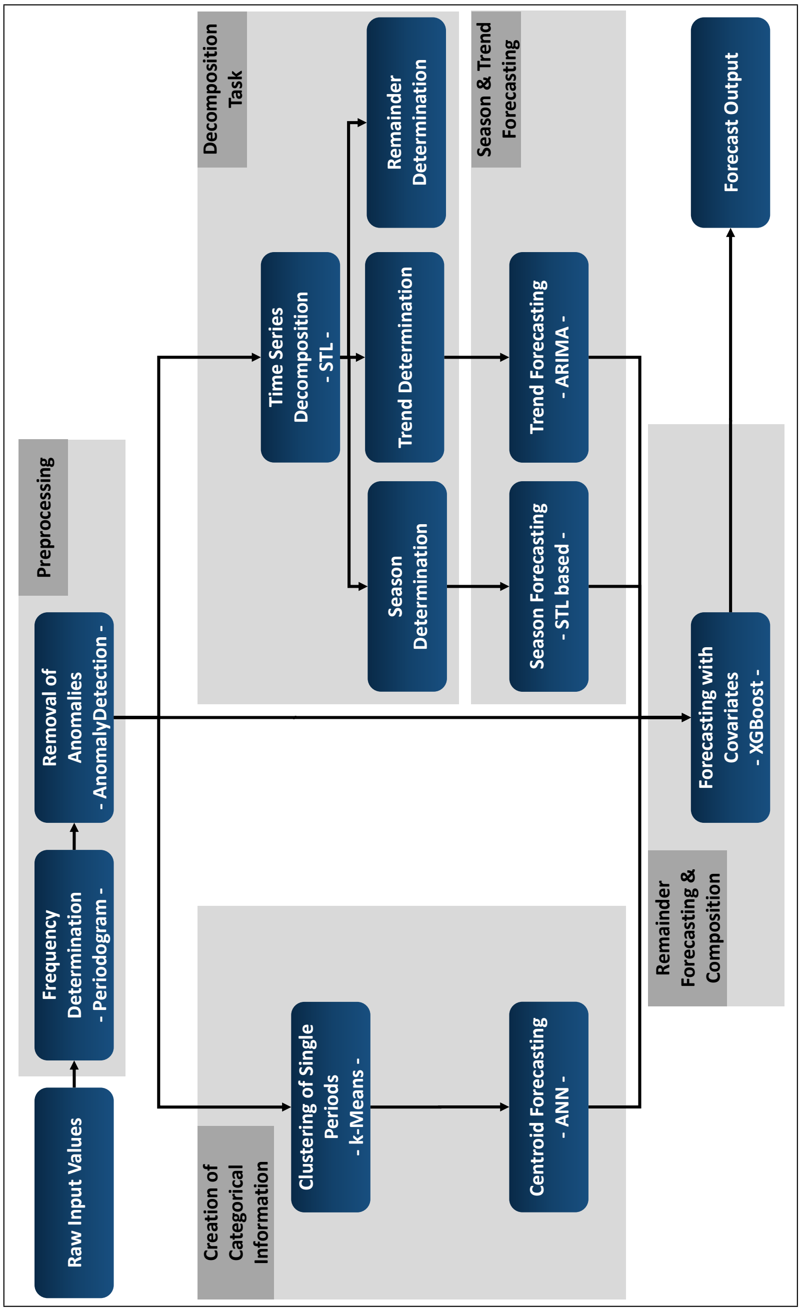

Raw Input Values
Preprocessing
Frequency Determination - Periodogram -
Removal of Anomalies - AnomalyDetection -
Decomposition Task
Time Series Decomposition - STL -
Trend Determination
Season Determination
Remainder Determination
Season & Trend Forecasting
Trend Forecasting - ARIMA -
Season Forecasting - STL based -
Creation of Categorical Information
Clustering of Single Periods - k-Means -
Centroid Forecasting - ANN -
Remainder Forecasting & Composition
Forecasting with Covariates - XGBoost -
Forecast Output

Figure 7.1: A simplified illustration of the Telescope approach.

and trend forecasting is executed. The seasonality determined by STL is simply continued, whereas the trend is forecast using the `auto.arima` function from the `forecast` R package by Hyndman [HK08, Hyn17]. Since there is no seasonal pattern left, seasonality is disabled in `auto.arima` for this purpose. Moreover, this seasonality disabling decreases the runtime of the algorithm. Additionally, the time series with removed anomalies is used to create categorical information. For this purpose, the time series is cut into single periods. Then, the single periods are clustered into two classes using the `kmeans` R function. Each class is represented by its centroid. Next, this history of centroids is forecast using an artificial neural network (ANN), i.e., the `nnetar` function of the `forecast` R package [HK08, Hyn17]. If a specific time series is forecast several times, this clustering task does not need to be performed every time. Finally, the last step is the remainder forecast and composition. XGBoost is used [CG16], which is an implementation of gradient boosted decision trees and it works best when obtaining covariates. XGBoost is applied using the trend, seasonality, and centroid forecasts as covariates and the raw time series history as labels. In addition, 10% of the history data are used for validation in the training phase to prevent XGBoost from overfitting. The sources of the Telescope approach are currently under publication as open-source[1].

7.2 Concluding Remarks

In the field of auto-scaling, typical time series from monitored metrics like arrival rates show seasonal patterns. As multiple values need to be predicted at once, most state-of-the-art forecasting methods cannot deal with such time series very well. Thus, we develop Telescope addressing RQ B.3 (How can a hybrid forecast approach based on decomposition be designed to be capable of providing accurate and fast multi-step-ahead forecasts of complex seasonal time-series within seconds?). Telescope is a novel hybrid, multi-step-ahead forecasting method for seasonal, univariate time series and based on multiple preprocessing steps and explicit time series decomposition. Then, each component is forecast by a certain forecaster that fits the requirements of the specific component. Moreover, Telescope learns categorical information for each period of a time series. Finally, dependencies between the forecasts are learned by the machine learning method XGBoost and then, XGBoost reconstructs the final forecast.

[1]Telescope: `https://descartes.tools/telescope`

Part IV

Evaluation

Chapter 8

Load Profile Model Accuracy Evaluation

In this chapter, we evaluate the automated model extraction methods for DLIM models as presented in Chapter 4 in terms of their achieved accuracy and processing time. By showing that ten different real-world server traces covering periods between two weeks and seven months can be captured in the descriptive DLIM model format with reasonable accuracy with our automated extraction processes, we demonstrate the expressiveness of our proposed modeling formalism. All traces exhibit human usage patterns. The extraction methods are applied to these traces in order to extract DLIM instances and compare them to the corresponding original traces by computing the median absolute percentage errors (MdAPE) [HK06]. The MdAPE is calculated by computing the median over the absolute values of the relative deviations of the extracted model for each arrival rate in the original trace. The mean absolute percentage error (MAPE) is not chosen as this measure is prone to deflection by positive outliers that are more likely to occur as negative outliers as also discussed earlier.

s-DLIM and hl-DLIM extraction are applied to extract model instances for all traces. For these extraction methods, we separately evaluate the effect of noise extraction, including noise reduction. The shape of the interpolating functions is always selected as the DLIM SinTrend, meaning that sin-flanks are always used for the interpolation between arrival rate peaks and lows. We chose SinTrend because it fits closest to the original trace in the majority of cases. For the same reason, ExponentialIncreaseAndDecline is always selected for Burst modeling (it is a child of Burst in the DLIM meta-model). Trends are selected to be multiplicative since this way they have a lower impact on arrival rate lows and a relatively high impact on arrival rate peaks (contrary to additive Trends, which have a constant impact on both). We do this, since arrival rate lows vary less than arrival rate peaks according to our observations.

s-DLIM is also configured with varying Trend lengths. Best results are expected at Trend length of one Seasonal period, whereas lower accuracy is expected at the longest evaluated Trend length of three Seasonal periods. For traces with a duration greater than one month, we also apply p-DLIM. p-DLIM is configured to extract weeks as a periodic Trend list with two Trend segments of the length of three and four. Additionally, it extracts a bi-weekly period with a Trend list using two Trend segments of the length of 7. Finally, it extracts a monthly (four week) period with a Trend list using two Trend segments of the length of 14.

We compare the extraction error and run-time on commodity hardware (Core i7 4770, 16 GB RAM) against the STL [CCMT90] and BFAST time-series decomposition [VHNC10] (which return both split data as opposed to a descriptive model). To enable a fair comparison, we configure STL and BFAST to extract one seasonal pattern and not more than one trend per day to match with the features of the DLIM extractors and to not further slow down BFAST. Please note that this configuration has no significant impact on the accuracy, but decreases processing speed. In contrast to DLIM, where seasonal patterns are represented by piece-wise interpolating functions, in STL and BFAST outputs, the seasonal pattern is represented as two less compact discrete function.

8.1 Internet Traffic Archive and BibSonomy Traces

The first batch of traces was retrieved from The Internet Traffic Archive[1]. The Internet Traffic Archive includes the following traces: ClarkNet-HTTP (Internet provider WWW server), NASA-HTTP (Kennedy Space Center WWW server), Saskatchewan-HTTP (Sask.) (University WWW server), and WorldCup98 (WC98) (official World Cup 98 WWW servers). Additionally, we used a six week long trace of access times to the social bookmarking system BibSonomy [BHJ+10], beginning on May 1st 2011[2]. All traces were parsed to arrival rate traces with a quarter-hourly resolution (96 arrival rate samples per day).

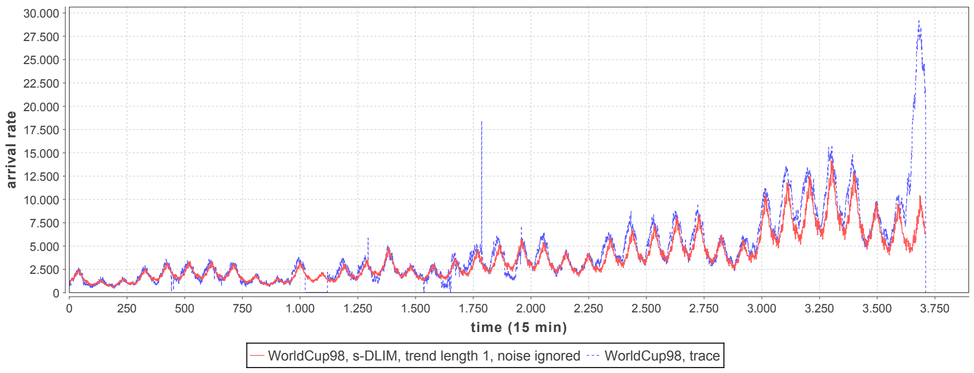


Figure 8.1: Arrival rates of the original WorldCup98 trace (blue) and the extracted DLIM instance (red) using s-DLIM with a Trend length of 1 and ignoring noise.

Table 8.1 shows the MdAPE for s-DLIM, p-DLIM, and the hl-DLIM extraction for different configurations. It also displays run-time of the overall most accurate extraction configuration (s-DLIM, ignoring noise, trend length 1) as an average value over ten runs. Accuracy and average run-times are also displayed for BFAST and STL. For some cases, BFAST did not terminate after more than one 1.5 hours.

The ClarkNet and NASA extraction results show that s-DLIM provides the best accuracy, especially with a Trend length of 1. Noise reduction does not seem to help for this particular trace during the DLIM extraction. The result does not improve when extracting the noise,

[1] Internet Traffic Archive: `http://ita.ee.lbl.gov/`

[2] The request log dataset is obtainable on request for research purposes: `http://www.kde.cs.uni-kassel.de/bibsonomy/dumps/`

Table 8.1: Model extraction errors for the Internet Traffic Archive and BibSonomy traces.

Trace	1. **ClarkNet**	2. **NASA**	3. **Sask.**	4. **WC98**	5. **BibSonomy**
Extraction Parameters	MdAPE	MdAPE	MdAPE	MdAPE	MdAPE
Extractor, Trend, Noise	[%]	[%]	[%]	[%]	[%]
p-DLIM, -, extracted	too short	32.223	43.293	52.304	37.387
p-DLIM, -, eliminated	too short	28.944	35.831	53.316	35.378
p-DLIM, -, ignored	too short	23.633	35.663	53.495	36.264
s-DLIM, 1, extracted	21.195	26.446	35.551	19.735	26.988
s-DLIM, 1, eliminated	17.509	23.560	**26.492**	16.882	**21.470**
s-DLIM, 1, ignored	**12.409**	**18.812**	29.171	**12.979**	23.831
s-DLIM, 2, ignored	14.734	20.800	30.273	15.691	26.786
s-DLIM, 3, ignored	14.919	27.577	32.085	19.161	28.218
hl-DLIM, 1, extracted	20.105	26.541	37.942	16.093	27.513
hl-DLIM, 1, eliminated	19.361	24.539	33.240	15.660	25.433
hl-DLIM, 1, ignored	72.924	55.575	80.792	43.957	42.268
STL	13.540	20.384	30.134	16.041	20.299
BFAST	12.243	no result	no result	no result	no result
avg. s-DLIM run-time	4.2 ms	25.2 ms	118.8 ms	11.8 ms	125 ms
avg. STL run-time	3.5 ms	15.0 ms	38.7 ms	13.2 ms	15.0 ms
avg. BFAST run-time	76276 ms	no result	no result	no result	no result

as noise generated by a random variable does not reproduce the exact measured results and increases the absolute arrival rate difference between trace and model. We trace the discrepancies between the extracted model instance and the original trace to three major causes:

- In some cases, bursts are not detected with full accuracy.
- The NASA server was shut down for maintenance between time-stamps 2700 and 2900. The extraction methods do not have contingencies for this case.
- Deviating Seasonal Patters are a major cause of inaccuracy in the extracted models. The extraction methods all assume a single, repeating Seasonal Part. Depending on the trace, this assumption may be valid to a different extent. In this case, the extracted Seasonal pattern is able to approximate most days in the trace, but a number of significant deviations occur. Manual modeling in the DLIM editor can circumvent this problem, as DLIM itself supports mixes of multiple seasonal patterns. We are currently working on extending the automated extractors to make use of this feature. Ideas range from the inclusion of additional meta-knowledge, such as calendar information, to the implementation of seasonal break detection.

In the case of the Saskatchewan-HTTP extraction, noise reduction improves the s-DLIM results. However, overall the results are not as good as they are for the other traces. The

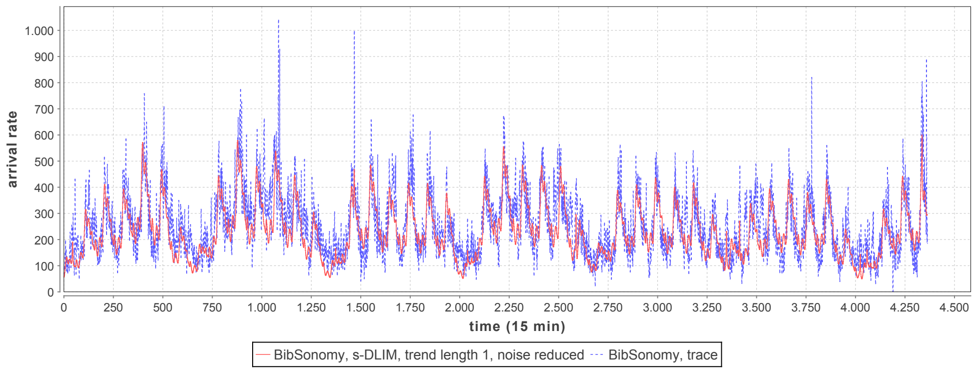


Figure 8.2: Arrival rates of the original BibSonomy trace (blue) and the extracted DLIM instance (red) using s-DLIM with Trend length 1 and noise reduction.

major explanation for the relatively poor results is once more the Seasonal pattern deviation. Since the Saskatchewan-HTTP trace extends over 7 months, the Seasonal patterns have a lot of room for deviation. The model extractors fail to capture this. This leads to an additional error in the Trend calibration, as trends are supposed to be calibrated, so that the greatest Seasonal peak in every Seasonal iteration matches the trace's nearest local arrival rate maximum. Since the Seasonal pattern deviation causes the extracted Seasonal peak's time of day to not match the trace's Seasonal peak's time of day, the calibration takes place at the wrong point of time. This also explains why a majority of extracted days have a lower peak then their counterparts in the original trace.

The major deviation from the trace's Seasonal patterns also explains why s-DLIM performs better using noise elimination for the Saskatchewan-HTTP extraction. Noise reduction helps to mitigate the effect of seasonal pattern changes over time, thus reducing the effect of the Seasonal pattern deviation.

Similarly to the Saskatchewan trace, s-DLIM extraction of the BibSonomy trace also improves with noise filtering. We explain this by the observation that the BibSonomy trace features a significant number of bursts, occurring at a relatively high frequency, as well as significant noise (as seen in Fig. 8.2). Without filtering, some of these bursts are included in the seasonal pattern by the s-DLIM extractor, distorting the extracted seasonal pattern. When applying noise reduction, the influence of these bursts is diminished. Therefore, the extracted seasonal pattern is more stable, leading to increased accuracy as major bursts are still extracted during s-DLIM's burst extraction. The BibSonomy trace demonstrates that s-DLIM (and also p-DLIM) are capable of dealing with traces featuring a significant amount of noise.

p-DLIM performs well compared to the other two extraction processes. p-DLIM assumes that all trends repeat. In the case of the NASA trace, this assumption seems to be quite accurate. Even for the Saskatchewan trace, p-DLIM performs better compared to s-DLIM.

The hl-DLIM extraction shows an entirely different picture. Considering that hl-DLIM uses only a small number of pre-defined parameters, the extracted hl-DLIM instances are surprisingly close to the detailed DLIM models. Contrary to what was observed in the

DLIM extraction, however, the hl-DLIM extraction strongly relies on noise reduction. If the noise is ignored and not filtered, hl-DLIM extraction accuracy drops dramatically. This can easily be attributed to the linear interpolation between the extracted peaks. Since hl-DLIM interpolates between the highest and lowest peak (thus only extracting two peaks), the non-filtered trace offers a high number of noisy peaks with minimal impact on the overall arrival rate. The filtered version, however, only offers a few remaining peaks, which have a much higher impact on the overall arrival rate. Applying noise reduction forces the hl-DLIM extractor to only consider the peaks with significant impact rather than accidentally choosing outliers as peaks.

The WorldCup98 extraction results are notable in that s-DLIM and hl-DLIM extraction perform relatively well, whereas p-DLIM performs worst for all considered traces. The obvious cause of this is the observation that the WorldCup98 trace does not feature recurring trends and only features increasing trends. The s-DLIM and hl-DLIM extraction methods can handle this easily, whereas p-DLIM cannot.

The times series decomposition method STL shows worse accuracy values with the exception of the BibSonomy trace, for which it achieves a slightly better accuracy. In all cases, STL terminates a few milliseconds faster than DLIM extraction. BFAST terminates only for ClarkNet within 1.5 hours and achieves a comparable accuracy compared to s-DLIM. Due to the order of magnitude by which BFAST run-time differs from STL and DLIM, using it in an autonomous management context seems difficult.

8.2 Wikipedia and CICS Traces

The second batch of traces was retrieved from the Wikipedia page view statistics[3]. They were parsed from the projectcount dumps, which already feature arrival rates with an hourly resolution. We restrict our analysis to the English, French, German and Russian Wikipedia projects, covering four of the six most requested Wikipedia projects and being distributed over different time-zones. All traces are from December 2013, with the exception of the English Wikipedia trace, which is from November 2013. The English December 2013 trace exhibits a major irregularity during the 4th day, which we attribute to a measurement or parsing error. While the French, German, and Russian Wikipedia projects are mostly accessed from a single time zone, the English Wikipedia is retrieved from all over the world. Thus, evaluating the impact of access behavior over different time zones and helping to assess how well the DLIM extraction methods deal with local vs. global access patterns.

In addition, we extract arrival rates from traces of the IBM CICS transaction processing system. These traces were logged in a banking environment with significantly different usage patterns during weekdays and weekends. These traces feature a quarter-hourly resolution (96 samples per day).

The Wikipedia extraction results in Table 8.2 confirm many of the observations made with the Internet Traffic Archive traces. Noise extraction is most useful for hl-DLIM extraction; Trend length of 1 as part of s-DLIM performs best. The overall accuracy, however, is significantly better than for the Internet Traffic Archive traces since the Seasonal pattern deviation, while still relevant, exhibits less impact than before.

[3]Wikipedia traces: `http://dumps.wikimedia.org/other/pagecounts-raw/2013/`

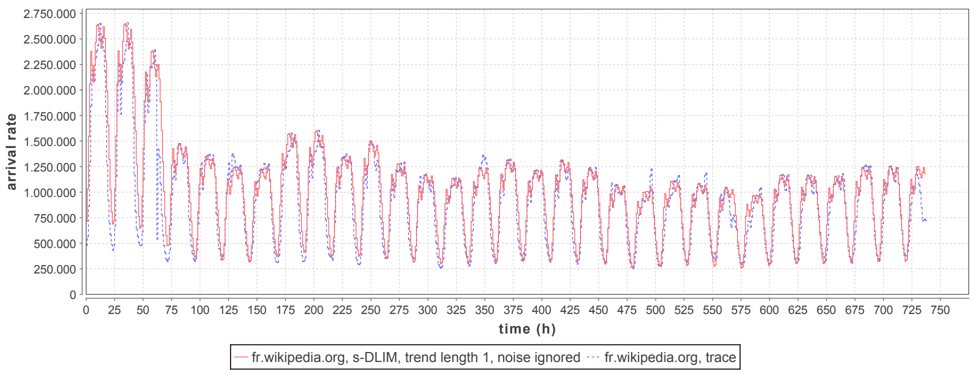


Figure 8.3: Arrival rates of the original French Wikipedia trace (blue) and the extracted DLIM instance (red) using s-DLIM with Trend length 1 and ignoring noise.

The Russian Wikipedia trace differs from the other Wikipedia traces. Noise reduction also improves s-DLIM, while, as usual, being useful for hl-DLIM extraction. The overall accuracy is similar to the other Wikipedia trace extractions. For this single trace, however, the Seasonal patterns are shaped in such a way that the noise reduction lessens the impact of the Seasonal pattern deviation.

The extraction results for the English Wikipedia trace exhibit by far the best overall accuracy across all examined traces. The reason for this is the unusually high arrival rate base level. Since wikipedia.org is accessed globally at all times, the load intensity variations on top of the base level have little impact on the load variations in general. As a result, all modeling errors are also reasonable small.

In terms of accuracy, our extraction processes perform on the same level compared to the STL and BFAST decompositions for the Wikipedia traces and the CICS trace. s-DLIM performs better than STL and BFAST for both the German and French Wikipedia traces. Here, s-DLIM's accuracy profits from its support of multiplicative trends. STL and BFAST, however, provide better accuracy for the English and Russian traces. Comparing run-times, s-DLIM is significantly faster than BFAST and sligthly faster than STL. Running on the same machine, LIMBO's s-DLIM implementation performed on average 8354 times faster than BFAST's R implementation and returned results in all cases in less than 0.2 seconds.

For the CICS trace, STL has a high MdAPE value of 72% while, s-DLIM achieves a better but still high value of 29%. The CICS trace once more demonstrates the effect of seasonal deviation. It does not reach the accuracy of the Wikipedia workloads as one seasonal pattern cannot model the strong differences between workdays and weekends. To demonstrate that the modeling error is in large part caused by seasonal deviation, we extract a DLIM instance for the first five days in the CICS trace (Monday to Friday). This weekday model only features an MdAPE of 13.745%.

Table 8.2: wikipedia.org model extraction errors.

Trace	1. **German Wikipedia**	2. **French Wikipedia**	3. **Russian Wikipedia**	4. **English Wikipedia**	5. **IBM CICS**
Extraction Parameters	MdAPE	MdAPE	MdAPE	MdAPE	MdAPE
Extractor, Trend, Noise	[%]	[%]	[%]	[%]	[%]
s-DLIM, 1, extracted	11.215	10.472	9.964	7.764	71.311
s-DLIM, 1, eliminated	10.511	8.566	**9.912**	7.838	40.822
s-DLIM, 1, ignored	**8.538**	**7.600**	11.251	**4.855**	**29.199**
s-DLIM, 2, ignored	9.956	8.973	11.683	5.270	34.746
s-DLIM, 3, ignored	11.771	9.813	11.420	7.230	38.785
hl-DLIM, 1, extracted	11.898	8.503	12.392	7.750	80.043
hl-DLIM, 1, eliminated	11.393	8.373	12.496	7.961	59.956
hl-DLIM, 1, ignored	13.126	10.816	13.310	8.868	92.400
STL	13.309	8.671	6.747	2.580	71.997
BFAST	11.223	8.511	5.809	2.302	no result
avg. s-DLIM run-time	3.9 ms	3.5 ms	5.8 ms	3.2 ms	11.3 ms
avg. STL run-time	7.5 ms	7.0 ms	7.0 ms	7.5 ms	11.3 ms
avg. BFAST run-time	23518 ms	23630 ms	23803 ms	21517 ms	no result

8.3 Summary

The results of our evaluation show that the proposed DLIM model extraction methods are capable of extracting DLIM instances with an average modeling error of only 15.2% (MdAPE) over ten different real-world traces that cover between two weeks and seven months. The model extraction performs best for the Wikipedia traces. Extracted seasonal patterns match the trace's days well and the overlying trends are precisely calibrated. Concerning the Internet Traffic Archive traces, we identified the seasonal pattern deviations for traces extending over several months as a major challenge for future work. Changes of daily usage patterns over the course of these particularly long traces lead to a decrease in accuracy. Nevertheless, the median error remains below 27%. Furthermore, the BibSonomy trace demonstrates that the extraction mechanisms are robust and capable of dealing with noisy arrival rate patterns. The results in general show that DLIM itself is capable of accurately capturing real-world load intensity profiles, independent of the explicit extraction processes we introduce.

Chapter 9

Evaluation of Elasticity Metrics and Bungee Measurement Methodology

This chapter first outlines an experiment setup to demonstrate the benchmarking capabilities of the Bungee approach with a realistic load profile and a realistic amount of dynamically scaled resources. The benchmark is applied to a private, CloudStack-based cloud, as well as a public, AWS-based cloud (Amazon EC2). Furthermore, we assess the reproducibility of measurements with a focus on Bungee's scalability analysis. In detail, we analyze the scalability behavior of the private cloud scenario, before we conduct for each individual elasticity metric a row of experiments on synthetic load profiles. The metrics results of these experiments show-case that the metric values are in line with the intuitive impression of the observed elastic behavior. We conduct a realistic case-study in both cloud environments that exhibit different performance and scalability characteristics. Leveraging DLIM capabilities, we generate realistic load profiles. By changing the configuration parameters of a standard threshold-based auto-scaler, we observe different timing and accuracy of elastic adaptations. We aggregate the resulting elasticity metrics in three different ways that all result in the same ranking consistently.

9.1 Experiment Setup

The experiment setup consists of three building blocks: The infrastructure nodes, the management and load balancing nodes and the benchmark controller nodes. The first two parts form the benchmarked system under test (SUT). In the private cloud setup, the infrastructure provides the physical resources of four 12 core AMD Opteron 6174 CPUs at 2.2 GHz and 256 GB RAM as fully virtualized resources using XenServer 6.2 as hypervisor. The management node (Intel Core i7 860 with 8 cores at 2.8 GHz and 8 GB RAM) runs the cloud management software (CloudStack 4.2) and the load balancer (Citrix Netscaler 10.1 VPX 1000) in separate Linux-based VMs. The benchmark controller node runs the Java-based benchmark harness and the load driver (JMeter) on a Windows desktop (Dell Precision T3400 (4 x 2.5 GHz) and 8 GB RAM). The three nodes are physically connected over a 1 GBit Ethernet network. Clock synchronization is ensured by using a Stratum 3 NTP server located in the same network. The template for virtual machines used by CloudStack bases on CentOS 5.6 as operating system with Java run-time environment and SNMP service installed. SNMP provides access to resource utilization information required for the elasticity mechanism.

Experiments on the private CloudStack-based environment employ VMs with 1 core at 2.2. GHz, 1 GB RAM and local storage is available. Experiments on the AWS EC2 public cloud employ the general purpose m1.small instance type.

9.1.1 Threshold-based Auto-Scaler Parameters

CloudStack (CS) and AWS allow to configure a rule based auto-scaling mechanism with the set of parameters in Table 9.1. The listed default values are used if not mentioned otherwise.

