



DISSERTATION | DOCTORAL THESIS

Titel | Title

Dynamic API Patterns Optimization: Using Predictive Analysis to
Adapt Microservices on Service Mesh Deployments

verfasst von | submitted by

tekn.mag. Amine El Malki

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of
Doktor der technischen Wissenschaften (Dr.techn.)

Wien | Vienna, 2024

Studienkennzahl lt. Studienblatt | Degree
programme code as it appears on the
student record sheet:

UA 786 880

Dissertationsgebiet lt. Studienblatt | Field of
study as it appears on the student record
sheet:

Informatik

Betreut von | Supervisor:

Univ.-Prof. Dr. Uwe Zdun

Acknowledgements

I wish to extend my most profound appreciation to my advisor, Uwe Zdun, for his invaluable guidance, encouragement, and unwavering support during my doctoral research. His insightful feedback and constructive criticism played a pivotal role in shaping both my research and academic development. I am also profoundly grateful to all my research colleagues whose expertise and resources were indispensable to realizing my research. Their contributions were instrumental in its success.

I want to express heartfelt gratitude to my family, my wife, Zineb, my parents, and my brother and sister, Chabib and Najiba, for their unconditional love, support, and encouragement throughout my academic journey. Their steadfast belief in me has consistently provided me with strength and motivation.

Lastly, I would like to acknowledge the faculty of Computer Science at the University of Vienna for providing the necessary resources and infrastructure for conducting my research. Their support has been invaluable in enabling me to accomplish my academic objectives.

To everyone who has provided unwavering support and encouragement throughout my doctoral journey, I extend my sincere thanks.

Abstract

Microservices are increasingly emerging as the predominant approach for developing software in the cloud and service-oriented computing. Service meshes have been introduced as specialized infrastructure to oversee networks of containerized microservices, addressing challenges such as complexity, manageability, and interoperability, particularly in extensive microservice architectures. Regrettably, there is currently a lack of specific architectural guidance for designing microservices and selecting technology options within a service mesh. Consequently, there exists significant uncertainty surrounding the design and utilization of microservices within a service mesh environment today.

On the other hand, numerous API patterns and best practices have been developed to enhance the quality attributes of microservices-based architectures, including Rate Limiting and Circuit Breaking, aimed at improving reliability, availability, scalability, and performance. While estimates regarding these properties would be beneficial, especially in the early stages of architectural design, the true impact of these patterns on such properties has not yet been rigorously examined. *Request Bundle* is one such pattern aimed at enhancing performance. While the pattern offers significant performance improvements, it can also entail notable drawbacks such as heightened development effort and application complexity. To date, there is a scarcity of rigorously acquired knowledge to assess whether implementing *Request Bundle* justifies the costs within a given Microservice architecture. Moreover, there has been no comprehensive study measuring the impact of these techniques and their combinations in specific configurations, particularly within a large distributed system workload setting.

In this thesis, we first conducted a qualitative, in-depth study of existing practices in the service mesh domain, utilizing a model-based approach. We systematically examined 40 reports of established practices from practitioners. Our study resulted in a rigorously specified, reusable architectural decision model. Within this model, we identified 14

Abstract

architectural design decisions, comprising 47 decision outcomes and influenced by 77 decision drivers in total.

After that, we thoroughly study the *Rate Limit* pattern by introducing an analytical model that takes into account specific workload configurations and predefined rate limits, accurately predicting the success and failure rates of backend services. Additionally, the model offers a method for adaptively fine-tuning rate limits. We utilized service meshes to set up the backend infrastructure, configure various workload scenarios, and facilitate monitoring and data collection. We conducted two extensive data experiments to validate the model and assess the impacts of rate limiting. The first experiment was conducted on a private cloud to minimize latency and potential biases, while the second was carried out on the Google Cloud Platform to test the model in a realistic cloud environment.

To study the *Request Bundle* pattern, we also opted for a service mesh-based infrastructure due to its capability to facilitate rapid and dynamic deployment of various versions of the same microservice. This significantly reduced the time and effort required to experiment with different pattern configurations. To assess the performance impact of request bundling, we developed a regression model and conducted a multivariate regression analysis. These selected regression models offer distributed system engineers and architects a means to estimate performance improvement in round-trip time, both with and without request bundling. The approach outlined can be adapted to other Microservice architectures or utilized to investigate other performance-related Microservice patterns.

Furthermore, we conduct experimental research to investigate the effects of combining the above patterns with *Load Balancing*. We base our study on a realistic, third-party microservice-based application deployed both in a private cloud environment and on the Amazon Web Services (AWS) cloud, employing 130 different configurations. Each configuration is executed 500 times in the private cloud, totaling over 4500 hours of runtime, and 200 times on AWS, totaling more than 3900 hours of runtime. The complexity of experimenting with such compound scenarios is made feasible by leveraging a service mesh-based infrastructure, which furnishes the necessary tools and functionalities to implement and monitor various combinations of these API patterns. We develop regression models from the collected data to predict the performance and reliability impacts of combining these techniques and patterns.

In addition to the specific analyses presented in this study, we propose a general and largely automated method that can be iteratively employed to evaluate similar techniques and patterns for their quality properties and a benchmark suite founded on a realistic open-source microservice application deployed on the service mesh-based infrastructure. This suite facilitates the automatic generation of comprehensive decision tree models. Subsequently, these models are analyzed to formulate API design recommendation algorithms aimed at enhancing API operations in terms of performance and reliability. These algorithms are stored in catalogs for future reuse. This API recommendation pipeline presents the basis for attaining self-adaptation and automation of API configuration and settings optimizations.

Kurzfassung

Microservices werden zunehmend als der vorherrschende Ansatz zur Entwicklung von Software in der Cloud und im serviceorientierten Computing angesehen. Service-Meshes wurden als spezialisierte Infrastruktur eingeführt, um Netzwerke containerisierter Microservices zu überwachen und Herausforderungen wie Komplexität, Verwaltbarkeit und Interoperabilität anzugehen, insbesondere in umfangreichen Microservice-Architekturen. Bedauerlicherweise fehlt derzeit eine spezifische architektonische Anleitung zur Gestaltung von Microservices und zur Auswahl von Technologieoptionen innerhalb eines Service-Meshes. Folglich besteht heutzutage erhebliche Unsicherheit hinsichtlich der Gestaltung und Nutzung von Microservices innerhalb einer Service-Mesh-Umgebung.

Andererseits wurden zahlreiche API-Patterns und bewährte Verfahren entwickelt, um die Qualitätseigenschaften von auf Microservices basierenden Architekturen zu verbessern, darunter Rate Limiting und Circuit Breaking, die darauf abzielen, Zuverlässigkeit, Verfügbarkeit, Skalierbarkeit und Leistungsfähigkeit zu verbessern. Während Einschätzungen zu diesen Eigenschaften insbesondere in den frühen Phasen der architektonischen Gestaltung nützlich wären, wurde der tatsächliche Einfluss dieser Patterns auf solche Eigenschaften bisher nicht gründlich untersucht. *Request Bundle* ist ein solches Pattern, das die Leistung verbessern soll. Obwohl das Pattern erhebliche Leistungsverbesserungen bietet, kann es auch bedeutende Nachteile wie erhöhten Entwicklungsaufwand und Anwendungskomplexität mit sich bringen. Bisher gibt es nicht genug gründlich erworbenes Wissen um zu bewerten, ob die Implementierung von *Request Bundle* die Kosten in einer gegebenen Microservice-Architektur rechtfertigt. Darüber hinaus liegt bisher keine umfassende Studie vor, die den Einfluss dieser Techniken und ihrer Kombinationen in spezifischen Konfigurationen, insbesondere innerhalb eines großen verteilten Workload-Systems, misst.

In dieser Arbeit führten wir zunächst eine qualitative, eingehende Untersuchung der bestehenden Praktiken im Bereich des Service Meshes durch, wobei wir einen modell-

Kurzfassung

basierten Ansatz verwendeten. Wir untersuchten systematisch 40 Berichte etablierter Praktiken von Praktiker*innen. Unsere Studie führte zu einem gründlich spezifizierten, wiederverwendbaren architektonischen Entscheidungsmodell. Innerhalb dieses Modells identifizierten wir 14 architektonische Entwurfsentscheidungen, die 47 Entscheidungsergebnissen und insgesamt 77 Entscheidungsantrieben unterliegen.

Anschließend untersuchten wir das Pattern *Rate Limit* gründlich, indem wir ein analytisches Modell einführten, das spezifische Arbeitslastkonfigurationen und vordefinierte Rate Limits berücksichtigt und die Erfolgs- und Ausfallraten von Backend-Services genau vorhersagt. Darüber hinaus bietet das Modell eine Methode zur adaptiven Feinabstimmung von Rate Limits. Wir nutzten Service-Meshes, um die Backend-Infrastruktur einzurichten, verschiedene Arbeitslastszenarien zu konfigurieren und Überwachung und Datensammlung zu erleichtern. Wir führten zwei umfangreiche Datenexperimente durch, um das Modell zu validieren und die Auswirkungen der Rate-Limitierung zu bewerten. Das erste Experiment wurde in einer privaten Cloud durchgeführt, um Latenz und potenzielle Bias-Effekte zu minimieren, während das zweite auf der Google Cloud Platform durchgeführt wurde, um das Modell in einer realistischen Cloud-Umgebung zu testen.

Um das Pattern *Request Bundle* zu untersuchen, entschieden wir uns ebenfalls für eine Service-Mesh-basierte Infrastruktur, da sie die schnelle und dynamische Bereitstellung verschiedener Versionen desselben Microservices ermöglicht. Dies reduzierte die Zeit und den Aufwand, um verschiedene Patternkonfigurationen auszutesten, signifikant. Um den Leistungseinfluss des Request Bundlings zu bewerten, entwickelten wir ein Regressionsmodell und führten eine multivariate Regressionsanalyse durch. Diese ausgewählten Regressionsmodelle bieten verteilten Systemingenieur*innen und -architekt*innen eine Möglichkeit, die Leistungsverbesserung in der Round-Trip-Zeit zu schätzen, sowohl mit als auch ohne Request Bundling. Der skizzierte Ansatz kann an andere Microservice-Architekturen angepasst oder genutzt werden, um andere leistungsbezogene Microservice-Patterns zu untersuchen.

Darüber hinaus führten wir experimentelle Forschung durch, um die Auswirkungen der Kombination der oben genannten Patterns mit *Load Balancing* zu untersuchen. Wir basierten unsere Studie auf einer realistischen, auf Drittanbieter-Microservices basierenden Anwendung, die sowohl in einer privaten Cloud-Umgebung als auch in der Amazon Web Services (AWS) Cloud bereitgestellt wird und 130 verschiedene Konfigurationen ver-

wendet. Jede Konfiguration wurde 500 Mal in der privaten Cloud ausgeführt, insgesamt über 4.500 Stunden Laufzeit, und 200 Mal in AWS, insgesamt mehr als 3.900 Stunden Laufzeit. Die Komplexität des Experimentierens mit solchen komplexen Szenarien wird durch die Nutzung einer Service-Mesh-basierten Infrastruktur ermöglicht, die die erforderlichen Tools und Funktionalitäten zur Implementierung und Überwachung verschiedener Kombinationen dieser API-Patterns bereitstellt. Wir entwickelten Regressionsmodelle aus den gesammelten Daten, um die Leistungs- und Zuverlässigkeitsauswirkungen der Kombination dieser Techniken und Patterns vorherzusagen.

Zusätzlich zu den spezifischen Analysen in dieser Studie schlagen wir eine allgemeine und weitgehend automatisierte Methode vor, die iterativ eingesetzt werden kann, um ähnliche Techniken und Patterns hinsichtlich ihrer Qualitätsmerkmale zu bewerten, sowie eine Benchmark-Suite, die auf einer realistischen Open-Source-Microservice-Anwendung basiert, die auf der Service-Mesh-basierten Infrastruktur bereitgestellt wird. Diese Suite erleichtert die automatische Generierung umfassender Entscheidungsbaummodelle. Im Anschluss werden diese Modelle analysiert, um Algorithmen zur API-Designempfehlung zu formulieren, die darauf abzielen, die API-Operationen hinsichtlich Leistung und Zuverlässigkeit zu verbessern. Diese Algorithmen werden in Katalogen für zukünftige Wiederverwendung gespeichert. Diese API-Empfehlungspipeline bildet die Grundlage für die Erreichung von Selbstanpassung und Automatisierung der API-Konfiguration und der Optimierung von Einstellungen.

Contents

Acknowledgements	i
Abstract	iii
Kurzfassung	vii
List of Tables	xvii
List of Figures	xix
List of Algorithms	xxiii
1 Introduction	3
1.1 Background	3
1.2 Thesis motivation	5
1.3 Research questions	8
1.4 Thesis structure	8
2 State of the Art	11
2.1 Research Context	11
2.2 Related Work	14
2.2.1 Related Works on Service Meshes	14
2.2.2 API Rate Limit and its Impact on Reliability	15
2.2.3 API Request Bundle and its Impact on Performance	16
2.2.4 Combining API patterns and its Impact of Performance and Reli-	
ability	16
2.2.5 Improving API Operations by Automated Recommendation	18

3 Research problems and contributions analysis	19
3.1 Research problems	19
3.2 Research contributions	20
3.3 List of publications	21
3.4 Research methods	23
4 Guiding Architectural Decision Making on Service Mesh Based Microservice	
Architectures	27
4.1 Introduction	27
4.2 Related Work	29
4.3 Research Method	30
4.4 Service Mesh Design Decisions	33
4.4.1 Managed Cross-Service Communication Decision	33
4.4.2 Managed Ingress Communication Decision	34
4.4.3 Traffic Control Decision	35
4.4.4 Service Mesh Expansion Decision	38
4.4.5 Central Services and Proxy Tasks	38
4.5 Challenges in API patterns implementation	42
4.6 Estimation of Uncertainty Reduction	44
4.7 Discussion and Threats to Validity	46
4.8 Conclusions	47
5 API Rate Limit Adoption – A pattern collection	49
5.1 Positioning of Rate Limiter	49
5.1.1 Internal Rate Limiter	49
5.1.2 External Rate Limiter	51
5.2 Scope of Rate Limiter	52
5.2.1 Global Rate Limiter	52
5.2.2 Local Rate Limiter	53
6.1 Introduction	57
6.2 Related work	59
6.3 Background on Rate Limiting	61
6.4 Analytical model	62

6.5	Empirical evaluation	65
6.5.1	Benchmark workload scenarios	66
6.5.2	Architecture overview	67
6.5.3	Configuration details	68
6.5.4	Experimental setup	69
6.5.5	Results analysis: Experiments using the private cloud	70
6.5.6	Results analysis: Experiments using GCP	72
6.6	Prediction error and threats to validity	72
6.6.1	Mean Absolute Percentage Error	72
6.6.2	Threats to validity	73
6.7	Conclusion and future work	74
7.1	Introduction	84
7.2	Related Work	85
7.3	Background: Request Bundling	86
7.4	Regression model	87
7.4.1	Independent and dependent variables	87
7.5	Empirical study	88
7.5.1	Scope definition	89
7.5.2	Lakeside Mutual application	90
7.5.3	Configuration details	90
7.5.4	Development and deployment details	91
7.5.5	Launching the experiment	92
7.5.6	Experiment design and hypotheses	92
7.6	Data collection and analysis	93
7.6.1	Multivariate regression analysis	93
7.6.2	Selected models	95
7.7	Validity evaluation	96
7.8	Conclusion and future work	97
8	Combining API Patterns in Microservice Architectures: Performance and Reliability Analysis	99
8.1	Introduction	100
8.2	Related work	103

8.3	Background about Load Balancing, Request Bundle and Rate Limit	105
8.4	Approach overview	106
8.5	Regression model description	107
8.5.1	Independent and dependent variables	107
8.6	Experimentation setup	109
8.6.1	Application under study	109
8.6.2	Infrastructure details	110
8.6.3	Running the experiment	111
8.7	Multivariate regression analysis	112
8.7.1	Data collection and analysis	112
8.8	Discussion and threats to validity	115
8.8.1	Models comparison	115
8.8.2	Tradeoff analysis	118
8.8.3	Models evaluation	121
8.8.4	Threats to validity	122
8.9	Conclusion and future work	124
9.1	Introduction	128
9.2	Related work	130
9.3	Research methods	131
9.3.1	Regression model	131
9.3.2	Benchmark description	132
9.4	Approach description	133
9.5	API analysis and recommendations	134
9.5.1	API analysis	134
9.5.2	API design recommendation algorithms	135
9.6	Discussion and threats to validity	138
9.6.1	Discussion	138
9.6.2	Threats to validity	142
9.7	Conclusion and future work	143
10	Microservices API Analytics: Bridging the Gap Between Runtime and Static	
	Analysis for Enhanced Software Quality	147
10.1	Challenges in Microservice API Analysis	148

10.2 Microservice API Analytics	151
10.2.1 Source Code Detector-based Analysis	152
10.2.2 Runtime Benchmark-based Model for Self-adaptation	155
10.3 The combined approach for microservices API-analytics	159
10.4 CONCLUSION	161
11 Conclusion	165
Bibliography	169

List of Tables

4.1 Knowledge Sources Included in the Study	31
4.2 Uncertainty Reduction Estimation	46
6.1 Definition of Parameters	65
6.2 GCP cost details (in Euros)	66
6.3 Workload configurations for experiments using the private cloud	70
6.4 Workload configurations for experiments using GCP	71
6.5 Results from the model and the experiments using the private cloud (50 runs)	79
6.6 Results from the model and the experiments using GCP (50 runs)	80
6.7 Calculated MAPE for all configurations up to 50 runs	81
7.1 Definition of Parameters	89
7.2 Data collected during the experiments	94
7.3 Models' description	94
7.4 Models' summary	95
8.1 Definition of Parameters	109
8.2 AWS cost details (in US Dollars)	111
8.3 Combined models' description on the private cloud: total time prediction	113
8.4 Combined models' description on AWS: total time prediction	113
8.5 Combined models' description on the private cloud: failure rate prediction	113
8.6 Combined models' description on AWS: failure rate prediction	113
8.7 Combined models' summary: total time & failure rate prediction	114
8.8 Models summary (on the private cloud): Pillai's Trace test	115
8.9 Models summary (on AWS): Pillai's Trace test	115

List of Tables

8.10 Calculated MAE and MAPE for failure rate prediction errors and total_time prediction errors respectively	122
9.1 Definition of Parameters	131
9.2 Sample data: LB=Load Balancing, RB=Request Bundle, OP=Operation, SZ=Bundle Size, TT=Total Time(ms), FR=Failure Rate(%)	145
9.3 Infrastructure details of the experiments	146
10.1 Metrics description (selected metrics from MZ21, MZP22, MZ23, GZ21, GZ22)	154
10.2 Models' description	158

List of Figures

1.1 Microservices API Architecture	4
1.2 Multi-level rate limiting	7
1.3 API patterns combined	7
2.1 Multiple microservices versions: Request Bundle On and Off	13
2.2 Multi-level load balancing	13
3.1 Research mapping overview	21
4.1 Managed Cross Service Communication Decision	35
4.2 Managed Ingress Communication Decision	36
4.3 Traffic Control Decision	37
4.4 Service Mesh Expansion Decision	38
6.1 <i>SR</i> results from all the experiments (50 runs)	76
6.2 <i>NRL_FR</i> results from all the experiments (50 runs)	77
6.3 <i>RL_FR</i> results from all the experiments (50 runs)	78
8.1 Approach overview	107
8.2 Performance analysis (on the private cloud)	116
8.3 Performance analysis (on AWS)	116
8.4 Reliability analysis (on the private cloud)	117
8.5 Reliability analysis (on AWS)	117
8.6 Load Balancing vs. Request Bundle: performance perspective (on the private cloud)	119
8.7 Load Balancing vs. Request Bundle: performance perspective (on AWS)	119
8.8 Load Balancing vs. Request Bundle: reliability perspective (on private cloud and AWS)	120

List of Figures

8.9 Load Balancing vs. Rate Limit: reliability perspective (on the private cloud)	120
8.10 Load Balancing vs. Rate Limit: reliability perspective (on AWS)	121
8.11 Load Balancing vs. Rate Limit: performance perspective (on the private cloud)	121
8.12 Load Balancing vs. Rate Limit: performance perspective (on AWS)	122
9.1 API design recommendation pipeline	133
9.2 Load balancing, request bundle & rate limit combined: Performance decision tree on AWS (milliseconds)	135
9.3 Load balancing, request bundle & rate limit combined: Reliability decision tree on AWS (% failure)	136
9.4 Load balancing & rate limit combined: Performance decision tree on AWS (milliseconds)	136
9.5 Load balancing & rate limit combined: Reliability decision tree on AWS (% failure)	138
9.6 Rate limit & request bundle combined: Performance decision tree on AWS (milliseconds)	138
9.7 Rate limit & request bundle combined: Reliability decision tree on AWS (% failure)	139
9.8 Load balancing, request bundle & rate limit combined: Performance decision tree on the private cloud (milliseconds)	139
9.9 Load balancing, request bundle & rate limit combined: Reliability decision tree on the private cloud (% failure)	140
9.10 Load balancing & rate limit combined: Performance decision tree on the private cloud (milliseconds)	140
9.11 Load balancing & rate limit combined: Reliability decision tree on the private cloud (% failure)	141
9.12 Rate limit & request bundle combined: Performance decision tree on the private cloud (milliseconds)	141
9.13 Rate limit & request bundle combined: Reliability decision tree on the private cloud (% failure)	142

10.1 Example of Excessive Data Exposure: The <i>collect-data</i> API request more	
data from the <i>customer</i> and <i>product</i> APIs than required by the client.	
This creates an avoidable security risk [GZ22].	153
10.2 Models (1 & 2) vs experiment using Create method	158
10.3 Approach Overview: The insights from the static analysis process are	
combined with the results from various benchmark runs to form a hol-	
istic quality model, allowing software architects to make more qualified	
architectural decisions.	160

List of Algorithms

1	API design recommendation (AWS)	137
2	API design recommendation (private cloud)	137

Part I

Research motivations and strategy

1 Introduction

1.1 Background

API stands for Application Programming Interface, which, as the name suggests, allows access to applications through predefined methods, including parameters and desired outcomes. Many technologies (including REST^[1]) have implemented this paradigm as an efficient and secure way to provide different services for API clients. As APIs have become the central point of communication between clients and servers, they have come up with many challenges regarding various quality properties, including performance and reliability.