Table 9.1: Default parameters of threshold-based auto-scaler of private cloud setup

Param. Name	Default Val. (CS | AWS)	Description
`evalInterval`	5s	Frequency for scaling rule evaluation
`quietTime`	300s	Period without scaling rule evaluation after supply change
`destroyVmGracePer`	30s	Time for connection closing before a VM is destroyed
`condTrueDurUp`	30s | 60s	Duration of condition true before a scale up triggered
`counterUp`	CPU util.	Monitored metric for `thresholdUp`
`operatorUp`	>	Operator comparing `counterUp` and `thresholdUp`
`thresholdUp`	90%	Threshold for `counterUp`
`condTrueDurDown`	30s | 60s	Duration of condition true before scale down triggered
`counterDown`	CPU util.	Monitored metric for `thresholdDown`
`operatorDown`	<	Operator comparing `counterDown` and `thresholdDown`
`thresholdDown`	50%	Threshold for `counterDown`
`responseTimeout`	1s | 2s	Period of time within that a response is expected from healthy instances
`healthCheckInterval`	5s	Time between two health checks
`healthyThreshold`	1 | 2	Number of subsequent health checks before instance is declared healthy
`unhealthyThreshold`	4 | 2	Number of subsequent health checks before instance is declared unhealthy

9.1.2 Benchmark Controller Configuration

The benchmark controller offers several configuration options that allow to configure it according to the targeted domain. Table 9.2 shows the different parameters and default values.

Table 9.2: Benchmark harness parameters

Name	Default
`requestSize`	50000
`requestTimeout`	1000ms
`SLO`	95% of all requests must be processed successfully within a maximum response time of 500ms.
`warmupCalibration`	180s
`warmupMeasurement`	300s

The amount of work executed within each request is defined by a `requestSize` parameter. It is set to 50000 for the evaluation meaning that each request issues a randomized calculation of the 50000_{th} element of the Fibonacci series. During the calibration phase, the benchmark needs a specified SLO in order to perform the System Analysis. Additionally, the benchmark has a `requestTimeout` parameter defining how long the benchmark waits for a response before the connection is closed.

9.2 System Analysis Evaluation

Two different activities precede the actual elasticity measurement: The System Analysis and the Benchmark Calibration. The latter depends on the correctness of the System Analysis. The System Analysis is therefore evaluated with respect to the following two questions:

- Is the result of the System Analysis reproducible on the test system?
- What is the deviation between the results of Detailed System Analysis and the results of Simple System Analysis, which assumes a linearly increasing resource demand?

9.2.1 Reproducibility

This subsection tests the following hypothesis:

Hypothesis 1. *Under the assumption that the analysis result follows a normal distribution, the error of the System Analysis for the first scaling stage is smaller than 5% on a confidence level of 95%.*

The reproducibility of the System Analysis is analyzed for three different system configurations that are different with respect to the processing performance of the underlying resources. To obtain different levels of performance for resource instances, CloudStack is configured to use service offerings that either assign one (Offering A), two (Offering B), or four (Offering C) virtual CPUs to the created. The evaluation is conducted for every configuration separately. Let $X_i \in N(\mu, \sigma)$ be the samples of the result of the analysis and n be the number of samples. The sample mean $\overline{X}$ can be expressed as $\overline{X} = \frac{\sum X_i}{n}$. The sample standard deviation S can be expressed as $S = \frac{\sum (X_i - \overline{X})^2}{n-1}$.

Claim to be proven:

$$\begin{aligned} & P(c_1 \leq \mu \leq c_2) \leq 1 - \alpha \\ & \text{with} \\ & c_1 = 0.95 * \bar{X},\ c_2 = 1.05 * \bar{X},\ \alpha = 0.05 \end{aligned} \tag{9.1}$$

According to [Man64], it is shown that $T = \frac{[\overline{X} - \mu]\sqrt{n}}{S}$ has a t-distribution with $(n-1)$ degrees of freedom (d.f.). It follows:

$$\begin{aligned} & P(c_{low} \leq \mu \leq c_{high}) \leq 1 - \alpha \\ & \text{with} \\ & c_{low} = \overline{X} - t_{1-\alpha/2;n-1} * S/\sqrt{n} \\ & c_{high} = \overline{X} + t_{1-\alpha/2;n-1} * S/\sqrt{n} \end{aligned}$$

where $t_{1-\alpha/2;n-1}$ is the upper $(1 - \frac{\alpha}{2})$ critical point of the t-distr. with n-1 d.f.

To prove the claim, we shown that

$$\begin{aligned} c_{low} = \overline{X} - t_{1-\alpha/2;n-1} * S/\sqrt{n} \geq 0.95 * \bar{X} = c_1 \\ c_{high} = \overline{X} - t_{1-\alpha/2;n-1} * S/\sqrt{n} \leq 1.05 * \bar{X} = c_2 \end{aligned} \tag{9.2}$$

holds true for a set of n scaling analysis samples.

Off.	Analysis Samples [#req./sec.]										$\overline{X}$	S	c_1	c_{low}	c_{high}	c_2
A	35	35	35	35	35	35	35	35	35	35	35.0	0.00	33.25	35.00	35.00	36.75
B	55	57	56	56	56	56	58	55	57	56	56.2	0.92	53.39	55.54	56.86	59.01
C	97	101	100	100	97	99	98	101	99	101	99.3	1.57	94.34	98.18	100.42	104.2

Table 9.3: Results of the reproducibility evaluation for the System Analysis

Table 9.3 shows the result of the System Analysis for the first scaling stage for $n = 10$ measurement samples and three different system configurations. For all configurations the equations (9.2) are true. Thus, it is not possible to reject Hypothesis 1.

9.2.2 Linearity Assumption

A simplified System Analysis only analyzes the load processing capabilities of the first scaling stage. It then assumes, that the resource demand increases linearly with the load intensity and therefore creates a mapping with steps of equal length, like illustrated in Figure 9.1. For the system depicted in Figure 9.1, only the load processing capability - here 25 - for one resource was determined by the simplified System Analysis. The load processing capabilities for two, three and four resources - here 50, 75, and 100 - are extrapolated based on the linearity assumption. Due to some overhead, e.g., overhead in the load-balancer

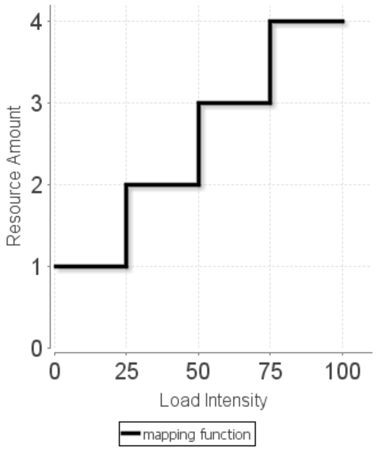


Figure 9.1: System with linear increasing resource demand

when four resources are used, it may be that the real load processing capability does not equal the one extrapolated based on the linearity assumption. This evaluation illustrates how big the error of not measuring the load processing capabilities for more than one resources is on the used test system.

Hypothesis 2. *The test system's resource demand scales linearly with the load intensity in a scale out scenario.*

It is notable that this hypothesis is not about a property of the benchmark but about a property of the tested system.

To test the hypothesis for a given system configuration, two measurements are taken. First, the simplified System Analysis is used to get the load processing capability $i_{unscaled}$ for one resource. Then, system is scaled manually to use n resources. Now, the load processing capability i_{scaled} for the scaled system is determined. In the next step, i_{scaled} is compared with the extrapolated intensity $i_{extrapolated}$ for n resources. Hereby, $i_{extrapolated}$ is calculated as $i_{extrapolated} = n * i_{unscaled}$. Using this numbers, the absolute deviation d_{abs}

and the relative deviation d_{rel} for the test system are:

$$d_{abs} = i_{extrapolated} - i_{scaled}$$
$$d_{rel} = \frac{d_{abs}}{i_{scaled}}$$

This analysis has been conducted for two different system configurations that are different with respect to the levels of performance of the underlying resources. To obtain different levels of performance for resource instances, CloudStack has been configured to use service offerings which either assign one (Offering A) or two (Offering B) virtual CPUs to the created VMs.

The result of this evaluation is shown Table 9.4 for Offering A and in Table 9.5 for Offering B respectively.

For Offering A, the maximum intensity i_{scaled} was analyzed for all system configurations from $n = 1..18$ resources. The result is shown in Table 9.4.

n	$i_{unscaled}$ [#req./sec.]	$i_{extrapolated}$ [#req./sec.]	i_{scaled} [#req./sec.]	dev_{abs} [#req./sec.]	dev_{rel} [%]
1	35	35	35	0	0.0
2	-	70	70	0	0.0
3	-	105	99	6	6.1
4	-	140	140	0	0.0
5	-	175	169	6	3.6
6	-	210	199	11	5.5
7	-	245	239	6	2.5
8	-	280	284	-4	-1.4
9	-	315	320	-5	-1.6
10	-	350	339	11	3.2
11	-	385	359	26	7.2
12	-	420	399	21	5.3
13	-	455	462	-7	-1.5
14	-	490	479	11	2.3
15	-	525	519	6	1.2
16	-	560	564	-4	-0.01
17	-	595	598	-3	-0.01
18	-	630	633	-3	-0.01

Table 9.4: Linearity analysis for Offering A

Within this experiment, the measured deviation from the linearity assumption is always below 10%. Since the accuracy of measurement itself is limited (95% confidence for relative accuracy of ±5%, compare Section 9.2.1), it can be assumed that the measured deviation is mainly due to inaccurate measurements.

For the first twelve resource scaling stages, the linearity analysis has been conducted three times. It is notable that these measurements have generated the same small deviations from the linearity assumption consistently. For six resources for example, all three measurements have returned 199 as measured maximum intensity i_{scaled}. Since the extrapolated maximum intensity $i_{extrapolated}$ for six resources is 210, this means a relative deviation d_{rel} of 5.5%. This observation contradicts the assumption that the measured deviations from the linearity assumption are mainly due to an inaccurate measurement. It rather indicates that the deviations from the linearity assumption are mainly system specific.

For Offering B, the maximum intensity i_{scaled} has been analyzed for all system configuration from $n = 1..22$ resources. For all scaling stages, maximum intensity i_{scaled} has been measured three times. Table 9.4 shows the averaged results.

As for Offering A, the deviation from the linearity assumption is below 10% for all analyzed scaling stages. Furthermore, the deviation tends to increase slowly with the number of used resources. A possible explanation for this increasing deviation is overhead within the hypervisor or the load balancer. Note that for the largest scale out scenario 92% of the underlying hardware resources of the test system have been used.

Hypothesis 2 holds for the tested scale out scenarios to a limited extend. Although the deviation from the linearity assumption is always below 10%, small deviations have been observed consistently. If possible, the usage of the Detailed System Analysis should therefore be preferred for new systems.

9.2.3 Summary of System Analysis Evaluation

The evaluation of the System Analysis demonstrated the reproducibility of its results for three system configurations. Within a second evaluation, it was shown that for scale-outs of up to 22 resources the linearity assumption holds for the test system to a limited extend. Constant small deviations from the linearity assumption have been observed for different scaling stages, consistently. The usage of the detailed System Analysis is therefore preferred.

During the evaluation period, several updates were installed on the hypervisor. The complete reproducibility evaluation and the evaluation of the linearity assumption for Offering A were conducted before, the linearity evaluation for Offering B has been conducted after the installation of hypervisor updates. An additional analysis for Offering A after the installation of updates on the hypervisor has not shown the deviations illustrated in Table 9.4 anymore. This change in the observed scaling behavior signifies that it is important to reanalyze the scaling behavior after any direct or indirect change of the tested system.

n	$i_{unscaled}$ [req./sec.]	$i_{extrapolated}$ [req./sec.]	i_{scaled} [req./sec.]	d_{abs} [req./sec.]	d_{rel} [%]
1	58	58	58	0	0.0
2	-	116	115.7	0.3	0.3
3	-	174	225.7	4.0	2.4
4	-	232	225.7	6.3	2.8
5	-	290	282.0	8.0	2.8
6	-	348	336.3	11.7	3.5
7	-	406	390.3	15.7	4.0
8	-	464	449.0	15.0	3.3
9	-	522	503.3	18.7	3.7
10	-	580	559.3	20.7	3.7
11	-	638	613.3	24.7	4.0
12	-	696	670.3	25.7	3.8
13	-	754	718.0	36.0	5.0
14	-	812	773.0	39.0	5.0
15	-	870	829.0	41.0	4.9
16	-	928	881.3	46.7	5.3
17	-	986	923.3	53.7	5.8
18	-	1044	1003.3	40.7	4.1
19	-	1102	1044.7	57.3	5.5
20	-	1160	1099.7	60.3	5.5
21	-	1218	1154.0	64.0	5.5
22	-	1276	1202.3	73.7	6.1

Table 9.5: Linearity analysis for Offering B
(i_{scaled} is averaged over three independent analysis runs)

9.3 Metric Evaluation

In the first set of experiment, we showcase that the accuracy and timing metrics introduced in Chapter 5.2 allow to rank systems according to their resource elasticity on an ordinal scale. Every metric is evaluated with the help of a simple synthetic load profile to induce controlled demand changes. For each metric, the elasticity mechanism of the private cloud systems is configured in four variants to exhibit different degrees of elasticity.

For comprehensibility reasons, the load profiles illustrated in this section are plotted with a vertically scaled axis such that the load intensity 100 is the maximum intensity one resource can withstand. However, in the calibration step the real intensity is adjusted in a way that the resource demand on the test system equals the resource demand in the illustrations. In the following, the evaluation for each metric is explained.

9.3.1 Experiment 1: Underprovisioning Accuracy

Load Profile

The under-provision accuracy metric a_U and its normalized version $theta_U$ are evaluated with the load profile illustrated in Figure 9.2. The load profile starts with an intensity that is just a little bit below the maximum intensity for two instances. After five minutes, the intensity changes stepwise - every five minutes - to a lower load intensity. However, the last intensity is still high enough to require two resources. A system with a low degree of elasticity may drop the second resource because of the shrinking demand although it is still needed. Thus, such a system lacks in accuracy.

System Configuration

The degree of elasticity of the test system is varied by changing the `thresholdDown` parameter. For increasing values of this parameter the system tends to drop resources earlier. The a_U and normalized θ_U metrics are evaluated for the following `thresholdDown` values: 55% (Config. A), 65% (Config. B), 75% (Config. C) and 85% (Config. D).

Results

Figure 9.2 shows the elasticity behaviors Configurations A-D for different values of `thres-holdDown`. It can be seen that due to lower degrees of elasticity for increasing `threshold-Down` values, the amount of under-provisioned resources increases. This is reflected by the metric results shown in Table 9.6. For decreasing degrees of elasticity, that means for increasing values of `thresholdDown`, the a_U and normalized θ_U metrics increase. Thus, the a_U and normalized θ_U metrics allow to rank elastic systems on an ordinal scale.

Remarks

Although the systems drops the second resource too early, they reallocate it again after a while. This behavior is due to the fact, that for every arrival rate above 100, the CPU utilization is very high if just one resource is used. Thus, the scale up rule triggers the allocation of a new resource shortly after each deallocation.

`thresholdDown` [%]	55 (A)	65 (B)	75 (C)	85 (D)
a_U [$\#res.$]	0.145	0.302	0.371	0.603
θ_U [%]	0.073	0.151	0.185	0.301

Table 9.6: Measurement results for the a_U and θ_U metrics

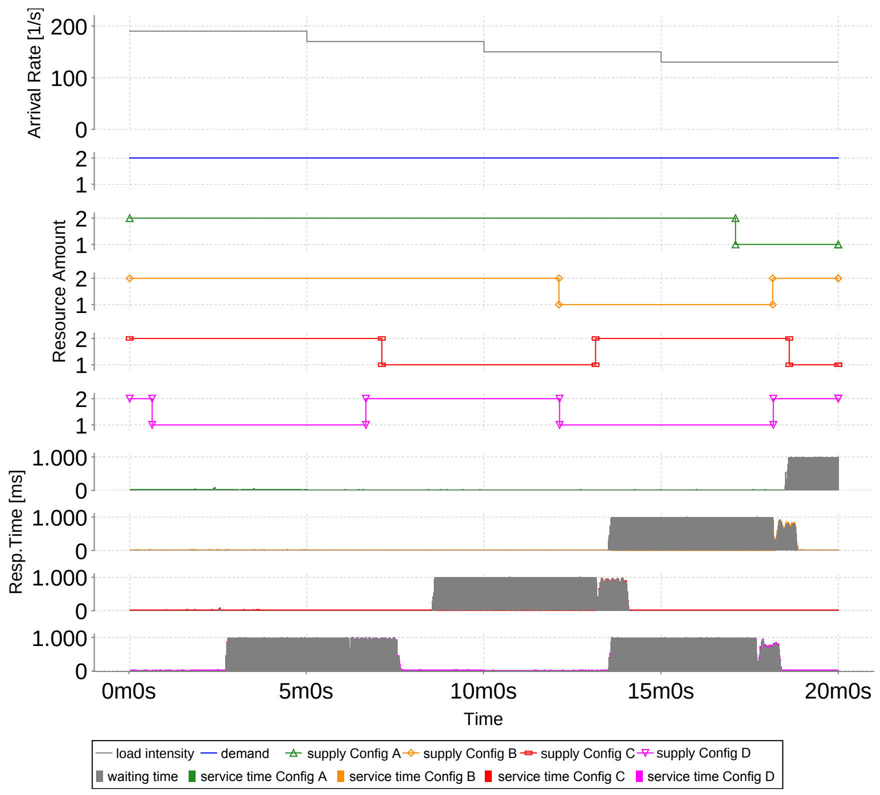


Figure 9.2: Evaluation of the a_U and θ_U metrics. Load profile (top), induced resource demand (second graph) and measured resource supply and response times for increasing `thresholdDown` values

9.3.2 Experiment 2: Overprovisioning Accuracy

Load Profile

The over-provisioning accuracy metric a_O and its normalized version θ_O are evaluated with the load profile illustrated in Figure 9.3. The load profile starts with 30% of the maximum intensity that one instance can handle. After one minute, the intensity changes stepwise - every four minutes with 20% increments - to 90% load intensity. A system with a low degree of elasticity will allocate a second resource instance when the CPU utilization increases although it is not needed.

System Configuration

The degree of elasticity of the test system is varied by changing the `thresholdUp` for the scale up rule. For decreasing values of this parameter the system tends to provision resources earlier, which leads to lower degrees of elasticity. The a_O and θ_O metric are evaluated for the following `thresholdUp` values: 80% (Configuration A), 60% (Configuration B), 40% (Configuration C) and 30% (Configuration D).

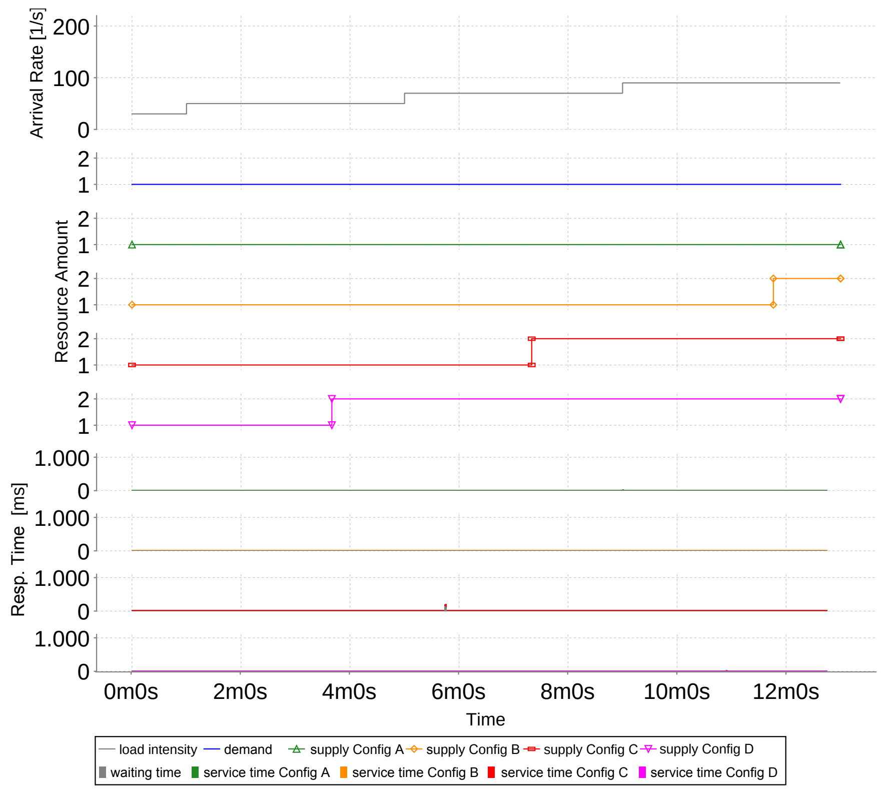


Figure 9.3: Evaluation of the a_O and θ_O metric. Load profile (top), induced resource demand (second graph) and measured resource supply and response times for decreasing `thresholdUp` values (A: 80%, B: 60%, C: 40%, D: 30%).

`thresholdUp` [%]	80 (A)	60 (B)	40 (C)	30 (D)
a_U [$\#res.$]	0	0.094	0.435	0.717
θ_U [%]	0	0.094	0.435	0.717

Table 9.7: Measurement results for the a_O and θ_O metrics

Results

Figure 9.3 shows the elasticity behaviors for different values of `thresholdUp`. It can be seen that due to lower degrees of elasticity for decreasing `thresholdUp` values, the amount of over-provisioned resources increases. This is reflected by the metric results shown in Table 9.7. For decreasing degrees of elasticity, that means for decreasing values of `thresholdUp`, the a_O respectively θ_O metric increases. Thus, the overprovisioning accuracy metrics allow to rank elastic systems on an ordinal scale. As the demand is always at one resource unit, the two metrics result in identical values.

9.3.3 Experiment 3: Underprovisioning Timeshare

Load Profile

The under-provision timeshare metric τ_U is evaluated using a simple step load profile, illustrated in Figure 9.4. The load profile starts with a constant load intensity i_{low} which can easily be handled with one resource instance. After one minute, the intensity changes to a higher load intensity i_{high} for which two resource instances are necessary for four minutes. The load profile is calibrated in a way, that intensity i_{low} is half of the maximum intensity the system can withstand using one resource and i_{high} is 150% of the maximum intensity the system can withstand using one resource. Thus, with the correct amount of resources the system should be able to handle the load easily, for any point in time.

System Configuration

With increasing values for the `condTrueDurUp` parameter the systems reacts delayed, which leads to lower degrees of elasticity. The τ_U metric is evaluated for the following `condTrueDurUp` values: 5s (Config. A), 10s (Config. B), 30s (Config. C) and 60s (Config. D).

Results

Figure 9.4 shows the elastic behaviors for different values of the `condTrueDurUp` parameter. It can be seen that due to lower degrees of elasticity for increasing `condTrueDurUp` values, the timeshare for under-provisioning increases. This is reflected by the metric results shown in Table 9.8. For decreasing degrees of elasticity, which means for increasing `condTrueDurUp` values, the τ_U metric increases. Thus, the τ_U metric allows ranking elastic systems on an ordinal scale.

condTrueDurUp [s]	5(A)	10(B)	30(C)	60(D)
τ_U [%]	25.1	26.0	29.3	34.3

Table 9.8: Measurement results for the τ_U metric

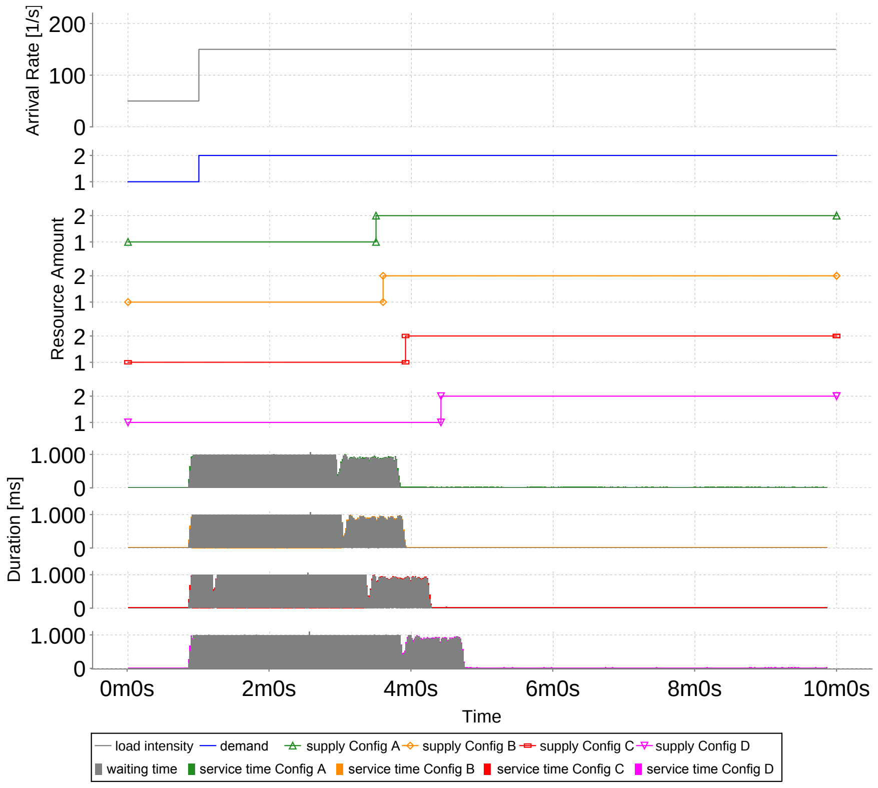


Figure 9.4: Evaluation of the τ_U metric. Load profile (top), induced resource demand (second graph) and measured resource supply and response times for increasing `condTrueDurUp` values.

9.3.4 Experiment 4: Overprovisioning Timeshare

Load Profile

Similar to the evaluation of the under-provision timeshare, the over-provision timeshare metric τ_O is evaluated using a simple step load profile, illustrated in Figure 9.5. The load profile starts with a constant load intensity i_{high} which can easily be handled with two resource instances. After one minute, the intensity changes to a lower load intensity i_{low} for which just one resource instance is enough for four minutes. The load profile is calibrated in a way, that intensity i_{high} is 175% of the maximum intensity the test system can withstand using one resource and i_{low} is 50% of the maximum intensity the test system can withstand using one resource. Thus, with the correct amount of resources the system should be able to handle the load easily, for any point in time.

System Configuration

The degree of elasticity of the test system is changed the same way as it was done for the under-provisioning timeshare evaluation.

Results

Figure 9.5 shows the elasticity behaviors for different values of the `condTrueDurDown` parameter. It can be seen that due to lower degrees of elasticity for increasing `condTrueDurDown` values, the timeshare where the system over-provisions increases. This is reflected by the metric results shown in Table 9.9. For decreasing degrees of elasticity, that means for increasing `condTrueDurDown` values, the τ_O metric increases. Thus, the τ_O metric allows to rank elastic systems on an ordinal scale.

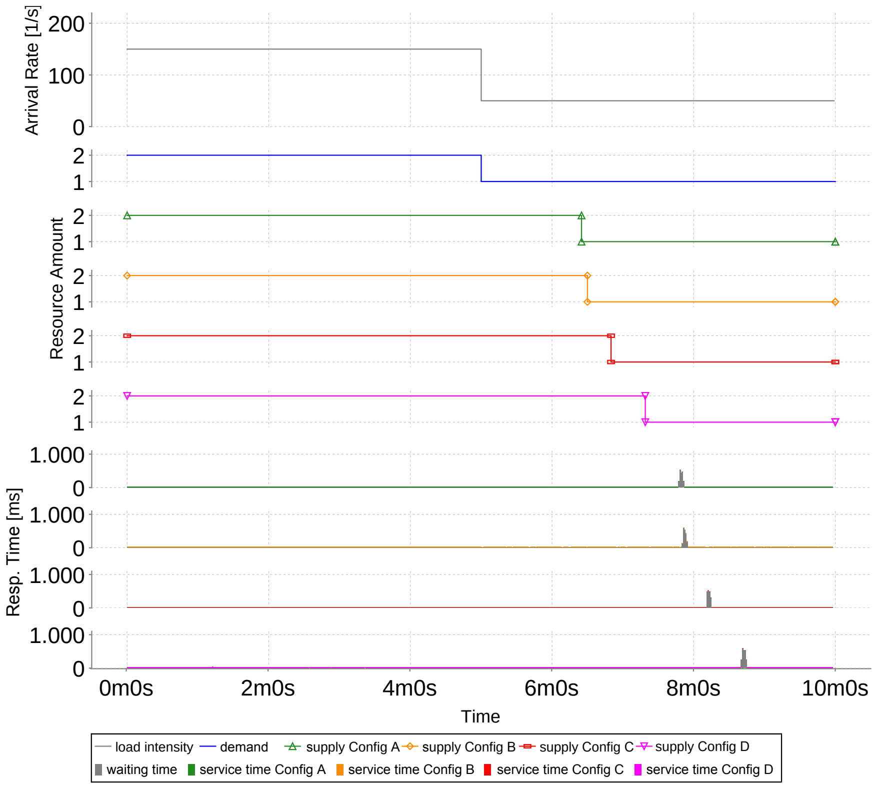


Figure 9.5: Evaluation of the τ_O metric. Load profile (top), induced resource demand (second graph) and measured resource supply and response times for increasing `condTrueDurDown` values (A: 5s, B: 10s, C: 30s, D: 60s).