Microservices^[2] originated from concepts already introduced in service-oriented architectures, including loose coupling and separation of concerns. Each microservice is a small containerized virtual machine that provides specific services related to its predefined domain through a set of APIs. The interaction between microservices should be minimal and only made through asynchronous API calls. Usually, each of these microservices runs its own database, but they can also share one shared database, which could considerably degrade performance and increase complexity.

API Gateway^[3] is a related paradigm introduced as a secure and efficient way to access APIs. It also ensures a high degree of interoperability between API clients and servers. However, API Gateways are usually regarded as single points of failure and have too much dependency on the backend, especially in highly volatile environments such as microservices and cloud-based architectures. Those architectures have the particularity of frequent change and evolution; thus, API Gateways have to cope with such changes dynamically and quickly, making the complexity even higher.

¹<https://www.ibm.com/topics/rest-apis>

²<https://microservices.io/>

³<https://www.ibm.com/blog/api-gateway/>

1 Introduction

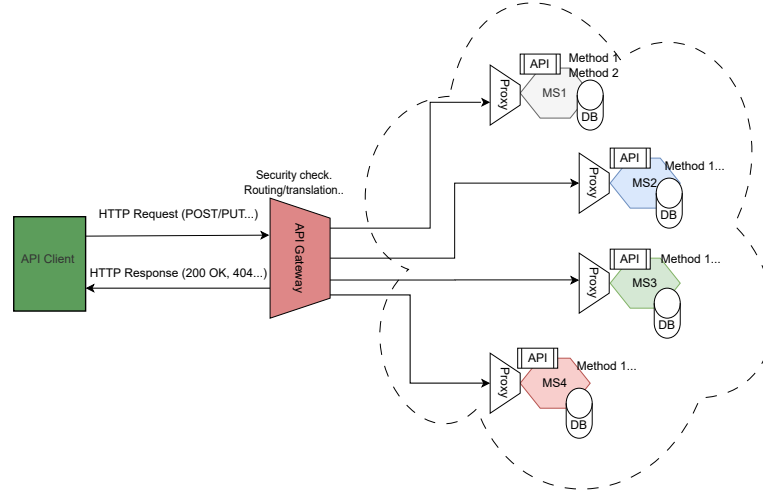


Figure 1.1: Microservices API Architecture

Service mesh is a new concept that has emerged with the widespread adoption of microservices-based architectures. As described more in detail in Chapter 4 it is a dedicated layer that manages such complex and large infrastructures. Instead of having one API Gateway, service meshes provide a microgateway or *Service Proxy* for each microservice in addition to the traditional API Gateway or *Front Proxy*. This design not only prevents the API Gateway from being overloaded but also provides a multi-level and multi-granularity control of the incoming traffic.

Figure 1.1 shows the typical architecture that comprises the concepts described above. This architecture is used to setup experimentation in Chapters 6, 7, 8 and 9. An *API Client* launches a series of HTTP-based requests to an *API Backend*, which pass through an *API Gateway*. The gateway is responsible for ensuring access control and routing to the proper microservice (MS1 ...). Each microservice is managed by its own *Service Proxy* and can handle specific API requests. We use the database per service model as it is simpler than the shared database model. Internal communication between microservices is minimized and realized, if needed, by calling their respective API. This type of communication and the responses to the *API Gateway* are omitted in the Figure for simplicity. Also, replication and other functionalities that will be realized in the

experimentation are not shown there. More details about the architecture are explained in Chapter 4.

1.2 Thesis motivation

API design has become a big challenge with the emergence of complex architectures, including microservices and cloud-based architectures. Such architectures' high versatility and volatility increase the risk of failures and performance issues [FML17]. Also, monitoring and tracing such failures is very difficult and sometimes impossible. Software architects are now not only responsible for implementing mechanisms to mitigate such failures but also for understanding the systems' behaviors and preparing for such failures in advance. Many API design patterns and best practices were proposed [ZSL⁺22, ZSZ⁺18, Hig20, Hig21], like *Circuit Breaker* and *Rate Limit* described in Chapters 5 and 6, to increase the reliability and performance of distributed systems. However, there is no guidance on how to use such patterns in different workload situations, for example. As a result, software developers and architects usually implement those patterns using their experience or ad hoc, which generates a high degree of uncertainty and doubt.

To alleviate that, we need to understand the system and its behavior with regard to API. However, such systems are usually huge and composed of thousands or even millions of microservices, making their understandability and predictability very hard. Especially when customers need to migrate from monolithic systems to microservices, the return on investment has to be evaluated in advance [LZJ⁺21, Li17]. Many tools have been proposed to assess quality attributes such as scalability and high availability when adopting microservices architectures like "TheArchitect" [PP18]. Still, these tools lack essential features, including generalizability and consistency. Also, several benchmark applications are available open-source to study microservices architecture, such as TeaStore [vKES⁺18]. They provide typical workload scenarios to simulate real-life microservices applications generating a wide range of interactions between API clients and servers. Unfortunately, none of these applications have been used to study such interactions, including API patterns and best practices.

With the widespread adoption of microservices [LF04, New15, Zim17] and cloud-based architectures, there has been a great interest in managing such complex environments

1 Introduction

effectively and efficiently. One of the most prominent platforms that has emerged for that purpose is *Service Mesh*. Indeed, this platform provides many functionalities to increase the observability and monitorability of such complex architectures. Unfortunately, there is no formal guidance on using such a tool apart from blogs and other informal literature. Also, it is very difficult to make the right decisions when using such a tool to choose between design options, including API patterns and best practices. On the other hand, understanding the impact of those design choices on quality attributes such as performance and reliability is a big challenge without the proper tooling and modeling.

Service meshes provide comprehensive tool support for managing microservices. *Rate Limit* and *Load Balancing* are some notable API patterns that can be implemented using a service mesh environment with low effort. This environment is ideal for having controlled experimentation as those patterns' variations can be realized during the runtime. As described later in Chapter 5, *Rate limit* can be realized in the *Front Proxy* or *API Gateway* to become a *Global Rate Limiter*. It can also be implemented on the *Edge Proxy* or *Sidecar* as an *Local Rate Limiter* (see Figure 1.2). Choosing between the two configurations depends on the scope of the control and the level of flexibility desired. Of course, software architects may combine both configurations to maximize control and flexibility. However, this might generate additional complexity and overhead. Service meshes also enable *External* or *Third-Party Rate Limiter* for maximum interoperability. *Internal Rate Limiters* are built-in in the service mesh at the level of *Front Proxy* or *Sidecar*. Both configurations can also be combined but would make complexity higher. More details about such configurations with their pros and cons are provided in Chapter 5. These variations possibilities make the service mesh a powerful decision support tool for comparing them experimentally.

Other functionalities provided by service meshes also include A/B testing and canary deployments or *Custom Routing* as described later in Chapter 4, with the possibility of deploying different microservices versions. *Request Bundle* (see Chapter 7) is an API pattern that could be evaluated by deploying the same microservices implementing different *Request Bundle* strategies. As a result, the time and effort of comparing them analytically is drastically reduced. Supporting *Request Bundle* is usually realized for all the microservices in the chain, starting from the front-end API to the database API. As a result, service meshes offer an extensive deployability capability for such experimentation.

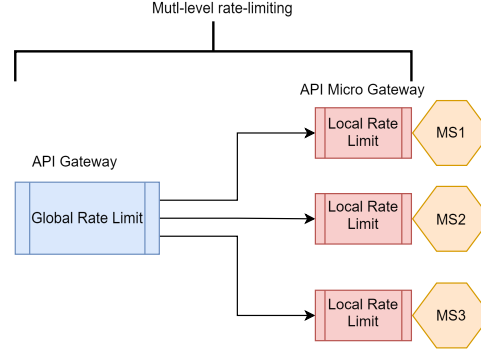


Figure 1.2: Multi-level rate limiting

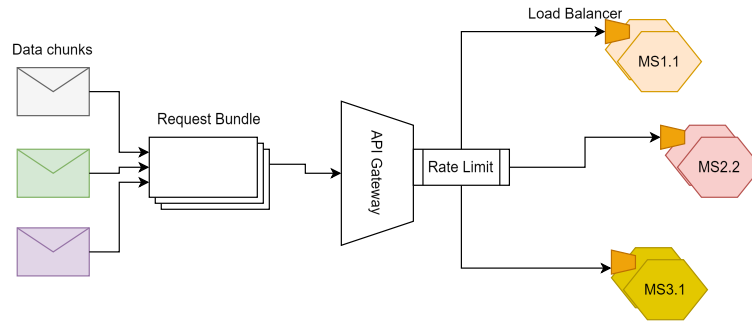


Figure 1.3: API patterns combined

Last but not least, *virtual services* (see Chapter 8) can simulate the *Load Balancing* pattern using one or multiple load balancers. Also, the simulation can use different *Load Balancing* algorithms and strategies, which can be changed at runtime at any time. As a result, decision-making gets faster regarding the best *Load Balancing* algorithm to choose, for instance. Service meshes also provide the possibility to create microservices replicas, so *Load Balancing* efficacy can be evaluated using as many replicas as the platform can comprehend. Most interestingly, combining the API patterns above and other patterns would further improve quality properties such as performance and reliability (see Figure 1.3 for an example scenario). For instance, using *Rate Limit* in combinations with *Request Bundle* would increase the reliability of the back-end by reducing the failure rate due to rate limiting since *Request Bundle* would encapsulate many requests into a single one, thus reducing the effect of setting too low rate limit values. Service meshes would make this scenario and others realizable and generate relevant data for analysis and decision support.

1.3 Research questions

To evaluate API patterns and best practices about quality properties like performance on a microservices-based architecture, we need a general experimental method that could be applied in different contexts and settings. This method has to support the implementation of the API patterns and their evaluation to provide accurate recommendations for the software architect. Realizing this method requires answering the following research questions:

RQ1 What are the service meshes' established best practices to implement API patterns? What are the design options that have to be considered when implementing such patterns?

RQ2 a) How can we evaluate API patterns like Rate Limit and Request Bundle using service meshes? b) What combinations of those API patterns are best for quality attributes like performance and reliability?

RQ3 a) What are the main criteria that should be considered to build a self-adaptive API recommendation pipeline? b) How can we combine this dynamic approach with static analysis for better and faster decision-making?

1.4 Thesis structure

Below is a description of the doctoral thesis structure:

Part I provides the state of the art describing the major research gaps we are trying to fill in our study. We also compare our research approaches with similar studies and provide the added values that make our research contribution unique.

Part II is the essence of our research plan as it describes API patterns and matches them with their different implementation strategies. We propose service meshes as a technology enabler that could bring API patterns from their conceptual focus to real-life practice.

Part III represents the core contribution of our research as it provides experimentally validated models to evaluate the impact of major API patterns and their

combinations on quality properties like performance and reliability. It also presents a general method that could be used to study other API patterns in different configurations and settings.

Part IV presents an improved approach based on a self-adaptive API recommendation pipeline. This approach is based on algorithms built using the benchmark data produced in **Part III**. We also discuss combining our analysis with static analysis for better-informed decision-making.

In **Part V**, we discuss the major contributions of the thesis.

2 State of the Art

2.1 Research Context

Many API patterns and best practices have been developed around microservices and Cloud-based architectures [ZSZ⁺18, PZA⁺17, SZZ⁺18] to improve quality properties like reliability and performance. However, there is only informal guidance on which API pattern to use in which situation, which is vague and inaccurate. Many other attributes need to be considered to attain accuracy and effectiveness when using API patterns, including the degree of flexibility, the scope, and the cardinality. While *Rate Limit* is a widely recognized aspect of Web API management [PH15, BLM21], its implementation varies across systems. Some systems may lack this feature entirely, posing challenges for API providers in implementing and enforcing rate limits [SF21]. In such cases, providers may resort to manual implementation or utilize specialized tools to ensure proper rate limits are in place, aiming to prevent excessive usage, thwart denial-of-service attacks, and mitigate security risks.

Moreover, there exists a lack of shared understanding or standardization regarding the configuration options for *Rate Limit* as described later in Chapter 6. This lack of common ground stems from the diverse approaches API providers take in implementing rate limiting, resulting in variations in configuration choices and terminology. Consequently, developers face challenges in fully grasping the nuances of the *Rate Limit* settings across different APIs. Additionally, the absence of a standard leads to inconsistencies not only in selecting the appropriate configuration and implementation mechanism but also in documenting these aspects. This poses a challenge for developers in understanding the Rate Limit policies of different APIs and their impact on their applications, as these policies are often not documented consistently [SMPZ23].

As a result, due to the lack of a comprehensive understanding of *Rate Limit* policies, developers may unintentionally exceed defined thresholds, leading to failures in their

2 State of the Art

client applications due to measures implemented by API providers to prevent abusive behaviors. Furthermore, the inconsistencies in documenting rate limits also hinder conducting comprehensive systematic studies on the various rate-limiting configurations adopted in real-world systems. For example, as presented briefly in Chapter 5, *Rate Limit* pattern configuration can be realized using different aspects. For instance, when considering the *scope* of the pattern, we can choose between *Global* or *Local* rate limiting. The first option is more common and usually implemented using *API Gateway*. The second option is more attractive when there is a need for a more fine-grained or service level rate limiting as depicted in Figure 1.2. Both options could also be combined to achieve a multi-level rate limiting. However, choosing the correct configuration is difficult as the real impact of each of those design options on quality properties like reliability remains unknown.

Other patterns like *Request Bundle* have to be implemented usually in the source code. So, decision-making regarding adopting such patterns must be done very early in project implementation, which is inconvenient. Such a pattern must be evaluated using real-life workload scenarios to get accurate results. Using *Request Bundle* would indeed reduce chatty communications. However, the impact of such an approach on the overall infrastructure is difficult to measure, as this could create large bundles of data that could cause network congestion. In addition, microservices systems are highly volatile, so the design choices must be adapted on demand. This is not straightforward for such complex and large infrastructures, especially for the API patterns whose design choices must be made early in the implementation phase. As a result, those systems need not only to self-adapt to manage sudden changes in workloads and settings but also to self-deploy different versions of the microservices application. By leveraging the *Custom Routing* capability provided by service meshes (see Chapter 4 for more details), we could deploy two distinct versions of the identical microservice, each capable of producing the same outcome. However, one version would manage incoming traffic with bundled data, while the other would handle traffic without *Request Bundle*. This setup, depicted in Figure 2.1, would enable us to seamlessly toggle the *Request Bundle* feature on and off, facilitating the evaluation of both scenarios using identical configurations and runtime environments.

Load Balancing is a prevalent API pattern whose main goal is to distribute the load over multiple microservices and increase scalability. This pattern has the particularity

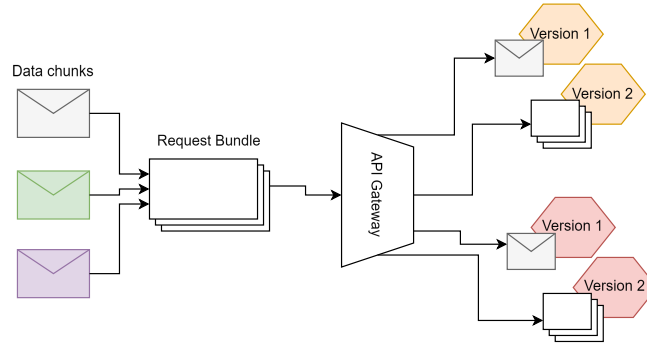


Figure 2.1: Multiple microservices versions: Request Bundle On and Off

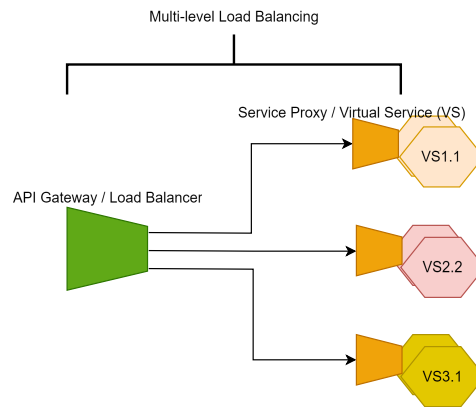


Figure 2.2: Multi-level load balancing

that it is usually implemented externally as a third-party service, which introduces additional costs and complexity. As a result, it is very important to evaluate the impact of such a pattern on the whole infrastructure to see if the implementation efforts are worth it. Service meshes offer the possibility to easily implement *Load Balancing* using *virtual services*, as depicted in Figure 2.2 and discussed in detail in Chapter 8, which can be realized at different levels (as in the case of *Rate Limit*) for more flexibility and control. This pattern can also be realized using different strategies or algorithms and by emulating even multiple load balancers per service.

As discussed in the previous Chapter, service meshes were introduced to support the implementation of the patterns described above and others with their different variations and configuration possibilities. Unfortunately, even with the widespread utilization of such a technology, there is still very high uncertainty in the decision-making process regarding its architectural design options as there is no formal guidance of such options,

which we try to address in Chapter 4. Also, there is no general method that we can use to implement API patterns like *Rate Limit* and *Request Bundle* using service meshes to evaluate their impact on various quality properties like performance and reliability. In Chapters 6, 7, and 8, we first try to implement those patterns individually and then in combination with the *Load Balancing* pattern. We also evaluate them regarding different quality properties and in different Cloud environments by building strong prediction models from the collected data. Secondly, we further develop that general method by building a complete API recommendation pipeline to attain self-adaptation to sudden changes in workload and configuration settings in Chapter 9, which will be the basis of combining this dynamic approach to the static analysis presented in Chapter 10.

2.2 Related Work

2.2.1 Related Works on Service Meshes

Service meshes are considered a very recent technology that has emerged due to the widespread utilization of microservices architecture [JPM⁺18]. Indeed, some research studies use this technology to uncover new challenges and opportunities [AGM21]. In this study, the author argues that service meshes present an unmatched opportunity to improve application performance by taking care of the network communication burden. Microservices decomposition has also been thoroughly studied from different perspectives in the scientific literature. Hasselbring et al. [HS17] claim that microservices architecture overcomes many issues in monolithic systems. This is mainly due to the high scalability, fault tolerance, and possibility of enabling DevOps best practices. In [BGDR17], the authors discuss issues regarding the proper granularity of microservices and boundaries definition. They propose a semantic-based approach to decompose microservices using their API specifications. Last, Zdun et al. [ZNL17] proposes an automated pattern-based method to assess microservices' architecture conformance and avoid architectural drift and erosion. While these studies and others have developed innovative approaches and ideas to maintain microservices architectural decomposition best practices, they lack application and translation into real-life scenarios.

Reusable ADD models have been employed by many authors. Zimmermann et al. [ZKL⁺09] extended those models to include dependencies, constraints, and rules.

They also argued that such models are crucial for a common understanding, especially in the software architecture community. ADD models have also been extended to cover software platforms or software sub-systems, which require a multi-level decision-making paradigm [LSZ12]. Those models were even suggested to compare Web Services and REST technologies in early adoption phases [PZL08], support microservices API design decisions [ZSZ⁺18, HWB17], and select the optimal big data system technology [GKN15]. Chapter 4 provides additional details about those related studies and others. In particular, we leverage the knowledge acquired in these areas regarding service meshes and microservices modeling approaches to develop guidelines about choosing the best ADD options for a specific scenario and workload situation. These guidelines are the basis of service meshes further development in the remaining chapters of the dissertation, where we employ such technology to evaluate those ADD options or API patterns about related quality properties.

2.2.2 API Rate Limit and its Impact on Reliability

Software architecture predictability regarding performance and reliability has been the focus of many research papers. Radliński et al. [Rad11] used the Bayesian model framework to predict software quality properties based on input from domain experts and empirical data. This approach is, unfortunately, too general and complex to implement, as it requires further customization for each use case. Also, a queuing-based model was proposed to forecast runtime quality properties using ARIMA techniques [DP06], which is only applicable in the Web Services domain. Hence, these approaches and methods have many limitations and are not applicable to microservices and Cloud-based architectures [AY16].

Indeed, only a few studies have provided models to deal with such architectures. Khazaei et al. [KBBML16] proposed a performance model for microservice provisioning in Amazon EC2 using Docker technology. A tool called Terminus was also developed to build performance models and identify MicroService capacity or maximal rate of requests [JPG19]. In our research, we propose a reliability model that is compatible with both the private Cloud (hosted in our University Data Center) and the Google public Cloud [LSL19] (GCP).

Many empirical studies on software architecture reliability have produced strong models.

Goseva-Popstojanova et al. [GPHP05] realized a large empirical study on existing software architecture reliability models. The authors suggested that those models need further refinements by interactively conducting experimentation. The Palladio Component Model (PCM) has also been proposed as a reliability prediction model that evaluates architectural aspects at runtime [BKBR10]. Calheiros et al. [CRB⁺11] developed the Cloudsim simulation and modeling tool to support application provisioning in Cloud systems. However, these studies do not focus on API and especially on *Rate Limit* pattern, which is the gap we try to close in Chapter 6. This is indeed crucial as *Rate Limit* has been introduced as a major API practice in several prominent studies, which we describe in more details in that Chapter.

2.2.3 API Request Bundle and its Impact on Performance

Request Bundle is an optimization technique that has been implemented in different areas of application [Sch15, AP20, Gys19, bra21]. For instance, Li et al. [LLGH16] developed an approach that bundles multiple HTTP requests using static analysis to improve energy efficiency. The authors evaluated their approach using Android mobile apps and suggested an energy consumption reduction of 15%. However, only a few studies have evaluated the real performance impact of such a pattern [Sin18]. In general, API performance has only been the focus of some empirical studies with strict technology and environment specificity [WCD⁺19, PJJ14] and around major API frameworks like GraphQL¹ and OpenAPI². GADOLINIUM, for instance, was presented as an OpenAPI framework to test the non-functional properties of REST API, including performance and availability [BCIEDC20]. In our research, we try to provide a complete technology-independent model to study the impact of *Request Bundle* on performance in Chapter 7.

2.2.4 Combining API patterns and its Impact of Performance and Reliability

Performance and reliability prediction models have been the subject of many experimental studies [WZ18]. Wartika et al. [WGNA19] focused on prediction software complexity using requirements. The authors developed a prediction model based on the Bayesian Regularization Algorithm. Component-based software engineering has also been the

¹<https://graphql.org>

²<https://swagger.io/specification/>

focus of a performance model prediction study [BKR09]. The authors used the concepts behind it to develop a simulation tool based on the Palladio Component Model (PCM). The tool supports architectural design decisions regarding non-functional requirements such as performance. Camili et al. [CGJ⁺22] proposed an integrated approach, called MiPaRT (Microservices Integrated Performance and Reliability Testing), to test the performance of microservices with regard to their reliability. They have also included DevOps best practices related to continuous testing and monitoring. However, none of these studies have specifically measured the impact of combining API patterns on reliability and performance. Chapter 8 closes this gap by presenting an experimentally validated regression model using a real-life microservice-based application and setting. More detailed related works are provided in this chapter.

Indeed, *Load Balancing* pattern has been the focus of many research papers to evaluate its impact on different quality properties like reliability. Wang et al. [WWL⁺21] highlighted the high importance of *Load Balancing*, especially in microservices-based systems. The authors have listed and explained the different *Load Balancing* strategies that could be implemented with their different pros and cons. COLBA, a chain-oriented load balancing algorithm, was also suggested to enhance those *Load Balancing* strategies efficiency [NLL18]. Liu et al. [LJSZ18] further proposed an online learning algorithm of time-weight (OLTP) to adaptively enhance those *Load Balancing* strategies' performance. Another algorithm called DWLOAD (Dynamic Weight Loading Algorithm) was also presented to cope with high load and concurrency situations in microservice systems by mitigating the resulting performance issues [LYW⁺22]. A distributed algorithm for *Load Balancing* based on Docker Swarm was also suggested as a way to decentralize the orchestration of the microservices system [RL18]. The authors argue that decentralized systems bring up more scalability and performance than centralized ones. Similarly, Bhattacharya et al. [BW22] developed a self-managing load balancing system (BLOC) that uses additional techniques like rate limiting to re-adapt the load balancing strategy. Zhu et al. [ZWB18] further formalized *Load Balancing* using scenario calculus to support the selection of the best strategy. Unfortunately, only a few studies considered combining *Load Balancing* with other related API patterns, which is the gap we try to reduce in Chapter 8.

2.2.5 Improving API Operations by Automated Recommendation

Self-adaptive systems have been studied in many research papers in improving quality properties like performance and reliability. Hamer et al. [dHS11] argued that SOA systems are used widely in different critical domains, including energy operations, and that automation is required to meet increasing and continuous business demands. The authors also point out that these continuous demands increase considerably the system's complexity. The Rainbow framework is a general framework proposed to monitor and adapt a running system based on externalized control mechanisms [GCH⁺04]. Layaida et al.- [LH05] also developed PLASMA, a component-based framework, to develop self-adaptive multimedia applications. A middleware-based was suggested to even bridge the gap between adaptation management and evolution management in self-adaptive systems [GEL⁺06]. These studies and others show the real importance of self-adaptation in distributed systems, but they do not consider API patterns and are not applicable to Cloud-based architectures.

In Chapter 9, we use decision tree modeling, which has been the subject of many studies to predict the performance and reliability of diverse applications. Rathore et al. [RK16] used Decision Tree Regression (DTR) to predict software faults for different open-source applications. The authors suggested that the use of DTR significantly improved the prediction error. Chiu et al. [CCW98] also cited that tree modeling improves the prediction error without impacting its accuracy for agent systems. Tree-based modeling was even used to detect faults in the early development of tactical military systems [KAB⁺96] and a real-time lane-changing system [HES14]. Those studies are further detailed in that Chapter.

3 Research problems and contributions analysis

In this chapter, we define the research problems we try to solve in this dissertation. Then, we explain the research contributions that we have realized through publications in renowned conferences.

3.1 Research problems

As previously argued in Chapter 1, APIs have become an important building block in microservices and cloud-based architectures. Hence, they have created many challenges regarding the backend infrastructure's quality properties, including reliability and performance. Unfortunately, such infrastructures' high volatility and versatility have made fine-tuning APIs very hard to realize. In particular, the following problems have been identified, which correspond respectively to the research questions defined in Section 1.3:

P1 High complexity of API patterns implementation in microservices and cloud-based architectures:

There exist several well-documented microservices API patterns that need to be evaluated in real-life scenarios. However, each pattern has its specifications, degree of complexity, and added value. To evaluate the actual value of those patterns, we need to find a simplified and flexible way to implement them. Also, it should be possible to turn an API pattern on and off, changing its parameters and combining it with other API patterns on the fly.

P2 Lack of methods to evaluate API patterns quality properties like performance and reliability:

Evaluating API patterns in such complex microservices and cloud-based systems

3 Research problems and contributions analysis

is not obvious. Such systems can change their properties in real-time and are so big that tracking each of these changes is impossible. Also, measuring the impact of such changes cannot be realized without the appropriate data collection and analysis. We need a general method that could be used to set up real-life experimentation and generate accurate models for predictive analytics.

P3 Missing a continuous and evolving approach for API patterns reconfiguration: Resolving problem P2 would not be sufficient as the general method has to run each time an API configuration changes. As microservices and cloud-based systems are highly evolving systems, running such a method manually (with human intervention) would not be efficient and is error-prone. The method has to be executed in a dedicated pipeline, and the human intervention should be minimized.

3.2 Research contributions

In this section, we describe our various contributions in this thesis dissertation to solve the research problems listed above. Most of the contributions were published in these renowned conferences: the European Conference on Software Architecture (ECSA), the International Conference on Service-Oriented System Engineering (SOSE), the International Conference on Services Computing (SCC), the International Conference on Software Services Engineering and the International Conference on Web Services (ICWS):

C1 In Chapter [4](#), we explore API patterns implementation strategies that are provided by *Service Meshes*. For that purpose, we propose guidelines to make accurate decisions regarding architectural design options when implementing such patterns in microservices and cloud-based architectures.

C2 In Chapters [6](#), [7](#) and [8](#), we propose a general method for API patterns reconfiguration. The method is validated for the *Rate Limit*, *Request Bundle* patterns and their combinations with *Load Balancing* regarding their impact on performance and reliability. We also realized strong prediction models that could be used for other API patterns and combinations of API patterns.

3.3 List of publications

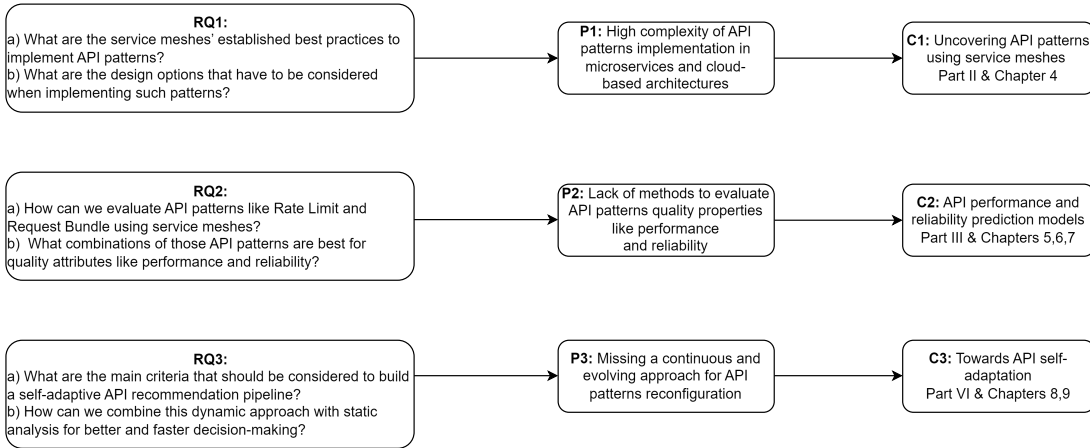


Figure 3.1: Research mapping overview

C3 In Chapter [9](#), we use the general method realized in C2 to automate the whole API reconfiguration process. This is done by building up a complete and automated API recommendation pipeline. We further combine this dynamic approach with static analysis in Chapter [10](#).

Figure [3.1](#) summarizes the research that we conducted by mapping each research question (RQ) to the corresponding problem (P) it tries to solve. The results are the list of contributions (C) we described above.

3.3 List of publications

The following is the detailed list of publications realized and linked to their respective chapters:

API Rate Limit Adoption – A pattern collection. Souhaila Serbout, Amine El Malki, Cesare Pautasso, Uwe Zdun 28th European Conference on Pattern Languages of Programs, July 5–9, 2023, Irsee, Germany (2023)

- Chapter [5](#): A summary of the API Rate Limit implementation strategies and known uses.

Combining API Patterns in Microservice Architectures: Performance and Reliability Analysis. Amine El Malki, Uwe Zdun IEEE International Conference On Web Services (ICWS 2023), 2-8 July 2023, Chicago, Illinois USA (2023)

3 Research problems and contributions analysis

- Chapter [8](#): A complex experimentation study to uncover best API patterns combination scenarios in Cloud-based architectures.

Automated Pattern-Based Recommendation for Improving API Operation Performance and Reliability in Cloud-Based Architectures. Amine El Malki, Uwe Zdun IEEE International Conference On Software Services Engineering (SSE 2023), 2-8 July 2023, Chicago, Illinois USA (2023)

- Chapter [9](#): Building up an API recommendation pipeline for self-adaptation.

Impact of API Rate Limit on Reliability of Microservices-Based Architectures. Amine El Malki, Uwe Zdun, Cesare Pautasso 16th IEEE International Conference on Service-Oriented System Engineering (SOSE2022), 15 -18 August 2022, San Francisco, USA (2022)

- Chapter [6](#): An analytical reliability model for the Rate Limit pattern evaluation in Cloud-based architectures.

Evaluation of API Request Bundling and its Impact on Performance of Microservice Architectures. Amine El Malki, Uwe Zdun IEEE International Conference on Services Computing (SCC 2021), 5-10 Sep 2021, Virtual (2021)

- Chapter [7](#): Realising a performance regression model for the Request Bundle pattern evaluation in microservices-based architectures.

Guiding Architectural Decision Making on Service Mesh Based Microservice Architectures. Amine El Malki, Uwe Zdun 13th European Conference on Software Architecture (ECSA), 9-13 September 2019, Paris France (2019)

- Chapter [4](#): Uncovering Service Meshes architectural design decision options.

On the Design and Architecture of Deployment Pipelines in Cloud- and Service-Based Computing – A Model-Based Qualitative Study. Uwe Zdun, Evangelos Ntontos, Konstantinos Plakidas, Amine El Malki, Daniel Schall, Fei Li 2019 IEEE International Conference on Services Computing (SCC 2019), 8-13 July 2019, Milan, Italy (2019)

3.4 Research methods

To answer RQ1, we comprehensively investigated the established practices within service mesh-based architectures. While various approaches, such as pattern mining [Cop96], have been proposed to examine these practices informally, we focus on providing a meticulously defined model. We intend to facilitate tool development and the establishment of metrics and constraints in our forthcoming endeavors. We opted to employ the model-based qualitative research method outlined in [ZSZ⁺18] to achieve this. This method systematically explores established practices within a specific domain and is rooted in the well-established qualitative research methodology known as Grounded Theory (GT) [GGLS67]. However, unlike GT, our approach generates inputs suitable for formal software modeling, such as model elements and relationship instances, rather than relying solely on informal textual codes.

Like GT, our methodology involved thoroughly examining each knowledge source. We gathered descriptions of established practices from what is commonly referred to as the "gray literature" (i.e., practitioner reports, system documentation, practitioner blogs, etc.), which served as unbiased depictions of established practices for further analysis. This contrasts sources like interviews, as typically used in traditional GT. We followed a similar coding process and employed a constant comparison procedure to derive our model, akin to the approach utilized in GT. However, unlike classical GT, our research commenced with initial research questions, aligning more closely with Charmaz's constructivist GT [Cha14]. While GT traditionally relies on textual analysis, we initially employed textual codes before transitioning them into formal UML models. We further develop this research methodology in Chapter 4.

To address the initial aspect of RQ2, we formulate an analytical model tailored to accurately predict the reliability impact of the *Rate Limit* pattern within microservice-oriented applications, specifically from the perspective of API clients. This model furnishes a robust technique for adaptively refining rate limits and undergoes empirical validation. To validate, we deploy a range of microservice-driven workload scenarios across both a private cloud (minimizing latency and mitigating biases) and the Google Cloud Platform (GCP), testing our model in a practical cloud environment. Supported by a service mesh-based infrastructure, we simulate real-world API client workloads employing various configurations and settings for empirical validation. Additionally, we endeavor to enhance

3 Research problems and contributions analysis

comprehension of the *Request Bundle* pattern through empirical evaluation within a service mesh-based framework. We aim to quantify potential performance enhancements facilitated by such a pattern and furnish a model for estimating it in applications and workloads akin to the open-source application we tested.

For the subsequent facet of RQ2, we introduce a regression model devised to quantify the performance and reliability impacts of combining the *Rate Limit*, *Request Bundle*, and *Load Balancing* patterns. Subsequently, we conduct experimental evaluations leveraging a large-scale setup and a practical, third-party microservices-based application deployed on a service mesh-based infrastructure (deployed on Amazon Web Services Cloud AWS). Our approach is a versatile and predominantly automated method, applicable iteratively for examining various optimization techniques and pattern combinations. Part III of the dissertation provides more details about the experimentation and the research process. In response to RQ3, we have significantly expanded upon these empirical investigations to introduce a novel approach for automatic and dynamic API design recommendations. This method, centered around decision tree modeling, represents the principal contribution of Chapter 9.

Part II

Uncovering API patterns using service meshes

4 Guiding Architectural Decision Making on Service Mesh Based Microservice Architectures

Microservices are becoming the de-facto standard way for cloud and service-oriented computing software development. Service meshes have been introduced as a dedicated infrastructure for managing a network of containerized microservices to cope with the complexity, manageability, and interoperability challenges in especially large-scale microservice architectures. Unfortunately, so far, no dedicated architecture guidance for designing microservices and choosing among technology options in a service mesh exists. As a result, there is substantial uncertainty in designing and using microservices in a service mesh environment today. To alleviate this problem, we have performed a model-based qualitative in-depth study of existing practices in this field in which we have systematically and in-depth studied 40 reports of established practices from practitioners. In our study, we modeled our findings in a rigorously specified reusable architectural decision model, in which we identified 14 architectural design decisions with 47 decision outcomes and 77 decision drivers in total. We estimated the uncertainty in the resulting design space with and without the use of our model and found that a substantial uncertainty reduction can be potentially achieved by applying our model.

4.1 Introduction

In this chapter, we present service meshes as the building block of the empirical studies that are described in the next chapters. To answer **RQ1**, we have elaborated a qualitative study about current service mesh technologies. Based on this study, we have uncovered an exhaustive list of architectural design decision options to set up service meshes. We

have also employed our reusable architectural decision model for faster and more accurate decision-making.

Microservices are a recent approach for designing service architectures that evolved from established practices in service-oriented architectures [LF04, New15, Zim17]. As microservices, especially in large-scale systems, introduce many challenges and high complexity in terms of manageability and interoperability, *service meshes* [Mor] have been introduced as an infrastructure for managing the communication of containerized microservices and perform many related tasks. For this, they usually use a network of lightweight proxies or sidecars that handle all the communication burden [JPM⁺18, Mor17]. As a result, the coupling between microservices and microservices to the infrastructure services can be drastically reduced. This also eases establishing interoperability between microservices developed in different programming languages and with other technologies. The proxies or sidecars form a *data plane* that is typically managed by a *control plane* [Smi18].

Unfortunately, so far, no dedicated architectural guidance exists on how to design and architect microservices in a service mesh environment apart from practitioner blogs, industry white papers, experience reports, system documentation, and similar informal literature (sometimes called gray literature). This includes that so far; there is no guidance for users of service mesh technologies or even their implementors to select the right design and technology options based on their respective properties. Very often, it is even difficult to understand the possible design options, their possible combination, and their impacts on relevant quality properties and other decision drivers. As a result, there is substantial uncertainty in architecting microservices in a service mesh environment, which can only be addressed by gaining extensive personal experience or gathering architectural knowledge from the diverse, often incomplete, and often inconsistent existing practitioner-oriented knowledge sources.

To alleviate these problems, we have performed a qualitative, in-depth study of 40 knowledge sources in which practitioners describe established practices. We have based our study on the model-based qualitative research method described in [ZSZ⁺18], which uses such documented practitioner sources as rather unbiased knowledge sources and systematically codes them using established coding and constant comparison methods [GGLS67] combined with precise software modeling, to develop a rigorously specified

software model of established practices and their relations. This chapter aims to study the following research questions:

RQ1.1 What established practices commonly appear in service mesh-based designs and architectures?

RQ1.2 What are the dependencies of those established practices? Especially which architectural design decisions (ADDs) need to be made in service mesh-based designs and architectures?

RQ1.3 What are the decision drivers in those ADDs to adopt the practices?

In addition to studying and answering these research questions, we have estimated the decision-making uncertainty in the resulting ADD design space, calculated the uncertainty left after applying the guidance of our ADD model, and compared the two. Our model shows a potential to substantially reduce the uncertainty not only by documenting established practices but also by organizing the knowledge in a model.

The remainder of this chapter is organized as follows: Section 4.3 explains the research method we have applied in our study. Section 4.4 describes a precise specification of the service mesh design decisions resulting from our research. The uncertainty estimation is discussed in Section 4.6, followed by a discussion in Section 4.7 and conclusions in Section 4.8.

4.2 Related Work

Service meshes have been identified in the literature as the latest wave of service technology [JPM⁺18]. Some research studies use service meshes in their solutions. For example, Truong et al. [TGH18] use a service mesh architecture to reduce rerouting effort in cloud-IoT scenarios. Studies on generic architecture knowledge specific to service meshes are rare in the scientific literature. One example that considers them is TeaStore, which introduces a microservice-based reference architecture for cloud researchers and considers practices used in service meshes [EKG⁺18]. More sources can be found on general microservice best practices. For instance, Richardson [Ric17] provides a collection of microservice design patterns. Another set of patterns on microservices has been published

by Gupta [Gup17]. Microservice best practices are discussed in [LF04], and similar approaches are summarized in a recent mapping study [PJ16]. So far, none of these approaches has put a specific focus on the service mesh practices documented in this chapter.

A field of study related to service mesh architectures is microservice decomposition, as this can lead to decision options and criteria related to the topology of the service mesh. While the microservice decomposition itself is studied in the scientific literature extensively (see, e.g. [HS17, BGDR17, ZNL17]), its influence on the design of the deployment in a service mesh and its topology is studied only rarely. For instance, Zheng et al. [ZZZ⁺19] study the SLA-aware deployment of microservices. Selimi et al. [SCASA⁺17] study the service placement in a microservice architecture. Both studies are not specific to service meshes but could be applied to them. In contrast to our research, which aims to cover a broad variety of architecting problems, these studies only cover a very specific design issue in a microservice architecture.

The model developed in this chapter can be classified as a reusable ADD model [ZKL⁺09]. Decision documentation models have been used by many authors before, and quite a number of them are focused on services, such as those on service-oriented solutions [ZKL⁺09], service-based platform integration [LSZ12], REST vs. SOAP [PZL08], microservice API quality [ZSZ⁺18], big data repositories [GKN15], and service discovery and fault tolerance [HWB17]; however, none of them considers service meshes yet. Also, the chapter has provided a thorough understanding of service meshes that motivates their use in the experiments described in the remaining chapters.

4.3 Research Method

This chapter aims to systematically study the established practices in the field of service mesh-based architectures. Several methods have been suggested to informally study established practices, including pattern mining (see, e.g., [Cop96]). As in our work, we rather aim to provide a rigorously specified model of the established practices, e.g., to support tool building or the definition of metrics and constraints in our future work, we decided to follow the model-based qualitative research method described in [ZSZ⁺18]. It aims to systematically study the established practices in a particular field and is based

on the established qualitative research method Grounded Theory (GT) [GGLS67]. Still, in contrast to GT, it produces inputs for formal software modeling like model elements or relation instances, not just informal textual codes. Like GT, we studied each knowledge source in depth. The method uses descriptions of established practices from the so-called gray literature (i.e., practitioner reports, system documentation, practitioner blogs, etc.). These sources are then used as unbiased descriptions of established practices in the further analysis (in contrast to sources like interviews as used in classic GT). We followed a similar coding process, as well as a constant comparison procedure, to derive our model as used in GT. In contrast to classical GT, our research began with initial research questions, as in Charmaz’s constructivist GT [Cha14]. Whereas GT typically uses textual analysis, we initially used textual codes and then transferred them into formal UML models.