`condTrueDurDown` [s]	5 (A)	10 (B)	30 (C)	60 (D)
τ_O [%]	14.3	15.0	18.4	23.3

Table 9.9: Measurement results for the τ_O metric

9.3.5 Experiment 5: Oscillations for Positive Jitter and Instability

The instability metric v captures the fraction of time, in which the sign of the change in demand is not equal to the sign of the corresponding change in the supply - in other words instable phases. The jitter metric j evaluates whether a system exhibits imperfect elasticity caused by too many or too few adaptations in the resource supply. We separate the jitter and instability experiments into a scenario with oscillations (Exp. 5) and with inertia (Exp. 6) in which the auto-scaler can not follow a quickly changing demand. In both cases the absolute value of the jitter metric increases for decreasing degrees of elasticity. However, for this experiment the sign of the jitter metric should be positive, and in contrast, for the latter case negative.

Load Profile

The load profile used for the first evaluation step is a constant load profile illustrated in Figure 9.6. The load profile is calibrated in a way that the intensity is 90% of the maximum intensity for one resource. Since the resource is utilized at a high level, rule based elasticity mechanisms tend allocate another instance, to be able to handle a possibly increasing load. As soon as the resource is provisioned the average resource utilization drops and - depending on the configuration - elasticity mechanisms may deallocate the superfluous resource again. Thus, this load profile tries to provoke unnecessary allocations and deallocations.

System Configurations

For the evaluation, the thresholds [`thresholdDown,thresholdUp`] are set to: [40,50]. With these thresholds the elasticity mechanism tends to allocate and deallocate a second resource. Furthermore, the `quietTime` is set to 30s, to allow faster reactions. The degree of elasticity of the system is changed by modifying the `condTrueDurUp/Down` parameters for both, the scale up and the scale down rule. For decreasing values of these parameters, the system reacts overly responsive, which leads to lower degrees of elasticity (to many unnecessary allocations). The jitter metric j and the instability metric v are evaluated for the following `condTrueDurUp/Down` values: 120s (Config. A), 60s (Config. B), and 30s (Config. C). The instability metric is computed based on an average provisioning time of 2 minutes.

Results

Figure 9.6 shows the elasticity behaviors for different values of the `condTrueDurUp/Down` parameter. It can be seen that due to decreasing degrees of elasticity for smaller `condTrue-DurUp/Down` values, the amount of unnecessary resource de-/allocation events increases. This is reflected by the metrics results shown in Table 9.10. As expected for decreasing degrees of elasticity, which means for smaller `condTrueDurUp/Down` values, both the jitter metric j and the instability metric v increase. Thus, the both metrics allow for ranking elastic systems whose imperfect elasticity is due to superfluous adaptations on an ordinal scale. When the smallest `condTrueDurUp/Down` is used (Config. C), the system even provisions a third resource in some cases although `maxInstances` is set to two. One explanation for this behavior is, that although a second instance is started before, the `quiettime` plus the `condTrueDurUp` was not enough time to allow the system to take advantage of the second instance and therefore a substitute instance is allocated. Shortly after the creation of the third instance, the system takes advantage of the second instance resulting in a quick deallocation.

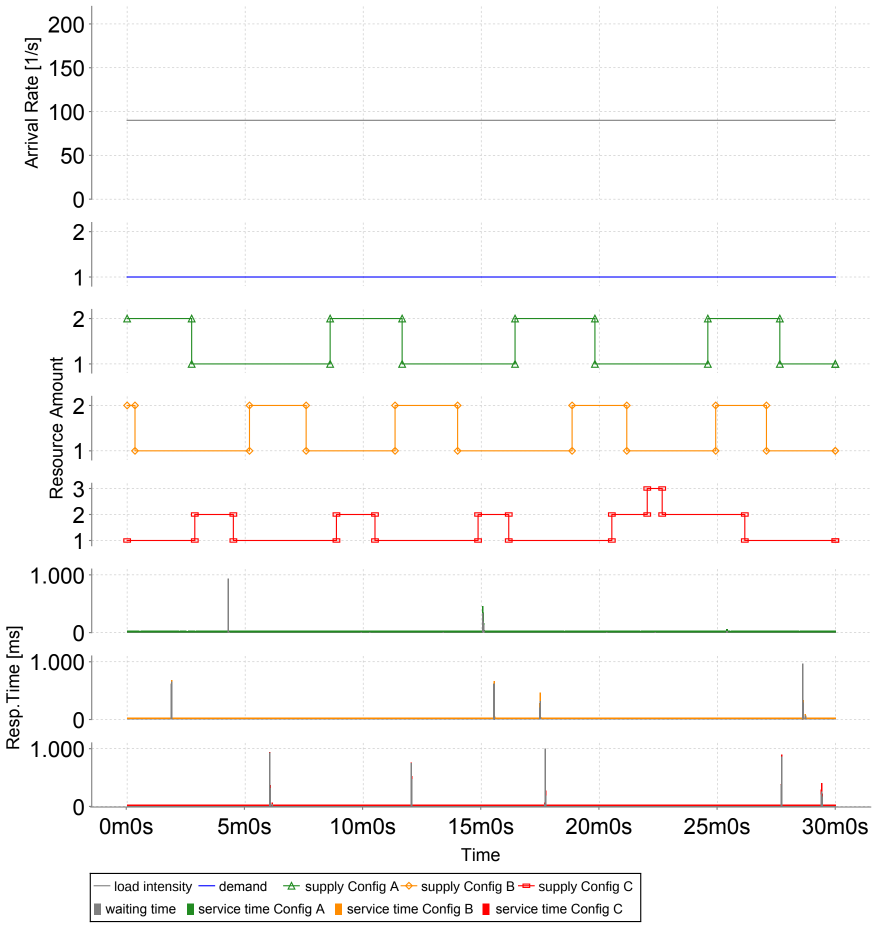


Figure 9.6: Evaluation of the jitter and instability metrics for superfluous adaptations. Load profile (top), induced resource demand (second graph) and measured resource supply and response times for decreasing `condTrueDurUp/Down` values.

`condTrueDurU/D` $[s]$	120(A)	60(B)	30(C)
$j\left[\frac{\#adap.}{min}\right]$	0.233	0.300	0.333
$\upsilon\,[\%]$	0.467	0.600	0.700

Table 9.10: Measurement results for the jitter metric j (positive) and instability metric υ.

9.3.6 Experiment 6: Inertia for Negative Jitter and Instability

Load Profile

The load profile used for showcasing different degrees of inertia is illustrated in Figure 9.7. The load profile changes between two intensity levels i_{low} and i_{high} in a repeated manner. Hereby, the Δt time for which the intensity is constant is seven minutes at the beginning. After every two intensity changes Δt is decrease by 15%. Like in the load profile for the timeshare evaluation, the load profile is calibrated in a way, that intensity i_{low} is half of the maximum intensity the system can withstand using one resource and i_{high} is 175% of the maximum intensity the system can withstand using one resource. Thus, with the correct amount of resources the system should be able to handle the load without stress, for any point in time.

System Configurations

The `quietTime` is set to 30s, to allow quick reactions. The degree of elasticity of the system is changed by modifying the `condTrueDurUp/Down` parameters for both, the scale up rule and the scale down rule. For increasing values of this parameter, the system reacts delayed leading to an increasing inertia. The metric j and v are evaluated for the following configurations: 5s (Conf. A), 30s (Conf. B) 60s (Conf. C) and 120s (Conf. D). The instability metric is computed based on an average provisioning time of 2 minutes.

Results

Figure 9.7 shows the elasticity behaviors for different values of the `condTrueDurUp/Down` parameters. It can be seen that due to increasing inertia caused by increasing `condTrueDur-Up/Down` values, the number of demand changes that the system is able to follow decreases. This is reflected by the jitter metric results shown in Table 9.11 as the absolute value of the metric j increases. Thus, the jitter metric j allows to rank elastic systems whose imperfect elasticity is due to missing adaptations on an ordinal scale. In contrast to the jitter metric, the instability rewards increasing inertia of the system as due to fewer instable phases. This is reflected by slightly increasing instability values. Thus, the instability metric is not capable to distinguish between oscillation and inertia as two different negative aspects of an elastic behavior.

`condTrueDurU/D` [s]	5(A)	30(B)	60(C)	120(D)
$j[\frac{\#adap.}{min}]$	-0.093	-0.107	-0.107	-0.133
$v[\%]$	0.665	0.630	0.620	0.582

Table 9.11: Measurement results for the jitter metric j (negative) and instability metric v

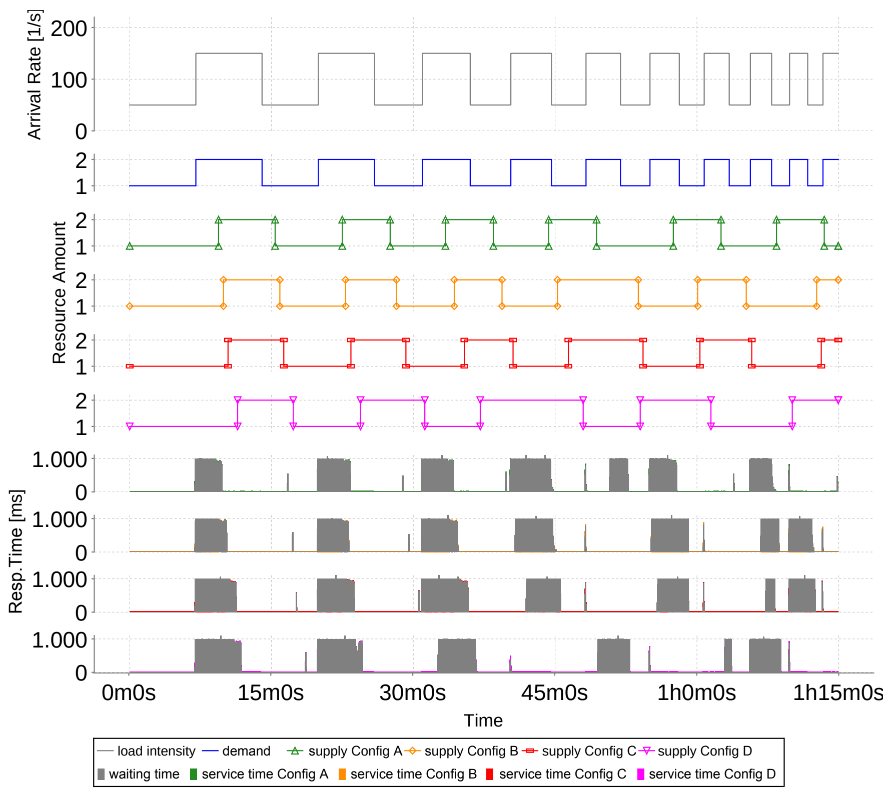


Figure 9.7: Evaluation of the jitter metric j and instability metric υ for missing adaptations. Load profile (top), induced resource demand (second graph) and measured resource supply and response times for increasing `condTrueDurUp/Down` values.

9.3.7 Summary of Metric Evaluation Experiments

The experiments for the evaluation of the elasticity metrics show that each metric allows ranking resource elastic systems on an ordinal scale reflecting the elastic behavior as expected. Varying different configuration parameters has different effects on different elasticity aspects and the corresponding metrics. For example, increasing the values for the `condTrueDurUp/Down` parameter values leads to lower degrees of elasticity in situations where the demand changes over time. Therefore, τ_U and τ_O increase with increasing `condTrueDurUp/Down` parameter values in the evaluation experiments. In contrast, the jitter metric j improves with higher `condTrueDurUp/Down` parameter for a stable workload, but not for a noisy one. The presented experiments indicate the load profile has a direct impact on how the configuration parameters of a trigger-based reactive auto-scaler should be chosen to optimize the trade off between reacting quick and remaining stable. Thus, the parameters need to be carefully tuned (or even learned) for any specific application scenario.

9.4 Benchmark Methodology Case Study

This section demonstrates the benchmarking capabilities for a realistic load profile and a representative amount of resources on the private and on a public cloud as described in Section 9.1.

The load profile (see Figure 9.8) used for this scenario is derived from a real intensity trace (see Figure 9.8a) previously used in Chapter 8. The trace features the amount of transactions on an IBM z196 Mainframe during February 2011 with a quarter-hourly resolution.

To reduce the experiment time, the first day is selected as the load profile and has been compacted from 24 hours to 6 hours. The load intensity within this load profile varies between two and 339 requests per second and contains about 2.7 million request timestamps, when adapted for a maximum of 10 one-core VMs in the private cloud setup. As a single AWS `m1.small` instance is capable of processing 71 requests per second, instead of 34 for the private cloud VMs, the load profile for public cloud experiments (see top of Figure 9.10a) is adapted to vary between five and 710 requests per second. This timestamp file contains about 5.6 million individual request submission timestamps and induces the same resource demand changes as the load profile for the private cloud setup. The System Analysis step has been evaluated for its reproducibility separately: After eight repetitions, the maximal deviation from the average processing capability per scaling stage was 6.8% for the public cloud and even lower for the private cloud setup.

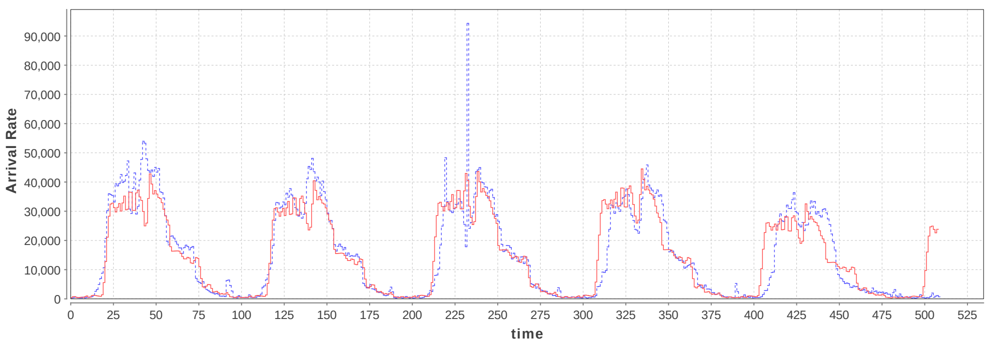
9.4.1 System Configuration

The resource elasticity of the cloud system is evaluated for different elasticity rule parameter settings as shown in Table 9.12: Configuration A serves as a baseline configuration for elasticity comparisons, as it is expected to exhibit the lowest degree of elasticity. Which one of the Configurations B, C and D shows the highest degree of elasticity is not directly visible from the parameters.

Table 9.12: Auto-scaler parameter configurations on private (A,B) and public (C,D) clouds

Configuration	`quietTime`	`condTrueDurUp`	`condTrueDurDown`	`threshOldUp`	`threshOldDown`
A	240s	120s	120s	90%	10%
B	120s	30s	30s	65%	50%
C	120s	60s	60s	65%	50%
D	60s	60s	60s	65%	40%

(a) Original (blue) and modeled (red) intensity [requests/15min] trace



(b) Compacted load profile of a single day adapted for one-core VMs in private cloud setup

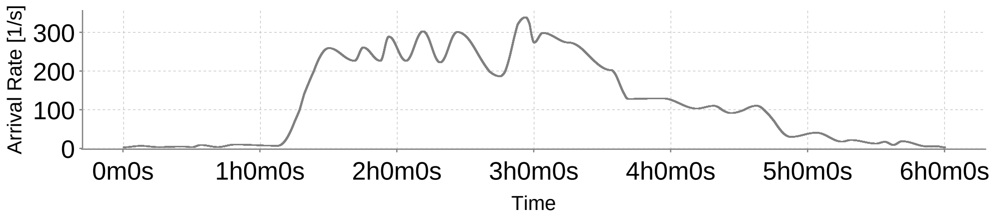


Figure 9.8: One day load profile derived from a real five day transaction trace

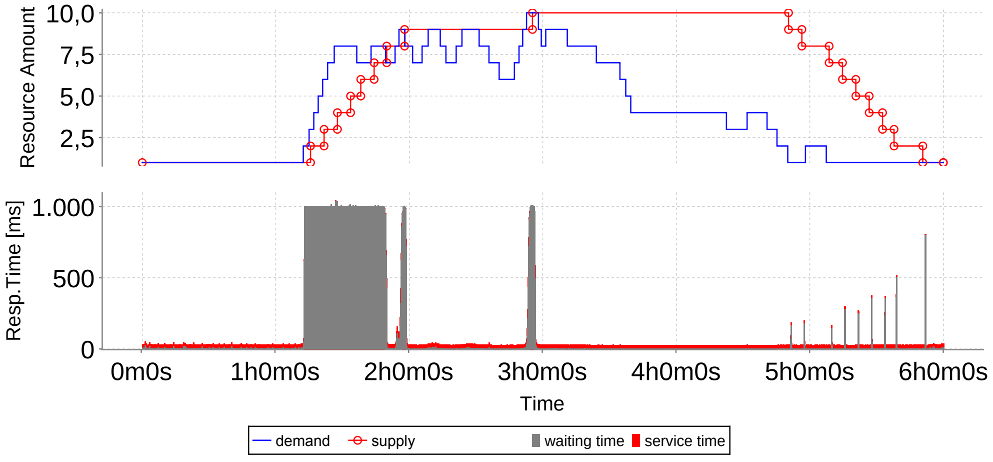
Table 9.13: Metric results for evaluated configurations on private cloud (A, B on Cloudstack) and on public cloud (C, D on AWS EC2)

Config-uration	θ_O [%]	θ_U [%]	a_O [#res.]	a_U [#res.]	τ_O [%]	τ_U [%]	υ [%]	j $\left[\frac{\#adap.}{min}\right]$	SLO viol. [%]
A	105.6	4.0	2.425	0.264	60.1	11.7	27.1	-0.067	20.3
B	14.1	1.9	0.815	0.080	48.7	6.5	26.4	-0.028	8.4
C	21.6	3.1	1.053	0.180	51.9	8.1	27.8	-0.033	9.1
D	31.8	1.4	1.442	0.049	57.6	4.7	28.6	-0.017	5.0

9.4.2 Results

Figure 9.9 and Figure 9.10 illustrate the exhibited elasticity of the four different Configurations A, B on the private cloud and C, D on the public cloud. Configuration D reacts fastest on the step increase starting at 90 min. The shape of resource allocations of Configuration B fits best to the demand changes. At the steep parts of the profile, when the systems are in

(a) Configuration A: Slow resource increase, very slow resource decrease



(b) Configuration B: Faster resource increase and decrease than Config. A

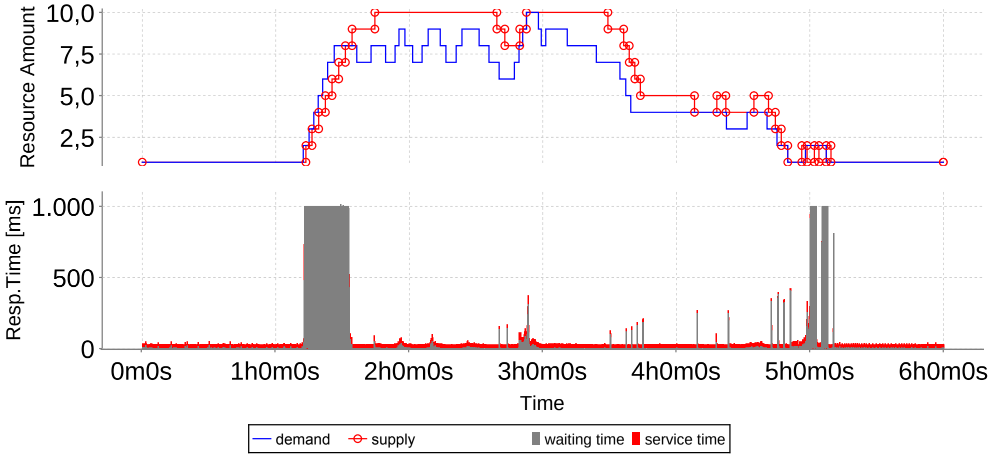
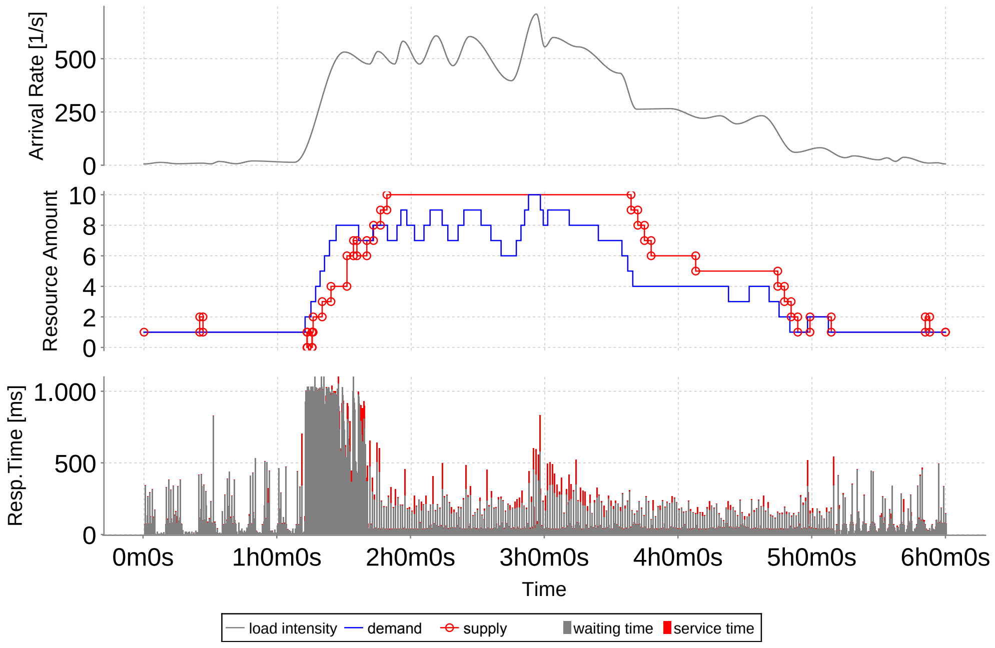


Figure 9.9: Elastic behavior for two elasticity rule parameter settings on CloudStack

(a) Configuration C on public cloud with adapted load profile



(b) Configuration D: Even faster resource in- and decrease compared to Configurations C

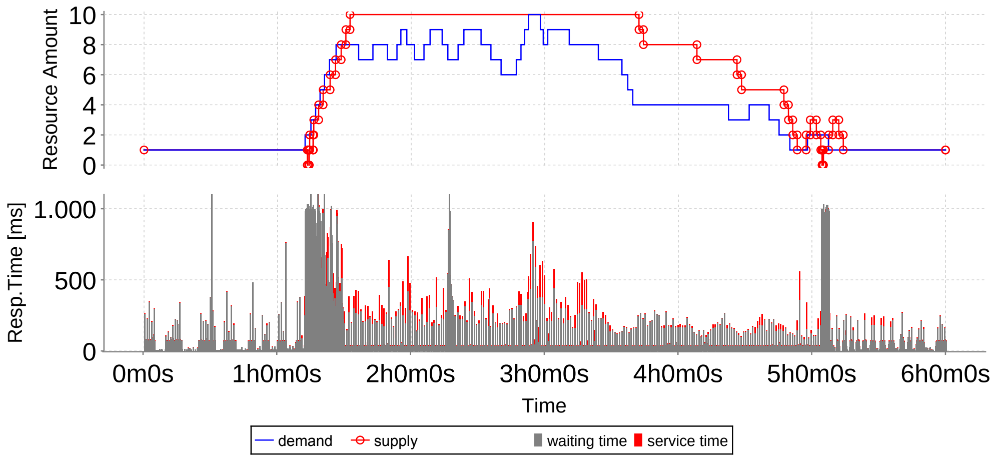


Figure 9.10: Elastic behavior for different elasticity rule parameter settings on AWS EC2

Table 9.14: Aggregated elasticity metrics

Configuration	AS deviation σ [%]	pairwise comp. κ [%]	elastic speedup ϵ_k
A	60.8	11.1	1.00
D	38.4	22.2	1.53
C	36.6	55.6	1.72
B	33.9	100.0	2.08

under-provisioned state, the request response time rises to the defined timeout of 1 second with the request violating the specified SLO. The response time graphs show higher and more variable response times for the public cloud experiments (C, D) compared to the private cloud experiments (A, B). Possible reasons are a higher performance variability due to imperfect isolation or overbooking on the public cloud.

Table 9.13 contains the metric results for the four configurations. Configuration B exhibits the lowest accuracy a_O metric value, whereas Configuration D achieves the lowest accuracy a_U result and is less than 5% of the experiment time in an underprovisioned state. All configurations exhibit a negative jitter j value as there are more demand changes than supply changes. The instability metric υ is not affected strongly as in general the stability behavior of the auto-scaler configuration is not changing significantly. Note that the percentage of SLO violating requests is correlated with the underprovisioning timeshare τ_U value and the amount of requests sent during under-provisioned states.

As for the aggregated elasticity metrics, the auto-scaler deviation σ [%], the pairwise competition κ [%], and the elastic speedup ϵ_k with no weights are summarized in Table 9.14. As expected from the visual representation, consistently for all aggregated metrics, Configuration B achieves the highest scores, followed by C, D, and lastly A. The unweighted aggregated elasticity metrics treat under-/over-provisioning equally. Thus, they are not correlated directly with the SLO violation rate.

9.5 Summary

In this evaluation chapter, we assess Contribution II presented in Chapter 5, namely the reproducibility of Bungee's system analysis and conduct for each individual elasticity metric a set of experiments on a tailored, synthetic load profile while altering the auto-scaler configurations of a threshold-based reactive approach. For all elasticity metrics, the parameter changes in the configurations are reflected according to intuition in the metric values.

In addition, the Bungee elasticity benchmark approach is applied to a complex realistic load scenario ranking different systems and configurations according to the exhibited degree of elasticity. Both, the System Analysis and the Elasticity Metrics, as well as the overall benchmarking methodology deliver reproducible results. In the realistic case study both in private and public cloud environments, we observe different timing and accuracy of elastic adaptations. We aggregate the resulting elasticity metrics in three different ways whereas all result in the same ranking consistently. Thus, this evaluation chapter answers RQ A.4 („How can the proposed elasticity metrics be measured in a reliable and repeatable way to enable fair comparisons and consistent rankings across systems with different performance?“).

Chapter 10

The Hybrid Auto-Scaler Chameleon in a Benchmark Competition

We evaluate the Chameleon auto-scaling mechanism by obtaining and analyzing the elasticity metrics we introduce in Section 5.2 and applying the Bungee measurement methodology (c.f. Section 5.5). First, we summarize the rationale behind the selected workloads used in the evaluation, and the auto-scaled application. Then, in Section 10.2, the five auto-scalers considered in the evaluation are introduced, before we step into an in-depth presentation and discussion of experiment results.

10.1 Workload and Application

To conduct representative experiments, authentic workloads with time-varying load intensity profiles are required. To this end, we collect existing traces from real-life systems that cover up to several months. For a feasible experiment run duration of up to 10 hours, we pick a randomly selected subset from the traces covering up to three days and accelerate the replay time by the factor 7.5 during the experiments so that one day in the traces corresponds to 3.2 hours in the experiments. This way, we trade experiment duration and covered time intervals for a realistic setup stressing the auto-scaling mechanisms. A higher time speed-up factor to replay more days within 10 hours experiments time would render the experiments unrealistic, e.g., as the induced changes in demand might exceed the provisioning delays and frequency of resource allocations technically supported by the cloud platforms.

- The FIFA World Cup 1998[1] trace is a widely known trace that represents the HTTP requests to the FIFA servers during the world championship between April and June 1998. This trace was analyzed in the paper of Arlitt and Tai [AJ00]. For our experiments, we use a sub-trace of three days.

- The BibSonomy trace consisting of HTTP requests to servers of the social bookmarking system BibSonomy (see the paper of Benz et al. [BHJ+10]) during April 2017. Here, we use 2 days for benchmarking the auto-scalers.

[1] FIFA Source: `http://ita.ee.lbl.gov/html/contrib/WorldCup.html`

- The IBM CICS transactions trace capturing four weeks of recorded transactions on a z10 mainframe CICS installation. From this trace, one weekday was extracted for the experiments.
- The German Wikipedia[2] trace containing the page requests to all German Wikipedia projects during December 2013. Here, we use two days from this trace.
- The Retailrocket [3] trace containing HTTP requests to servers of an anonymous real-world e-commerce website during June 2015. Similar to German Wikipedia, we use 2 days for the evaluations.