A crucial question in GT is when to stop this process; here, theoretical saturation [GGLS67] has attained widespread acceptance in qualitative research: We stopped our analysis when 5 to 7 additional knowledge sources did not add anything new to our understanding of the research topic. As a result of this very conservative operationalization of theoretical saturation, we studied a rather large number of knowledge sources in depth (40 in total, summarized in Table 4.1), whereas most qualitative research often saturates with a much lower number of knowledge sources. Our search for knowledge sources was based on popular search engines (e.g., Google, Bing), social network platforms used by practitioners (e.g., Twitter, Medium), and technology portals like InfoQ and DZone.

Proof-of-Concept Implementation

Our proof-of-concept implementation is based on our existing modeling tool implementation CodeableModels¹, a Python implementation for precisely specifying meta-models, models, and model instances in code with an intuitive and lightweight interface. We implemented all models described in this chapter together with automated constraint checkers and PlantUML code generators to generate graphical visualizations of all meta-models and models.

Table 4.1: Knowledge Sources Included in the Study

¹<https://github.com/uzdun/CodeableModels>

4 Guiding Architectural Decision Making on Service Mesh Based Microservice Architectures

Code	Description	Reference
S1	Istio Prelim 1.2 / Traffic Management (documentation)	http://bit.ly/2Js3JXj
S2	Using Istio to support Service Mesh on Multiple ... (blog)	http://bit.ly/2FqMce5
S3	Service mesh data plane vs. control plane (blog)	http://bit.ly/2EtC8z6
S4	The Importance of Control Planes with Service Meshes ... (blog)	http://bit.ly/2He7JYu
S5	Envoy Proxy for Istio Service Mesh (documentation)	https://bit.ly/2HaNdrE
S6	Our Move to Envoy (blog)	https://bit.ly/2Vyyefd
S7	Envoy Proxy 101: What it is, and why it matters? (blog)	https://bit.ly/2HaNhYq
S8	Service Mesh with Envoy 101 (blog)	https://bit.ly/2UjPuVn
S9	Microservices Patterns With Envoy Sidecar Proxy (blog)	https://bit.ly/2t0Wo9C
S10	Ambassador API Gateway as a Control Plane for Envoy (blog)	http://bit.ly/2TuaZWj
S11	Streams and Service Mesh - v1.0.x Kong ... (documentation)	http://bit.ly/2UGX7W1
S12	Istio Prelim 1.2 / Security (documentation)	http://bit.ly/2Hy0IkH
S13	Consul Architecture (documentation)	https://bit.ly/2ITnhU2
S14	Global rate limiting — envoy ... (documentation)	http://bit.ly/2Js3JXj
S15	Cilium 1.4: Multi-Cluster Service Routing, ... (blog)	http://bit.ly/2Cv49pU
S16	Proxy Based Service Mesh (blog)	https://bit.ly/2VzpbL2
S17	Smart Networking with Consul and Service Meshes (blog)	http://bit.ly/2Uk14jg
S18	A sidecar for your service mesh (blog)	http://bit.ly/2ThMrvF
S19	Istio Prelim 1.2 / Multicluster Deployments (documentation)	http://bit.ly/2udsxI3
S20	Microservices Reference Architecture from NGINX (blog)	http://bit.ly/2U3tNw1
S21	Comparing Service Mesh Architectures (blog)	http://bit.ly/2tQ2Gwd
S22	Istio Multicluster on OpenShift – Red Hat OpenShift ... (blog)	https://red.ht/2FcMyn4
S23	Amazon ElastiCache for Redis FAQs (documentation)	https://amzn.to/2TgGML8
S24	Service Mesh for Microservices (blog)	http://bit.ly/2TCd6Is
S25	Designing microservices: ... (documentation)	http://bit.ly/2tPQk04
S26	HashiCorp Consul 1.2: Service Mesh (blog)	http://bit.ly/2Fnj1It
S27	Connect-Native App Integration (documentation)	http://bit.ly/2NEDM1L
S28	Service discovery — envoy ... (documentation)	http://bit.ly/2Tfp59H
S29	Linkerd2 Proxy (open source implementation)	https://bit.ly/2HaiFqa
S30	Multi-Cluster Support for Service Mesh ... (blog)	http://bit.ly/2Jp6isS
S31	Linkerd Architecture (documentation)	http://bit.ly/2Uki3lt
S32	Federated Service Mesh on VMware PKS ... (blog)	http://bit.ly/2TNRitD
S33	Consul vs. Istio (documentation)	http://bit.ly/2Tdx5gd

S34	Guidance for Building a Control Plane to Manage Envoy ... (blog)	http://bit.ly/2CCAYRU
S35	Comparing Service Meshes: Linkerd vs. Istio ... (blog)	http://bit.ly/2TWQAtT
S36	Connect - Proxies - Consul by HashiCorp (documentation)	http://bit.ly/2UViLWG
S37	Approaches to Securing Decentralised Microservices ... (blog)	http://bit.ly/2Wp50jn
S38	Istio Routing Basics – Google Cloud Platform ... (blog)	http://bit.ly/20oR0Dn
S39	Integrating Istio 1.1 mTLS and Gloo Proxy ... (blog)	http://bit.ly/2UTpctm
S40	Kubernetes-based Microservice Observability ... (blog)	http://bit.ly/2FvE4aT

4.4 Service Mesh Design Decisions

Following our study results, we identified 14 ADDs for service meshes described in detail below. *Service Meshes* are usually used together with a *Container Orchestrator* such as Kubernetes or Docker Swarm. That is, the services in the mesh, the central services of the service mesh, and service mesh proxies are usually containerized, and the containers are orchestrated. Very often, service meshes are used to deal with heterogeneous technology stacks. That is, a major goal is that microservices can be written as HTTP servers with any programming language or technology, and without modification, these services get containerized and managed in a mesh, including high-level services like service discovery, load balancing, circuit breaking, and so on. In the first four sections, we describe ADDs that characterize a service mesh as a whole. The remaining section describes ADDs that can be made for specific components of a service mesh.

4.4.1 Managed Cross-Service Communication Decision

As stated previously, a *Service Mesh* is composed of a set of networked proxies or *Sidecars* that handle the communication between microservices [JPM⁺18, Smi18]. The decision regarding managed communication across the services in a *Service Mesh* is made for the *Service Endpoints* of these microservices, as illustrated in Figure 4.1. Not using managed cross-service communication is a decision option for each service endpoint, but please note that this essentially means not following a service mesh architecture for the endpoint. Alternatively, we can select between the two following design options: *Service Proxy* and *API-Based Service Integration*. *Service Proxy* is the commonly supported option. If the *Service Proxy* is hosted in a container that runs alongside the service container (i.e., in

the same pod of the *Container Orchestrator*), the service proxy is called a *Sidecar Proxy*. A few service meshes offer the additional option *API-Based Service Integration*, which means that the service uses a service mesh API to register itself in the mesh and is then integrated without a dedicated proxy. The entire cross-service communication handled by proxies or otherwise integrated services is called the *Data Plane* of the *Service Mesh*. Centralized services of the service mesh are usually called the *Control Plane* (discussed below).

The *Service Proxy* option has the benefit to make it easier to protect the service from malicious or overloaded traffic by achieving *access control*, *TLS termination*, *rate limiting and quotas*, *circuit breaking*, *load balancing*, and other tasks; this is discussed in more depth in Section 4.4.5. Also, the independence of the service from its proxy increases the *extensibility* and *maintainability* of the service, which is, as a result, unaware of the network at large and only knows about its local proxy. However, this option might produce additional *communication overheads* and *congestions*. The major benefit of choosing an *API-Based Service Integration* over a *Service Proxy* is that it makes the service mesh less *complex* and there is less *communication overhead*. However, doing so limits its *extensibility* and *interoperability*. The option not to manage cross-service communication basically means that all benefits of service mesh are not achievable.

An example realizing the *API-Based Service Integration* is Connect Proxy used in Consul that is implemented using language-specific libraries that are used directly in microservices code. *Service Proxy* and *Sidecar Proxies* are more frequently supported; examples are Envoy Proxy [Env19] in the Istio Service Mesh [Ist19], Kong Proxy, NGINX Proxy, and Linkerd Proxy. Most such service proxy technologies can be deployed as a sidecar or a service proxy running in a different environment (e.g., different server or VM); they usually also offer the option to be used as a *Front Proxy* as discussed in the next section.

4.4.2 Managed Ingress Communication Decision

In addition to handling cross-service communication, service meshes often intercept incoming traffic, usually called ingress traffic. The decision for managed ingress communication is usually made for the *Service Mesh* as a whole. The ingress traffic then needs to be routed to the containers orchestrated in the mesh. Of course, we might choose not

4.4 Service Mesh Design Decisions

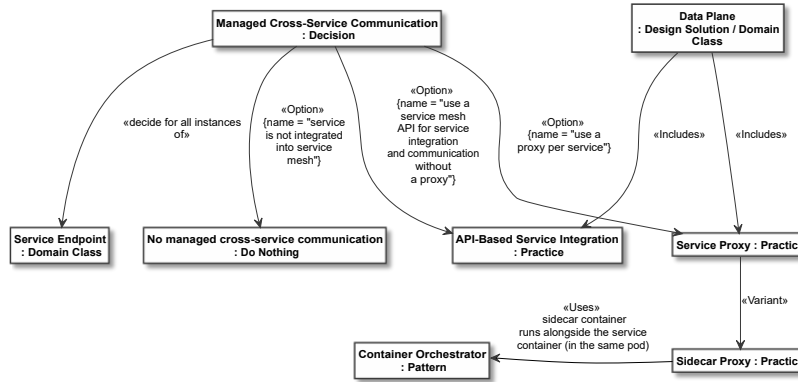


Figure 4.1: Managed Cross Service Communication Decision

to manage ingress communication, but this is a risky option since it might expose the service mesh to malicious or overloaded traffic. This option may be adopted in the case of a private service mesh, but such meshes seem to be very rare. The typical design option chosen is a *Front Proxy*, which is used by the *Control Plane* to intercept ingress traffic as shown in Figure 4.2. An *API Gateway* [Ric17], a common microservice pattern with the goal to provide a common API for several services, can be realized based on a *Front Proxy* of a service mesh. A *Front Proxy* can protect the service mesh from *malicious traffic*. It can provide *proxy tasks* such as *load balancing* and *multi-protocol support* at the perimeter of the service mesh. Clients are *shielded from details about the inner workings* of the service mesh and are provided with an *API at the client-needed granularity*; this reduces *complexity for clients*. The additional proxy increases *complexity for developers* of the service mesh. The *performance of requests* can be improved, as fewer roundtrips from clients to services are needed if the *Front Proxy* can retrieve data from multiple services for one request from a client. However, the additional network hop for accessing the *Front Proxy* decreases the *performance*. An example of this type of proxy is the NGINX Ingress Controller. Most of the proxies from the previous section can also be used as *Front Proxies*.

4.4.3 Traffic Control Decision

Communication in service meshes generates a lot of traffic and data that needs to be controlled and captured, e.g., to distribute access control and usage policies and observe and collect telemetry, traces, and metrics. The traffic control decision is usually made for

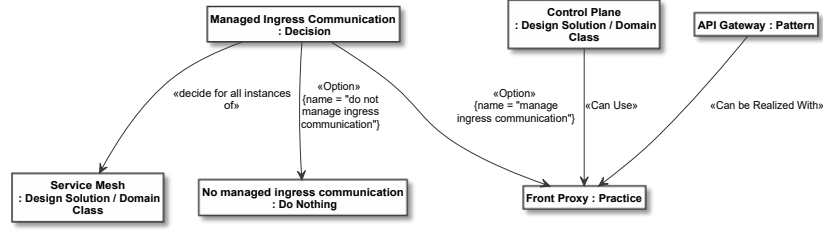


Figure 4.2: Managed Ingress Communication Decision

the whole *Service Mesh* as illustrated in Figure 4.3. There are four traffic control options:

Centralized Control Plane – A central component, called the *Control Plane*, controls traffic of a service mesh. It is responsible for managing and configuring sidecars in addition to distributing access control and usage policies, observing and collecting telemetry, traces, and metrics, in addition to numerous other services like service discovery, as described in Section 4.4.5.

Distributed Control Plane – Each service mesh service has its own cache that is efficiently updated from the rest of the services. This helps to enforce policies and collect telemetry at the edge.

Front Proxy – The proxy is responsible for intercepting incoming traffic from outside the service mesh as described in Section 4.4.2. It might also be extended to handle traffic control at the entry point of the service mesh. This option can potentially be combined with the two previous options (for that reason, the decision is marked with a stereotype indicating multiple answers can be selected).

Finally, no dedicated traffic control can be used as well.

The most obvious benefit of using a *Centralized Control Plane* is its *simplicity* and ease of *administration*. However, especially when using one single control plane, it produces a *single point of failure* and is hard to *scale*. Also, it might cause *traffic congestion*, which *increases latency*. *Centralized Control Planes* provide policies to the *Service Proxy* on how to perform routing, load balancing, and access control. In that case, the following optional decision to take is related to service mesh expansion. Istio service mesh, for example, which is based on *Centralized Control Plane*, supports service mesh expansion in a multi-cluster environment. On the other hand, a *Distributed Control Plane* is *highly*

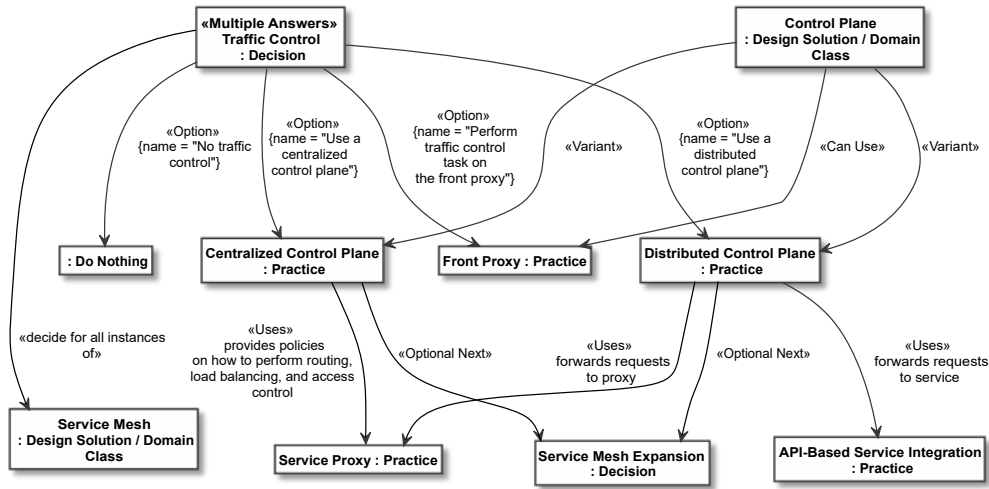


Figure 4.3: Traffic Control Decision

scalable, and there is *no single point* of failure. However, this option is the most *complex* and thus *risky* option. Using this option, traffic may be forwarded to either a *Service Proxy* or directly to a service via *API-Based Service Integration* as described in Section 4.4.1. Consul, for example, which implements *Distributed Control Plane*, consists of a set of client agents that are efficiently updated from server agents to control traffic at the edge. An example using *Front Proxy*, described in Source S10 in Table 4.1, uses Envoy, which can be extended to become an *API Gateway*, which then can do traffic control for the service mesh by integrating with Istio. The *Front Proxy* solution does not have the *fine-grained control* offered by the other options, as it is only applied in a central place. It is also a *single point of failure* and is negative for *congestion* and *latency*, but is a *simple* and *non-complex* solution. If used together with one of the other options, it increases the *complexity* of these options even further. Still, it enables more *fine-grained control* for the ingress traffic and thus can reduce the overall traffic in the mesh. These options lead us to the following optional decision regarding distributing traffic control and other tasks among *Control Plane* and *Data Plane* (see Section 4.4.5 for a list of these follow-on decisions, not shown in Figure 4.3 for brevity). Figure 4.3 shows these decision options and their relations.

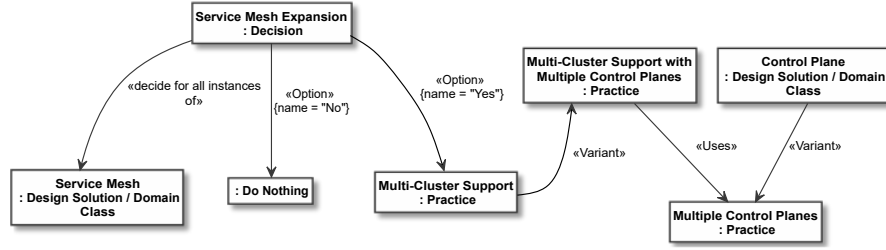


Figure 4.4: Service Mesh Expansion Decision

4.4.4 Service Mesh Expansion Decision

To *scale* and achieve *redundancy*, service meshes can be expanded and form multi-clustered service meshes, leading to the selection of the option *Multi-Cluster Support* in the decision illustrated in Figure 4.4. The service mesh expansion decision is made for the *Service Mesh* itself. Selecting *Multi-Cluster Support* may result in higher *complexity* and increased *network bandwidth* need and *cost*. The decision option *Multi-Cluster Support* is in its simple form, just using one *Centralized Control Plane* (see Section 4.4.3) that controls multiple service meshes. The most obvious benefit of this option is its *simplicity* and *ease of administration*. However, it is creating a *single point of failure* and might produce traffic bottlenecks which increase *latency*. *Multi-Cluster Support* has one variant *Multi-Cluster Support with Multiple Control Planes* with no *single point of failure*. This option variant uses *Multiple Control Planes*, which is a variant of *Control Plane* as shown in Figure 4.4.

Istio Multicluster on Openshift, described in source S22 of Table 4.1, is an example that implements *Multi-Cluster Support* using one *Centralized Control Plane*. In this example, one cluster is hosting the *Control Plane* and the others host the *Data Plane* as well as some parts of the *Control Plane* for distributing certificates and *Sidecar Proxy* injection. Another example is NSX service mesh, described in source S32 of Table 4.1, which is also based on Istio but implements the variant *Multi-Cluster Support with Multiple Control Planes* by enabling a local service mesh per Kubernetes cluster.

4.4.5 Central Services and Proxy Tasks

As explained above, the *Control Plane* and *Data Plane* provide numerous central services and proxy tasks, many of which are achieved jointly. The decisions on central services

and proxy tasks are usually made for the *Service Mesh* itself but can, in many cases, be changed for individual services or service clusters from the default configured for the service mesh. Proxy tasks generally can be implemented on the *Service Proxies* or, in some cases, alternatively on a *Front Proxy*. All solutions relying on a central service or the *Front Proxy* are introducing a *single point of failure* (not repeated per case below). The decisions, options, and decision drivers for central services and proxy tasks are discussed below. In addition, we have found evidence for decisions that are needed for the basic functioning of the service mesh, such as *policy distribution*, which we have not included in our catalog, as the user does not have to make a decision about them. Based on the services and tasks listed below, we found evidence for many possible follow-on decisions such as support for *rate limits*, *quotas*, *circuit breaking*, *retries*, *timeouts*, *fault injection*, *dashboards*, *analytics*, and so on. We did not include those in our ADD model either, as the possible list of such higher-level services is excessive and will likely grow over time.

Service Discovery Decision

To communicate in a service mesh, services need to locate others based on information like IP address and port number. Of course, this might be *hard-coded in each of these services*. If a service changes its address, fails, or is unavailable for other reasons like congestion, it becomes *not reachable* anymore. Then, there is a huge problem since all service code needs to be changed and the mesh needs to be restarted, which impacts negatively *availability*. To resolve this issue, the *Control Plane* and *Data Plane* use *service discovery system* usually provided by platforms like Kubernetes, for example. An alternative is using a central *Lookup* service [VKZ04]. The *distributed service discovery* option requires a consistency protocol and caching of discovery information, i.e., it is more *complex*. However, the lookup is local, thus, it offers better *performance*. Without service discovery, the *manageability*, *changeability*, and *evolvability* of the service mesh would severely suffer.

Load Balancing Decision

Service meshes, especially at scale, have to handle tremendous traffic loads, which might overload services, increase their *latency*, and decrease their *availability*. To avoid such

a situation and maintain *scalability*, services are replicated, and load is distributed over these instances by both the *Control Plane* and *Data Plane* using a *load balancing* algorithm. *Load balancing* can also be based on geographical location, especially in the case of service mesh expansion described in Section 4.4.4. If load balancing is used, the typical option is *Load balancing on the Service Proxies*. An alternative that offers *balancing loads for the whole ingress traffic* is *Load balancing on the Front Proxy*; this option offers less *fine-grained control over the load balancing* than, e.g., to balance per service cluster. Both solutions can also be *combined*, offering the benefits of both solutions but also increasing the *complexity*.

Custom Routing Decisions

To manage cross-service and ingress communication, the *Control Plane* and *Data Plane* need to know where each packet should be headed to or routed; routing is usually configured on the *Control Plane* and enacted by the proxies on the *Data Plane*. In addition to such basic routing, the service mesh often offers *Custom Routing* options, which can be based on URL path, host header, API version, or other application-level rules for *control over the routing* in the mesh. Such routing rules can be dynamically changed, increasing the *flexibility* of the architecture. Custom routing can, in follow-on decisions, be used for extra tasks; a prominent one is to support continuous experimentation techniques such as *staged rollouts*, *A/B testing* and *canary deployment* (or not). The latter can help for more *controlled deployments* to production, which helps to minimize *deployment risks*.