With exception of the German Wikipedia trace, all traces contain 96 data points per day, i.e., the traces contain 15 minutes averages of the arrival rates. In case of German Wikipedia, we transform the hourly samples using interpolation so that it ends up with 96 data points per day. We apply s-DLIM model extraction (see Section 4.3) with noise filtering and sinus-flanks as interpolating functions to obtain accurate models of the raw data. This step is required to adapt the original traces a) to the maximum supported load level, b) speed up the replay time, and c) adapt the load profile to the scaling behavior of the different platforms to trigger identical demand changes in all experiments for one trace.

The auto-scaling mechanisms are configured to monitor and auto-scale a CPU-intensive Java Enterprise application - an implementation of the LU worklet from SPEC's Server Efficiency Rating Tool SERT™2 - as a benchmark application. The application calculates the LU Decomposition [BH74] of a random generated $n \times n$ matrix, where n is the GET parameter of each HTTP request. The application is deployed on WildFly application servers in three private and public infrastructure cloud environments.

For the evaluation in our private cloud infrastructure, the application is deployed in an Apache CloudStack[4] cloud that manages virtualized Xen-Server hosts. This cloud environment is running in a cluster of 11 identical HPE servers. Eight of them are managed by CloudStack. Overbooking and hyper-threading are deactivated on the managed hosts. The last three servers are not part of the cloud and are used for hosting the software for the cloud management, as well as the benchmark framework: (i) the load-balancer (Citrix Netscaler[5]) and the cloud management system for CloudStack, (ii) Chameleon and the other auto-scaling mechanisms, and (iii) the load driver and the experiment controller. The specifications of each physical machine and of each VM can be seen in Table 10.1 and in Table 10.2, respectively.

In order to cover setups with background noise, the application is deployed in both the public AWS EC2 IaaS cloud and in the OpenNebula[6]-based IaaS cloud of the Distributed ASCI Supercomputer 4 (DAS-4) [BEdLm16] - a shared research cloud infrastructure. The specification of each VM in both setups is listed in Table 10.2. In the public AWS EC2 cloud, the quota for the number of VMs is given by the provider as 20 instances at maximum in parallel. In order to have comparable scaling ranges over the experiments, the amount of

[2]Wikipedia Source: `https://dumps.wikimedia.org/other/pagecounts-raw/2013/`
[3]Retailrocket Source: `https://www.kaggle.com/retailrocket/ecommerce-dataset`
[4]Apache CloudStack: `https://cloudstack.apache.org/`
[5]Citrix Netscaler: `https://www.citrix.de/products/netscaler-adc/`
[6]OpenNebula: `https://opennebula.org/`

Component	Server Specification
Model	HP DL160 Gen9
CPU	8 cores @2.6Ghz (Intel E5-2630v3)
Memory	32GB

Table 10.1: Specification of the servers.

Component	CloudStack	AWS (m4.large)	DAS-4
Operating System	CentOS 6.5	CentOS 6.5	Debian 8
vCPU	2 cores	2 cores	2 cores
Memory	4GB	8GB	2GB

Table 10.2: Specification of the VMs.

VMs is limited to 18 (20 VMs minus a load-balancer VM for the experiment and a domain controller VM for the WildFly cluster). The experimentally derived mappings showing how many requests can be handled by a certain number of VMs without violating the SLO is shown in Figure 10.1. Here, the x-axis shows the current number of VMs and the y-axis the maximal sustainable requests per second. The blue curve represents the private cloud scenario, the red one AWS EC2, and the black one DAS-4. The mappings have been extracted three times for public environments and averaged to account for the acceptable small variance present in shared environments.

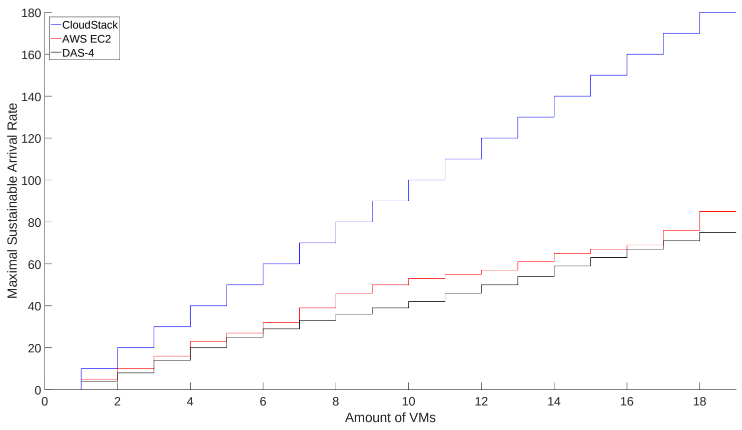


Figure 10.1: System Analysis results of all scenarios.

10.2 Competing Auto-Scalers

For the evaluation, we select five representative auto-scalers that have been published in the literature over the past decade. We identify two groups of auto-scalers differing in the way they treat the workload information. The first group are auto-scalers that build a predictive model based on long-term historical data [U$^+$05, IDCJ11, FPK14]. The second group consists of auto-scalers that only use recent history to make auto-scaling decisions [C$^+$09, AETE12]. The selected methods have been published in the following years: 2008 [USC$^+$08] (with an earlier version published in 2005 [U$^+$05]), 2009 [C$^+$09], 2011 [IDCJ11], 2012 [AETE12], and 2014 [FPK14]. This is a representative set of the development of cloud auto-scalers designs across the past 10 years. We describe each of these in more detail.

10.2.1 Reactive

Based on the work of Chieu et al. [C$^+$09], this auto-scaler realizes a scaling mechanism based on thresholds as provided, e.g., by AWS EC2. Here, the user can set a condition to add new instances in increments or based on the amount of currently running resources when the average utilization is higher than a specified threshold over a specified period. Similarly, the user can set a condition to remove instances. An additional cool-down parameter defines a duration after an action, during which the metrics are not evaluated. This allows to avoid possible oscillations by delaying the next possible action. In our experiments, we set the cool-down parameter to 0 and the condition-true period to 2 minutes for both directions. As thresholds, we use 80% CPU utilization for scaling up and 60% for scaling down while adding/removing the fixed amount of 1 unit.

10.2.2 Adapt

Ali-Eldin et al. [AETE12] propose an autonomic elasticity controller that changes the number of VMs allocated to a service based on both monitored load changes and predictions of future load. We refer to this technique as Adapt. The predictions are based on the rate of change of the request arrival rate, i.e., the slope of the workload, and aims at detecting the envelope of the workload. The designed controller adapts to sudden load changes and prevents premature release of resources, reducing oscillations in the resource provisioning. Adapt tries to improve the performance in terms of number of delayed requests and the average number of queued requests, at the cost of some resource over-provisioning.

10.2.3 Hist

Urgaonkar et al. [USC$^+$08] propose a provisioning technique for multi-tier Internet applications. The proposed methodology adopts a queueing model to determine how many resources to allocate in each tier of the application. A predictive technique based on building Histograms of historical request arrival rates is used to determine the amount of resources to provision at an hourly time scale. Reactive provisioning is used to correct errors in the long-term predictions or to react to load bursts caused by unanticipated flash crowds. The authors also propose a novel data center architecture that uses Virtual Machine (VM) monitors to reduce provisioning overheads. The technique is shown to be able to improve responsiveness of the system, also in the case of a flash crowd. We refer to this technique as Hist.

10.2.4 Reg

Iqbal et al. propose a regression-based auto-scaler (hereafter called Reg) [IDCJ11]. This auto-scaler has a reactive component for scale-up decisions and a predictive component for scale-down decisions. When the capacity is less than the load, a scale-up decision is taken and new VMs are added to the service in a way similar to Reactive. For scale-down, the predictive component uses a second order regression to predict future load. The regression model is recomputed using the complete history of the workload each time a new measurement is available. When the current load is lower than the provisioned capacity, a scale-down decision is taken using the regression model. This auto-scaler does not perform on the level of other mechanisms in our experiments due to two factors; first, building a regression model for the full history of measurements for every new monitoring data point is a time consuming task. Second, distant past history becomes less relevant as time proceeds. After contacting the authors, we have modified the algorithm such that the regression model is evaluated only for the past 60 monitoring data points.

10.2.5 ConPaaS

ConPaaS, proposed by Fernandez et al. [FPK14], scales a web application in response to changes in throughput at fixed intervals of 10 minutes. The predictor forecasts the future service demand using standard time series analysis techniques, e.g., Linear Regression, Auto Regressive Moving Averages (ARMA), etc. The code for this auto-scaler is open source. This is the only auto-scaler out of the five with an open-source implementation that we used in our evaluation.

The auto-scalers are running in a Python venv[7] environment as a web application with a REST-interface based on Flask[8]. The interface of the auto-scalers is called every 2 minutes via a Java-based wrapper. Each auto-scaling mechanism receives a set of input values and returns the amount of VMs that have to be added or removed. The input consists of the following parameters: (i) the accumulated number of requests during the last 2 minutes, (ii) the estimated resource demand per request determined by LibReDE as used in Chameleon, and (iii) the number of currently running VMs. The interface also provides the average CPU utilization across the running instances as input. However, the other proactive auto-scalers do not use it.

[7]Python venv: `https://docs.python.org/3/library/venv.html`
[8]Flask: `http://flask.pocoo.org/`

10.3 Experiment Results

In this section, we benchmark Chameleon and the other auto-scalers. We have conducted 7 different sets of experiments with 6 auto-scalers. Section 10.3.1 explains how to interpret the results of the measurements. In Section 10.3.2, the variability of the auto-scalers is investigated. Afterwards, in Section 10.3.3, the measurements in the private cloud and the public cloud are presented and compared. We evaluate the impact of forecasting in Chameleon in Section 10.3.4. In Section 10.4, the results of all measurements are combined, illustrated and discussed. Afterwards, we discuss threats to validity and, finally, conclude the evaluation with a list of key findings. Detailed results of the experiments per selected workload can be found in the appendix.

10.3.1 Introduction to the Results

We first introduce the format in which results are presented before discussing the detailed results. To this end, a single experiment of the Retailrocket trace is explained and depicted in Figure 10.2. For all measurements, we set the timeout limit of requests to 5000 milliseconds. Furthermore, the condition that 95% of all response times should be less than 4000 milliseconds is set as Service Level Objective (SLO). In Figure 10.2, the timeout limit is represented by a grey dashed line and the SLO violation border as blue line in the lower part of the diagram. The x-axis shows the time of the measurements in minutes and the y-axis the response time in milliseconds. Green cycles represent requests that meet the SLO, whereas red dots represent requests that violate the SLOs. The upper part of the figure shows the scaling behavior of the auto-scaler. Here, the x-axis shows the time of the measurement in minutes since the beginning of the experiment, whereas the y-axis shows the amount of concurrently running VMs. The blue curve represents the VMs that the auto-scaler supplied to the system. The black dashed curve shows the ideal number of supplied VMs, determined beforehand by the experiment controller Bungee, which is unknown to the auto-scalers and is used to calculate the metrics afterwards. For all measurements, the auto-scalers start with 1 VM and therefore, they start in an under-provisioned state.

In the example of Figure 10.2, from minute 0 to minute 40 the system in an under-provisioned state. Hence, there are a lot of SLO violations depicted by a cloud of red dots. During the period from minute 40 to minute 185 the system has at least as many VMs as required or more. Therefore, the average response period during this time is 800 ms. The few red dots that occur in that period can be explained by the fact that VMs are shutdown when scaling down and therefore, some requests either timeout or experience a degraded performance.

The experiment results for the German Wikipedia trace are shown Figure 10.3. This diagram shows the Chameleon measurement (top left) compared with the measurements for all competing auto-scalers: Reactive (top right), Adapt (middle left), Hist (middle right), ConPaaS (bottom left) and Reg (bottom right). For each auto-scaler, the x-axis shows the time of the measurement in minutes; the y-axis shows the number of concurrently running VMs. The blue curves represent the supplied VMs of each auto-scaler; the black dashed curves represent the required amount of VMs. The interpretation of the curves are the same as in the single experiment example.

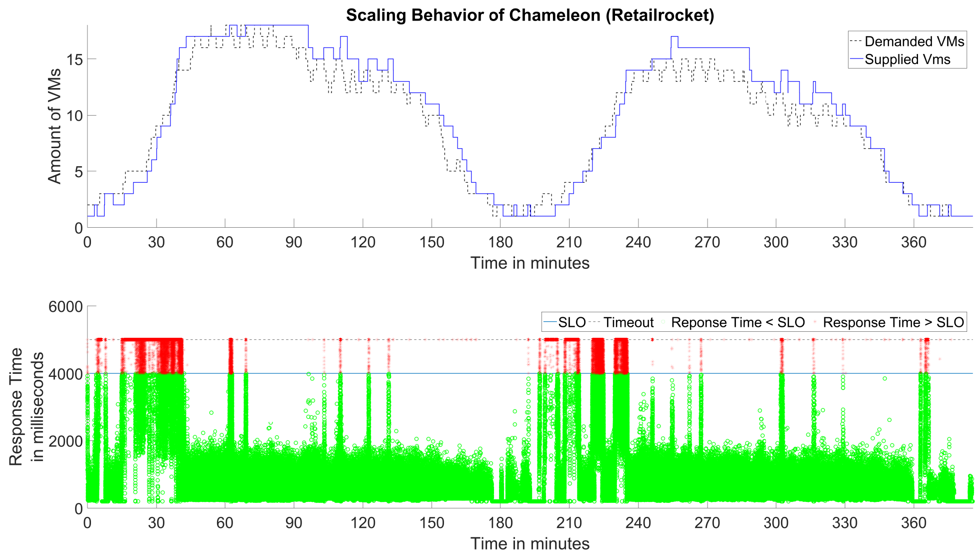


Figure 10.2: Scaling behavior of Chameleon for the Retailrocket trace.

When comparing the scaling behavior of the different auto-scalers for the German Wikipedia trace, a first observation is that the auto-scalers can be grouped into two categories: (i) tendency to over-provision the system (Hist, Reactive, Adapt and Chameleon), (ii) tendency to under-provision the system (ConPaaS and Reg). Furthermore, Reg and ConPaaS have a high rate of oscillations during the measurement. Chameleon is the first auto-scaler that meets the demand surge in the beginning of the experiment and then tends to allocate slightly more VMs than required for the remaining time. Adapt, for instance, has a similar behavior as Chameleon but it allocates more VMs. Reactive allocates almost the right amount of VMs during the increasing and constant load, but is too slow when scaling down. In contrast, Hist roughly meets the demand and holds the amount of allocated VMs for about 30 to 60 minutes after which it drops to the current demand.

To provide a quantitative comparison as in Table 10.3, we compute the elasticity metrics (see Section 5.2), the aggregated elasticity metrics without custom weight (see Section 5.3) and report user-oriented quantities like average and median response times, absolute accounted instance minutes, the average number of VMs as well as the absolute number of resource adaptations. The first column denotes the various metrics and the following ones the values obtained by the different auto-scalers with the last one corresponding to the no auto-scaling scenario. The rows of the table show the values of different metrics with the best value highlighted in bold. For the elasticity metrics, as well as for the metrics SLO violations, auto-scaling deviation, instance minutes, average response time and median response time it generally holds that the lower the value the better the auto-scaler performs. In contrast, for the pairwise competition and elastic speedup, a higher value is better.

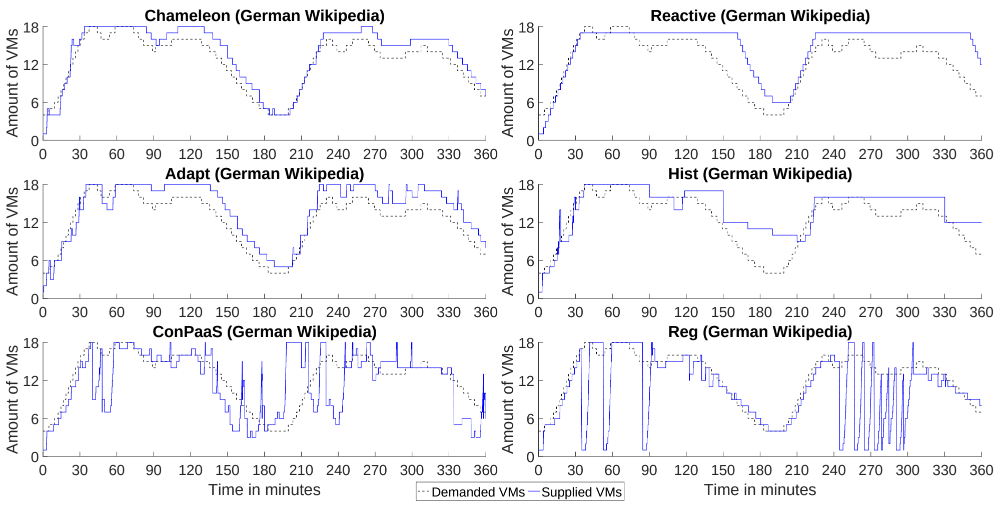


Figure 10.3: Comparison of the auto-scalers for the German Wikipedia trace.

Metric	Chameleon	Adapt	Hist	ConPaas	Reg	Reactive	No Scaling
θ_U (accuracy$_U$)	**1.57**%	1.68%	2.37%	14.69%	16.08%	2.10%	25.93%
θ_O (accuracy$_O$)	11.15%	17.51%	33.55%	15.67%	**4.34**%	28.94%	19.38%
τ_U (time share$_U$)	**5.70**%	9.16%	12.75%	47.41%	51.04%	12.27%	70.48%
τ_O (time share$_O$)	73.25%	80.94%	71.95%	32.07%	**25.24**%	80.77%	26.28%
υ (instability)	5.83%	7.09%	4.75%	12.66%	12.88%	4.97%	**2.94**%
ψ (SLO violations)	**2.95**%	15.47%	11.86%	59.73%	80.71%	7.15%	85.95%
σ (as deviation)	**39.49**%	45.24%	42.75%	62.54%	81.71%	46.67%	88.13%
κ (pairwise comp.)	**77.78**%	50.00%	50.00%	41.67%	41.67%	52.78%	36.11%
ϵ (elastic speedup)	**2.30**	1.78	1.51	0.91	1.19	1.56	1.00
#Adaptations	61	66	26	112	102	49	0
Avg. #VMs	10.795	12.343	11.571	11.191	9.3761	10.451	9
Instance minutes	5103.50	5267.70	5416.00	4380.70	3913.00	5551.30	**3471.00**
Avg. response time	**0.62** s	1.22 s	1.06 s	3.31 s	4.20 s	0.82 s	4.38 s
Med. response time	**0.43** s	0.48 s	0.48 s	5.00 s	5.00 s	0.46 s	5.00 s

Table 10.3: Metric overview for the German Wikipedia trace.

When comparing individual elasticity metrics only the aspect characterized by the respective metric is considered. For instance, in the German Wikipedia scenario, Chameleon has the best values for under-provisioning accuracy, under-provisioning time share, and SLO violations. Reg has the best values for under-provisioning accuracy and under-provisioning time share, but the highest amount of SLO violations compared to all other auto-scalers. Therefore, we evaluate the performance of the auto-scalers based on the auto-scaling deviation, the pairwise competition, and the elastic speedup. Chameleon has the best values for each of these unweighted aggregate metrics. This is supported by also looking at the average and median response time for which Chameleon also exhibits the best values among the compared auto-scalers.

10.3.2 Auto-Scaler Performance Variability

Besides good elasticity metrics, an essential characteristic of auto-scalers is that they exhibit low variability in their scaling behavior. Hence, we investigate how the scaling behavior varies when rerunning the same experiment multiple times and comparing the elasticity metrics and SLO violations. As the IBM trace consists of one day only, we repeat the experiment three times for each auto-scaler. The scaling behaviors are depicted in Figure 10.4. This diagram has the same structure as described in Section 10.3.1 with the exception that the blue solid curve shows the supplied VMs in experiment 1, the dashed cyan curve with dots the supplied VMs in experiment 2, and the red solid curve with circles the supplied VMs in the last experiment. Table 10.4 shows the metric value ranges of the achieved elasticity metrics and the SLO violations. The lowest level of variability is achieved by Chameleon and Reactive, followed by ConPaaS, Reg and Adapt. In contrast to the mentioned auto-scalers, Hist exhibits the highest variability. In general, all auto-scalers exhibit a reasonably low degree of performance variability. This allows to analyze the different auto-scalers in the context of the conducted experiments without the threat of non-reproducible results.

10.3.3 Auto-Scaling in Private vs. Public IaaS Clouds

In this section, we run the experiments with the FIFA World Cup 1998 trace in the AWS EC 2 and in the DAS-4 environment in order to evaluate the behavior of the different auto-scalers when running in a public cloud. In contrast to our private cloud, we observe a high

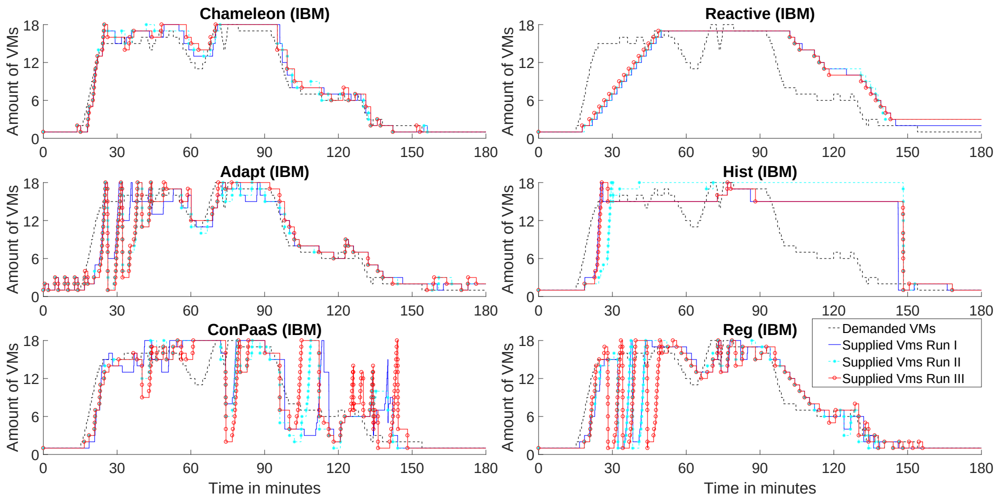


Figure 10.4: Comparison of the auto-scalers for the IBM trace.

Metric	Chameleon	Adapt	Hist	ConPaaS	Reg	Reactive
θ_U (accuracy$_U$)	**5.3% ± 0.1**	7.6% ± 1.2	6.0% ± 0.4	13.1% ± 0.6	12.4% ± 1.0	7.9% ± 0.3
θ_O (accuracy$_O$)	**7.7% ± 0.4**	33.3% ± 0.9	88.6% ± 16.0	21.5% ± 1.7	7.8% ± 0.9	60.7% ± 11.5
τ_U (time share$_U$)	**14.9% ± 0.6**	23.7% ± 5.1	17.2% ± 7.2	36.0% ± 1.7	29.8% ± 2.6	19.7% ± 0.5
τ_O (time share$_O$)	43.1% ± 1.3	52.4% ± 4.3	52.9% ± 12.5	**26.9% ± 4.2**	27.9% ± 1.9	60.0% ± 2.6
υ (instability)	8.8% ± 0.3	14.2% ± 0.6	**7.6% ± 0.2**	14.1% ± 1.4	13.6% ± 1.5	8.2% ± 0.1
ψ (SLO violations)	**12.1% ± 2.5**	37.1% ± 7.2	14.8% ± 4.6	40.7% ± 3.3	67.7% ± 6.4	29.9% ± 2.2

Table 10.4: Metric value ranges (IBM trace).

performance variation for each VM and a high degree of background noise in AWS. The resulting scaling behavior of a subset of the auto-scalers is depicted in Figure 10.5. The figures are structured as explained in Section 10.3.1; the left column shows Chameleon, Reactive and Adapt in the private cloud scenario and the right column shows the same auto-scalers in the public AWS EC2 scenario.

While Chameleon and Adapt scale in a similar manner in both scenarios, a bigger difference is observed for Reactive, as the diagrams in Figure 10.5 show. In contrast to the proactive auto-scalers that make decisions based on the amount of requests and the resource demands, the scaling decisions of Reactive are only based on the observed CPU load of the system. While the system is in an under-provisioned state, the CPU load drops significantly for a certain time and Reactive scales down as since CPU usage indicates a low load. After the CPU load starts increasing, Reactive tries to scale up the system again. Such drops in the CPU load seem to follow a scheme. Therefore, we assume that AWS EC2 performs migrations in the background for migrating the VMs from overloaded hosts to hosts with less load. After such migrations, the CPU usage drops given that more resources are available on the new host.

The metrics for all auto-scalers are listed in Table 10.5. While in the private scenario, Chameleon achieved the best value for the auto-scaling deviation metric, in the public scenario, Chameleon achieved the best values for the pairwise competition and elastic speedup metrics, in addition to the auto-scaling deviation metric. In the private scenario, Reactive has the best score for pairwise competition and elastic speedup, while in the public scenario, it exhibits average performance for the aggregate metrics.

10.3.4 Side-Evaluation: The Impact and Quality of Forecasting in Chameleon

In order to quantify the performance of the reactive cycle and the proactive cycle (in particular its forecast component) separately, we perform two additional measurements on the Retailrocket trace. In the first experiment, the scaling component of the reactive cycle is deactivated, that is, Chameleon scales the system solely proactively relying on the forecast component and without applying the strategies for resolving reactive and proactive scaling events mentioned in Section 6.4. The average MASE metric (see Section 6.2) during this experiment was 31.37%. In the second experiment, we use Chameleon without any historical data and thus the forecast component may not be able to provide trustworthy forecasts. Indeed, as the average MASE-value is 51.10% and the minimum is 30.14%, which

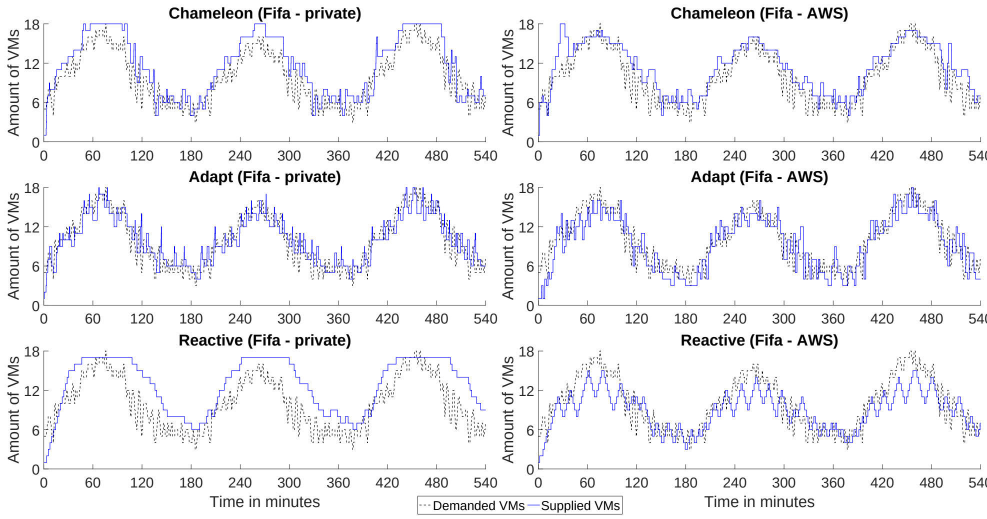


Figure 10.5: Comparison of the auto-scalers in both the private cloud scenario and public AWS EC2.

Metric	Chameleon		Adapt		Reactive		No Scaling
	private	AWS EC2	private	AWS EC2	private	AWS EC2	
θ_U (accuracy$_U$)	3.23%	1.64%	7.09%	12.93%	**1.56**%	13.36%	14.75%
θ_O (accuracy$_O$)	21.95%	21.31%	11.81%	10.85%	40.20%	**9.19**%	27.41%
τ_U (time share$_U$)	13.09%	11.04%	41.47%	52.02%	**5.20**%	57.49%	48.93%
τ_O (time share$_O$)	74.05%	67.92%	35.81%	32.43%	87.14%	**25.49**%	44.77%
υ (instability)	16.36%	15.95%	21.56%	17.98%	14.68%	19.02%	**12.66**%
ψ (SLO violations)	8.65%	5.04%	43.67%	21.56%	**2.70**%	49.30%	62.14%
σ (as deviation)	43.88%	**39.81**%	49.68%	43.31%	46.76%	54.80%	66.83%
κ (pairwise comp.)	61.11%	**69.44**%	55.56%	52.78%	66.67%	44.44%	44.44%
ϵ (elastic speedup)	1.58	**1.93**	1.33	1.21	**1.93**	1.27	1.00
#Adaptations	122	126	159	169	81	254	0
Avg. #VMs	9.532	9.649	10.692	9.3815	10.488	8.8372	9
Instance minutes	6500.20	6360.30	5562.00	5350.30	7262.50	**5004.00**	5190.00
Avg. response time	0.89 s	0.62 s	2.60 s	1.32 s	**0.60** s	2.68 s	3.32 s
Med. response time	0.45 s	0.26 s	1.63 s	**0.25** s	0.45 s	3.23 s	5.00 s

Table 10.5: Metric overview for the private vs AWS EC2 cloud scenario.

is still greater than the configured tolerance value, all proactive events are considered as not trustworthy, that is, only reactive events are scheduled and the proactive ones are skipped. Thus, this experiment can be considered as using Chameleon without the proactive cycle, although the proactive cycle is still active as it has to estimate the resource demand that is used in the reactive cycle.

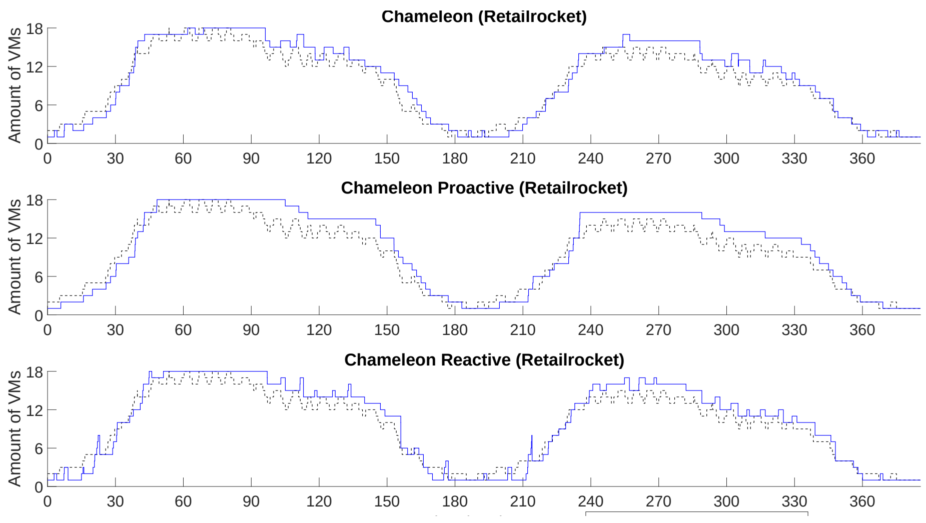


Figure 10.6: Comparison of standard Chameleon (top) vs. Chameleon with only proactive adaptations (middle) vs. Chameleon with only reactive adaptations (bottom).

Metric	Chameleon			No Scaling
	original	proactive	reactive	
θ_U (accuracy$_U$)	6.37%	**6.17**%	9.46%	17.68%
θ_O (accuracy$_O$)	11.71%	14.79%	**9.87**%	110.38%
τ_U (time share$_U$)	19.53%	**19.36**%	20.35%	55.75%
τ_O (time share$_O$)	51.34%	63.14%	53.80%	**38.81**%
υ (instability)	10.98%	9.38%	11.41%	**7.22**%
ψ (SLO violations)	8.68%	8.69%	**8.51**%	82.56%
σ (deviation)	**35.59**%	41.34%	37.22%	90.52%

Table 10.6: Metric overview for the components of Chameleon when executed in isolation.

The resulting two scaling behaviors, in addition to the original measurement, are depicted in Figure 10.6. Here, the first row shows Chameleon without any restrictions, the second one shows Chameleon using only the proactive cycle, and finally, the third row shows Chameleon without proactive events. Each diagram is structured as described in Section 10.3.1. The elasticity metrics, the SLO violations, and the auto-scaling deviation are listed in Table 10.6. Here, the columns represent the auto-scalers, the rows the metrics, and the best values are highlighted in bold. Although the difference between the standard Chameleon, Chameleon with only proactive adaptations, and Chameleon with only reactive adaptations is quite low, the combination of both cycles (57% proactive events and 43% reactive events) increases the performance of Chameleon. Despite the assumption (i) from Section 6.5 that Chameleon needs historical data, Chameleon achieves a similar score without using historical data.

Besides the improved scaling performance, the combination of both cycles reduces the reliability of scaling decisions as for instance proactive events based on forecast values are replaced by reactive events. Still, we identify potential for further improvements in the proactive cycle that relies on the selected forecast method. From time to time, tBATS and ARIMA forecast methods deliver inaccurate results or do note complete computation efficiently in time. Our idea towards leveraging this potential is the hybrid forecasting method Telescope sketched in Chapter 7 with promising exemplary results and an auto-scaling case study in Chapter 11.

10.4 Overall Evaluation Results

As the previous subsections show only selected subsets of the experiment results, an overview of all results is presented and discussed here. Table 10.7 shows the conducted experiments in terms of considered traces and cloud environments. In our experiment design, each auto-scaler is observed for 18 days on 5 different traces where one day takes 3.2 hours. The experiments took over 400 hours in total, during which about 107 million requests were sent and 5000 adaptations were performed by the auto-scalers. In other words, during one hour about 275.000 requests arrived at the system on average and the system experienced about 13 adaptions on average.

Scenario	**IBM**	**German Wikipedia**	**FIFA World Cup 1998**	**BibSonomy**	**Retailrocket**
CloudStack	3 × 1 day	2 days	3 days	2 days	2 days
AWS EC2	-	-	3 days	-	-
DAS-4	-	-	3 days	-	-

Table 10.7: Overview of the conducted experiments.

Table 10.8 shows the average metrics over all traces. Here, each row represents a metric and each column an auto-scaler. As previously, the best values are highlighted in bold. When comparing the auto-scalers based on the auto-scaling deviation, the lowest deviation from the theoretically optimal auto-scaler for all experiments is achieved by Chameleon, followed by Hist, Adapt, Reactive, ConPaaS, and finally, Reg. In a pairwise competition, the most points are collected by Chameleon, followed by Hist, Reactive, Adapt, Reg, and finally, ConPaaS. When taking the elastic speedup into account, the best performance is achieved by Chameleon, followed by Reactive, Adapt, Reg, Hist, and finally, Hist. To summarize, Chameleon achieves the best results for all three aggregate metrics. This is seen in the last three lines of the table, which show for each aggregate metric, how many times the auto-scalers dominate the respective metric. The maximum number is 7 as there are 7 different experiments per auto-scaler. As in the case of the pairwise competition, some auto-scalers achieved the same score, resulting in the sum of this row being greater than 7.

In order to rank the auto-scalers, we compute the average rank of each auto-scaler over all experiment setups. First, we compute the three ranks of each auto-scaler when considering the auto-scaling deviation, the pairwise competition, and the elastic speedup over all experiments. Then, we determine the average rank of each auto-scaler as the arithmetic mean

Metric	**Chameleon**	**Adapt**	**Hist**	**ConPaaS**	**Reg**	**Reactive**
$\overline{\theta}_U$ (avg. accuracy$_U$)	**3.63**%	6.45%	4.70%	15.55%	15.69%	6.98%
$\overline{\theta}_O$ (avg. accuracy$_O$)	17.88%	19.94%	52.64%	25.98%	**10.51**%	34.47%
$\overline{\tau}_U$ (avg. time share$_U$)	**13.32**%	30.43%	22.75%	42.04%	43.71%	25.41%
$\overline{\tau}_O$ (avg. time share$_O$)	65.06%	51.41%	62.35%	41.69%	**33.42**%	62.08%
$\overline{v}$ (avg. instability)	13.91%	16.60%	**11.95**%	17.42%	17.02%	12.99%
$\overline{\psi}$ (avg. SLO violations)	**10.29**%	32.76%	15.59%	44.11%	60.16%	21.96%
$\overline{\sigma}$ (avg. as deviation)	**39.63**%	46.90%	46.43%	54.03%	63.46%	48.14%
$\overline{\kappa}$ (avg. pairwise comp.)	**69.44**%	50.00%	58.33%	36.51%	42.46%	55.56%
$\overline{\epsilon}$ (avg. elastic speedup)	**2.02**	1.48	1.38	1.10	1.41	1.49
#Win deviation	**7**	0	0	0	0	0
#Win Pairwise Comp.	**5**	0	1	0	0	3
#Win Elasticity Score	**5**	0	0	0	0	2

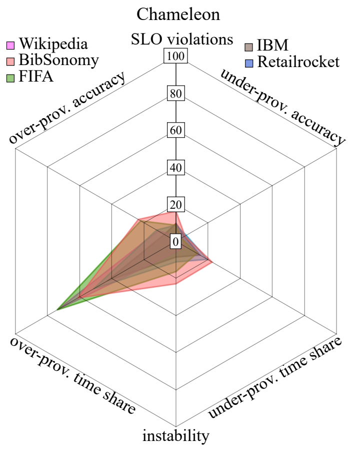
Table 10.8: Average metrics over all experiments.

Trace	**Chameleon**	**Adapt**	**Hist**	**ConPaaS**	**Reg**	**Reactive**
BibSonomy	**1.33**	5.33	2.00	5.00	4.33	2.00
FIFA World Cup 1998 (AWS EC2)	**1.00**	3.00	3.33	5.33	4.33	4.00
FIFA World Cup 1998 (DAS-4)	**1.33**	2.33	3.00	5.66	5.33	3.00
FIFA World Cup 1998 (private)	**1.67**	3.67	3.00	5.33	5.67	**1.67**
IBM	**1.00**	4.67	3.33	3.67	3.67	4.67
German Wikipedia	**1.00**	2.67	3.00	5.33	5.33	3.00
Retailrocket	**1.00**	2.33	3.33	6.00	4.33	4.00
Average Ranking	**1.19**	3.43	3.00	5.19	4.71	3.19

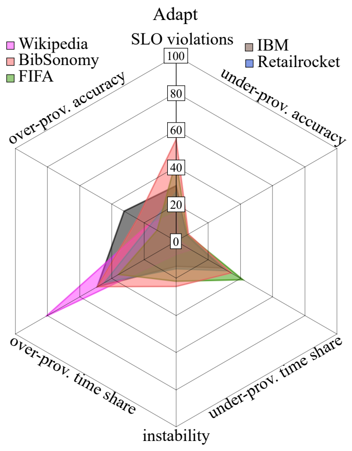
Table 10.9: Average ranking for each experiment over the three competitions.

over the three ranks. The ranks are listed in Table 10.9. Here, each column represents an auto-scaler and each row an experiment. The entries in each cell represent the rank in the associated experiment. As usual, the best values are highlighted in bold. Chameleon has the smallest average rank of 1.19. This means that Chameleon exhibits the best rank on average among all traces and experiments. The second best auto-scaler based on this ranking is Hist, followed by Reactive, Adapt, Reg, and finally, ConPaaS. Except for Chameleon, the auto-scalers in competition show a significant variance in their ranks over the experiments, that is, none of competing auto-scalers , with exception of Chameleon, outperforms the others in all 7 experiment rows.

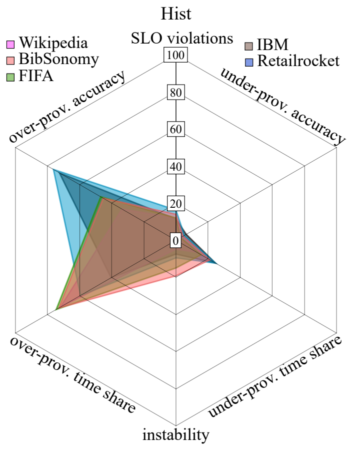
In order to visualize the scaling behavior of all auto-scalers, Figure 10.7 shows a spider chart, also known as radar chart, of each auto-scaler. Each spiderweb contains six edges, where the edges represent the elasticity metrics and the SLO violations, and shows the results for each trace in the private cloud scenario. The smaller and thinner the stretched areas are, the better the respective auto-scaler performs. Based on these diagrams, we can conclude that Chameleon tends to slightly over-provision allocating slightly more VMs than required. In contrast, Hist tends to over-provision but has a worse over-provisioning accuracy than Chameleon. Reactive has the worst over-provisioning time share and the second worst over-provisioning accuracy. Adapt tends to a balance of under- and over-provisioning with a slight tendency to over-provision. In the under-provisioned states, Adapt has only a few VMs less than required Reg and ConPaaS have (due to their oscillations) no tendency to either under-provision or over-provision the system. Both auto-scalers exhibit the highest SLO violations. Besides comparing the tendencies, the stability of each auto-scaler over all traces can be investigated. Chameleon exhibits a stable scaling behavior for all five traces as all areas are oriented in one direction and the areas have less deviation to each other. The other auto-scalers have either areas with different orientation or areas with a higher variance than Chameleon. Hence, the other auto-scalers can be seen as less stable than Chameleon.



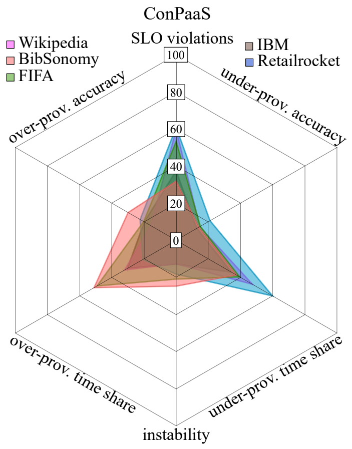
(a) Metric overview Chameleon.



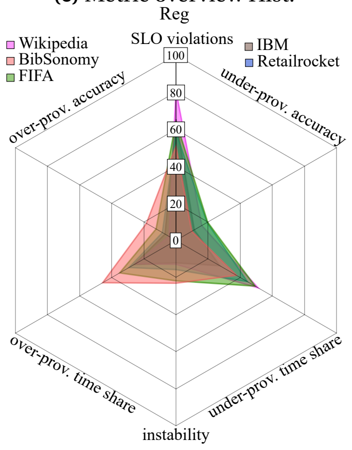
(b) Metric overview Adapt.



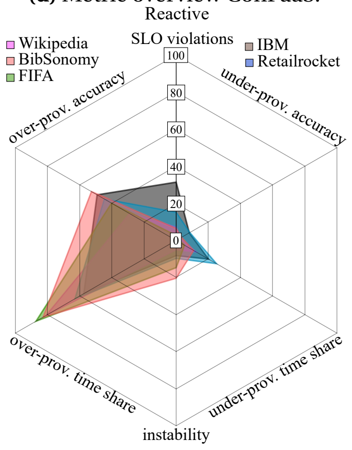
(c) Metric overview Hist.



(d) Metric overview ConPaaS.



(e) Metric overview Reg.



(f) Metric overview Reactive.

Figure 10.7: Scaling behavior of all auto-scalers.

10.5 Threats to Validity

Although our experimental analysis covered a wide range of different scenarios, the results may not be generalizable to other types of applications, for example, applications that are not interactive or CPU intensive. In principle, for the evaluated competing auto-scalers a comparable behavior has been observed in related works on auto-scaler evaluation for workflows [IAEH$^+$17] and in simulation [PAEÅ$^+$16]: platoons for Hist, some oscillations for ConPaaS and Reg, and a tightly following Adapt policy with many adaptations. It is still worth noting that, as discussed in previous studies [IAEH$^+$17, PAEÅ$^+$16], most of the auto-scalers evaluated in this paper are sensitive to their configuration for a given scenario.

The repeatability of performance related experiments in public cloud environments is limited due to the fact that there is no control given regarding the placement and co-location of VMs with other workloads running on the cloud. This can cause significant performance variability [IYE11]. To alleviate this problem, we conduct most experiments in a private environment under controlled conditions, however, for completeness, we also include experiments conducted in two different public cloud deployments (DAS-4 and AWS EC2) with a different degree of background load. Furthermore, we conduct long running experiments while not stressing the cloud's APIs much with on average 13 adaptations per hour.

We address the threat of a possible bias in metric values by using multiple and established sets of metrics that have been officially endorsed by SPEC [HKO$^+$16]. These elasticity metrics, as proposed in Section 5.2 have been developed in several iteration since 2012 reflecting a significant amount of discussions also with a broad spectrum of researchers and representatives from industry. The individual elasticity metrics are combined to aggregate metrics in an unweighted manner, treating under-provisioning and over-provisioning as being equally bad. In the case where under-provisioning is considered worse than over-provisioning, the results for Chameleon would further improve due to its tendency to slightly over-provision.

10.6 Summary

In this evaluation chapter, we answer RQ B.2 („How well does a proposed hybrid auto-scaling approach perform compared to state-of-the-art mechanisms in realistic deployments and application scenarios?"). We summarize the main evaluation findings as follows: (i) The Chameleon auto-scaler performs best in the evaluated scenarios based on average competition results, under-provisioning time share, under-provisioning accuracy and SLO violations. Chameleon tends to a reliable slight over-provisioning. (ii) The proposed way of combining proactive and reactive scaling decisions improves the auto-scaling performance (see Section 10.3.4). This answers RQ B.1 („How can conflicting auto-scaling decisions from independent reactive and proactive decision loops be combined to improve the overall quality of auto-scaling decisions?"). More efficient and accurate forecasting mechanisms could leverage the potential of further improving the proactive cycle. (iii) The Adapt auto-scaler manages to closely follow the demand with a relatively high number of adaptations. (iv) Hist and Reactive auto-scalers tend to stronger over-provisioning than others. (v) ConPaaS and

Reg auto-scalers exhibit unstable behavior in some situations and can not be considered as reliable in the covered scenarios. (vi) The Reactive auto-scaler relies on accurate CPU utilization measurements. It shows decreased performance in a public cloud context, where overbooking of virtual resources may cause significant interference with background load. (vii) In the conducted experiments, Chameleon exhibits consistently reliable scaling behavior, whereas the other auto-scalers show higher variance. (viii) Among the other investigated state-of-the-art auto-scalers, no mechanism outperforms the other ones in all traces, see Table 10.9. The proposed Chameleon approach dominates all covered scenarios.

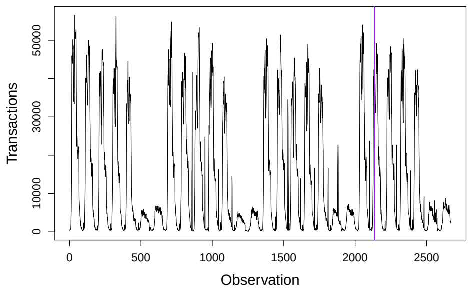
Chapter 11

Telescope Forecasting Prototype: Preliminary Evaluation and Case Study

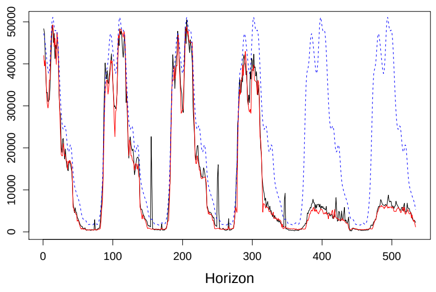
Answering RQ B.3 („How can a hybrid forecast approach based on decomposition be designed to be capable of providing accurate and fast multi-step-ahead forecasts of complex seasonal time-series within seconds?"), we assess the performance of the Telescope hybrid forecasting approach by conducting initial experiments presented in this chapter. As example time series, we use a trace of completed transactions on an IBM z196 Mainframe during February 2011 and a trace of monthly international airline passengers from 1949 to 1960. In addition, we present an auto-scaling case study comparing the resulting auto-scaler performance using Telescope against relying on the less timely and accurate forecast results of tBATS forecasts. This is our answer to the corresponding RQ B.4 („Is such a hybrid forecast approach capable to improve the performance and reliability of auto-scaling mechanisms?").

11.1 Preliminary Forecast Accuracy Evaluation based on Selected Time-Series

Each observation in the IBM trace contains the quarter-hourly amount of transactions (e.g., bookings or money transfers). The trace contains 2670 observations and is depicted in Figure 11.1a. It shows a typical seasonal pattern with a daily and weekly cycle, where the amount of transactions differs completely during weekdays and weekends. The trace exists of about 28 daily periods and 4 weekly periods. Since the approach is designed to perform multi-step-ahead forecasts, the last 20% observations of the time series are chosen as forecast horizon. Thus, the history of the IBM trace incorporates 2136 observations and the forecast horizon is set to 534 observations. The border between history and horizon is shown as vertical purple line in Figure 11.1a. The forecast of the IBM trace is shown in Figure 11.1b. The original time series is depicted in black, whereas the forecast of Telescope is represented by the red line. As a reference, the second best forecast produced by the tBATS approach [LHS11] is shown as dashed blue line. Besides the good fitting of the observed history, the hybrid approach succeeds in capturing the weekdays and weekends. In contrast to Telescope, tBATS only repeats a single pattern for the whole horizon. For weekdays, the forecast of Telescope and tBATS are very close to each other. However, tBATS misses capturing the weekends.



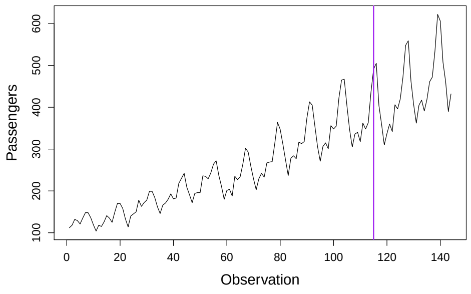
(a) All observations of the history and forecast horizon of the IBM trace.



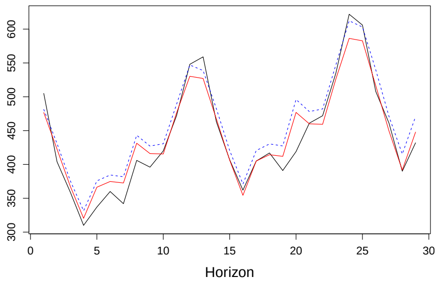
(b) Telescope (red) and tBATS (dashed blue) forecast of the IBM trace.

Figure 11.1: Observations and forecast of the IBM trace.

The airline passengers trace consists of 144 observations and shows an exponential trend pattern as well as a seasonal pattern with yearly cycle. Furthermore, the amplitude of the seasonal pattern increases as the trend rises. The airline passengers trace is shown in Figure 11.2a. Since the forecast horizon is set to 20%, the history contains 115 observations and the forecast horizon consists of 29 observations. Again, the border between history and horizon is shown as vertical purple line in Figure 11.2a.



(a) All observations of the history and forecast horizon of the airline passengers trace.



(b) Telescope (red) and tBATS (dashed blue) forecast of the airline passengers trace.

Figure 11.2: Observations and forecast of the airline passengers trace.

Figure 11.2b shows the forecast of the airline passengers trace. Again, the original time series is depicted as black line, the forecast of Telescope is shown as red line, and the tBATS forecast is depicted as dashed blue line. Both, tBATS and Telescope succeed in capturing the trend and season pattern. Though, besides the first value of the horizon, the tBATS forecast is always greater than the Telescope forecast.

To evaluate and compare the forecasting accuracy in a mathematical way, we use the mean absolute percentage error (MAPE) and mean absolute scaled error (MASE) measures. The MAPE is a widely-used measure to assess forecasting accuracy based on the forecasting error normalized with the observations. However, MAPE has some serious limitations, i.e., it cannot be used for time series with zeros in the forecasting horizon and it punishes positive errors harder than negative errors. Thus, we additionally use a second measure called MASE. Both measures are independent of the data scale, but in contrast to MAPE, MASE is suitable for almost all situations and the error is based on the in-sample mean absolute error from the random walk forecast. For a multi-step forecast horizon, the random walk forecast would forecast the last value of the history for the entire horizon. Thus, the investigated forecast is better than the random walk forecast if the MASE value is smaller than 1 and worse in the other case. The MAPE and MASE values are calculated as follows:

$$MAPE = 100 \times \frac{1}{n} \sum_{i=1}^{n} |\frac{e_i}{Y_i}| \tag{11.1}$$

$$MASE = \frac{\frac{1}{n} \sum_{i=1}^{n} |e_i|}{\frac{1}{n-1} \times \sum_{i=2}^{n} |Y_i - Y_l|} \tag{11.2}$$

Where Y_l is the observation at time l with l being the index of the last observation of the history. Y_i is the observation at time $l+i$. Thus, Y_i is the value of the i-th observation in the forecast horizon. The forecast error at time $l+i$ is calculated as $e_i = Y_i - F_i$ where F_i is the forecast at time $l+i$. The amount of observations in the forecast horizon is represented by n. Fore details on forecast accuracy metrics, please refer to Section 2.3.2. Another important measure to evaluate the performance of the forecasting approach is the elapsed time for the forecasting process. The total time elapsed for the forecast is measured in seconds. Table 11.1 shows the MASE and MAPE values and runtime of the hybrid approach for the IBM and airline passengers traces compared to six state-of-the-art forecasting methods:

- **ARIMA**: auto-regressive integrated moving averages (`auto.arima` with seasonality in package `forecast` [HK08, Hyn17]),
- **ANN**: artificial neural nets (`nnetar` in package `forecast` [HK08, Hyn17]),
- **ETS**: extended exponential smoothing (`ets` in package `forecast` [HK08, Hyn17]),
- **tBATS**: trigonometric, Box-Cox transformed, ARMA errors using trend and seasonal components (`tbats` in package `forecast` [HK08, Hyn17, LHS11]),
- **SVM**: support vector machine (`svm` in package `e1071` [Mey17]),
- **XGBoost**: scalable tree boosting (`xgboost` in package `xgboost` [CG16]) using only the index of the observation as covariate.

Concerning forecasting accuracy, the experiment shows that the hybrid approach reaches the lowest MASE and MAPE values for both time series. The Telescope forecast reaches a MASE value of about 0.064 for the IBM trace and 0.179 for the airline passengers trace. The MAPE values are about 51.628% and 3.382%. The second best MASE values are achieved by tBATS with 0.191 for the IBM trace and 0.276 for the airline passengers trace. Furthermore, tBATS reaches the second best MAPE value for the airline passengers trace with about 5.472%.

However, ANN outperforms tBATS in matters of the MAPE value for the IBM trace, i.e., ANN achieves a MAPE value of about 179.537. Concerning time-to-result, the experiment shows that Telescope has a very short runtime with about 8.557 and 2.679 seconds compared to its competitors of comparable accuracy. On the IBM trace, ETS, SVM, and XGBoost itself achieve shorter runtimes compared to Telescope. Though, each of these forecasting methods delivers an in-acceptable accuracy. On the airline passengers trace, only tBATS has a longer runtime than Telescope. Since the IBM trace is about 15 times as long as the airline passengers trace, this experiment implies that the runtime of the Telescope approach does not depend as much on the time series length as some state-of-the-art forecasting methods, i.e., ARIMA, ANN, and tBATS, do.

Table 11.1: Accuracy and runtime of state-of-the-art forecasting methods and Telescope.

Forecasting Method	IBM Trace			Passengers Trace		
	MASE	MAPE [%]	Time [s]	MASE	MAPE [%]	Time [s]
Telescope	0.064	51.628	8.557	0.179	3.382	2.679
ARIMA	0.343	813.570	12.301	0.358	6.255	1.065
ANN	0.788	179.537	12.172	0.400	7.473	0.801
ETS	0.986	2992.701	0.531	0.358	6.361	2.371
tBATS	0.191	253.243	38.078	0.276	5.472	4.538
SVM	0.276	574.624	2.312	3.711	67.909	0.233
XGBoost	0.736	545.469	0.484	0.692	11.936	0.278

11.2 Chameleon Auto-Scaling Case Study leveraging Telescope Forecasts

Due to the problems of the limited timeliness and accuracy of forecast results provided by existing methods like tBATS in the context of auto-scalers, we have been motivated to start working the development of the new hybrid forecasting approach Telescope. Thus, we now provide a showcase that underlines the potential of Telescope to further improve auto-scaling performance. A typical time series in auto-scaling show daily peaks and troughs that occur due to the Internet usage behavior of people. Thus, time series in auto-scaling mostly show seasonal patterns. Besides, the arriving load is measured with a relatively high frequency and so, there are many observations within a single seasonal period. In order to scale the amount of virtual machines properly, the future demand has to be forecast timely and accurately. The future incoming load needs to be forecast since virtual machines (and also stateful containers) need some time to start and, the resources should be running just before the demand arrives. Therefore, forecasters in auto-scaling have to fulfill three requirements. As usual, a high forecasting accuracy is desired. Additionally, the forecasting method should be stable, especially for seasonal time series with many observations within one period. Furthermore, since there are many observations within one period, the forecaster has to accurately predict multiple values at once. This can also include several hundreds of values at once. Finally, the results of the forecaster are required within a fixed time slot. Thus, the runtime of the forecaster should be as low and reliable as possible while keeping the accuracy of the forecast high. However, most common forecasting methods, e.g., sARIMA and ETS, cannot handle time series with a higher amount of data points per period very well. For example, ETS ignores the season component if it contains more than 24 data points. For sARIMA, the computation time explodes for more than 80 data points per period. So we only consider tBATS and Telescope as competitors in this use case. As service level objective (SLO), a request response time of at most 2 seconds is set. The time series describes the amount of requests sent to the social bookmark and publication sharing system BibSonomy. Due to a high ratio of noisy fluctuations, this trace is a challenge to be forecast accurately enough for planning auto-scaling actions.