Health Checking Decision

In highly versatile environments such as service meshes, services go up and down unexpectedly, which decreases *availability*. To overcome this issue, periodic *health checks* on services can be applied, and, e.g., mitigation strategies like service restarts can be applied. Health checks are usually performed by the service proxy and a central service collecting the information. Alternatively, simple health checks like pinging service proxies can also be done solely on the central service, but then *more complex health checking* is not possible. Of course, another decision option is not to perform health checks.

Security-Related Decisions

Communication in service meshes usually uses encryption-based generated keys and certificates; if not used, the service mesh might be exposed to malicious traffic and manipulations unless a key and certificate management service outside the service mesh can or must be used. A simple option is using *API Keys* [ZSZ⁺18] and local key management. The alternative is to introduce a *central certificate authority*, residing in the *Control Plane*, that takes care of storing and distributing security keys and certificates to the *Data Plane*. This option is more *secure* than the other options and creates in large installations less *maintenance overhead* for managing various *API Keys* in the clients and service proxies. Still, it is also more *complex* than, e.g., *API Keys*. Once authentication is handled, authorization needs to be considered. This can be achieved by setting up access control in the *Control Plane* or the *Data Plane*. If we choose not to control access after authentication, services are exposed to unintentional and unwanted modifications. *Security* is the most important driver in this decision; a solution on the data plane supports more *fine-grained control* but is more *complex* than a solution on the control plane. Encryption in service meshes, usually based on mutual TLS, must be handled at both ends; not using encryption means *security* is endangered. There are three decision options for *TLS Termination*: either we offer TLS termination directly in the service, at the *Front Proxy*, or – the most common option – in the *Data Plane*. The first option brings *boilerplate code* to the service, which might also decrease its *performance*. The second option is only viable if the service mesh is in a private environment where *internal unencrypted communication* is an option (or another encryption than the one used for communication with clients).

Collect Telemetry, Traces, and Metrics Decision

To observe telemetry, traces, and metrics in a service mesh, they first need to be collected. Otherwise, we have to access each service and manually upload this data. This is usually done by a *control plane service collecting data from data plane proxies*. With few services, we can choose not to *collect them centrally*. At *large scale*, this might make control and management tasks complex, and *central features* such as dashboard or metrics are hard to impossible to build. Some telemetry might also be needed anyway for the functioning of the *service mesh itself*.

Multi-Protocol Support Decision

In heterogeneous environments like service meshes, *multi-protocol support* is required. It helps to have a *unified API interface* that can be used by services using different protocols, which increases *interoperability* and *extensibility* of the service mesh. This can be *offered by data plane proxies* or *on the front proxy*, where the latter option offers less *fine-grained support* and is suitable if *the mesh uses only one protocol inside*. Of course, we might choose not to use this *API interface* and relieve the service mesh from the resulting processing *overhead*. Then, we need to add *boilerplate code* to services to support different protocols or suffer from *interoperability* issues.

4.5 Challenges in API patterns implementation

Microservices and cloud-based architectures have come up with many challenges to achieve the desired level of quality properties like performance and reliability. Although many techniques are available to fine-tune these quality properties as described in the previous section, there is still high uncertainty on what configuration and setting to choose in such unpredictable environments. Indeed, software developers can build libraries to achieve fault tolerance and resiliency features such as circuit breakers and timeouts. They can also use those features that are already implemented in service meshes proxies as described above. However, their proper configuration could only be realized through personal experience and guessing. As a result, there is a need for tool support to decide on which configuration to select in different situations.

APIs are considered as the entry point of each application, including microservices and Cloud-based. Many API best practices and patterns have been identified (see Chapter 2), including *Rate Limit*, *Request Bundle*, and *Load Balancing* to increase quality properties related to performance and reliability, for example. However, there exists no formally defined link between configuring such API patterns and its impact on those quality properties. *Rate Limit*, a feature already provided by service meshes as described in the last section, has long been used by major service providers like Google and eBay to prevent overloading their backends. They use a pricing model to define the rate limit value to assign to each user, which is not an efficient solution in the long term. As the number of paying users gets higher, there is a higher risk that the backend will still be

overloaded and reject their requests. This is an unwanted situation for both the users and the service provider. Of course, service providers cannot have unlimited resources, so they need to adjust them on demand. Indeed, rate limit values need to be adjusted in an adaptive way, taking into consideration parameters from both API client and server sides. Service meshes provide multi-level rate limiting capability as already depicted in Figure 1.2, which would allow us to fine-tune rate limits from different perspectives (see Chapter 5 for more details) and on demand. As a result, realizing rate limiting adaptation on the fly is feasible as the service meshes control the traffic, as discussed in the last section, from the entry point or *API Gateway* to the backend or microservice.

Request Bundle is another important API pattern, but has the particularity that it is usually implemented directly in the source code (see Chapter 7 for more details). The role of that pattern is to bundle multiple requests into a single one. This approach has its advantages and disadvantages. Using the *Request Bundle* would reduce drastically the communication between API client and server. However, having a big request bundle could create network congestion and longer processing time on the server. As a result, it is not obvious to make the right decision for the implementation of this pattern. Service meshes provide the possibility to have different versions of the same microservice as described in Section 4.4.5 which could make analyzing such a pattern feasible. Using *Custom Routing* capability, we would be able to deploy two different versions of the same microservice that deliver the same result. However, one version would handle incoming traffic with bundled data, and the second one without bundled data. As a result, we would be able to switch on and off *Request Bundle* on the fly to evaluate both cases using the same configuration and running environment (see Figure 2.1).

The *Load Balancing* pattern is a widely adopted API strategy aimed at distributing workload across multiple microservices to enhance scalability. Notably, this pattern is often externally implemented through third-party services, adding complexity and extra expenses. Consequently, it's crucial to assess the pattern's impact on the entire infrastructure to determine if the implementation efforts are justified. Service meshes offer a convenient approach to implementing *Load Balancing* using *virtual services*, as illustrated in Figure 2.2 and elaborated upon in Chapter 8. This implementation can be executed at various levels (such as with *Rate Limit*) to provide greater flexibility and control. Moreover, this pattern can be realized through different strategies or algorithms,

potentially including the emulation of multiple load balancers per service.

In general, combining API patterns is expected to have a positive impact on the related quality properties. However, it is not obvious to what degree this combination can be beneficial. For instance, *Rate Limit* has to be adapted carefully when used with *Request Bundle*. Without *Request Bundle*, it is expected to have chatty communication between API client and server and so *Rate Limit* could prevent to have excessive resource usage. However, when using *Request Bundle*, the chunks of communication become larger, and it would make sense to reduce the rate-limiting values. On the other hand, those bundles of data could also create a bottleneck and traffic congestion on the backend. More combinations scenarios, including the *Load Balancing* pattern, are implemented using service meshes and analyzed in detail in Chapter 8.

4.6 Estimation of Uncertainty Reduction

There are many different kinds of uncertainties involved in making ADDs in a field in which the architect's experience is limited. The obvious contribution of our ADD model is that it helps to reduce the uncertainty of whether all relevant, necessary, and sufficient elements for making a correct decision have been found. Another kind of uncertainty reduction is our ADD model's uncertainty reduction compared to using the same knowledge but in a completely unorganized fashion. We want to estimate this kind of uncertainty reduction here, following the approach described in detail in [ZSZ⁺18]. Here, we estimate the uncertainty reduction only for each individual decision. Please note that in most decisions, combinations of options from different decisions need to be taken. Still, as many decisions in our ADD model have different decision contexts, this can only be calculated precisely for actual decisions made, not for the reusable decisions in the ADD model. But a consequence is that the actually achievable uncertainty reduction is much higher than the numbers below when decisions need to be made in combination. We calculate each number both using our ADD model (denoted with $\oplus$ below) and not using our model (denoted with $\ominus$ below). Let DEC denote the decisions in our ADD model. For each $d \in DEC$, there are a number of decision options OPT_d possible to choose for decision d . Finally, a set of criteria CRI_d needs to be considered when making a decision d .

4.6 Estimation of Uncertainty Reduction

Number of decisions nodes (ndec): Our ADD model represents each decision separately. So the number of decision nodes for a single decision d is always $ndec_d^{\oplus} = 1$. Without our ADD model, each decision option in the design space that is not *Do Nothing* is a possible decision node, and it can either be selected or not: $ndec_d^{\ominus} = |OPT_d \setminus \{Do\ Nothing\}|$. Please note that if a design solution has variants, OPT_d contains the base variant plus each possible variant.

Number of required criteria assessments in a decision (ncri): Our ADD model includes explicit decision criteria per decision, and for all decisions described above all criteria are pre-decided in the sense that we have assigned a qualitative value $\{++, +, o, -, --\}$ to it, represented in the range: very positive, positive, neutral, negative, and very negative. Thus, the required criteria assessments per decision are one assessment per decision, $ncri_d^{\oplus} = 1$. Without our ADD model, we need to assess each criterion for each decision node (as we have no pre-decided choices): $ncri_d^{\ominus} = |CRI_d| \times |ndec_d^{\ominus}|$.

Number of possible decision outcomes (ndo): Our ADD model already models each decision option separately in $|OPT_d|$ including *Do Nothing* options, so $ndo_d^{\oplus}$ usually equals $|OPT_d|$ unless the design space allows explicit combinations of solutions as additional outcomes. For instance, in the decision on *managed ingress communication*, the *API Gateway* can be combined with the base variant *Front Proxy*. Let the function $solComb()$ return the set of possible solution combinations in the options of a decision; then $ndo_d^{\oplus} = |OPT_d| + |solComb(OPT_d)|$. The same is true in principle for the decisions made without our ADD model. Still, as the decision d is here split into multiple separate decision nodes $ndec_d^{\ominus}$ and without the ADD model, no information on which combinations are possible is present, we need to consider any possible combination in $ndec_d^{\ominus}$, i.e., the size of the powerset of the decision nodes: $ndo_d^{\ominus} = |\mathcal{P}(ndec_d^{\ominus})| = 2^{|ndec_d^{\ominus}|}$.

Table 4.2 shows the results of the uncertainty reduction estimation. It can be seen that the number of decisions to be considered $ndec$ can be, in total, reduced from 33 to 14, with an average improvement of 49.76% when using our ADD model. As all decisions have multiple criteria and when not using our ADD model, no decisions are pre-decided, the improvement for criteria assessments is higher: on average, an 86.70% improvement is possible. Finally, the possible decision outcomes are improved from 96 to 47, with an average 32.59% improvement.

Table 4.2: Uncertainty Reduction Estimation

Decision	$ndec^{\oplus}$	$ndec^{\ominus}$	Imp.	$ncri^{\oplus}$	$ncri^{\ominus}$	Imp.	$ndo^{\oplus}$	$ndo^{\ominus}$	Imp.
Managed Cross-Service Communication	1	3	66.67%	1	24	95.83%	4	8	50.00%
Managed Ingress Communication	1	2	50.00%	1	16	93.75%	3	4	25.00%
Traffic Control	1	5	80.00%	1	45	97.78%	6	32	81.25%
Service Mesh Expansion	1	2	50.00%	1	16	93.75%	3	4	25.00%
Service Discovery	1	2	50.00%	1	16	93.75%	3	4	25.00%
Load Balancing	1	3	66.67%	1	21	95.24%	4	8	50.00%
Custom Routing	2	4	50.00%	2	8	75.00%	6	10	40.00%
Health Checks	1	2	50.00%	1	6	83.33%	3	4	25.00%
Security	3	7	57.24%	3	24	87.50%	10	16	37.50%
Telemetry	1	1	0.00%	1	4	75.00%	2	2	0.00%
Multi-Protocol Support	1	2	50.00%	1	16	93.75%	3	4	25.00%
Total	14	33		14	196		47	96	
Average Improvement Per Decision			49.76%			86.70%			32.59%

4.7 Discussion and Threats to Validity

We have studied knowledge on established practices in service mesh architectures, relations among those practices, and decision drivers to answer our research questions **RQ1.1-3**, respectively, with multiple iterations of open coding, axial coding, and constant comparison to first codify the knowledge in informal codes and then in a reusable ADD model. Precise impacts on decision drivers of design solutions and their combinations were documented as well; for space reasons, we only summarized those in the text and did not show them in detailed tables. In addition, we estimated in Section 4.6 the uncertainty reduction achievable through the organization of knowledge in our ADD model. We may conclude that our ADD model (and similar models) has the potential to lead to substantial uncertainty reduction in all evaluation variables due to the additional organization it provides and the pre-selections it makes. For individual decisions, mastering and keeping in short-term memory the necessary knowledge for design decision-making seems very hard for the case without the ADD model (see numbers in Table 4.2), but it is quite feasible in the case of our ADD model. Our model also helps to maintain an overview of the decisions $ndec^{\oplus}$ and criteria assessments $ncri^{\oplus}$ in the combinations of contexts. Only the number of possible decision outcomes for the combination of multiple decisions

seems challenging to handle, both in the $ndo^{\oplus}$ and $ndo^{\ominus}$ case. That is, despite all the benefits of our approach, the uncertainty estimations show that a limitation of the approach is that when multiple decisions need to be combined in a context, maintaining an overview of possible outcomes and their impacts remains a challenge – even if a substantial uncertainty reduction and guidance is provided as in our ADD model. Further research and tool support is needed to address this challenge. As our numbers are only rough estimates, further research is needed to harden them and confirm them in empirical studies, maybe based on a theory developed based on such preliminary estimations.

While we believe the generalizability of our results beyond the knowledge sources we have studied is possible to a large extent, our results are limited to those sources and, to a lesser extent, to very similar service mesh architectures. Most sources were public Web sources; there might be in-house practices not reported to the public by practitioners not covered here. Some sources were from the technology vendors, which might have introduced bias, but this is mitigated to a certain extent as we considered sources from most major service mesh vendors. Our results are only valid in our set scope; we do not claim any form of completeness. Possible misinterpretations or biases of the author team cannot be entirely excluded and might have influenced our results. We aimed to mitigate this threat through our own in-depth experience and by carefully cross-checking the sources in many iterations.

4.8 Conclusions

We have performed in this chapter a qualitative study in which we have studied service mesh established practices and proposed a formally defined ADD model. In total, based on our findings, we modeled 14 architectural design decisions with 47 decision outcomes and 77 decision drivers. In our uncertainty reduction estimations, we were able to indicate that the knowledge organization in our ADD model can lead to a significant reduction of uncertainty. We plan in our future work to combine our ADD model with other aspects of microservice design and DevOps practices and empirically validate a theory based on the preliminary uncertainty reduction estimations. We also plan to validate our ADD model using real-life case studies with field practitioners.

In this chapter, we uncovered architectural design decision options when setting up service meshes. We found that this technology provides strong support for API pattern implementation. In the following chapters, we start by using service meshes to empirically evaluate some central services and proxy tasks (described in Section 4.4.5) or API patterns including *Rate Limit* and *Load Balancing*. Before that and in the next chapter, we exhibit a deep dive into the first API pattern that will be empirically studied: *Rate Limit*.

5 API Rate Limit Adoption – A pattern collection

This Chapter refers to the work established [\[SMPZ23\]](#) that presents an analysis of the prevalent patterns utilized by developers for defining Rate Limit configurations and strategies, alongside their practical applications in real-world APIs. Moreover, a thorough examination is conducted on the extent and granularity with which Rate Limit constraints can be enforced. Furthermore, multiple implementation-related patterns are identified, and a compilation of noteworthy technological examples is provided as known uses. The patterns identified in the study revolve around the configuration and parameters of Rate Limit policies, the breadth and specificity of the enforced policies, methods employed to combat abusive clients, and diverse server-side implementation strategies. The API rate limit adoption pattern collection is structured by grouping patterns with similar objectives.

In the following sections, we identify specific patterns delineating various potential implementations of Server-Side rate limiting, which we categorized based on the rate limiter component, its placement, and its scope. The choice of solution relies on the type of system architecture housing the API. By pinpointing the relevant pattern, developers can efficiently apply rate limiting at the interface level of the system.

5.1 Positioning of Rate Limiter

5.1.1 Internal Rate Limiter

Context: A service provider has control over their infrastructure and systems, enabling them to regulate the rate of incoming requests.

Problem: How can a controllable rate limiter be implemented without relying on

5 API Rate Limit Adoption – A pattern collection

external services?

Solution: Rate Limit is fully integrated within the server-side code. The rate limiter intercepts and filters clients' requests upon reaching the backend service that implements the API.

Solution details: In monolithic service architectures [KZH⁺20], a rate limiter can be part of the backend service.

In microservice architectures [Ric16], the Rate Limit policy can also be internally implemented within the backend architecture. The precise placement of the rate limiter depends on the chosen infrastructure to implement the microservices architecture:

Service Mesh based architecture. In a service mesh architecture, the Rate Limit policy is positioned in the control plane to ensure centralized and consistent enforcement. Different strategies can be adopted to implement rate limits internally within the backend:

- Configuring the sidecar proxy to enforce rate limits on inter-services communications helps shift the Rate Limit control from the application to the networking layer.
- Designating one service within the mesh as a rate limiter service can centrally manage and enforce Rate Limits for all microservices within the mesh.

API Gateway based architecture. In this architecture, an API gateway sits between the client and the microservices, and it can host a rate limiter component to regulate incoming requests to each microservice.

Consequences:

- + *Customization:* Since it's part of the provider's code, the rate limiter component can be tailored to match specific rate limits required by the application and handle different request types differently.
- + *Integration:* Seamless integration with the application codebase facilitates easier updates as the application evolves.
- *Increased complexity:* Additional development effort and maintenance overhead are required, potentially adding complexity to the application codebase.

Known uses:

API Gateway: Many API gateways, such as Kong (which we employ in Chapter 6) or Tyk, have built-in rate-limiting functionality, allowing enforcement of Rate Limits across multiple APIs or endpoints.

Envoy Proxy: Used by service mesh technologies like Istio and Kuma, it can serve as a Front or Edge Proxy for granular rate limiting. We implement this approach in Chapters 8 and 9.

5.1.2 External Rate Limiter

Context: A readily available solution to rate limiting is provided by an external system.

Problem: How can implementing rate limiting be expedited and simplified?

Solution: Rate Limit is implemented using third-party services or libraries.

Solution details: Select an appropriate rate-limiting service or tool, configure rate-limiting rules, integrate with the API, test, monitor, and fine-tune as needed.

Consequences:

- + *Speed of implementation:* Reusing an existing rate-limiting component can save time compared to building one from scratch.
- + *Compliance:* Ensures compliance with regulations and industry standards related to rate limiting.
- *Security:* May introduce security risks, such as exposure of sensitive data to external service providers.
- *Performance:* May introduce additional latency and overhead in request processing.
- *Reliability:* The system becomes dependent on external factors such as network connectivity and availability of the rate limiter.

Known uses:

Cloud provider's rate-limiting service: AWS, Google Cloud, and Azure offer rate-limiting services that can be integrated with an application or API.

5 API Rate Limit Adoption – A pattern collection

Third-party rate-limiting services: Cloudflare, Akamai, and Fastly offer advanced features like DDoS protection and real-time monitoring.

Open-source rate-limiting components: Redis, Guava, and Bucket4j can provide basic rate-limiting functionality.

Related Patterns: A hybrid approach combining an External Rate Limiter and an Internal Rate Limiter can enhance system reliability by providing a backup mechanism in case of failure or performance issues with the external rate limiter. However, integration challenges may arise between the Internal and External Rate Limiters.

5.2 Scope of Rate Limiter

5.2.1 Global Rate Limiter

Context: The system architecture is composed of multiple services.

Problem: How to enforce a shared and uniform Rate Limit value for all services of a system?

Solution: Rate Limit is implemented using a Front Proxy to manage the overall incoming traffic to a system.

Solution details: This involves identifying operational thresholds that require a global rate limiter. The Front Proxy can enforce the Rate Limit either at the application layer or the network layer.

Consequences:

- + *Consistency:* Ensures a uniform rate limit for all services of the architecture.
- *Flexibility:* Adjusting or fine-tuning the rate limit to specific resource consumption rates can be challenging.

Known uses:

Envoy's service proxy is often used to implement global rate limiting in Service Mesh architectures.

Nginx ingress allows configuring global rate limits using the `limit_req_zone` directive.

Redis can be used to implement global rate limiting and is supported by several API Gateway pattern implementations.

5.2.2 Local Rate Limiter

Context: The system architecture is composed of multiple services.

Problem: How to define and enforce different Rate Limit values for each service?

Solution: Rate Limit is implemented within the service mesh using Edge Proxies, enabling each service to independently manage its incoming requests.

Consequences:

- + *Flexibility:* Local rate limiting provides more control over the rate-limiting logic, allowing for more fine-grained control and customization of the rate-limiting rules for each service.
- *Complexity:* Implementing local rate limiters requires additional code and configuration, increasing system complexity and potentially introducing bugs or performance issues.

Known uses:

Envoy supports local non-distributed rate limits where the Rate Limit can be configured for specific endpoints.

GitHub API uses different rate limits for various services it allows access to, prioritizing authentication requests over unauthenticated ones.

Related Patterns: Combining both Global and Local rate limiters can offer better control and flexibility over resource utilization in a system. Global rate limiters ensure overall system load remains manageable, while local rate limiters provide finer control over specific services, enhancing resource efficiency and performance.

Part III

API Performance and reliability prediction models

6 Impact of API Rate Limit on Reliability of Microservices-Based Architectures

Many API patterns and best practices have been developed around microservices-based architectures, such as Rate Limiting and Circuit Breaking, to increase quality properties such as reliability, availability, scalability, and performance. Even though estimates on such properties would be beneficial, especially during the early design of such architectures, the real impact of the patterns on these properties has not been rigorously studied yet. This chapter focuses on API Rate Limit and its impact on reliability properties from the perspective of API clients. We present an analytical model that considers specific workload configurations and predefined rate limits and then accurately predicts the success and failure rates of the back-end services. The model also presents a method for adaptively fine-tuning rate limits. We used service meshes to set up the back-end infrastructure as well as the configuration of the different workload scenarios, the monitoring, and the data collection. We performed two extensive data experiments to validate the model and measured Rate Limiting impacts, firstly on a private cloud to minimize latency and other biases and secondly on the Google Cloud Platform to test our model in a realistic cloud environment. In both experiments, we observed a low percentage of prediction errors. Thus, we conclude that our model can provide distributed system engineers and architects with insights into an acceptable value for the rate limits to choose for a given workload. Very few works empirically studied the impact of Rate Limit or similar API-related patterns on reliability.

6.1 Introduction

In the current chapter, we have selected *Rate Limit* as the first API pattern to study to answer the first part of **RQ1**. This is a prominent and widely used API pattern or best

6 Impact of API Rate Limit on Reliability

practice by several Cloud and online service providers like eBay, Google, and Amazon.

The increasing adoption of microservices-based architectures has posed many challenges regarding API design, including runtime properties such as reliability, availability, and performance [FML17]. Many API best practices and patterns have been identified [ZSZ⁺18, PZA⁺17, SZZ⁺18]. A prominent and often applied practice is *API Rate Limit* [ZSL⁺20b]. *Rate Limit* aims to protect services from excessive usage and nudge *API clients* into becoming paying customers. It is supposed to increase the reliability of the services from *API clients* perspective. Many techniques and solutions exist that are, e.g., deployed in a Cloud-based architecture [Goo20] or provided as configurable options of an *API Gateway* [Pap17]. However, there is little guidance on which technique to use, how to use them, in which specific circumstances, and using which configurations.

Our study aims to fill this gap by developing an analytical model for accurately predicting the reliability impacts of *Rate Limiting* in microservice-based applications from *API clients* perspective. This model also presents a solid method for adaptively fine-tuning rate limits. We empirically validated this model using a set of microservice-based workload scenarios deployed on a private cloud (to minimize latency and other biases) and on the Google Cloud Platform (GCP) (to test our model in a realistic cloud environment) and supported by a service mesh-based infrastructure. Note that *Rate Limit* also substantially impacts overall system performance and scalability, but this is beyond the scope of this chapter. We simulate real-world API client workloads using different configurations and settings for empirical validation. Our goal is to answer the following research questions:

RQ2.1 How can we accurately predict *API Rate Limit* impacts on reliability properties of a microservices-based system from *API clients* perspective?

RQ2.2 What are the effects of *API Rate Limit* on those reliability properties?

We propose an analytical model to accurately predict the influence of *Rate Limit* on reliability properties from *API clients* perspective. We model each complete simulation run as a Bernoulli process composed of a series of Bernoulli trials, each mapping to a predefined rate limit.

To validate our model, we have set up an experiment using a representative, modern

Cloud-based architecture based on microservices and the Istio service mesh¹, testing realistic workload scenarios (see Section 6.5.1). Then, we ran the experiment simulating 20 different configurations of API client workloads and repeated each experiment 50 times on the private cloud and GCP. The total runtime of the experiments was more than 2,000 hours. We compared the results from the model with the data generated from each of these configurations. We have found that the error is significantly reduced with more runs or stays stable and stands at the maximum values of 17,7% for the private cloud and 16,73% for the GCP experiment, respectively. Those values are substantially below the target prediction error of up to 30% for Cloud-based architectures [MA02] and explainable with network infrastructure imperfections such as latency and unforeseen errors. The predictions are acceptable since they are close to reality and are sufficient to make broad or early architectural decisions.

The chapter is organized as follows. We give an overview of existing *Rate Limit* techniques and the technique we chose in our research in Section 6.3. In Section 6.4, we present our analytical model. After that, we describe the experiment we have set up to empirically assess the model in Section 6.5. In Section 6.6, we compare the results from the model and the experiments and discuss threats of validity. Finally, we conclude in Section 6.7.

6.2 Related work

This chapter focuses on designing a prediction model to help software architects select the best *Rate Limit* strategy for a specific workload and a particular server configuration. Many approaches and methods have been presented for performance and reliability prediction in software architecture, such as performance model derivation and analysis based on model transformations [DMM06] and software quality prediction based on Bayesian Nets [Rad11]. Duzbayev et al. [DP06] present a runtime prediction model of software architecture QoS specific to queued systems. Adjepon-Yamoah [AY16] argues that most approaches have limitations, especially in unpredictable environments like Cloud-based architectures. Only a few studies proposed performance models dedicated to Cloud environments [KBBML16, JPG19]. Likewise, our approach is based on an

¹<https://istio.io/docs/concepts/what-is-istio/>

analytical model and empirical data measured in a private cloud and GCP. It is generally cloud-compatible and can be applied to any microservice-based system. Since we cannot cover all microservice technologies and concepts, slight adaptations of measurements and models might be needed in environments with other characteristics, such as other public clouds [LSL19].

Empirical assessments of software architecture reliability have also been the subject of many research papers. Katerina et al. [GPHP05] discuss the issue that most software reliability models research papers do not present empirical evaluation and assessment of those models. A simulation tool called the Palladio Component Model (PCM) was extended by Brosch et al. [BKBR10] to close the gap between the (UML-based) modeling of a system and its empirical evaluation. Another simulation tool is Cloudsim [CRB⁺11], which is sometimes applied for service simulations but generally more low-level, i.e., with a focus on cloud resources rather than architectural components. Chaos engineering [BBd⁺16] has been introduced as a discipline-specific to large-scale microservice-based systems for real-time reliability testing. Our proposed reliability model is a novel parametric approach assessed using a widely used service mesh deployable at a considerable scale. None of these approaches have been applied with an API-centric perspective or focused on specific API patterns to the best of our knowledge. Thus, the research question of what the introduction of *Rate Limit* means for the results achieved with such approaches has not yet been answered, and our study aims to close this gap.

Rate Limit and similar techniques have been introduced as best practices or patterns [ZSZ⁺18, PZA⁺17, SZZ⁺18, ZSL⁺20b, MZ21] and in the context of Service-Level Agreements [GDFRC⁺19, GDFMRC18]. However, the effects of *Rate Limit* have not yet been studied empirically in the scientific literature. This is interesting because *Rate Limit* is a major API practice, and it has a substantial influence not only on reliability and availability properties but also on other critical properties such as security and performance. Thus, measuring the impact of *Rate Limit* and related patterns is crucial to unveil their impacts on those properties and tradeoffs for decision-making. We believe it is necessary to study and integrate the effects of such major practices that might influence every single distributed call between clients and servers in a system to improve the results of existing simulators and prediction models.

6.3 Background on Rate Limiting

Nowadays, service providers usually expose their APIs through API Gateways [Ric20] to provide a central access point and ensure the security and availability of the services. *Rate Limit* [ZSL⁺20b, ZSZ⁺18, SZZ⁺18] is one of the significant patterns that are used to ensure that the API provider is not overwhelmed by excessive requests of *API clients*, either intentional in API abuse or Denial of Service attacks or unintentional, e.g., as a result of system or user errors. *Rate Limit* also solves scalability (and thus overall system performance) and system reliability issues, as it can help cope with situations where an API provider unexpectedly receives a burst of client requests. In some cases, this practice might be linked to extra functionalities like limiting access to a specific *API clients* from a specific device (e.g., mobile device) at a particular period. It can also detect *API clients* with shallow frequency access and who intentionally aim to keep connections open for a long time.

Rate Limit is usually linked to a *Rate Plan* [ZSL⁺20c], which enforces a billing strategy on API usage. Combining those patterns solves most of the problems described lately, as it provides an efficient way to control the behavior of *API clients*. However, this produces an additional computation and resource consumption overhead for authenticating and authorizing *API clients* and keeping track of their usage, especially when the *usage-based pricing* is selected as the base billing policy. Usually, a request is sent by an *API client* along with an *API Key* [ZSL⁺20a] or information for some other authorization mechanism to verify the legitimate use of the requested API resource.

There exist two types of *Rate Limit*: First, backend rate limiting is enforced by the backends' physical capacity and measured as Transaction Per Second (TPS). Second, application rate-limiting enforces limited requests per period or a quantity of data per user [Sim19]. Our study focuses on the second rate-limiting type by enforcing limited user requests per minute.

Several techniques can enforce rate limiting, such as the token bucket, leaky bucket, or fixed and sliding window practices [Goo20]. When using the token bucket technique, each request is considered a set of operations, and each of these operations consumes one token until it reaches the maximum bucket capacity. The leaky bucket technique is similar to the first one, but the rate limit is applied directly to the requests, and any request that exceeds the capacity of the bucket is leaked. Our study covers all these

rate-limiting techniques.

6.4 Analytical model

In this section, we present our analytical model for the reliability of services with *Rate Limit* from *API clients* perspective, modeled as a Bernoulli process. The Bernoulli process represents an experiment during time T , where each time interval t represents a Bernoulli trial. We define n_t as the number of times we measure the success and failure rates of the system where:

$$\sum_{t=1}^{n_t} t = T \quad \text{with} \quad n_t = \frac{T}{t} \quad (6.1)$$

We model user profiles based on different usage levels, from low to high. On the client side, let SU_i denote the share of users with level i of usage, where i ranges from 0 to ∞ . Further, to precisely define each of these user groups, let RPM_i denote the number of requests per minute the users in SU_i send. Let TU denote the total number of users that send these requests. The total client requests TR^T in the time interval T can then be calculated as:

$$TR^T = T \cdot TU \cdot \sum_{i=0}^{\infty} SU_i \cdot RPM_i \cdot LF_i \quad (6.2)$$

Such that,

$$\sum_{i=0}^{\infty} SU_i = 100\%$$

And LF_i is the *load frequency occurrence* corresponding to SU_i . It represents the proportion of time T where the workload is being executed and is calculated using the formula:

$$LF_i = \frac{\frac{LOAD_i}{RPM_i}}{\frac{LOAD_i}{RPM_i} + SLEEP_i} \quad (6.3)$$

Where $LOAD_i$ is the number of concurrent requests sent by users in SU_i during time T and $SLEEP_i$ is the idle or no execution time corresponding to the users in SU_i during time T .

On the server side, let $TR_{ratelimit}^T$ denote the total number of requests that failed due to rate limiting, and let $TR_{failure}^T$ indicate the total number of requests that failed for other reasons than rate-limiting during the time interval T . Then $TR_{success}^T$, the total

number of succeeded requests, assuming that the reliability of the network infrastructure is very high, can be calculated as:

$$TR_{success}^T = TR^T - (TR_{failure}^T + TR_{ratelimit}^T) \quad (6.4)$$

Additionally, let $FR_{failure}^t$ and $FR_{ratelimit}^t$ denote the failure rates, during the time interval t , corresponding to $TR_{failure}^T$ and $TR_{ratelimit}^T$, respectively. $SR_{success}^t$ denotes the success rate, during the time interval t . The respective number of failed requests $TR_{failure}^t$, succeeded requests $TR_{success}^t$, and failed requests due to rate limiting $TR_{ratelimit}^t$, during the time interval t can be written as:

$$TR_{failure}^t = t \cdot FR_{failure}^t \quad (6.5)$$

$$TR_{success}^t = t \cdot SR_{success}^t \quad (6.6)$$

$$TR_{ratelimit}^t = t \cdot FR_{ratelimit}^t \quad (6.7)$$

We consider each time interval t as a Bernoulli trial. Let $P_{failure}^t$ denote the probability of request failure, $P_{ratelimit}^t$ the probability of request failure due to rate limiting, and $P_{success}^t$ the probability of a succeeded request. The corresponding expected numbers of failed requests $E_{failure}^T$, requests failed because of rate limiting $E_{ratelimit}^T$, and succeeded requests $E_{success}^T$, during the time interval T , can be written as:

$$E_{failure}^T = n_t \cdot P_{failure}^t \quad (6.8)$$

$$E_{ratelimit}^T = n_t \cdot P_{ratelimit}^t \quad (6.9)$$

$$E_{success}^T = n_t \cdot P_{success}^t \quad (6.10)$$

Where:

$$P_{success}^t = 1 - (P_{failure}^t + P_{ratelimit}^t) \quad (6.11)$$

Thus, the total number of failures $TR_{failure}^T$, failures due to rate limiting $TR_{ratelimit}^T$, and succeeded requests $TR_{success}^T$ can be expressed as:

6 Impact of API Rate Limit on Reliability

$$TR_{failure}^T = \sum_{n_t} E_{failure}^T \cdot TR_{failure}^t \quad (6.12)$$

$$TR_{ratelimit}^T = \sum_{n_t} E_{ratelimit}^T \cdot TR_{ratelimit}^t \quad (6.13)$$

$$TR_{success}^T = \sum_{n_t} E_{success}^T \cdot TR_{success}^t \quad (6.14)$$

which can be rewritten using equations (6.1), (6.5), (6.6), (6.7), (6.8), (6.9) and (6.10) as follows:

$$TR_{failure}^T = T \cdot \sum_{n_t} P_{failure}^t \cdot FR_{failure}^t \quad (6.15)$$

$$TR_{ratelimit}^T = T \cdot \sum_{n_t} P_{ratelimit}^t \cdot FR_{ratelimit}^t \quad (6.16)$$

$$TR_{success}^T = T \cdot \sum_{n_t} P_{success}^t \cdot SR_{success}^t \quad (6.17)$$

Using equations (6.2), (6.3), (6.4), (6.15), (6.16) and (6.17), the resulting formula including both client-side and server-side parameters is the following:

$$\begin{aligned} & \overbrace{\sum_{n_t} \frac{P_{failure}^t}{TU} \cdot FR_{failure}^t}^{\text{non-rate-limiting failure rate}} = \\ & \text{workload} \left\{ \sum_{i=0}^{\infty} SU_i \cdot RPM_i \cdot \frac{\frac{LOAD_i}{RPM_i}}{\frac{LOAD_i}{RPM_i} + SLEEP_i} \right. \\ & \left. - \underbrace{\sum_{n_t} \frac{P_{success}^t}{TU} \cdot SR_{success}^t}_{\text{success rate}} - \underbrace{\sum_{n_t} \frac{P_{ratelimit}^t}{TU} \cdot FR_{ratelimit}^t}_{\text{rate-limiting failure rate}} \right. \end{aligned} \quad (6.18)$$

Such that:

$$\begin{aligned} NRL_FR &= \sum_{n_t} \frac{P_{failure}^t}{TU} \cdot FR_{failure}^t \\ SR &= \sum_{n_t} \frac{P_{success}^t}{TU} \cdot SR_{success}^t \\ RL_FR &= \sum_{n_t} \frac{P_{ratelimit}^t}{TU} \cdot FR_{ratelimit}^t \end{aligned} \quad (6.19)$$

The resulting equation shows that the *non-rate-limiting failure rate* NRL_FR might decrease if both the *success rate* SR and *rate-limiting failure rate* RL_FR increase. Indeed, *Rate Limit* can considerably decrease NRL_FR . However, if not configured well, it can also lead to the decrease of SR and thus increase NRL_FR . Table 6.1 summarizes the variables used in our model.

Table 6.1: Definition of Parameters

Independent variable	Description
SU_i	Percentage of the user group with usage level i in the total users share.
RPM_i	Number of requests per minute the users in the share of users of level i usage send.
$LOAD_i$	Number of concurrent requests executed by level i usage users during time T .
$SLEEP_i$	Idle or no execution time for level i usage users during time T .
TU	Total number of users in the experiment.
T	Total duration of the experiment in minutes.
t	Trial duration in minutes.
Dependent variable	Description
NRL_FR	Per-minute number of requests failed for other reasons than rate limiting.
RL_FR	Per-minute number of requests failed due to rate limiting.
SR	Per-minute number of requests succeeded.

6.5 Empirical evaluation

We have followed the eight methodological principles for reproducible Cloud performance evaluation [PVB⁺19]. Principle 1 emphasizes repeating the experiments many times and quantifying the results. Indeed, we repeated the experiments 50 times until we reached a reasonably low prediction error. As per Principle 2, each of these experiments has to run a different configuration, which is the case in our work since we have run 50 different configurations (10 workload configurations $\times$ five rate limit configurations) using a private cloud and 50 additional configurations using GCP. Section 6.5.3 describes in detail the hardware and software configurations used in the experiments, which is required for fulfilling Principle 3. To satisfy Principle 4 and enable the reproducibility of our study, we published all artifacts as open-access². In Section 6.5.5, we analyze the results using box plots as recommended for Principle 5. Then, we evaluate the data's accuracy by comparing the experiments' results to our model, as described in Section 6.6, and required as Principle 6. As per Principle 7, all the measurement units are explicitly displayed. Principle 8 is about reporting cloud costs, which we do for the GCP

²To enable reproducibility of our study, all artifacts used in the experiment, including code and data, are published as an open-access artifact on Zenodo: <https://doi.org/10.5281/zenodo.6560270>.

6 Impact of API Rate Limit on Reliability

Table 6.2: GCP cost details (in Euros)

Service	Cost	Discounts	Promotions and others	Subtotal
Compute Engine	416,56	-3,72	-243,22	169,62
Cloud Logging	74,2	0	-13,65	60,55
Kubernetes Engine	94,99	-94,99	0	0
Total	585,75	-98,71	-256,87	230,17

experiment in Table 6.2 (not applicable for the private cloud experiment).

6.5.1 Benchmark workload scenarios

Existing benchmarks for studying application-level microservices, such as TeaStore [vKES⁺18], are based on examples implemented by researchers. We essentially followed the same approach, but to design realistic workload scenarios, we decided to perform our experiments using workload scenarios derived from typical kinds of interactions observed in a prior empirical study of open-source microservice architectures realized by practitioners [NZP⁺20] (e.g., we tried to model similar call chains length, number of database interactions, processing steps, etc.). In this context, we observed that typical E-commerce or business interactions such as authentication, product selection, basket interactions, and payment occur relatively often in open-source systems. Thus, we used such interactions for our workload scenarios. We have extended our study to other benchmarks as described in the remaining chapters.

Many open-source systems showcase specific technologies, such as specific tracing or deployment technologies, or microservice concepts, such as event sourcing. To avoid the substantial bias by those specific technologies or concepts in the implementations, we decided not to experiment based on one of those open-source systems. Instead, we created a benchmark based on those workload scenarios to model typical interactions in microservice-based business applications. In particular, we (1) modeled eight typical interactions in four services covering authentication, product, basket, and payment functionality as benchmark scenarios based on the steps repeatedly occurring in the open-source systems, and (2) implemented those scenarios from scratch as plain RESTful HTTP services. Based on the mentioned empirical study [NZP⁺20], we believe those scenarios to be representative of typical interactions in many business systems. We developed

our RESTful microservices using the widely used Java/Spring Boot³ technology. Each microservice exposes a set of REST API endpoints that execute the corresponding operations on its dedicated Mongo⁴ database instance.

A typical workload scenario in our experiments starts with the number – predefined earlier during workload initiation (see Tables 6.3 and 6.4) – of *API clients* of type *customer* sending *POST* login API calls to the *authentication* microservice. For instance, then, each of these customers views the list of available products using the provided *GET* API endpoint in the *product* microservice and adds his reserved product (for the reason described earlier) to the basket by calling *POST* API Endpoint in the *basket* microservice. After that, each customer proceeds to payment and checkout by sending a *POST* request to the *payment* microservice. The latter sends *DELETE* requests to both *basket* and *product* microservices to remove the purchased product from the customer’s basket and from the list of available products to purchase, respectively. Numerous executions of such scenarios can be executed concurrently during our experiment runs.

6.5.2 Architecture overview

Service meshes have been studied as an infrastructure layer that manages communication in microservice architectures and performs numerous *Proxy Tasks* including API rate-limiting (as described in Chapter 4). It has also been pointed out that service meshes provide numerous *Central Services* like collecting telemetry, traces, and metrics, considerably reducing the complexity of data collection and analysis. We have chosen to use the Istio service mesh as a representative modern microservice platform in our current study. To handle *Ingress Communication*, we have decided to use a *Front Proxy* as an API Gateway [Ric20], as we intend to assess rate limiting on the traffic coming from outside the service mesh. We have also selected the *Multi-Cluster Support* option for our private cloud, as it is simple and easy to administrate.

We have deployed our benchmark workload scenarios on the Istio service mesh. We aim to provide a realistic setup for a modern microservice deployment configuration. Istio is installed using its multi-cluster option with one shared control plane⁵. Each cluster

³<https://spring.io/projects/spring-boot>

⁴<https://www.mongodb.com>

⁵<https://archive.istio.io/v1.4/docs/setup/install/multicluster/shared-vpn/>

6 Impact of API Rate Limit on Reliability

represents a Minikube^[6] instance running a Kubernetes engine^[7] under a preconfigured Virtual Machine (VM). The benchmark workload scenarios run on four Java-based microservices: authentication, product, basket, and payment. Each microservice is a Docker^[8] image deployed on a single cluster along with a dedicated Mongo database Docker image and exposes REST API endpoints accessible via the HTTP Protocol. The internal communication between these microservices is established through Sidecars (see Chapter 4) over a private network.

We have integrated the Kong API Gateway [KC20] into Istio to intercept incoming client communication. The reason for choosing this API Gateway is that it is open source and can be seamlessly integrated with Istio. Also, it natively supports rate limiting, considerably reducing its implementation complexity. In addition, this way, we can avoid the threat to validity that – by implementing our rate limiting solution – we could have introduced some unrealistic implementation variant or other bias into the used rate limiting solution. Finally, the Kong API Gateway already provides *API metrics* [Bag20b], using the time series `kong_http_status`, that can be directly collected and visualized using tools like Prometheus^[9] or Grafana^[10].