Figure 11.3 shows the auto-scaling behavior and the requests per second for applying tBATS as forecasting method. In the upper plot, the demand is shown as black line, whereas the red line illustrates the supply. The requests per second are shown on the bottom plot. Here, the black line represents the requests sent, the green line shows the amount of requests that confirm the SLO, and the red line depicts the SLO violating request. The upper plot shows that the supply does not fit the demand very well. Especially from minute 30 to 80 and from minute 250 to 310, it can be seen that the supply crashes, whereas the demand keeps high.

That indicates that the tBATS forecasts either do not provide accurate forecasts or require too much computation time so that the results are dismissed. This can also be seen by taking a look at the lower plot. Here, the SLO violations increase considerably for the periods of time mentioned before, i.e, almost all requests violate the SLO. Besides these two long periods of time, most requests confirm the SLO and thus, they are responded within less than 2 seconds.

Table 11.2: Results of the Chameleon auto-scaling comparison using Telescope and tBATS.

Metric	**Telescope**	**tBATS**
θ[%] (accuracy)	0.1035	0.1510
τ[%] (time share)	0.3130	0.3806
υ[%] (instability)	0.4525	0.4620
σ (auto-scaling deviation)	0.50	0.60
ψ[%] (SLO violations)	0.09	0.36
avg. response Time [ms]	1071.08	1339.94

The auto-scaling behavior and the requests per second for applying Telescope is shown in Figure 11.4. The figure is organized as Figure 11.3. In contrast to the tBATS forecast, the upper plot of Figure 11.4 shows that the supply fits the demand quite well. In most cases, the amount of virtual machines is scaled just before a change in the demand occurs. The lower plot implies the same conclusion since there are almost no requests violating the SLO. There are only few and short periods of time where a small amount of requests exceed the

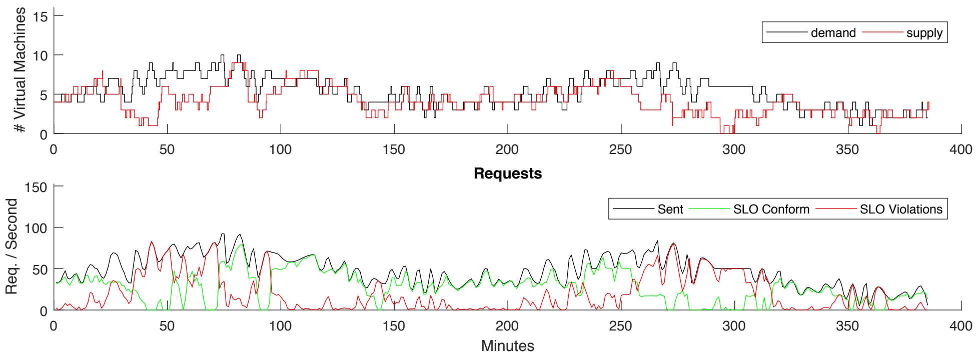


Figure 11.3: Demand, supply, and requests of Chameleon with **tBATS** on the BibSonomy trace.

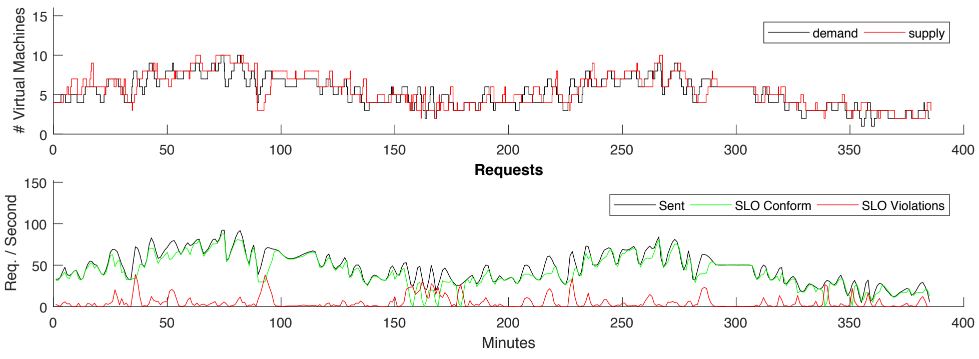


Figure 11.4: Demand, supply, and requests of Chameleon with **Telescope** on the BibSonomy trace.

SLO, e.g., around minutes 90 and 160. In general, the green line almost equals the black line and thus, most requests confirm the SLO. Compared to tBATS, it can be seen that Telescope improves the auto-scaling performance significantly.

In order to compare the auto-scaling performance of tBATS and Telescope in a mathematical way, we compute the elasticity metrics (not distinguishing between under-/over-provisioning for simplicity) as shown in Table 11.2. The accuracy θ, the wrong provisioning time share τ, as well as the instability υ improve applying Telescope compared to tBATS. Thus, the auto-scaling deviation σ summarizes that the Chameleon auto-scaling behavior by applying Telescope is closer to the optimal auto-scaler than auto-scaling by using tBATS. This results for using Telescope in only 9% of all requests exceeding the SLO of 2 seconds, whereas 36% of all requests violate the SLO for tBATS. Thus, we can conclude that Telescope considerably improves the auto-scaling performance in this case-study.

11.3 Summary

The evaluations show that Telescope improves the multi-step-ahead forecasting accuracy for univariate, seasonal time series compared to three state-of-the-art forecasting methods from time-series analysis and three forecasting methods based on machine learning techniques. Moreover, Telescope reduces the time-to-result immensely compared to the best competitors in terms of accuracy. The advantages of Telescope expose especially for long time series with many observations within single periods. Most classical forecasting methods cannot handle forecasting horizons consisting of several hundreds of values for time series with a high frequency. However, Telescope forecasts these time series very well. Furthermore, Telescope benefits from a second dominant seasonal pattern as it mostly succeeds in also capturing this second frequency, whereas the other forecasting methods in competition did not find the second seasonal pattern. For all these reasons, as also shown in an auto-scaling case study using the Chameleon auto-scaler, Telescope is suited for auto-scaling purposes very well. Thus, with the Telescope approach we can positively answer RQ B.4 („Is such a hybrid forecast approach capable to improve the performance and reliability of auto-scaling mechanisms?“). Telescope appears at this stage of research to be superior to existing black-box forecasting methods without any manual feature engineering.

Part V

Conclusions and Outlook

Chapter 12

Conclusions and Outlook

In this chapter, we first summarize the contributions of this thesis before stepping into a discussion of open and emerging challenges due to recent technological developments.

12.1 Thesis Summary

The contributions of this thesis lay in the domain of experimental performance evaluation and autonomic resource management in cloud environments. We are guided by two main goals, each accompanied by four research questions.

We approach our first main Goal A („Establish a benchmark for state-of-the-art auto-scalers to increase the trust in novel proactive mechanisms fostering a broader adoption of auto-scalers in production.") by introducing a modeling formalism for load intensity profiles to allow for realistic generation of changing loads. We define a set of elasticity metrics and a measurement methodology for reproducible and fair comparisons of elastic behavior across elastic cloud platforms.

The second main Goal B („Reduce the risk of using novel auto-scalers in operation by leveraging multiple different proactive mechanisms applied in combination with a conventional reactive mechanism.") is tackled in two ways: We propose Chameleon, a hybrid auto-scaling mechanism integrating multiple reactive and proactive methods, to increase quality and reliability of scaling actions. Second, we develop the hybrid forecasting approach Telescope to achieve more accurate and time-efficient predictions of the observed arriving load and thus further increase the performance and reliability of the Chameleon auto-scaler.

In the following, the four individual contributions are summarized in more detail:

Contribution I (addressing RQ A.1 and A.2): By introducing a modeling framework - called DLIM with its Limbo tool chain - for flexible handling and automated extraction of load intensity profiles, we enable benchmarking with more realistic workload models especially when complemented with user behavior profiles. For example, we provide extensions to load generation frameworks like Apache JMeter for emulating open and intensity varying workloads. Our evaluations demonstrate DLIM's model expressiveness and accuracy based on a representative set of web-based workload traces.

Contribution II (addressing RQ A.3 and A.4): With a focus on the experimental evaluation of the elasticity of cloud computing environments, we contribute by defining and assessing a set of metrics for quantifying the elasticity that auto-scaling mechanisms achieve in practice. The covered elasticity aspects are provisioning accuracy and timing behavior both distinguishing between under- and over-provisioning states and the stability/inertia of the auto-scaling mechanism in a given context. In individual sets of experiments, we show for each metric how it is impacted by changing configuration parameters. We establish a level-playing field for benchmarking auto-scaling mechanisms in practice and across platforms of different performance characteristics by defining and applying a sound measurement methodology (i.e. run rules) called Bungee. With a set of experiments both in private and public cloud environments, we showcase how a set of configurations of a standard reactive auto-scaler results in different levels of achieved elasticity and can be ranked consistently by applying three different metric aggregation approaches.

Contribution III (addressing RQ B.1 and B.2): This major contribution lays in the domain of autonomic management of compute resources. We present the novel self-aware auto-scaler Chameleon and benchmark it against four other state-of-the-art proactive auto-scalers. Chameleon combines forecasting (time series analysis) and service demand estimation (queueing theory) enriched with application knowledge (in the form of a descriptive software performance model) at system run-time to increase the timeliness and accuracy of auto-scaling reconfigurations. The forecast and the service demand estimation are realized by integrating established open-source tools provided by the research community. In the evaluation, we employ our proposed set of elasticity metrics, supplemented by user-oriented measures, to rate Chameleon in comparison to five other auto-scalers in a private CloudStack-based environment, on the public AWS EC2 IaaS cloud, and in an OpenNebula-based IaaS cloud of a medium-scale multi-cluster experimental environment DAS-4 [BEdLm16]. The workload scenarios consist of a CPU intensive Java Enterprise application (endorsed by SPEC SERT™2) driven by five different real-world load traces. The load profiles are extracted and adapted to the different platforms with the support of our load modeling framework DLIM (Contribution I). We apply the Bungee elasticity measurement methodology (Contribution II) and achieve reproducible and consistent results. For the five 3rd-party auto-scalers in our competition, we observe typical scaling behavior characteristics. Furthermore, the performance of the state-of-the-art auto-scalers depends on the workload characteristics and thus, none of the competing auto-scalers outperforms the others for all traces we cover. In contrast, Chameleon achieves in all setups and among all traces the best scaling behavior.

Contribution IV (adressing RQ B.3 and B.4:) In the context of applying forecasts in auto-scaling, we identify potential to further improve existing forecasting methods for more accurate and timely request arrival rate predictions. As a side-contribution of this thesis, we introduce the self-aware hybrid forecast mechanism Telescope that combines artificial neural networks and boosted random trees from the machine learning domain with classical time series analysis (spectral analysis, anomaly filtering, trend forecasting with ARIMA models) via a fast and reliable time-series decomposition approach STL. Telescope provides highly accurate forecasts including multi-step ahead ones with a time-to-result reduced up to a factor of 5. As forecast computation times do not explode for long or high resolution time series, a sweet-spot of Telescope are extensive time series with complex seasonal patterns. We show that Telescope is capable of significantly improving the auto-scaling performance of Chameleon by replacing the state-of-the-art forecasting methods tBATS or seasonal ARIMA.

We are confident that the four core contributions of this thesis have the potential to change the way cloud resource management approaches are assessed leading to improved quality of autonomic management algorithms as a result. To support such a development, we published the code artifacts of all four contributions of this thesis as open-source tools, which are actively maintained and accompanied by user and developer guides [BHK17][1].

12.2 Open Challenges and Outlook

In this section, we first discuss the challenges of extending the presented approaches for auto-scaling and its benchmarking towards scaling of multiple distributed services with a possibly inhomogeneous resource landscape which is out of scope for this work. Then, we step into a list of emerging challenges that gain increasing importance due to the adoption of container technologies on top of the operating system virtualization layer. We see the need for further work here due to the increased nesting of virtualized resource entities, the reflection of priorities of resources, and resource fragmentation in combination with increasing dynamics of user behavior triggered by DevOps practices. At the end of this section, we outline a few points of further work based on our self-aware forecasting mechanism Telescope.

12.2.1 Challenges of Elasticity in Applications Scaling Multiple Services

The proposed elasticity metrics are designed to quantify the elasticity on systems that host single-tier applications where on one group of homogeneous resources is scaled. Hence, with the consideration of multi-service applications and multiple auto-scalers in place for sub-groups of resources, or one multi-scaler that distinguishes between groups of resources, some challenges arise: (i) While the search of the demand-intensity curve on a single-service application takes n steps (n is the number of resource units), the amount of steps for a naive search for a multi-tier application is $n_1 \cdot \dots \cdot n_m$ where m is the number of tiers and n_i the number of resources of tier i. (ii) As the demanded resource units for a given intensity for a single-service application is distinct, the demand derivation for a multi-tier application becomes more complex because for one workload intensity there could be more than one optimal resource configuration. (iii) While investigating each service separately, the proposed metrics remain fully applicable for multi-tier scenarios, however, while characterizing the metrics along all services, the proposed set of metrics is not completely sufficient. For instance, consider the scenario where in the first service, there is one resource too less and in the second service, there is one resource too much. In the sum, the number of resources are equal to the number of the demanded resource units. Hence, the proposed elasticity measurement methodology and the metrics need extensions to handle such scenarios.

[1]Descartes Tools: `https://descartes.tools`

We started to work on extending the elasticity benchmarking framework BUNGEE to support multiple scaling dimensions and even inhomogeneous resource landscapes. The major challenge here is to automatically derive optimal deployment configurations for given load intensity levels in the analysis phase. This includes the exploration of a multi-dimensional search space with load tests for a Pareto-optimal front. Without this feature, it is not possible to derive the optimal demand over time, as done for the one-dimensional case. Having these features, Chameleon could be benchmarked against sets of other auto-scalers, each managing a different service of a distributed application, or against another multi-dimensional auto-scaler.

12.2.2 Emerging Challenges due to Technological Evolution

In the recent years, the industry has increasingly been moving towards micro-service and so-called serverless or Function-as-a-Service (FaaS) architectures in order to take advantage of the flexibility, scalability and time-to-market in cloud environments [BHJ16]. The growing popularity of micro-service and FaaS/serverless architectures is introducing new challenges and opportunities to the field of non-functional performance evaluation in cloud computing environments. We discuss potential extensions, assumptions and changes in the applicability of the elasticity metrics in order to address the concerns and challenges specific to these new types of cloud architectures.

Resource Nesting

The increasing popularity of micro-services coincides with technological advances in containerization. In contrast to traditional VM-based virtualization, containerization technology, such as LXC[2] or Docker[3], offer reduced performance overhead by executing applications as isolated processes, commonly referred to as containers, directly on the host operating system [FFRR15]. As a result, systems are increasingly deployed as a set of smaller micro-services in a container cloud. However, due to the security concerns associated with this relatively immature technology, it is currently a common practice to deploy micro-service containers on top of a VM. Moreover, to take advantage of the container-based, environment-agnostic cloud infrastructure, FaaS functions are typically deployed on top of containers [Rob16]. This *resource nesting*, consisting of FaaS functions, containers, VMs and physical resources, introduces challenges for many existing metrics. Where existing metrics for the evaluation of cloud environments focused on a single virtual layer, i.e. VMs, on top of physical hardware, adaptations will need to be made to ensure quality aspects take into account multiple virtual layers. One implication of this increasing resource nesting is that metrics concerned with elasticity typically assume that underlying resources are available and can be requested with a consistent provisioning delay. However, this assumption no longer holds true, scaling higher-level resources depends on the scaling capabilities of the underlying, lower-level resources. To scale up a FaaS function, in an optimistic scenario, the underlying VM and container are already provisioned, leading to a very short provisioning time. However, in an alternative scenario, the VM and the container might need to be scaled

[2]LXC: `https://linuxcontainers.org/`
[3]Docker: `https://www.docker.com/`

first before having sufficient resources for the function to scale. Therefore, performance metrics of systems comprised of these nested resources should not only account for their observed performance, but also of that of the underlying resources.

We see potential to extend our proposed Chameleon approach towards multi-dimensional and nested auto-scaling. Here, Chameleon would have to scale multiple services of a distributed application either horizontally (by adding further nodes) or vertically (by adding resources to existing nodes). Deciding between the two options would require a significant extension to the decision logic that may, for example, be based on machine learning. Among the benefits of a multi-dimensional auto-scaler is that it may be aware of hidden bottlenecks and thus able to prevent scaling delays or instability. Auto-scaling on nested resource layers like virtual machines or containers poses a new challenge on its own.

Resource Priorities and Cost Awareness

One of the important advantages to cloud computing, is the notion of "on-demand" resources. This powerful notion has allowed cloud users to avoid large upfront costs and static maintenance costs by only paying the cloud provider for the resources actually used. By effectively managing and multiplexing its resources, a cloud provider can make a profit, while reducing the costs for the cloud users. However, offering the same priority for every application might not be the most effective. A background application collecting log data would most likely not have the same urgency or priority as a business-critical web server. Resource cost instead of absolute, immediate performance is the most important factor for a background application, while in the business-critical web server the performance is required at any expense. This scenario is especially true, with regards to micro-services, where the diverse set of micro-services all have different performance/cost trade-offs. Large providers are experimenting with the notion of spot markets, where cloud users can get resources with fewer performance guarantees at a reduced price. For example, Amazon Spot Instances allows you to bid on spare capacity at their data-centers and in return allow Amazon to preempt the resources when the capacity is needed [ZZB11]. On the other hand, Microsoft Azure and Google Cloud offer a substantial, fixed discount on preemptable VMs compared to regular VMs. Yet, current metrics focus on optimizing performance, elasticity and similar properties without a regard for the costs. Though metrics for absolute, functional characteristics, such as performance, will continue to be of value, extensions will be needed to reflect how well systems can match the expected functional and non-functional objectives. Others have started to investigate this performance versus cost trade-off [SIm15], but more research is needed on how to adapt existing metrics and methodologies to take the costs of performance into account.

We started working on integrating awareness of the applied cost model as extension to auto-scaling mechanisms [LBHK18]. In the presence of long accounting intervals per resource, Chameleon can leverage knowledge from the forecast executions to delay or skip scaling events in case they are not cost-efficient. This would result in a shifted trade-off with decreased elasticity results but at lowered costs.

Resource Fragmentation

The increased adoption of the cloud by various industries comes with a wide variety of different types of workloads, which also introduce more and specific demand for various types of resources [Ber14]. High-performance resources, such as VMs with large memory allocation and CPU shares, or specific resources, such as machines with GPUs, generally cost more relative to less efficient machines. This diverse demand requires providers to be flexible in the resource options that they offer. However, this diverse demand does lead to physical resources being sub-optimally utilized, as it leads to *resource fragmentation*. For example, a single application might take up most of the CPU while under-utilizing the memory. Due to the inability to change resource characteristics on-demand, this prevents any other application being able to make use of the resource. In the context of micro-services this resource fragmentation would increase even more as micro-services are generally optimized to focus on a similar set of related responsibilities. Thus, micro-services tent to being a CPU-heavy, I/O-heavy or network-heavy specific resource consumer. Metrics for elasticity and performance typically consider only a single resource. When they do consider multiple resources, they fail to take into account over-provisioning in multiple resource dimensions. Therefore, adaptations will need to be made in the performance evaluation, in order to put more emphasis on how effectively multiple resource dimensions are taken into account.

Dynamic User Behavior

Current approaches for performance evaluation of cloud applications assume a relatively static user behavior as part of workload definitions. However, with the rising popularity of DevOps practices, automated frequent deployments of new versions of services is increasingly becoming a common industry practice, which is referred to as Continuous Deployment (CD) [Fow13]. According to the 2015 State of DevOps Report [FVm15], a significant portion of organizations already conduct multiple software deployments per day. For example, the retailer Otto conducts more than 500 deployments per week without interrupting the services [HS17]. The rapid changes in the behavior and resource consumption of services causes variations in the workload that affect the entire system under test, making it more difficult to predict workloads based on historical data. Accordingly, the methodology will need to be adapted to focus more on evaluating the initial, ’warm-up time’, of mechanisms. The warm-up time is the time needed to obtain a workload characterization needed by a mechanism or policy to perform close to optimally. For example, in auto-scaling research, many policies make use of historical data to evaluate scaling decisions using time series analysis [LBMAL14]. These policies need historical data to function optimally. With the frequent deployments and changing configurations of services, the amount of the historical workload data that needs to be recorded to have a policy function near-optimally becomes a concern. Yet, this concern is not yet represented in existing metrics for cloud environments.

In summary, the industry is rapidly transforming to an increasingly dynamic, on-demand model. This trend raises a number of challenges regarding the current cloud metrics and methodology. First, extensions to existing metrics will need to be considered to address increasing resource nesting. Second, there is an increased focus on applications being cost-effective, meaning that the operational costs should match the desired performance.

Besides focusing on the performance of applications, more research is needed on adaptations to the metrics to consider the performance/cost ratio. Third, adaptations to metrics regarding resource fragmentation should be considered. Finally, these dynamic systems are accompanied by dynamic workloads. The methodology of how to evaluate resource management policies, which may depend on a statically defined user behavior model for profiling purposes, will need to be revisited.

12.2.3 Future Work on Self-Aware Forecasting

As our presented Telescope approach is evaluated based on an early prototype implementation, we are working on a forecaster benchmark including a broad and representative time-series dataset covering various domains. We will evaluate Telescope's accuracy in comparison to competing approaches like Facebook's Prophet [4] from late 2017. Furthermore, several extensions and improvements of Telescope will be targeted as part of our future work:

(1) The automated feature engineering of Telescope can be extended. The selection of proper features is a crucial task for machine learning algorithms. Currently, the features are trend, season, and learned cluster labels. Nevertheless, there are also algorithms specifically designed to identify good features [KST17]. Such an approach could be integrated as optional task in Telescope.

(2) A second point for extending the Telescope approach would be towards detection of structural changes in the history. This includes trend level shifts as well as breakpoints in the trend and seasonal patterns. Currently, structural changes in the history are the main reason for poor forecasting performance of Telescope in certain situations. In these situations, Telescope fits a sub-optimal trend model based on the entire history instead of a meaningful recent part. If the trend model is only fitted according to the observations after the last break point, Telescope would likely fit the right trend model and thus, deliver a better forecasting accuracy.

Beyond the scope of this thesis, Telescope has further potential for usage in various fields of active research like predictive maintenance for anomaly detections and failure predictions with root cause analysis.

[4] Prophet Forecasting by and tailored for Facebook: `https://facebook.github.io/prophet/`

Appendices

Detailed Auto-Scaler Evaluation Results

This appendix chapter contains the detailed set of experimental evaluation results that have been omitted in Chapter 10.

German Wikipedia Trace

Table 10.3 represents the metric values and Figure 10.3 the scaling behavior the IBM trace.

IBM Trace

Table A.1 represents metric values and Figure 10.4 the scaling behavior of the three runs for the IBM trace.

Metric	Chameleon	Adapt	Hist	ConPaas	Reg	Reactive	No Scaling
θ_U (accuracy$_U$)	**5.36**%	6.48%	6.37%	12.81%	11.39%	8.17%	16.24%
θ_O (accuracy$_O$)	7.31%	32.36%	72.56%	19.79%	**6.86**%	49.16%	270.30%
τ_U (time share$_U$)	**14.88**%	28.81%	24.39%	34.43%	27.25%	20.16%	40.74%
τ_O (time share$_O$)	42.91%	49.05%	40.40%	30.28%	**25.95**%	57.44%	58.48%
υ (instability)	8.56%	13.58%	7.44%	12.80%	12.11%	8.22%	**5.88**%
ψ (SLO violations)	**9.57**%	29.92%	11.44%	37.41%	65.32%	31.44%	80.30%
σ (auto-scaling deviation)	**29.05**%	42.54%	43.40%	42.14%	65.79%	44.50%	147.16%
κ (pairwise competition)	**80.56**%	41.67%	66.67%	38.89%	55.56%	50.00%	16.67%
ϵ (elastic speedup)	**3.10**	1.73	1.79	1.79	2.47	1.74	1.00
#Adaptations	34	57	12	37	41	32	0
Avg. #VMs	10.035	10.081	9.8056	10.296	9.9806	8.7879	9
Instance minutes	1630.00	1529.70	1953.50	1544.30	**1454.20**	1716.00	1734.00
Avg. response time	**0.95** s	1.93 s	1.06 s	2.27 s	3.56 s	1.94 s	4.12 s
Med. response time	**0.46** s	0.59 s	0.50 s	0.66 s	5.00 s	0.58 s	5.00 s

Table A.1: Metric overview for the IBM trace.

BibSonomy Trace

Table A.2 represents the metric values and Figure A.1 the scaling behavior for BibSonomy trace.

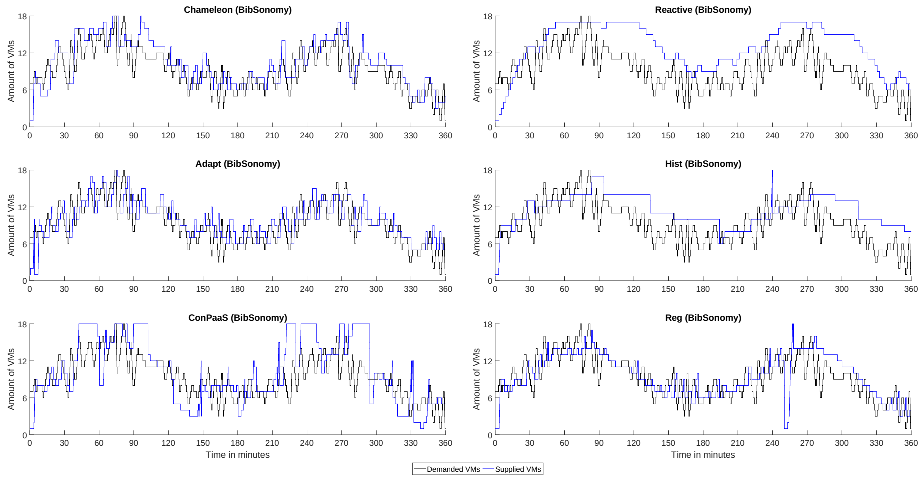


Figure A.1: Comparison of the auto-scalers for the BibSonomy trace.

Metric	Chameleon	Adapt	Hist	ConPaas	Reg	Reactive	No Scaling
θ_U (accuracy$_U$)	5.60%	7.62%	4.37%	14.04%	9.89%	**3.32**%	10.73%
θ_O (accuracy$_O$)	23.45%	24.11%	45.35%	29.87%	**18.40**%	52.67%	36.84%
τ_U (time share$_U$)	22.52%	34.10%	21.09%	37.42%	37.60%	**10.85**%	43.04%
τ_O (time share$_O$)	61.02%	48.96%	72.90%	51.04%	45.64%	82.71%	**44.86**%
υ (instability)	22.95%	24.37%	19.14%	24.72%	23.08%	20.40%	**17.59**%
ψ (SLO violations)	16.20%	54.85%	13.99%	31.75%	50.74%	**6.47**%	65.91%
σ (auto-scaling deviation)	**43.05**%	59.39%	48.28%	48.31%	56.17%	48.60%	69.27%
κ (pairwise competition)	**61.11**%	44.44%	**61.11**%	30.56%	50.00%	**61.11**%	41.67%
ϵ (elastic speedup)	1.27	1.12	1.18	0.93	1.13	**1.33**	1.00
#Adaptations	116	144	35	117	115	75	0
Avg. #VMs	10.112	9.7755	10.058	9.7426	8.9485	10	9
Instance minutes	3862.50	3650.70	4256.20	3857.30	3534.30	4778.80	**3471.00**
Avg. response time	1.28 s	3.10 s	1.18 s	1.99 s	2.87 s	**0.79** s	3.51 s
Med. response time	0.49 s	5.00 s	0.49 s	0.59 s	4.44 s	**0.47** s	5.00 s

Table A.2: Metric overview for the BibSonomy trace.

Retailrocket Trace

Table A.3 represents the metric values and Figure A.2 the scaling behavior for Retailrocket trace.