In GCP, we used the previously described setup but without the *Multi-Cluster Support* option. Instead, we used only one Google Kubernetes Cluster (GKE^[11]) composed of four nodes pool.

6.5.3 Configuration details

We have run the experiments on a private cloud, composed of 4 Ubuntu^[12] 18.04.4 LTS Virtual Machines (VMs) running on vSphere^[13] 6.7 environment. Each of these VMs runs a Minikube instance (version ranging from 1.8.2 to 1.9.2) with eight dedicated CPU cores Intel Xeon(R) CPU E5-2650 v4 @ 2.20GHz and 20 GB of system memory. Each Minikube instance runs Kubernetes engine version 1.14.2 and Istio version 1.4.3.

⁶<https://minikube.sigs.k8s.io/docs/>
⁷<https://kubernetes.io>
⁸<https://hub.docker.com>
⁹<https://prometheus.io>
¹⁰<https://grafana.com>
¹¹<https://cloud.google.com/kubernetes-engine>
¹²<https://releases.ubuntu.com/18.04/>
¹³<https://www.vmware.com/products/vsphere.html>

The central Minikube instance hosts the central *Control Plane*, Kong Ingress Controller¹⁴ version 0.8.0, the authentication microservice built using Java version 8 and its dedicated Mongo database (latest version used). The *Ingress Controller* accesses the 3 remaining microservices through YAML-defined Kubernetes endpoints¹⁵ and Ingress Rules [Bag20a]. The data are collected using Prometheus version 11.0.3 and visualized using Grafana version 5.0.8.

On the client side, nine physical desktop clients and six virtual desktop clients are used to send HTTP requests to the private cloud. All desktop clients run Ubuntu 18.04.4 LTS, and eight physical desktops have 4 CPU cores, Intel Core i3-2120T CPU @ 2.60GHz with 8 GB of system memory. The remaining physical desktop client has 4 CPU cores: Intel Core i5-4670S CPU @ 3.10GHz, and 30 GB of system memory. All virtual desktop clients have 2 CPU cores Intel Xeon(R) CPU E5-2650 0 @ 2.00GHz with 8 GB of system memory. An experiment involves three desktop clients, each dedicated to a specific category of users in terms of the number of requests per minute sent and a specific percentage share as described in Table 6.1.

In GCP, we used one Cluster composed of 4 identical e2-medium VMs with eight dedicated Intel Broadwell vCPUs and 16 GB of system memory. The Cluster runs these newer Kubernetes engine 2.6, Istio 1.6.14, and Kong Gateway 2.5. We also used three virtual desktop clients identical to the ones described above.

6.5.4 Experimental setup

Each experiment starts by launching a workload script on three selected desktop clients, each corresponding to a specific category of users, as described previously. These workloads execute the necessary setup requirements on the server side, creating passwords and a product database by calling the corresponding microservices using their appropriate API endpoints. Tables 6.3 and 6.4 list the different workload configurations (C1 ... C10) used in the experiments. Note that the following description of the values used in those workload configurations is specific to the current application and cloud settings and cannot be generalized to other experimentation settings without further adjustments. The value of $SU_{i \leq 10}$, reflecting the most common type of users that never overloads

¹⁴<https://github.com/Kong/kubernetes-ingress-controller>

¹⁵<https://kubernetes.io/docs/concepts/services-networking/service>

6 Impact of API Rate Limit on Reliability

Table 6.3: Workload configurations for experiments using the private cloud

Client parameter	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
TU	100	150	100	150	100	150	100	150	100	150
$SU_{i \leq 10}$	50 %	46 %	50 %	50 %	48 %	50 %	50 %	47 %	54 %	53 %
$RPM_{i \leq 10}$	5	3,12	5	2	4	10	4	5	4	7
$SU_{i > 10}$	40 %	46 %	30 %	35 %	35 %	30 %	45 %	46 %	40 %	42 %
$RPM_{i > 10}$	16,17	13	22	11	12	11	24	10,78	24,8	11
$SU_{i \geq 50}$	10 %	8 %	20 %	15 %	17 %	20 %	5 %	7 %	6 %	5 %
$RPM_{i \geq 50}$	80	100	140	62	85	150	260	70	180	150

the system and has *API freemium* subscriptions only, is generally set as the greatest percentage of at least 46% of total users. The corresponding values of $RPM_{i \leq 10}$ are usually less than ten requests per minute. Then, the value of $SU_{i > 10}$, representing a lower proportion of users that might overload the system and possess *API premium* subscriptions, fluctuates between 30% and 46%, without exceeding the value of $SU_{i \leq 10}$. The respective values of $RPM_{i > 10}$ are estimated at least ten requests per minute. Finally, the value of $SU_{i \geq 50}$, reflecting the undesired set of users that usually creates bursts, crashes the system, and don't possess any valid API subscription, fluctuates between 5% and 20%, without exceeding the value of $SU_{i > 10}$. These users can generate at least 50 requests per minute. On the server side, a configuration script is launched with the predefined variables T and t described in Table 6.1. The script executes mainly two tasks:

Enabling (and disabling) rate-limiting with a predefined value in the following: 5, 25, 75, 150, Infinite (for disabled rate limiting).

Collecting data by querying the Prometheus database and storing them in JSON¹⁶ format.

6.5.5 Results analysis: Experiments using the private cloud

To cover a wide range of possible workload configurations, the workload-related independent variables described in Table 6.1 are varied by respecting the constraints detailed in Section 6.5.4. These workload configurations are listed in Table 6.3.

¹⁶<https://www.json.org/json-en.html>

Table 6.4: Workload configurations for experiments using GCP

Client parameter	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
TU	100	150	100	150	100	150	100	150	100	150
$SU_{i \leq 10}$	50%	50%	55%	55%	50%	45%	50%	47%	55%	50%
$RPM_{i \leq 10}$	3,12	3,01	3,08	2,97	3,11	3,02	3,13	3	3,17	2,99
$SU_{i > 10}$	40%	40%	35%	35%	35%	35%	45%	42%	40%	30%
$RPM_{i > 10}$	28,01	35,09	27,54	24,43	35	33,24	25,68	33,15	30,75	34,29
$SU_{i \geq 50}$	10%	10%	10%	10%	15%	20%	5%	11%	5%	20%
$RPM_{i \geq 50}$	303,49	450	295,61	400	430	500	342,88	420	745,18	380

The results of the experiments are collected and displayed as boxplots in Figures [6.1](#), [6.2](#) and [6.3](#). From these plots, we can gather a few general conclusions, such as: These plots show that as the rate limit increases for all experiments, both SR and NRL_FR generally increase. In contrast, RL_FR naturally decreases. However, we observe a slight decrease of SR for C4, C6, C7, and C10, and a considerable reduction of NRL_FR for C5 and C9 when we turn off rate limiting. Then, a rate limit value of approximately 75 requests per minute should be selected for C4 and C6, where NRL_FR starts decreasing with a negative impact on RL_FR , especially for C4. However, for C7 and C10, applying a more significant rate limit value of around 150 requests per minute is more appropriate, where NRL_FR starts decreasing and RL_FR only slightly increases. On the other hand, for C5 and C9, rate-limiting only makes sense when applying a value of fewer than 25 requests per minute. However, the RL_FR increase is too high in this case, and SR dramatically decreases, especially for C5. Applying rate-limiting for C1 and C3 is an obvious choice for a value of around 150 requests per minute, with a bit of impact on SR and a very slight increase in RL_FR for C3. Finally, disabling rate limiting is best for C2; unfortunately, we cannot make any visual decision regarding C8.

To evaluate the accuracy of our model, we compare the data collected from the experiments with the data calculated using the model. To have an approximation of the probabilities $P_{success}^t$, $P_{failure}^t$ and $P_{ratelimit}^t$, we have run each experiment many times and selected the probabilities that best fit our model. We plan to improve our proposed model and omit the last step in future work. The resulting data are shown in Table [6.5](#). First of all, we notice that $P_{failure}^t$ is highest (more than 83%) when rate limiting is disabled. Also, $P_{success}^t$ is highest when rate limiting is enabled, except for C9, whose variation is hardly noticeable. This shows that *Rate Limit* plays an essential role in

increasing system reliability properties, but it may also degrade these properties if not correctly configured.

6.5.6 Results analysis: Experiments using GCP

To generalize the results of our experiment on the private cloud, we have run it on GCP as well. This is to avoid any bias that might affect the results from using a private cloud, test our model on a realistic public cloud environment, and open the door to experimentation in other public clouds and environments. We used the workload configurations described in Table 6.4 and compared the data collected from the experiments with the data calculated using the model in Table 6.6. We have used the workload configurations in the private cloud with few adjustments.

From Table 6.6, we conclude that for C6, enabling rate limiting does not make sense since it drastically decreases SR and increases RL_FR , with only a slight decrease in NRL_FR . However, we notice a significant decline of NRL_FR when applying a rate limit of 150 requests per minute for all remaining configurations, although the increase in RL_FR and decrease in SR are still very high. Unlike when using the private cloud, both $P_{success}^t$ and $P_{failure}^t$ have the highest values when rate limiting is disabled, which confirms the latter conclusions. This is mainly due to the GCP's high reliability against the selected workload configurations. Notice that box plots corresponding to GCP experiments are not shown here for space reasons and are included in the replication package¹⁷.

6.6 Prediction error and threats to validity

To evaluate the accuracy of our model, we calculate its prediction error using the Mean Absolute Percentage Error (MAPE) [TB17]. Then, we discuss the threats to validity.

6.6.1 Mean Absolute Percentage Error

We have run each experiment, using the workload configurations described in Tables 6.3 and 6.4, more than 50 times to show that the prediction error gets smaller with the increase of the number of runs and then stays relatively stable. A single run of

¹⁷<https://doi.org/10.5281/zenodo.6560270>

6.6 Prediction error and threats to validity

an experiment takes about 2.5 hours without considering the intermittent crashes and recovery times. Let n_{config} be the number of configurations in our experiments, equal to 50 using the private cloud and 30 using GCP. Also, let $rate_{model}^i$ and $rate_{experiment}^i$ denote the model and experiment values respectively, corresponding to SR , NRL_FR and RL_FR for each configuration i . Then, we have:

$$MAPE = \frac{100\%}{n_{config}} \cdot \sum_{i=1}^{n_{config}} \left| \frac{rate_{model}^i - rate_{experiment}^i}{rate_{experiment}^i} \right|$$

As seen in Table [6.7](#), the overall prediction errors decrease with the increasing number of runs for the experiment on the private cloud and reach stable values relatively early for the GCP experiment. After 50 runs we reach a prediction error of 17,7% for $SR\%$, 2% for $NRL_FR\%$, and 16,89% for $RL_FR\%$ using a private cloud and 16,73% for $SR\%$, 3,69% for $NRL_FR\%$, and 16,24% for $RL_FR\%$ using GCP. All the errors are below the target prediction error of up to 30% for Cloud-based architectures [MA02](#), and thus explainable with network infrastructure imperfections such as latency and unforeseen errors. Thus, the predictions are acceptable since they are close to reality and are sufficient to make broad architectural decisions.

6.6.2 Threats to validity

In our study, we have simulated workloads from only a limited number of desktop clients per experiment, simultaneously generating the workload configurations of users' requests, as described in Section [6.5.3](#). This could be a threat to validity since, in real-world scenarios, we usually have one desktop client per user. In our case, the shared network bandwidth would have reduced the values configured for $RPM_{i \leq 10}$, $RPM_{i > 10}$ and $RPM_{i \geq 50}$. We have monitored these values in real time and adjusted them accordingly to mitigate this issue.

The benchmark workload scenarios we used in our study are based on four homogeneous microservices, with only one service and one database per server. We have not considered the case where we have a heterogeneous environment, and we did not replicate the services to increase their availability, for example. Thus, our model does not consider server-side details like the number and type of services (such as database services, third-party services, and so on). Also, our model assumes that computational and network resources are continuously available with a low risk for errors (e.g., issues with locks,

6 Impact of API Rate Limit on Reliability

authorization/authentication) and bottlenecks, which is not valid in the real world. To exclude this threat, server-side configurations, and these resources, including potential errors, need to be integrated into our study. Please note that our model focuses on the API level only; from an abstract point of view, it does not matter whether a service is unreliable or slow because of the database or the service implementation. While this mitigates this threat to a large extent, it cannot be excluded that specific backend effects would influence the results on *Rate Limit* that can be observed at the API level.

A similar threat is that we used a limited number of E-commerce or business-related benchmark workload scenarios that might not represent non-business workflows. There is the threat that different applications and scenarios would yield substantially different experimental results. However, we plan to extend our study to other benchmarks and applications in the future. Again, we are only interested in the properties observable at the API level. Nonetheless, the model might have to be extended or changed to be applied to substantially different kinds of microservice technologies and concepts. However, as many microservice technologies and concepts exist, covering them in one experiment is virtually impossible. Hence, the study is based on plain (vanilla) RESTful services, and we have chosen the E-commerce/business domain as enterprise applications with many users are often rate-limited.

6.7 Conclusion and future work

Concerning **RQ2.1**, we have developed an analytical model to provide distributed system engineers and architects insights into which specific value to choose as a rate limit given a workload situation. We have empirically validated the model using 50 different configurations in a private cloud and an additional 50 different configurations in GCP with more than 2000 hours of runtime. We found that the prediction error generally decreases with more runs or stays stable and stands at 16,89% for *RL_FR%*, 2% for *NRL_FR%* and 17,7% for *SR%* using a private cloud and 16,73% for *SR%*, 3,69% for *NRL_FR%*, and 16,24% for *RL_FR%* using GCP, after 50 runs. These prediction errors are reasonably close to reality, given some indeterministic effects, such as network latency, that we have not included in our model.

Concerning **RQ2.2**, we conclude that applying *Rate Limiting* may increase the reliab-

ility properties of APIs, given a specific workload situation. However, finding the right balance between improving the success rate and keeping the failure rate at a certain minimum level is not trivial. We have provided the first step towards a solution by providing an empirically tested analytical model to accurately predict a configuration's and workload's impact on those variables. This model also presents a solid method for adaptively fine-tuning rate limits. Since we cannot cover all microservice technologies and concepts, we provide an approach that can be extended to other technologies and concepts. Hence, the study is based on plain RESTful services. We have also realized a ready-to-go service mesh-based infrastructure that could be used to evaluate *Rate Limit* using its other different variations [SMPZ23]. It can also be used to experiment with other API patterns as described in the next chapters.

In future work, we aim to improve the analytical model by varying other parameters like the total duration of the experiment T and trial duration t and including different server configurations, workloads, benchmark scenarios, and resource management. Also, we plan to generalize the model by applying it to other technologies, applications, and architectures.

In this chapter, we have partly responded to **RQ2** by providing an analytical model to predict the reliability impact of *Rate Limit*. In Chapter 8, we use that analytical model when combining *Rate Limit* with *Request Bundle* and *Load Balancing* for performance and reliability evaluation.

6 Impact of API Rate Limit on Reliability

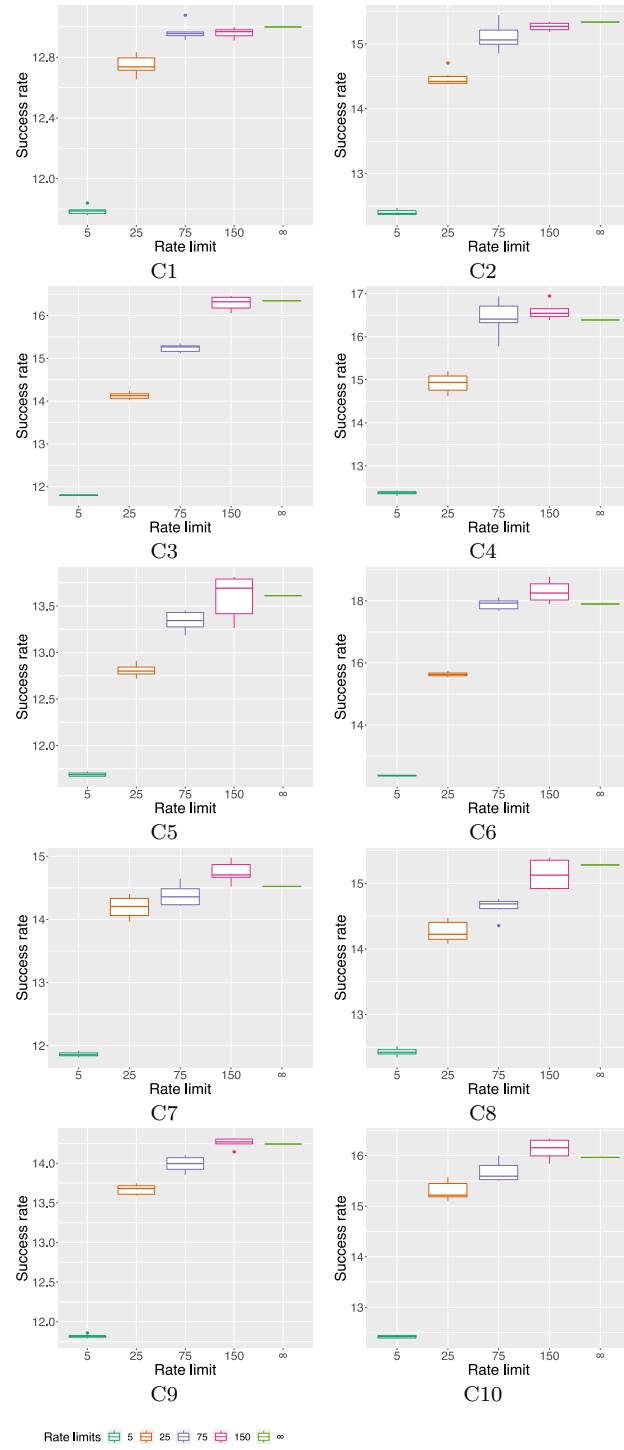


Figure 6.1: *SR* results from all the experiments (50 runs)

6.7 Conclusion and future work

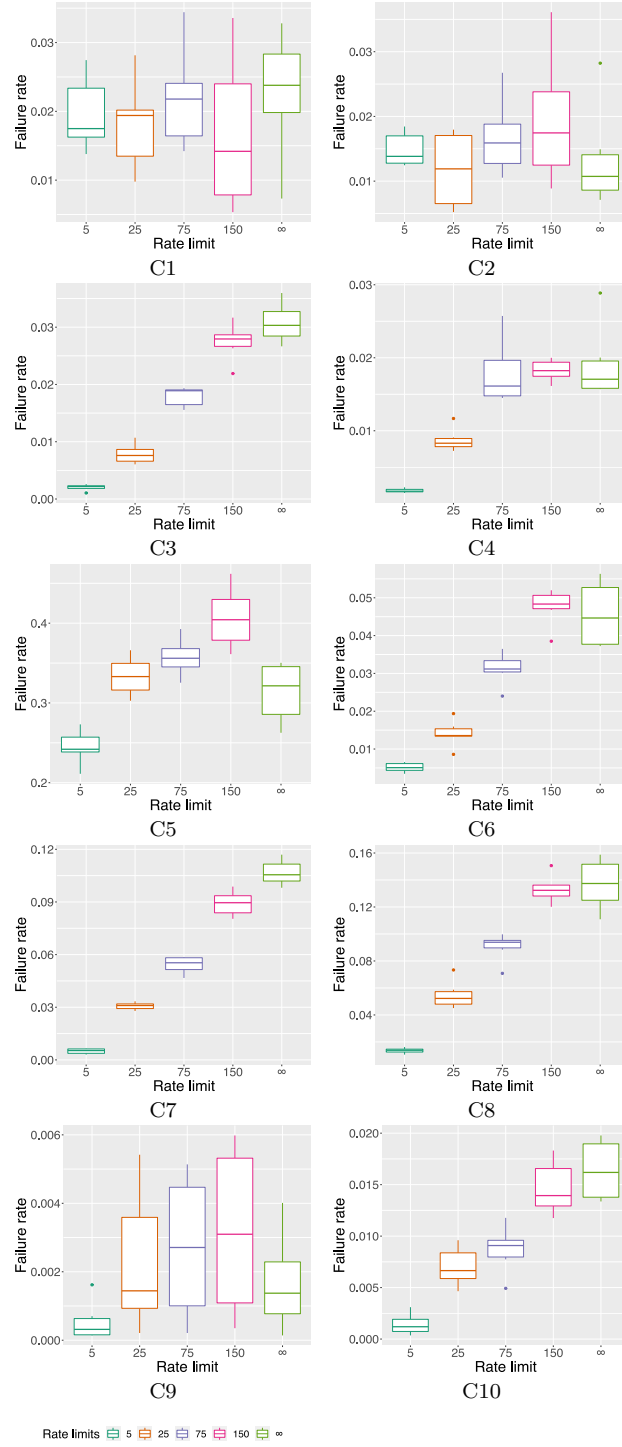


Figure 6.2: NRL_FR results from all the experiments (50 runs)