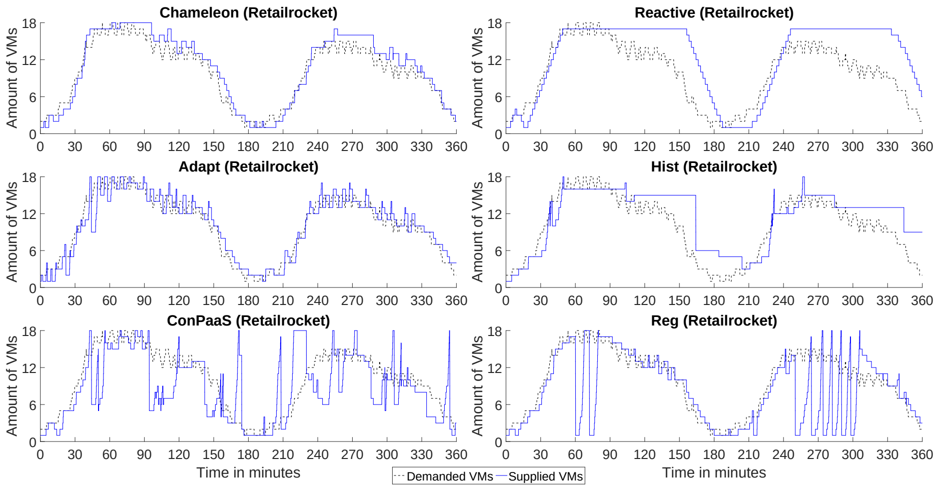


Figure A.2: Comparison of the auto-scalers for the Retailrocket trace.

Metric	Chameleon	Adapt	Hist	ConPaas	Reg	Reactive	No Scaling
θ_U (accuracy$_U$)	6.37%	6.55%	**5.22**%	21.05%	19.37%	8.50%	17.68%
θ_O (accuracy$_O$)	11.71%	16.45%	76.29%	22.41%	**8.44**%	44.64%	110.38%
τ_U (time share$_U$)	**19.53**%	31.76%	24.59%	59.90%	44.38%	25.02%	55.75%
τ_O (time share$_O$)	51.34%	44.60%	59.42%	**20.74**%	29.21%	62.92%	38.81%
υ (instability)	10.98%	14.82%	9.46%	18.24%	15.73%	9.68%	**7.22**%
ψ (SLO violations)	**8.68**%	26.56%	16.26%	60.87%	58.20%	15.20%	82.06%
σ (auto-scaling deviation)	**35.59**%	40.49%	49.39%	63.93%	60.51%	45.53%	90.52%
κ (pairwise competition)	**75.00**%	52.78%	61.11%	27.78%	47.22%	50.00%	36.11%
ϵ (elastic speedup)	**2.06**	1.68	1.41	1.23	1.56	1.39	1.00
#Adaptations	82	125	36	122	113	69	0
Avg. #VMs	9.301	11.009	11.013	10.196	9.0686	8.2361	9
Instance minutes	3890.50	3687.30	4279.30	3211.80	**3020.30**	4568.70	3471.00
Avg. response time	**0.89** s	1.82 s	1.25 s	3.33 s	3.22 s	1.19 s	4.23 s
Med. response time	**0.44** s	0.59 s	0.49 s	5.00 s	5.00 s	0.49 s	5.00 s

Table A.3: Metric overview for the Retailrocket trace traceo.

FIFA World Cup 1998 Trace in Private Scenario

Table A.4 represents the metric values and Figure A.3 the scaling behavior for FIFA World Cup 1998 trace in the private scenario.

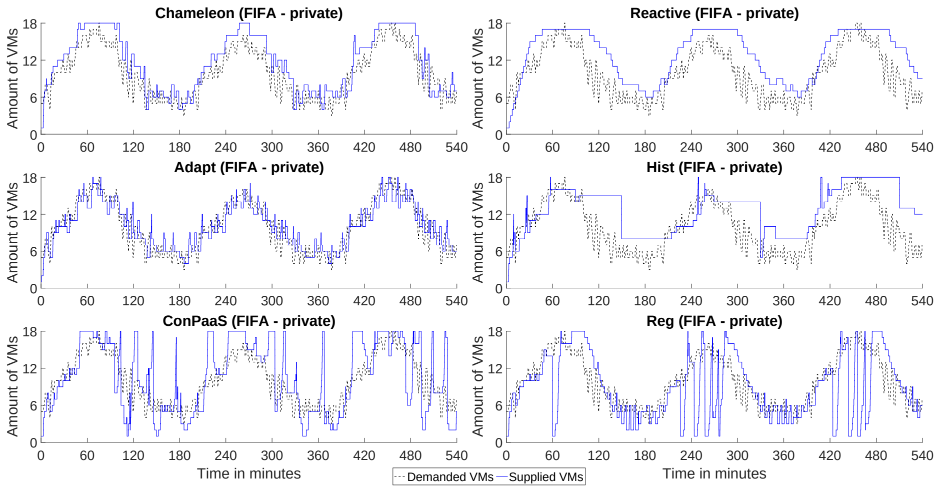


Figure A.3: Comparison of the auto-scalers for the FIFA World Cup 1998 trace in private scenario.

Metric	**Chameleon**	**Adapt**	**Hist**	**ConPaas**	**Reg**	**Reactive**	**No Scaling**
θ_U (accuracy$_U$)	3.23%	7.09%	2.83%	15.01%	19.70%	**1.56**%	14.75%
θ_O (accuracy$_O$)	21.95%	**11.81**%	46.87%	20.18%	12.37%	40.20%	27.41%
τ_U (time share$_U$)	13.09%	41.47%	15.61%	39.48%	49.25%	**5.20**%	48.93%
τ_O (time share$_O$)	74.05%	35.81%	74.57%	49.22%	**35.03**%	87.14%	44.77%
υ (instability)	16.36%	21.56%	14.91%	20.93%	21.79%	14.68%	**12.66**%
ψ (SLO violations)	8.65%	43.67%	12.01%	53.23%	64.28%	**2.70**%	62.14%
σ (auto-scaling deviation)	**43.88**%	49.68%	46.27%	59.09%	67.30%	46.76%	66.83%
κ (pairwise competition)	61.11%	55.56%	50.00%	41.67%	30.56%	**66.67**%	44.44%
ϵ (elastic speedup)	1.58	1.33	1.37	0.98	1.04	**1.93**	1.00
#Adaptations	122	159	45	153	169	81	0
Avg. #VMs	9.532	10.692	11.773	11.232	8.7312	10.488	9
Instance minutes	6500.20	5562.00	7203.30	5899.70	**4939.00**	7262.50	5190.00
Avg. response time	0.89 s	2.60 s	1.06 s	2.96 s	3.46 s	**0.60** s	3.32 s
Med. response time	**0.45** s	1.63 s	0.48 s	5.00 s	5.00 s	**0.45** s	5.00 s

Table A.4: Metric overview for the FIFA World Cup 1998 trace in private scenario.

FIFA World Cup 1998 Trace on AWS EC2

Table A.5 represents the metric values and Figure A.4 the scaling behavior for FIFA World Cup 1998 trace executed on AWS EC2.

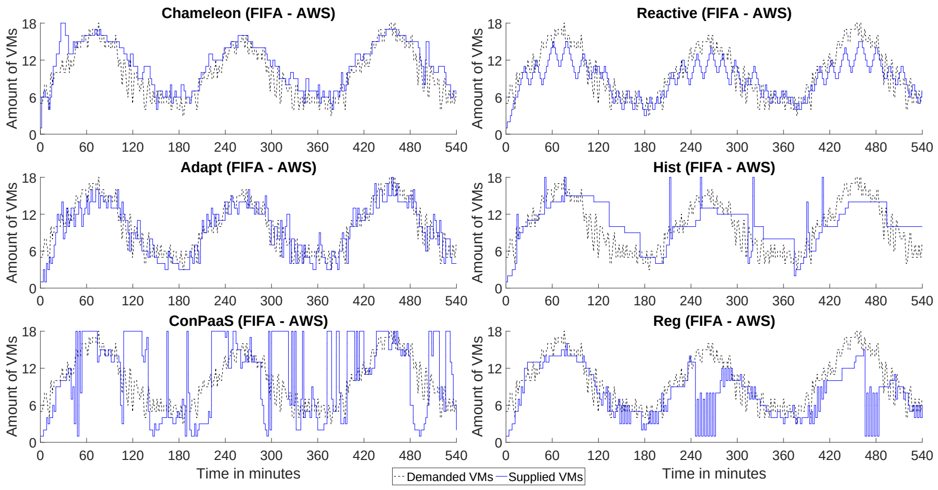


Figure A.4: Comparison of the auto-scalers for the FIFA World Cup 1998 trace in AWS EC2.

Metric	Chameleon	Adapt	Hist	ConPaas	Reg	Reactive	No Scaling
θ_U (accuracy$_U$)	**1.64**%	12.93%	9.19%	21.11%	21.44%	13.36%	14.75%
θ_O (accuracy$_O$)	21.31%	10.85%	23.98%	35.51%	**5.46**%	9.19%	27.41%
τ_U (time share$_U$)	**11.04**%	52.02%	44.10%	49.02%	67.80%	57.49%	48.93%
τ_O (time share$_O$)	67.92%	32.43%	44.68%	41.94%	**18.44**%	25.49%	44.77%
υ (instability)	15.95%	17.98%	13.96%	16.50%	16.79%	19.02%	**12.66**%
ψ (SLO violations)	**5.04**%	21.56%	16.60%	22.17%	54.36%	49.30%	51.21%
σ (auto-scaling deviation)	**39.81**%	43.31%	44.93%	47.83%	59.22%	54.80%	58.76%
κ (pairwise competition)	**69.44**%	52.78%	66.67%	36.11%	38.89%	44.44%	41.67%
ϵ (elastic speedup)	**1.93**	1.21	1.13	0.85	1.35	1.27	1.00
#Adaptations	126	169	47	131	164	254	0
Avg. #VMs	9.649	9.3815	10.202	9.8456	6.8119	8.8372	9
Instance minutes	6360.30	5350.30	5905.50	6016.80	**4471.30**	5004.00	5190.00
Avg. response time	**0.62** s	1.32 s	1.07 s	1.32 s	2.84 s	2.68 s	2.67 s
Med. response time	0.26 s	**0.25** s	**0.25** s	**0.25** s	5.00 s	3.23 s	5.00 s

Table A.5: Metric overview for the FIFA World Cup 1998 trace in AWS EC2.

FIFA World Cup 1998 Trace in DAS-4

Table A.6 represents the metric values and Figure A.5 the scaling behavior for FIFA World Cup 1998 trace in DAS-4.

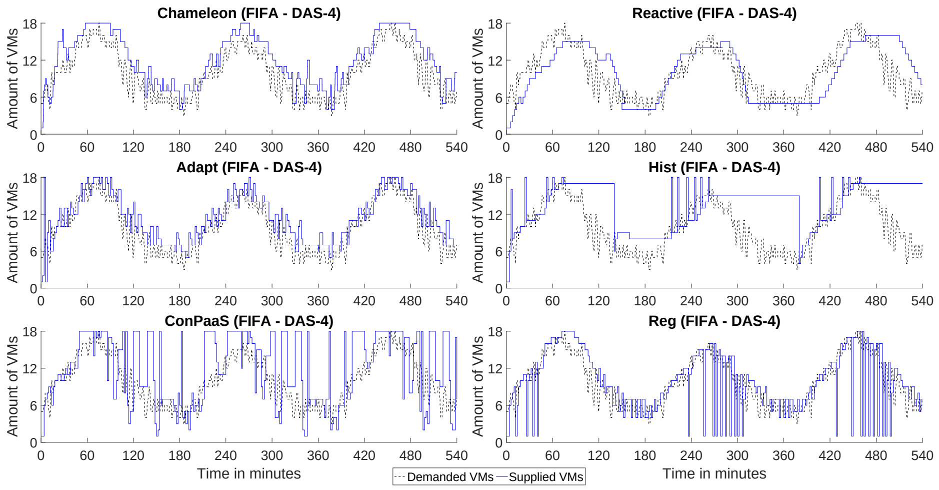


Figure A.5: Comparison of the auto-scalers for the FIFA World Cup 1998 trace in DAS-4.

Metric	**Chameleon**	**Adapt**	**Hist**	**ConPaas**	**Reg**	**Reactive**	**No Scaling**
θ_U (accuracy$_U$)	**1.66**%	2.79%	2.54%	10.11%	11.95%	11.83%	14.75%
θ_O (accuracy$_O$)	28.28%	26.51%	69.89%	38.45%	17.68%	**16.52**%	27.41%
τ_U (time share$_U$)	**6.45**%	15.69%	16.73%	26.65%	28.64%	46.85%	48.93%
τ_O (time share$_O$)	84.97%	68.06%	72.54%	66.53%	54.45%	**38.12**%	44.77%
υ (instability)	16.76%	16.79%	13.99%	16.10%	16.79%	13.96%	**12.66**%
ψ (SLO violations)	**20.95**%	37.31%	26.96%	43.61%	47.51%	41.49%	43.56%
σ (auto-scaling deviation)	**46.53**%	47.61%	50.02%	54.38%	53.51%	50.10%	54.22%
κ (pairwise competition)	61.11%	52.78%	52.78%	38.89%	33.33%	**63.89**%	44.44%
ϵ (elastic speedup)	**1.92**	1.53	1.30	1.00	1.15	1.18	1.00
#Adaptations	131	288	288	288	288	135	0
Avg. #VMs	9.8796	11.187	13.803	11.716	9.5675	10.978	9
Instance minutes	6786.00	6452.00	7955.30	6770.70	5519.30	5629.70	**5190.00**
Avg. response time	**1.34** s	2.14 s	1.68 s	2.51 s	2.60 s	2.32 s	2.50 s
Med. response time	**0.33** s	0.40 s	**0.33** s	1.43 s	1.63 s	**0.33** s	1.50 s

Table A.6: Metric overview for the FIFA World Cup 1998 trace in DAS-4.

List of Figures

List of Tables

Bibliography

[AETE12] Ahmed Ali-Eldin, Johan Tordsson, and Erik Elmroth. An Adaptive Hybrid Elasticity Controller for Cloud Infrastructures. In IEEE NOMS 2012, pages 204–212. IEEE, 2012. [see pages 40, 41, and 148]

[AFG$^+$10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. Commun. ACM, 53(4):50–58, April 2010. [see pages 13 and 66]

[AJ00] Martin Arlitt and Tai Jin. A Workload Characterization Study of the 1998 World Cup Web Site. IEEE Network, 14(3):30–37, 2000. [see page 145]

[Arm85] J Scott Armstrong. From crystal ball to computer. New York, 1985. [see page 21]

[ASLM13] Rodrigo F Almeida, Flávio RC Sousa, Sérgio Lifschitz, and Javam C Machado. On Defining Metrics for Elasticity of Cloud Databases. In Proceedings of the 28th Brazilian Symposium on Databases, 2013. [see pages 37 and 66]

[AW96] Martin F. Arlitt and Carey L. Williamson. Web server workload characterization: The search for invariants. SIGMETRICS Perform. Eval. Rev., 24(1):126–137, May 1996. [see page 34]

[Bar10] P. Bartlett. Introduction to time series analysis. lecture 19, 2010. [see page 28]

[BC64] George EP Box and David R Cox. An analysis of transformations. Journal of the Royal Statistical Society. Series B (Methodological), pages 211–252, 1964. [see page 31]

[BC98] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems, SIGMETRICS '98/PERFORMANCE '98, pages 151–160, New York, NY, USA, 1998. ACM. [see page 35]

[BEdLm16] Henri Bal, Dick Epema, Cees de Laat, and more. A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term. IEEE Computer, 49(5):54–63, May 2016. [see pages 95, 146, and 174]

[Bel16] Marta Beltrán. Becloud: A new approach to analyse elasticity enablers of cloud services. Future Generation Computer Systems, 64:39–49, 2016. [see pages 37, 38, and 39]

[Ber14] David Bernstein. Containers and cloud: From LXC to Docker to Kubernetes. IEEE Cloud Computing, 1(3):81–84, 2014. [see page 178]

[BFB09] Gordon Blair, Robert B. France, and Nelly Bencomo. Models@ run.time. Computer, 42:22–27, 2009. [see page 48]

[BG69] John M Bates and Clive WJ Granger. The combination of forecasts. Journal of the Operational Research Society, 20(4):451–468, 1969. [see page 42]

[BGHK18] André Bauer, Johannes Grohmann, Nikolas Herbst, and Samuel Kounev. On the Value of Service Demand Estimation for Auto-Scaling. In 19th International GI/ITG Conference on Measurement, Modelling and Evaluation of Computing Systems (MMB 2018). Springer, February 2018. [see pages xiv and 7]

[BH74] James R Bunch and John E Hopcroft. Triangular factorization and inversion by fast matrix multiplication. Mathematics of Computation, 28(125):231–236, 1974. [see page 146]

[BHIP17] Gunnar Brataas, Nikolas Herbst, Simon Ivansek, and Jure Polutnik. Scalability Analysis of Cloud Software Services. In Companion Proceedings of the 14th IEEE International Conference on Autonomic Computing (ICAC 2017), Self Organizing Self Managing Clouds Workshop (SOSeMC 2017). IEEE, July 2017. [see page xv]

[BHJ$^+$10] Dominik Benz, Andreas Hotho, Robert Jäschke, Beate Krause, Folke Mitzlaff, Christoph Schmitz, and Gerd Stumme. The social bookmark and publication management system BibSonomy. The VLDB Journal, 19(6):849–875, December 2010. [see pages 114 and 145]

[BHJ16] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: migration to a cloud-native architecture. IEEE Software, 33(3):42–52, 2016. [see page 176]

[BHK17] André Bauer, Nikolas Herbst, and Samuel Kounev. Design and Evaluation of a Proactive, Application-Aware Auto-Scaler. In Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE 2017), April 2017. [see pages xvi, 8, and 175]

[BHS$^+$18] André Bauer, Nikolas Herbst, Simon Spinner, Samuel Kounev, and Ahmed Ali-Eldin. Chameleon: A Hybrid, Proactive Auto-Scaling Mechanism on a Level-Playing Field. IEEE Transactions on Parallel and Distributed Systems (TPDS), 2018. Minor Revision July 2018. [see pages xiii and 7]

[BJ$^+$15] George EP Box, Gwilym M Jenkins, et al. Time Series Analysis: Forecasting and Control. John Wiley & Sons, 2015. [see pages 94 and 98]

[BKK09] Fabian Brosig, Samuel Kounev, and Klaus Krogmann. Automated Extraction of Palladio Component Models from Running Enterprise Java Applications. In VALUETOOLS ’09, pages 1–10, 2009. [see pages 34 and 35]

[BKKL09] Carsten Binnig, Donald Kossmann, Tim Kraska, and Simon Loesing. How is the Weather Tomorrow?: Towards a Benchmark for the Cloud. In Proceedings of the Second International Workshop on Testing Database Systems, DBTest '09, pages 9:1–9:6, New York, NY, USA, 2009. ACM. [see pages 37 and 66]

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The palladio component model for model-driven performance prediction. Journal of Systems and Software, 82(1):3 – 22, 2009. Special Issue: Software Performance - Modeling and Analysis. [see page 34]

[BLB15] Matthias Becker, Sebastian Lehrig, and Steffen Becker. Systematically deriving quality metrics for cloud computing systems. In Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, ICPE '15, pages 169–174, New York, NY, USA, 2015. ACM. [see page 37]

[BLY$^+$10] Aaron Beitch, Brandon Liu, Timothy Yung, Rean Griffith, Armando Fox, and David A. Patterson. Rain: A workload generation toolkit for cloud computing applications. Technical Report UCB/EECS-2010-14, EECS Department, University of California, Berkeley, Feb 2010. [see pages 34 and 48]

[Bm06] Gunter Bolch and more. Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications. John Wiley & Sons, 2006. [see pages 97 and 99]

[Bro56] Robert G Brown. Exponential smoothing for predicting demand. cambridge, mass., arthur d. little, 1956. [see page 32]

[BSL$^+$13] Gunnar Brataas, Erlend Stav, Sebastian Lehrig, Steffen Becker, Goran Kopčak, and Darko Huljenic. Cloudscale: Scalability management for cloud systems. In Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering, ICPE '13, pages 335–338, New York, NY, USA, 2013. ACM. [see page 49]

[BvHW$^+$15] Andreas Brunnert, Andre van Hoorn, Felix Willnecker, Alexandru Danciu, Wilhelm Hasselbring, Christoph Heger, Nikolas Herbst, Pooyan Jamshidi, Reiner Jung, Joakim von Kistowski, Anne Koziolek, Johannes Kross, Simon Spinner, Christian Vögele, Jürgen Walter, and Alexander Wert. Performance-oriented DevOps: A research agenda. Technical Report SPEC-RG-2015-01, SPEC Research Group — DevOps Performance Working Group, Standard Performance Evaluation Corporation (SPEC), August 2015. [see page xvii]

[BZ86] Herman J Blinchikoff and Anatol I Zverev. Filtering in the time and frequency domains. Krieger Publishing Co., Inc., 1986. [see page 61]

[C$^+$09] Trieu Chieu et al. Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment. In IEEE ICEBE 2009, pages 281–286. IEEE, 2009. [see page 148]

[CA92] Fred Collopy and J Scott Armstrong. Rule-based forecasting: Development and validation of an expert systems approach to combining time series extrapolations. Management Science, 38(10):1394–1414, 1992. [see page 42]

[CCB$^+$12] Dean Chandler, Nurcan Coskun, Salman Baset, Erich Nahum, Steve Realmuto Masud Khandker, Tom Daly, Nicholas Wakou Indrani Paul, Louis Barton, Mark Wagner, Rema Hariharan, and Yun seng Chao. Report on Cloud Computing to the OSG Steering Committee. Technical report, April 2012. [see pages 36 and 66]

[CCMT90] Robert B Cleveland, William S Cleveland, Jean E McRae, and Irma Terpenning. Stl: A seasonal-trend decomposition procedure based on loess. Journal of Official Statistics, 6(1):3–73, 1990. [see pages 23, 49, 55, 108, and 113]

[CG16] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 785–794. ACM, 2016. [see pages 7, 29, 110, and 165]

[CGRGG$^+$17] Gaspar Cano, Jose Garcia-Rodriguez, Alberto Garcia-Garcia, Horacio Perez-Sanchez, Jón Atli Benediktsson, Anil Thapa, and Alastair Barr. Automatic selection of molecular descriptors using random forest: Application to drug discovery. Expert Systems with Applications, 72:151–159, 2017. [see page 108]

[CKKR12] Giuliano Casale, Amir Kalbasi, Diwakar Krishnamurthy, and Jerry Rolia. Burn: Enabling workload burstiness in customized service benchmarks. IEEE Transactions on Software Engineering, 38(4):778–793, 2012. [see page 35]

[Cle89] Robert T Clemen. Combining forecasts: A review and annotated bibliography. International Journal of Forecasting, 5(4):559–583, 1989. [see page 42]

[Coh09] Reuven Cohen. Defining Elastic Computing, September 2009. `http://www.elasticvapor.com/2009/09/defining-elastic-computing.html`, last consulted Jan. 2017. [see page 14]

[CST$^+$10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In Proceedings of the 1st ACM symposium on Cloud computing, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM. [see page 37]

[CW09] Alpha C. Chiang and Kevin Wainwright. Fundamental methods of mathematical economics. McGraw-Hill [u.a.], Boston, Mass. [u.a.], 4. ed., internat. ed., [repr.] edition, 2009. [see page 13]

[Dav87] Herbert A David. Ranking from Unbalanced Paired-Comparison Data. Biometrika, 74:432–436, 1987. [see page 75]

[DLHS11] Alysha M De Livera, Rob J Hyndman, and Ralph D Snyder. Forecasting time series with complex seasonal patterns using exponential smoothing. Journal of the American Statistical Association, 106(496):1513–1527, 2011. [see page 32]

[DMBT00] Lilian M De Menezes, Derek W Bunn, and James W Taylor. Review of guidelines for the use of combined forecasts. European Journal of Operational Research, 120(1):190–204, 2000. [see page 42]

[DMRT11] Thibault Dory, Boris Mejías, Peter Van Roy, and Nam-Luc Tran. Measuring Elasticity for Cloud Databases. In Proceedings of the The Second International Conference on Cloud Computing, GRIDs, and Virtualization, 2011. [see pages 37, 38, and 66]

[DRW07] Leticia Duboc, David Rosenblum, and Tony Wicks. A Framework for Characterization and Analysis of Software System Scalability. In Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07), pages 375–384. ACM, 2007. [see page 15]

[EHPG$^+$17] Neska El Haouij, Jean-Michel Poggi, Raja Ghozi, Sylvie Sevestre-Ghalila, and Mériem Jaïdane. Random forest-based approach for physiological functional variable selection: Towards driver's stress level classification. 2017. [see page 108]

[Fab06] Faban. http://faban.org, 2006. last accessed in February 2018. [see page 48]

[FAS$^+$12] Enno Folkerts, Alexander Alexandrov, Kai Sachs, Alexandru Iosup, Volker Markl, and Cafer Tosun. Benchmarking in the Cloud: What It Should, Can, and Cannot Be. In Raghunath Nambiar and Meikel Poess, editors, Selected Topics in Performance Evaluation and Benchmarking, volume 7755 of Lecture Notes in Computer Science, pages 173–188. Springer Berlin Heidelberg, 2012. [see pages 37, 66, and 78]

[Fei02] DrorG. Feitelson. Workload modeling for performance evaluation. In MariaCarla Calzarossa and Salvatore Tucci, editors, Performance Evaluation of Complex Systems: Techniques and Tools, volume 2459 of Lecture Notes in Computer Science, pages 114–141. Springer Berlin Heidelberg, 2002. [see page 35]

[FFRR15] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In IEEE ISPASS 2015, pages 171–172. IEEE, 2015. [see page 176]

[FHA99] E. Freeman, S. Hupfer, and K. Arnold. JavaSpaces Principles, Patterns, and Practice. Java series. Addison-Wesley, 1999. [see page 81]

[Fow13] Martin Fowler. Continuous delivery, 2013. Accessed: 2017-05-17. [see page 178]

[FPK14] Hector Fernandez, Guillaume Pierre, and Thilo Kielmann. Autoscaling Web Applications in Heterogeneous Cloud Infrastructures. In IEEE IC2E, 2014. [see pages 2, 148, and 149]

[Fri91] Jerome H Friedman. Multivariate adaptive regression splines. The annals of statistics, pages 1–67, 1991. [see page 35]

[FVm15] N Forsgren Velasquez and more. State of devops report 2015. Puppet Labs and IT Revolution, 2015. [see page 178]

[FW86] Philip J. Fleming and John J. Wallace. How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results. Commun. ACM, 29(3):218–221, March 1986. [see page 76]

[GB12] Guilherme Galante and Luis Carlos E. de Bona. A Survey on Cloud Computing Elasticity. In Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing, UCC '12, pages 263–270, Washington, DC, USA, 2012. IEEE Computer Society. [see pages 33 and 66]

[GdB12] Guilherme Galante and Luis de Bona. A Survey on Cloud Computing Elasticity. In IEEE UCC 2012, pages 263–270. IEEE, 2012. [see page 39]

[GHSK17] Johannes Grohmann, Nikolas Herbst, Simon Spinner, and Samuel Kounev. Self-Tuning Resource Demand Estimation. In IEEE ICAC 2017, July 2017. [see pages xv, 7, 35, and 98]

[GHSK18] Johannes Grohmann, Nikolas Herbst, Simon Spinner, and Samuel Kounev. Using Machine Learning for Recommending Service Demand Estimation Approaches (Position Paper). In Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018). SciTePress, March 2018. [see page xv]

[HAA$^+$17] Nikolas Herbst, Ayman Amin, Artur Andrzejak, Lars Grunske, Samuel Kounev, Ole J. Mengshoel, and Priya Sundararajan. Online Workload Forecasting. In Samuel Kounev, Jeffrey O. Kephart, Xiaoyun Zhu, and Aleksandar Milenkoski, editors, Self-Aware Computing Systems. Springer Verlag, Berlin Heidelberg, Germany, 2017. [see pages xv and 33]

[Hal08] Emily H Halili. Apache JMeter: A Practical Beginner's Guide to Automated Testing and performance measurement for your websites. Packt Publishing Ltd, 2008. [see page 48]

[HB13] Mor Harchol-Balter. Performance Modeling and Design of Computer Systems: Queueing Theory in Action. Cambridge University Press, New York, NY, USA, 1st edition, 2013. [see page 13]

[HBK$^+$17] Nikolas Herbst, Steffen Becker, Samuel Kounev, Heiko Koziolek, Martina Maggio, Aleksandar Milenkoski, and Evgenia Smirni. Metrics and Benchmarks for Self-Aware Computing Systems. In Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu, editors, Self-Aware Computing Systems. Springer Verlag, Berlin Heidelberg, Germany, 2017. [see page xv]

[HBK$^+$18] Nikolas Herbst, André Bauer, Samuel Kounev, Giorgos Oikonomou, Erwin van Eyk, George Kousiouris, Athanasia Evangelinou, Rouven Krebs, Tim Brecht, Cristina L. Abad, and Alexandru Iosup. Quantifying Cloud Performance and Dependability: Taxonomy, Metric Design, and Emerging Challenges. ACM Transactions on Modeling and Performance Evaluation of Computing Systems (ToMPECS), 2018. Submitted Sept 2017 - Major Revision Jan 2018 - Minor Revision June 2018. [see pages xiii, 6, and 67]

[HBS$^+$17] Nikolaus Huber, Fabian Brosig, Simon Spinner, Samuel Kounev, and Manuel Bähr. Model-Based Self-Aware Performance and Resource Management Using the Descartes Modeling Language. IEEE Transactions on Software Engineering (TSE), 43(5), 2017. [see pages 6 and 34]

[Her11] Nikolas Herbst. Quantifying the Impact of Configuration Space for Elasticity Benchmarking. Study Thesis, Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, 76131 Karlsruhe, Germany, 2011. [see page xvii]

[Her12] Nikolas Herbst. Workload Classification and Forecasting. Diploma Thesis, Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, 76131 Karlsruhe, Germany, 2012. Forschungszentrum Informatik (FZI) Prize "Best Diploma Thesis". [see page xvii]

[HGGG12] Rui Han, Li Guo, Moustafa M Ghanem, and Yike Guo. Lightweight Resource Scaling for Cloud Applications. In IEEE/ACM CCGrid 2012, pages 644–651. IEEE, 2012. [see page 39]

[HHK$^+$14] Nikolaus Huber, André Hoorn, Anne Koziolek, Fabian Brosig, and Samuel Kounev. Modeling run-time adaptation at the system architecture level in dynamic service-oriented environments. Serv. Oriented Comput. Appl., 8(1):73–89, March 2014. [see page 40]

[HHKA13] Nikolas Herbst, Nikolaus Huber, Samuel Kounev, and Erich Amrehn. Self-Adaptive Workload Classification and Forecasting for Proactive Resource Provisioning. In Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE 2013), pages 187–198, New York, NY, USA, April 2013. ACM. Among TOP 3 most cited ICPE papers (according to Google Scholar). [see page xiv]

[HHKA14] Nikolas Herbst, Nikolaus Huber, Samuel Kounev, and Erich Amrehn. Self-Adaptive Workload Classification and Forecasting for Proactive Resource Provisioning. Concurrency and Computation - Practice and Experience (CCPE), John Wiley and Sons, Ltd., 26(12):2053–2078, March 2014. 2nd most cited CCPE article (according to Google Scholar). [see pages xiii, 2, and 42]

[HK06] Rob J Hyndman and Anne B Koehler. Another look at measures of forecast accuracy. International Journal of Forecasting, pages 679–688, 2006. [see pages 19, 97, 103, and 113]

[HK08] Rob J Hyndman and Yeasmin Khandakar. Automatic Time Series Forecasting: The Forecast Package for R. Journal of Statistical Software, 26(3):1–22, 2008. [see pages 98, 110, and 165]

[HKO$^+$16] Nikolas Herbst, Rouven Krebs, Giorgos Oikonomou, George Kousiouris, Athanasia Evangelinou, Alexandru Iosup, and Samuel Kounev. Ready for Rain? A View from SPEC Research on the Future of Cloud Metrics. Technical Report SPEC-RG-2016-01, SPEC Research Group — Cloud Working Group, Standard Performance Evaluation Corporation (SPEC), 2016. [see pages xvii, 6, 67, 95, and 161]

[HKR13] Nikolas Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in Cloud Computing: What it is, and What it is Not. In Proceedings of the 10th International Conference on Autonomic Computing (ICAC 2013). USENIX, June 2013. Top 1 most cited ICAC papers (according to Google Scholar). [see pages xv, 6, 36, 37, 38, and 67]

[HKSG02] Rob J Hyndman, Anne B Koehler, Ralph D Snyder, and Simone Grose. A state space framework for automatic forecasting using exponential smoothing methods. International Journal of Forecasting, 18(3):439 – 454, 2002. [see page 32]

[HKWG15a] Nikolas Herbst, Samuel Kounev, Andreas Weber, and Henning Groenda. BUNGEE: An Elasticity Benchmark for Self-Adaptive IaaS Cloud Environments. In SEAMS 2015, pages 46–56. IEEE Press, 2015. [see pages xiv, 94, and 95]

[HKWG15b] Nikolas Herbst, Samuel Kounev, Andreas Weber, and Henning Groenda. BUNGEE: An Elasticity Benchmark for Self-Adaptive IaaS Cloud Environments. In Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2015), May 2015. Acceptance rate: 29%. [see pages 6 and 38]

[HS17] Wilhelm Hasselbring and Guido Steinacker. Microservice architectures for scalability, agility and reliability in e-commerce. In Proceedings of the IEEE International Conference on Software Architecture Workshops (ICSAW) 2017, pages 243–246. IEEE, April 2017. [see page 178]

[Hup09] Karl Huppler. The art of building a good benchmark. In Raghunath Nambiar and Meikel Poess, editors, Performance Evaluation and Benchmarking, volume 5895 of Lecture Notes in Computer Science, pages 18–30. Springer Berlin Heidelberg, 2009. [see pages 38 and 78]

[Hup12] Karl Huppler. Benchmarking with Your Head in the Cloud. In Raghunath Nambiar and Meikel Poess, editors, Topics in Performance Evaluation, Measurement and Characterization, volume 7144 of Lecture Notes in Computer Science, pages 97–110. Springer Berlin Heidelberg, 2012. [see page 78]

[HVK17] Jordan Hochenbaum, Owen S Vallis, and Arun Kejariwal. Automatic anomaly detection in the cloud via statistical learning. arXiv preprint arXiv:1704.07706, 2017. [see page 108]

[Hyn11] R. Hyndman. Cyclic and seasonal time series, 2011. [see page 26]

[Hyn17] Rob J Hyndman. forecast: Forecasting Functions for Time Series and Linear Models, 2017. R package version 8.1. [see pages 98, 110, and 165]

[IAEH$^+$17] Alexey Ilyushkin, Ahmed Ali-Eldin, Nikolas Herbst, Alessandro V. Papadopoulos, Bogdan Ghit, Dick Epema, and Alexandru Iosup. An Experimental Performance Evaluation of Autoscaling Policies for Complex Workflows. In Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE 2017), New York, NY, USA, April 2017. ACM. Best Paper Candidate (1/4). [see pages xiv, 6, 41, 67, and 161]

[IAEH$^+$18] Alexey Ilyushkin, Ahmed Ali-Eldin, Nikolas Herbst, André Bauer, Alessandro V. Papadopoulos, Dick Epema, and Alexandru Iosup. An Experimental Performance Evaluation of Autoscalers for Complex Workflows. ACM Transactions on Modeling and Performance Evaluation of Computing Systems (ToMPECS), 3(2):8:1–8:32, April 2018. [see pages xiii, 6, and 67]

[IDCJ11] Waheed Iqbal, Matthew N Dailey, David Carrera, and Paul Janecek. Adaptive Resource Provisioning for Read Intensive Multi-tier Applications in the Cloud. Future Generation Computer Systems, 27(6):871–879, 2011. [see pages 2, 41, 148, and 149]

[ILFL12] Sadeka Islam, Kevin Lee, Alan Fekete, and Anna Liu. How a Consumer Can Measure Elasticity for Cloud Platforms. In Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering, ICPE '12, pages 85–96, New York, NY, USA, 2012. ACM. [see pages 37, 38, and 66]

[IYE11] A. Iosup, N. Yigitbasi, and D. Epema. On the Performance Variability of Production Cloud Services. In CCGrid 2011, pages 104–113, 2011. [see pages 2 and 161]

[JS15] Brendan Jennings and Rolf Stadler. Resource Management in Clouds: Survey and Research Challenges. Journal of Network and Systems Management, 23(3):567–619, 2015. [see pages 33, 39, and 66]

[JW00] Prasad Jogalekar and Murray Woodside. Evaluating the scalability of distributed systems. IEEE Transactions on Parallel and Distributed Systems, 11:589–603, 2000. [see page 15]

[Kag15a] Kaggle Team. Avito winner's interview: 1st place, Owen Zhang, 2015. [see page 30]

[Kag15b] Kaggle Team. Caterpillar winners' interview: 1st place, Gilberto, Josef, Leustagos, Mario, 2015. [see page 30]

[Kag15c] Kaggle Team. Liberty mutual property inspection, winner's interview: 1st place, Qingchen Wang, 2015. [see page 30]

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. Computer, 36(1):41–50, January 2003. [see page 48]

[KHBZ16] Samuel Kounev, Nikolaus Huber, Fabian Brosig, and Xiaoyun Zhu. A Model-Based Approach to Designing Self-Aware IT Systems and Infrastructures. IEEE Computer, 49(7):53–61, July 2016. [see pages 34, 94, and 96]

[KHvKR11] Michael Kuperberg, Nikolas Herbst, Jóakim Gunnarsson von Kistowski, and Ralf Reussner. Defining and Quantifying Elasticity of Resources in Cloud Computing and Scalable Platforms. Technical report, Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, 76131 Karlsruhe, Germany, 2011. [see pages xvii, 36, and 90]

[KKMZ17] Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu. Self-Aware Computing Systems. Springer Publishing Company, Incorporated, 1st edition, 2017. [see page 17]

[KKZ$^+$00] Tiruvalam Natarajan Krishnamurti, CM Kishtawal, Zhan Zhang, et al. Multimodel ensemble forecasts for weather and seasonal climate. Journal of Climate, 13(23):4196–4216, 2000. [see page 42]

[KLB$^+$17] Samuel Kounev, Peter Lewis, Kirstie Bellman, Nelly Bencomo, Javier Camara, Ada Diaconescu, Lukas Esterle, Kurt Geihs, Holger Giese, Sebastian Götz, Paola Inverardi, Jeffrey Kephart, and Andrea Zisman. The Notion of Self-Aware Computing. In Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu, editors, Self-Aware Computing Systems. Springer Verlag, Berlin Heidelberg, Germany, 2017. [see pages 17, 18, and 189]

[KPSCD09] Stephan Kraft, Sergio Pacheco-Sanchez, Giuliano Casale, and Stephen Dawson. Estimating service resource consumption from response time measurements. In VALUETOOLS '09, pages 1–10, 2009. [see pages 35 and 98]

[KSH14] Rouven Krebs, Philipp Schneider, and Nikolas Herbst. Optimization Method for Request Admission Control to Guarantee Performance Isolation. In Proceedings of the 2nd International Workshop on Hot Topics in Cloud Service Scalability (HotTopiCS 2014), co-located with the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014). ACM, March 2014. [see page xvi]

[KST17] Udayan Khurana, Horst Samulowitz, and Deepak S. Turaga. Feature engineering for predictive modeling using reinforcement learning. CoRR, abs/1709.07150, 2017. [see page 179]

[KTZ09] Dinesh Kumar, Asser Tantawi, and Li Zhang. Real-time performance modeling for adaptive software systems. In VALUETOOLS '09, pages 1–10, 2009. [see pages 34 and 35]

[KWKC12] Max Kuhn, Steve Weston, Chris Keefer, and Nathan Coulter. Cubist models for regression, 2012. `https://cran.r-project.org/web/packages/Cubist/vignettes/cubist.html`, Last accessed: Dec 2017. [see page 35]

[LB14] Sebastian Lehrig and Matthias Becker. Approaching the cloud: Using palladio for scalability, elasticity, and efficiency analyses. In Proceedings of the Symposium on Software Performance 2014, 26-28 November 2015, Stuttgart, Germany, 2014. [see page 49]

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. Nature, 521(7553):436, 2015. [see page 108]

[LBHK18] Veronika Lesch, André Bauer, Nikolas Herbst, and Samuel Kounev. FOX: Cost-Awareness for Autonomic Resource Management in Public Clouds. In Proceedings of the 9th ACM/SPEC International Conference on Performance Engineering (ICPE 2018), New York, NY, USA, April 2018. ACM. Full paper acceptance rate: 23.7%. [see pages xiv and 177]

[LBMAL14] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments. Journal of Grid Computing, 12(4):559–592, 2014. [see pages 2, 33, 39, 66, 94, and 178]

[LHS11] Alysha M. De Livera, Rob J. Hyndman, and Ralph D. Snyder. Forecasting Time Series With Complex Seasonal Patterns Using Exponential Smoothing. Journal of the American Statistical Association, 106(496):1513–1527, 2011. [see pages 94, 98, 163, and 165]

[Li10] Hui Li. Realistic workload modeling and its performance impacts in large-scale escience grids. IEEE Transactions on Parallel and Distributed Systems, 21(4):480–493, 2010. [see page 35]

[LOZC12] Zheng Li, L. O'Brien, He Zhang, and R. Cai. On a Catalogue of Metrics for Evaluating Commercial Cloud Services. In Grid Computing (GRID), 2012 ACM/IEEE 13th International Conference on, pages 164–173, Sept 2012. [see pages 36 and 66]

[LTZ$^+$14] Nian Liu, Qingfeng Tang, Jianhua Zhang, Wei Fan, and Jie Liu. A hybrid forecasting model with parameter optimization for short-term load forecasting of micro-grids. Applied Energy, 129:336–345, 2014. [see page 42]

[LWXZ06] Zhen Liu, Laura Wynter, Cathy H. Xia, and Fan Zhang. Parameter inference of queueing models for IT systems using end-to-end measurements. Elsevier Perform. Evaluation, 63(1):36–60, 2006. [see page 35]

[LYKZ10] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. CloudCmp: Comparing Public Cloud Providers. In Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement, IMC '10, pages 1–14, New York, NY, USA, 2010. ACM. [see pages 36 and 66]

[Man64] John Mandel. The Statistical Analysis of Experimental Data. Dover, 1964. [see page 124]

[MAR+03] D. A. Menascé, V. A. F. Almeida, R. Riedi, F. Ribeiro, R. Fonseca, and W. Meira, Jr. A hierarchical and multiscale approach to analyze e-business workloads. Perform. Eval., 54(1):33–57, September 2003. [see page 36]

[MBS11] Michael Maurer, Ivona Brandic, and Rizos Sakellariou. Enacting Slas in Clouds Using Rules. In Euro-Par 2011, pages 455–466. Springer, 2011. [see page 39]

[MDA04] Daniel A. Menascé, Lawrence W. Dowdy, and Virgilio A. F. Almeida. Performance by Design: Computer Capacity Planning By Example. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004. [see pages 13 and 34]

[Men08] Daniel Menascé. Computing missing service demand parameters for performance models. In CMG Conference, pages 241–248, 2008. [see pages 35 and 98]

[Mey17] David Meyer. Support vector machines, 2017. [see page 165]

[MG11] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. Technical report, U.S. National Institute of Standards and Technology (NIST), 2011. Special Publication 800-145, `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`. [see page 14]

[NBKR13] Qais Noorshams, Dominik Bruhn, Samuel Kounev, and Ralf Reussner. Predictive performance modeling of virtualized storage systems using optimized statistical regression techniques. In ACM/SPEC ICPE 2013, ICPE ’13, pages 283–294, New York, NY, USA, 2013. ACM. [see page 35]

[New99] H. Newton. The periodogram, 1999. [see page 28]

[NKK+11] Jiquan Ngiam, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y Ng. Multimodal deep learning. In Proceedings of the 28th International Conference on Machine Learning (ICML-11), pages 689–696, 2011. [see page 108]

[Noo15] Qais Noorshams. Modeling and Prediction of I/O Performance in Virtualized Environments. PhD thesis, Karlsruhe Institute of Technology (KIT), 2015. [see page 35]

[NSG+13] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. AGILE: Elastic distributed resource scaling for infrastructure-as-a-service. In Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13), pages 69–82, San Jose, CA, 2013. USENIX. [see page 2]

[OCD12] OCDA. Master Usage Model: Compute Infratructure as a Service. Technical report, Open Data Center Alliance (OCDA), 2012. `http://www.opendatacenteralliance.org/docs/ODCA_Compute_IaaS_MasterUM_v1.0_Nov2012.pdf`. [see page 14]

[Om14] P-O Östberg and more. The CACTOS Vision of Context-Aware Cloud Topology Optimization and Simulation. In Proceedings of the Sixth IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pages 26–31, Singapore, December 2014. IEEE Computer Society. [see page 49]

[P$^+$09] Pradeep Padala et al. Automated control of multiple virtualized resources. In ACM European Conference on Computer Systems, pages 13–26. ACM, 2009. [see page 40]

[PAEÅ$^+$16] Alessandro Papadopoulos, Ahmed Ali-Eldin, Karl-Erik Årzén, Johan Tordsson, and Erik Elmroth. PEAS: A Performance Evaluation Framework for Auto-Scaling Strategies in Cloud Applications. ACM ToMPECS, 1(4):1–31, August 2016. [see pages 37, 41, 67, and 161]

[PL05] Ping-Feng Pai and Chih-Sheng Lin. A hybrid arima and support vector machines model in stock price forecasting. Omega, 33(6):497–505, 2005. [see page 42]

[Pla11] P. Plavchan. What is a periodogram?, 2011. [see page 28]

[PS12] Guillaume Pierre and Corina Stratan. ConPaaS: a Platform for Hosting Elastic Cloud Applications. IEEE Internet Computing, 16(5):88–92, 2012. [see page 40]

[PSB$^+$09] D. C. Plummer, D. M. Smith, T. J. Bittman, D. W. Cearley, D. J. Cappuccio, D. Scott, R. Kumar, and B. Robertson. Study: Five Refining Attributes of Public and Private Cloud Computing. Technical report, Gartner, 2009. `https://www.gartner.com/newsroom/id/1035013`, last consulted Jan. 2017. [see page 13]

[PVB$^+$18] Alessandro Vittorio Papadopoulos, Laurens Versluis, André Bauer, Nikolas Herbst, Jóakim von Kistowski, Ahmed Ali-Eldin, Cristina Abad, J. Nelson Amaral, Petr Tuma, and Alexandru Iosup. Methodological Principles for Reproducible Performance Evaluation in Cloud Computing. Under Resubmission to IEEE Transactions on Cloud Computing (TCC), 2018. [see pages xiii, 3, and 6]

[Q$^+$92] John R Quinlan et al. Learning with continuous classes. In Proceedings of the 5th Australian joint Conference on Artificial Intelligence, volume 92, pages 343–348. Singapore, 1992. [see page 35]

[RBX+09] Jia Rao, Xiangping Bu, Cheng-Zhong Xu, Leyi Wang, and George Yin. VCONF: a Reinforcement Learning Approach to Virtual Machines Auto-configuration. In ACM ICAC 2009, pages 137–146. ACM, 2009. [see page 40]

[RLGPC+99] Arcadio Reyes-Lecuona, E González-Parada, E Casilari, JC Casasola, and A Diaz-Estrella. A page-oriented www traffic model for wireless system simulations. In Proceedings ITC, volume 16, pages 1271–1280, 1999. [see page 36]

[Rob16] Mike Roberts. Serverless architectures. `https://martinfowler.com/articles/serverless.html`, 2016. Last accessed in January 2018. [see page 176]

[RV95] Jerome Rolia and Vidar Vetland. Parameter estimation for performance models of distributed application systems. In CASCON '95, page 54. IBM Press, 1995. [see pages 34 and 35]

[SA12] D.M. Shawky and A.F. Ali. Defining a Measure of Cloud Computing Elasticity. In Systems and Computer Science (ICSCS), 2012 1st International Conference on, pages 1–5, Aug 2012. [see pages 37 and 38]

[SCBK15] Simon Spinner, Giuliano Casale, Fabian Brosig, and Samuel Kounev. Evaluating Approaches to Resource Demand Estimation. Elsevier Performance Evaluation, 92:51 – 71, October 2015. [see pages 6, 34, and 94]

[Sch12] Edwin Schouten. Rapid Elasticity and the Cloud, September 2012. `http://thoughtsoncloud.com/index.php/2012/09/rapid-elasticity-and-the-cloud/`, last consulted Jan. 2017. [see page 14]

[SCZK14] Simon Spinner, Giuliano Casale, Xiaoyun Zhu, and Samuel Kounev. LibReDE: A library for Resource Demand Estimation. In ACM/SPEC ICPE 2014, pages 227–228. ACM, 2014. [see pages 96 and 98]

[SHK+15] Simon Spinner, Nikolas Herbst, Samuel Kounev, Xiaoyun Zhu, Lei Lu, Mustafa Uysal, and Rean Griffith. Proactive Memory Scaling of Virtualized Applications. In Proceedings of the 2015 IEEE 8th International Conference on Cloud Computing (IEEE CLOUD 2015), pages 277–284. IEEE, June 2015. Acceptance Rate: 15%. [see page xiv]

[SIm15] Siqi Shen, Alexandru Iosup, and more. An availability-on-demand mechanism for datacenters. In IEEE/ACM CCGrid, pages 495–504, 2015. [see page 177]

[SMC+08] Piyush Shivam, Varun Marupadi, Jeff Chase, Thileepan Subramaniam, and Shivnath Babu. Cutting Corners: Workbench Automation for Server Benchmarking. In USENIX 2008 Annual Technical Conference on Annual Technical Conference, ATC'08, pages 241–254, Berkeley, CA, USA, 2008. USENIX Association. [see page 83]

[SSGW11] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In Proceedings of the 2nd ACM Symposium on Cloud Computing, SOCC '11, pages 5:1–5:14, New York, NY, USA, 2011. ACM. [see page 2]

[Sul12] Basem Suleiman. Elasticity Economics of Cloud-Based Applications. In Proceedings of the 2012 IEEE Ninth International Conference on Services Computing, SCC '12, pages 694–695, Washington, DC, USA, 2012. IEEE Computer Society. [see pages 37 and 66]

[SWHB06] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: a cautionary tale. In Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3, NSDI'06, pages 18–18, Berkeley, CA, USA, 2006. USENIX Association. [see pages 19 and 81]

[SWK16] Simon Spinner, Jürgen Walter, and Samuel Kounev. A Reference Architecture for Online Performance Model Extraction in Virtualized Environments. In ACM/SPEC ICPE 2017, pages 57–62. ACM, 2016. [see page 96]

[TJDB06] Gerald Tesauro, Nicholas K Jong, Rajarshi Das, and Mohamed N Bennani. A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation. In IEEE ICAC 2006, pages 65–73. IEEE, 2006. [see page 40]

[TTP14] Christian Tinnefeld, Daniel Taschik, and Hasso Plattner. Quantifying the Elasticity of a Database Management System. In DBKDA 2014, The Sixth International Conference on Advances in Databases, Knowledge, and Data Applications, pages 125–131, 2014. [see pages 37 and 66]

[U+05] Bhuvan Urgaonkar et al. An Analytical Model for Multi-Tier Internet Services and its Applications. In ACM SIGMETRICS, 2005. [see page 148]

[USC+08] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile Dynamic Provisioning of Multi-tier Internet Applications. ACM TAAS, 3(1):1, 2008. [see pages 2, 40, 41, and 148]

[VHNC10] Jan Verbesselt, Rob Hyndman, Glenn Newnham, and Darius Culvenor. Detecting trend and seasonal changes in satellite image time series. Remote Sensing of Environment, 114(1):106 – 115, 2010. [see pages 24, 52, and 113]

[vHRH08] André van Hoorn, Matthias Rohr, and Wilhelm Hasselbring. Generating probabilistic and intensity-varying workload for web-based software systems. In Proceedings of the SPEC international workshop on Performance Evaluation: Metrics, Models and Benchmarks, SIPEW '08, pages 124–143, Berlin, Heidelberg, 2008. Springer-Verlag. [see page 34]

[vHVS+14] André van Hoorn, Christian Vögele, Eike Schulz, Wilhelm Hasselbring, and Helmut Krcmar. Automatic extraction of probabilistic workload specifications for load testing session-based application systems. In Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS '14, pages 139–146, ICST, Brussels, Belgium, Belgium, 2014. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). [see page 34]

[vKHK14a] Jóakim von Kistowski, Nikolas Herbst, and Samuel Kounev. LIMBO: A Tool For Modeling Variable Load Intensities. In Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014), ICPE '14, pages 225–226, New York, NY, USA, March 2014. ACM. [see pages xvi, 66, and 81]

[vKHK14b] Jóakim von Kistowski, Nikolas Herbst, and Samuel Kounev. LIMBO Load Intensity Modeling Tool. Research Group of the Standard Performance Evaluation Corporation (SPEC), Peer-reviewed Tools Repository, `https://research.spec.org/tools/overview/limbo.html`, 2014. [see page xvii]

[vKHK14c] Jóakim von Kistowski, Nikolas Herbst, and Samuel Kounev. Modeling Variations in Load Intensity over Time. In Proceedings of the 3rd International Workshop on Large-Scale Testing (LT 2014), co-located with the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014), pages 1–4, New York, NY, USA, March 2014. ACM. [see page xvi]

[vKHK14d] Jóakim von Kistowski, Nikolas Herbst, and Samuel Kounev. Using and Extending LIMBO for the Descriptive Modeling of Arrival Behaviors. In Proceedings of the Symposium on Software Performance 2014, pages 131–140. University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, November 2014. Best Poster Award. [see page xvi]

[vKHK+17] Jóakim von Kistowski, Nikolas Herbst, Samuel Kounev, Henning Groenda, Christian Stier, and Sebastian Lehrig. Modeling and Extracting Load Intensity Profiles. ACM Transactions on Autonomous and Adaptive Systems (TAAS), 11(4):23:1–23:28, January 2017. [see pages xiii, 5, and 38]

[vKHZ+15] Jóakim von Kistowski, Nikolas Herbst, Daniel Zoller, Samuel Kounev, and Andreas Hotho. Modeling and Extracting Load Intensity Profiles. In Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2015), May 2015. Acceptance rate: 29%. [see page xiv]

[vMvHH11] Robert von Massow, André van Hoorn, and Wilhelm Hasselbring. Performance simulation of runtime reconfigurable component-based software architectures. In Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors, Software Architecture, pages 43–58, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. [see page 40]

[Wei11] Joe Weinman. Time is Money: The Value of "On-Demand", 2011. (accessed July 9, 2014). [see pages 37 and 66]

[WHGK14] Andreas Weber, Nikolas Herbst, Henning Groenda, and Samuel Kounev. Towards a Resource Elasticity Benchmark for Cloud Environments. In Proceedings of the 2nd International Workshop on Hot Topics in Cloud Service Scalability (HotTopiCS 2014), co-located with the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014), HotTopiCS '14, pages 5:1–5:8, New York, NY, USA, March 2014. ACM. [see page xvi]

[WM97] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation, 1(1):67–82, Apr 1997. [see pages 41, 107, and 108]

[Wm12] Wei Wang and more. Application-Level CPU Consumption Estimation: Towards Performance Isolation of Multi-tenancy Web Applications. In IEEE CLOUD 2012, pages 439–446, June 2012. [see pages 34, 35, and 98]

[Wol11] Rich Wolski. Cloud Computing and Open Source: Watching Hype meet Reality, May 2011. `http://www.ics.uci.edu/~ccgrid11/files/ccgrid-11_Rich_Wolsky.pdf`, last consulted Jan. 2017. [see page 14]

[WSMH09] Xiaozhe Wang, Kate Smith-Miles, and Rob Hyndman. Rule Induction for Forecasting Method Selection: Meta-learning the Characteristics of Univariate Time Series. Neurocomputing, 72(10 - 12):2581 – 2594, 2009. [see pages 31, 42, 57, and 98]

[ZBH+17] Marwin Züfle, André Bauer, Nikolas Herbst, Valentin Curtef, and Samuel Kounev. Telescope: A Hybrid Forecast Method for Univariate Time Series. In Proceedings of the International work-conference on Time Series 2017, September 2017. [see pages xv and 7]

[ZCS07] Qi Zhang, Ludmila Cherkasova, and Evgenia Smirni. A Regression-based Analytic Model for Dynamic Resource Provisioning of Multi-tier Applications. In IEEE ICAC 2007, pages 27–27. IEEE, 2007. [see page 40]

[ZF13] Netanel Zakay and Dror G. Feitelson. Workload resampling for performance evaluation of parallel job schedulers. In Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering, ICPE '13, pages 149–160, New York, NY, USA, 2013. ACM. [see page 34]

[Zha03] G Peter Zhang. Time series forecasting using a hybrid arima and neural network model. Neurocomputing, 50:159–175, 2003. [see pages 32 and 42]

[ZZB11] Qi Zhang, Quanyan Zhu, and Raouf Boutaba. Dynamic resource allocation for spot markets in cloud computing environments. In IEEE UCC 2011, pages 178–185. IEEE, 2011. [see page 177]