6 Impact of API Rate Limit on Reliability

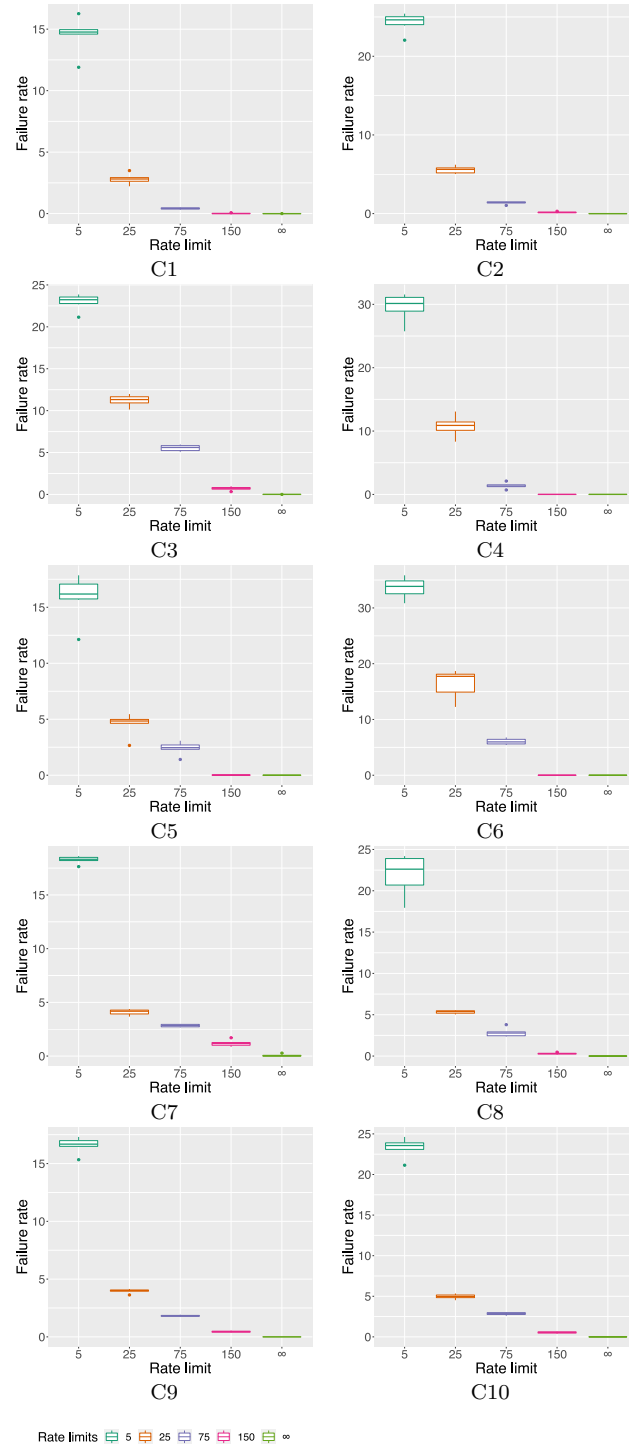


Figure 6.3: RL_FR results from all the experiments (50 runs)

6.7 Conclusion and future work

Table 6.5: Results from the model and the experiments using the private cloud (50 runs)

Config	RL	Probability %			Model			Experiment		
		$P_{success}^t$	$P_{failure}^t$	$P_{ratelimit}^t$	SR	NRL_FR	RRL_FR	SR	NRL_FR	RRL_FR
C1	5	16,61	66,85	16,54	2,96	0,05	4,91	7,01	0,05	8,66
	25	16,65	67,03	16,32	6,16	0,05	1,72	7,64	0,04	1,65
	75	16,59	68,61	14,8	7,73	0,05	0,14	7,74	0,05	0,22
	150	16,56	83,44	0	7,87	0,05	0	7,73	0,05	0
	∞	16,52	83,48	0	7,85	0,07	0	7,73	0,07	0
C2	5	16,62	66,77	16,61	2,64	0,02	4,77	5,8	0,02	9,68
	25	16,634	66,776	16,59	5,48	0,019	1,58	6,63	0,02	2,22
	75	16,648	67,252	16,1	6,9	0,026	0,51	6,89	0,03	0,53
	150	16,58	83,42	0	7,4	0,04	0	6,92	0,04	0
	∞	16,61	83,39	0	7,41	0,02	0	6,97	0,03	0
C3	5	16,61	66,709	16,681	3,71	0,005	9,22	8,33	0,005	13,79
	25	16,6	66,68	16,72	8,36	0,02	4,56	9,72	0,02	6,75
	75	16,56	66,63	16,81	10,3	0,04	2,59	10,36	0,04	3,35
	150	16,56	83,44	0	12,85	0,08	0	10,99	0,08	0
	∞	16,55	83,45	0	12,84	0,09	0	11,01	0,09	0
C4	5	16,591	66,717	16,692	2,33	0,003	5,11	5,78	0,003	11,85
	25	16,543	66,607	16,85	5,16	0,01	2,27	6,77	0,01	4,36
	75	16,59	83,41	0	7,41	0,034	0	7,4	0,036	0
	150	16,585	83,415	0	7,41	0,04	0	7,45	0,04	0
	∞	16,581	83,419	0	7,4	0,04	0	7,37	0,04	0
C5	5	13,63	69,59	16,81	2,98	0,61	6,16	6,78	0,61	9,61
	25	14,92	68,98	16,1	6,25	0,83	2,67	8,03	0,83	2,65
	75	15,1	68,1	16,8	8,42	0,87	0,47	8,41	0,88	1,46
	150	14,5	85,5	0	8,49	1,27	0	8,22	1,25	0
	∞	15	85	0	8,78	0,98	0	8,5	0,96	0
C6	5	19,84	63,468	16,692	3,53	0,008	7,07	6,9	0,008	13,47
	25	16,47	66,63	16,9	6,6	0,02	3,99	7,02	0,02	6,67
	75	16,1	65,6	18,3	7,97	0,05	2,59	7,74	0,05	2,65
	150	16,52	83,48	0	10,52	0,09	0	8,1	0,09	0
	∞	16,52	83,48	0	10,51	0,09	0	7,94	0,09	0
C7	5	16,24	66,96	16,8	2,97	0,01	8,41	8,18	0,01	11,05
	25	16,32	65,87	17,81	9,37	0,07	1,94	9,59	0,07	2,62
	75	16,4	66,6	17	9,8	0,13	1,46	9,75	0,13	1,73
	150	16,31	66,79	16,9	10,31	0,21	0,86	9,91	0,21	0,72
	∞	16,2	83,8	0	11,07	0,32	0	9,72	0,32	0
C8	5	15,27	67,41	17,32	2,24	0,02	4,87	5,34	0,02	9,13
	25	15,9	65,35	18,75	5,45	0,09	1,59	6,26	0,09	2,4
	75	16,24	83,76	0	6,95	0,18	0	6,54	0,18	0
	150	16,1	83,9	0	6,89	0,24	0	6,67	0,27	0
	∞	16	84	0	6,85	0,28	0	6,69	0,28	0
C9	5	16,654	66,677	16,669	3,09	0,001	7,63	8,37	0,001	9,95
	25	16,653	66,656	16,691	9,23	0,005	1,49	9,47	0,005	2,38
	75	16,652	66,648	16,7	9,71	0,006	1,01	9,67	0,006	1,09
	150	16,654	66,646	16,7	10,43	0,007	0,29	9,83	0,008	0,27
	∞	16,659	83,341	0	10,72	0,005	0	9,82	0,005	0
C10	5	16,62	66,69	16,69	3,34	0,002	5,06	5,81	0,002	9,33
	25	16,54	66,46	17	6,41	0,01	1,99	6,92	0,01	2,03
	75	16,55	66,25	17,2	7,19	0,01	1,21	7,08	0,014	1,18
	150	16,61	83,39	0	8,38	0,03	0	7,28	0,03	0
	∞	16,6	83,4	0	8,37	0,03	0	7,21	0,03	0

6 Impact of API Rate Limit on Reliability

Table 6.6: Results from the model and the experiments using GCP (50 runs)

Config	RL	Probability %			Model			Experiment		
		$P_{success}^t$	$P_{failure}^t$	$P_{ratelimit}^t$	SR	NRL_FR	RL_FR	SR	NRL_FR	RL_FR
C1	5	16,62	66,711	16,669	0,27	0	11,22	0,21	5E-5	14,88
	25	16,42	66,889	16,691	1,1	0,001	10,39	0,99	0,001	13
	75	16,51	66,79	16,7	2,96	0,01	8,52	2,76	0,01	10,92
	150	16,61	66,68	16,71	5,76	0,004	5,72	5,26	0,006	7,65
	∞	16,63	83,37	0	11,47	0,025	0	8,83	0,02	0
C2	5	16,62	66,711	16,669	0,15	0	10,67	0,14	0	13,93
	25	16,28	67,029	16,691	0,65	0,001	10,13	0,65	0,001	12,57
	75	16,378	66,922	16,7	1,89	0,015	8,91	1,83	0,015	11,04
	150	16,563	66,727	16,71	3,68	0,004	7,13	3,59	0,004	8,54
	∞	16,647	83,353	0	10,81	0,012	0	6,47	0,012	0
C3	5	16,62	66,711	16,669	0,24	0	10,79	0,21	2E-5	13,76
	25	16,473	66,836	16,691	1,14	0	9,89	1,01	0,001	12,31
	75	16,51	66,79	16,7	2,97	0,012	8,04	2,81	0,012	9,92
	150	16,61	66,68	16,71	5,71	0,005	5,31	5,23	0,005	6,56
	∞	16,63	83,37	0	11,01	0,02	0	8,52	0,02	0
C4	5	16,62	66,711	16,669	0,15	0	9,05	0,14	0	11,86
	25	16,34	66,969	16,691	0,68	0,001	8,52	0,66	0,001	10,79
	75	16,37	66,93	16,7	1,84	0,018	7,35	1,89	0,018	9,24
	150	16,586	66,704	16,71	3,55	0,002	5,65	3,63	0,002	6,4
	∞	16,655	83,345	0	9,2	0,006	0	5,81	0,007	0
C5	5	16,62	66,711	16,669	0,24	0	12,28	0,2	0	12,28
	25	16,367	66,942	16,691	0,96	8E-4	11,56	0,97	8E-4	11,56
	75	16,45	66,85	16,7	2,45	0,012	10,06	2,69	0,012	10,06
	150	16,583	66,707	16,71	4,58	0,002	7,94	5,15	0,002	7,94
	∞	16,656	83,344	0	12,52	0,008	0	8,94	0,008	0
C6	5	16,62	66,711	16,669	0,16	0	11,2	0,14	E-4	15,23
	25	16,3	67,009	16,691	0,73	9E-4	10,62	0,66	8E-4	14,21
	75	16,39	66,91	16,7	1,88	0,012	9,46	1,88	0,013	12,22
	150	16,56	66,73	16,71	3,56	0,002	7,79	3,53	0,002	9,83
	∞	16,662	83,338	0	11,35	0,003	0	6,67	0,003	0
C7	5	16,62	66,711	16,669	0,30	0	10,38	0,20	0	13,36
	25	16,473	66,836	16,691	1,23	6E-4	9,45	1	6E-4	12,09
	75	16,49	66,81	16,7	2,71	0,013	7,96	2,76	0,012	9,51
	150	16,61	66,68	16,71	4,94	0,002	5,73	5,29	0,003	5,75
	∞	16,646	83,354	0	10,67	0,01	0	8,96	0,01	0
C8	5	16,62	66,711	16,669	0,15	0	9,82	0,14	0	9,97
	25	16,31	66,999	16,691	0,69	0,001	9,28	0,66	0,001	10,79
	75	16,37	66,93	16,7	1,87	0,017	8,08	1,89	0,018	9,24
	150	16,581	66,709	16,71	3,64	0,002	6,32	3,63	0,002	6,4
	∞	16,655	83,345	0	9,96	0,006	0	5,81	0,007	0
C9	5	16,62	66,711	16,669	0,25	0	9,66	0,21	0	13,2
	25	16,436	66,873	16,691	0,99	9E-4	8,93	1	9E-4	12,66
	75	16,39	66,91	16,7	1,73	0,01	8,17	2,72	0,01	10,38
	150	16,525	66,765	16,71	2,65	0,003	7,26	5,01	0,003	7,08
	∞	16,65	83,35	0	9,91	0,009	0	9,42	0,009	0
C10	5	16,62	66,711	16,669	0,15	0	11,12	0,14	0	15,65
	25	16,345	66,964	16,691	0,66	0	10,6	0,66	0,001	14,84
	75	16,41	66,89	16,7	1,80	0,009	9,46	1,85	0,009	12,94
	150	16,556	66,734	16,71	3,47	0,002	7,79	3,60	0,003	10,68
	∞	16,85	83,42	0	11,21	0,05	0	6,71	0,06	0

6.7 Conclusion and future work

Table 6.7: Calculated MAPE for all configurations up to 50 runs

Runs	Private Cloud			GCP		
	SR%	NRL_FR%	RL_FR%	SR%	NRL_FR%	RL_FR%
10	21,92	822,73	59,66	17,01	61,69	16,41
20	21,18	94,51	23,45	17,09	16,49	16,41
30	16,85	8,31	16,33	16,38	16,58	17,07
40	20,04	24	17,65	16,29	14,05	16,90
50	17,7	2	16,89	16,73	3,69	16,24

7 Evaluation of API Request Bundling and its Impact on Performance of Microservice Architectures

The widespread adoption of Microservice architectures has posed many challenges regarding API design for these architectures. Several API best practices and patterns have been proposed that could help API designers ensure API quality attributes such as reliability, availability, and performance. API Request Bundling, which is the focus of this chapter, is one of those patterns that aims at optimizing performance. The pattern promises substantial performance gains but can also lead to significant drawbacks such as increased development effort and application complexity. So far, there is little to no rigorously acquired knowledge to judge whether applying Request Bundling is worth the costs in a given Microservice architecture. To improve this situation, we performed an empirical study based on a Microservice-based open source business application using realistic workload scenarios. We used a service mesh-based infrastructure as it provides support for fast and dynamic deployment of different versions of the same microservice. This tremendously reduced the time and efforts of experimenting different configurations of the *Request Bundle* pattern. To estimate the performance impact of Request Bundling, we derived a regression model and performed a multivariate regression analysis. These selected regression models can help distributed system engineers and architects estimate the gain in performance in terms of round-trip time, with or without Request Bundling. Our approach, followed in the chapter, can be customized to other Microservice architectures or to study other performance-related Microservice patterns.

7.1 Introduction

In this chapter, we study *Request Bundle*, which is not as prominent as *Rate Limit*, but it has the particularity that it needs to be evaluated very early in any software project development. However, the evaluation has to be realized at runtime, which is the main challenge that we try to address to answer the first part of **RQ2**.

Numerous quality aspects of Microservices APIs have been identified in the context of critical application design decisions to be made by software architects [ZSZ⁺18]. The *Request Bundle* pattern [ZSL⁺21], described as part of the Microservice API (MAP) patterns¹, is a commonly used Microservice API pattern that can be used to optimize Microservice API performance-related qualities. In particular, it can be used to reduce communication overhead, such as throughput, latency, and bandwidth use between API clients and backends. Instead of sending many messages, API clients can bundle all these messages and send them in fewer chunks, which can considerably reduce network use and overall roundtrip times of messages. As a downside, some complexity and latency may be introduced on both the client and server-side due to the larger amount of data to be processed at once. Usually, the number of messages is decreased, and the size increases. Support for request bundling requires design and implementation efforts both on the client and server sides. It also increases the complexity of the messages and the message processing architecture.

To determine the real impact of request bundling on performance and whether the aforementioned complexity and efforts are justifiable and worth the costs, we have conducted an empirical study based on a Microservice-based open-source business application using realistic workload scenarios. Firstly, we aim to establish a better understanding of the *Request Bundle* pattern through its empirical evaluation using a service mesh-based infrastructure. Secondly, we try to quantify the possible performance gains using such a pattern and provide a model to estimate it in applications and workloads similar to the tested open-source application. Thirdly, we provide a method of studying such optimization techniques, which can be easily adapted to other patterns or to study the *Request Bundle* pattern in the context of substantially different applications. We aim to answer the following research questions:

¹See: <https://microservice-api-patterns.org/>

RQ2.3 What is the performance impact of the *Request Bundle* pattern in a realistic application and workload setting?

RQ2.4 a) How can we evaluate the *Request Bundle* pattern empirically, and how can we measure its performance impacts? b) What method(s) can be used to study such optimization techniques (or related patterns) for Microservices API communication?

In our empirical study, we have gathered data from a representative and modern Microservice-based setup. We have deployed the Lakeside Mutual application², an open-source system implemented based on experiences with real-life applications, on the Istio³ service mesh. The installation is hosted on a private Cloud. We have studied different scenarios in this application with and without request bundling and with different message bundle sizes. In total, we ran 30 tests on a private Cloud with a total running time of more than 15 hours (including setup times) and a total number of 8160 messages per repeated experiment run (at least 5 times). Based on the data from this empirical experiment, we created four regression models that can be used to estimate the total round trip time from the API client to the backend to evaluate the response time or performance of an API request with or without request bundling.

The chapter is organized as follows. We present some background about request bundling in Section 7.3. In Section 7.4, we provide details about our proposed regression model. We evaluate this model empirically in Section 7.5. Section 7.6 presents an analysis of the data collected and the regression models selected. Finally, threats of validity are discussed in Section 7.7, and we conclude in Section 7.8.

7.2 Related Work

There are two types of request bundling. The first one consists of sending one single bundled request and receiving one single bundled response. The second one sends one single bundled request but receives multiple responses instead. Further, a *Request Bundle* might consist of non-dependent or dependent batch requests [Sin18]. This chapter focuses on the first type of request bundling using non-dependent batch requests.

²c.f. <https://github.com/Microservice-API-Patterns/LakesideMutual>

³<https://istio.io/v1.4/>

7 Evaluation of API Request Bundling and its Impact on Performance

Request bundling is a well-known technique already implemented widely for optimization in Web browsers [Sch15, AP20, Gys19]. Also, it has been introduced as a way to optimize energy consumption in mobile applications by detecting and bundling multiple HTTP requests into one [LLGH16]. Some Cloud providers (e.g. [bra21]) propose request bundling as a paid feature for reaching better performance and lowering costs. However, there are no indications or guidelines on how and when to use this technique. In addition, although some studies have praised the overall performance gains achievable with request bundling [Sin18], there is no concrete information about what performance improvements to expect in a given situation.

Many API-related empirical studies have exhibited a set of recommendations to improve API performance. Wittern et al. [WCD⁺19] have studied GraphQL⁴ interfaces in practice as a query language that can improve API performance by using fewer client-server roundtrips and reduced response message sizes. An OpenAPI⁵-based framework has been proposed by Bucaille et al. [BCIEDC20] to test non-functional properties of REST APIs such as performance and availability. Park et al. [PJJC14] presented a REST-MapReduce framework specific to mobile Cloud computing to increase the performance of both REST OpenAPI service and MapReduce. However, those studies are technology-dependent and environment-specific. There is no platform-agnostic model proposed, as is the case in our study, by using the well-known and widely-used *Request Bundle* pattern as the object of the research.

7.3 Background: Request Bundling

Request Bundle advises defining a bundle of messages as a container that assembles multiple individual requests in a single request message. The bundle message usually contains metadata such as the number and identifiers of individual messages. *Request Bundle* can be found in many public and private APIs (see, e.g., the known uses in [ZSL⁺21]). The basis of this optimization is that in many cases, clients are able to “foresee” that requests can be bundled automatically; that is, in those cases, if supported by the Microservice API, the optimization can be applied automatically by client-side tools without human intervention. For example, consider users on the client side visiting

⁴<https://graphql.org>

⁵<https://swagger.io/specification/>

linked data pages, and a message is needed to retrieve each page. The client application could profile user behavior and prefetch the typical next pages visited by the user with a *Request Bundle*.

To make the decision for or against using a *Request Bundle*, regardless of whether it is human or automated decision making, it needs to be understood whether the effort is worth the costs and if, in the specific design situation (for human decision making) or runtime decision (if decision making is automated), a substantial performance gain can be achieved. In particular, the additional effort needed to implement a more complex API endpoint and corresponding clients needs to be considered as the costs of the use of the pattern. Reducing the number of messages does not imply that less information is exchanged. Thus, the remaining messages need to carry more complex payloads. This chapter deals with the problem that, so far, empirical data and models that can help to make such decisions based on solid and empirical ground are missing.

7.4 Regression model

7.4.1 Independent and dependent variables

We define the dependent variable *total_time* as the total round-trip time taken to process an API client request:

$$total_time = network_time + backend_time \quad (7.1)$$

Where *network_time* is the round-trip time spent between the API client and the API Gateway, and *backend_time* is the round-trip time spent from the API Gateway to the database or backend service (in case no database call is made).

On the backend, we define a number of variables to characterize the operations executed in the backend (see Table 7.1). Those variables should enable our model to estimate the time spent during backend computations.

Then, we have:

$$\begin{aligned} backend_time &= inmem_time + db_time + dist_time \\ &= inmem_weight \cdot inmem_calls \\ &\quad + db_weight \cdot db_calls + dist_weight \cdot dist_calls \end{aligned} \quad (7.2)$$

7 Evaluation of API Request Bundling and its Impact on Performance

Here, *inmem_weight* is the weight factor corresponding to the number of in-memory operations *inmem_calls*, *db_weight* is the weight factor corresponding to the number of database operations *db_calls*, and *dist_weight* is the weight factor corresponding to the number of distributed operations *dist_calls*. We introduce the weights to ease the application of our model and enable adjusting our model to different applications and environments. That is, the effort in applying the model can be eased because by using the weights, it is possible to simply count the number of operations to generate a new estimate using our model if acceptable weight values have been measured before.

In summary, to make an estimation with our model, a model user needs to count the number of calls of the different types in the operations that are candidates for request bundling. In addition, to adjust our model, the number of calls of each type should be measured to obtain mean performance measurements to determine the weights. Both are rather straightforward (e.g., with a simple counting script and simple source code instrumentation).

Using equations (7.1) and (7.2), we get:

$$\begin{aligned} total_time &= network_time + inmem_time + db_time \\ &\quad + dist_time \\ &= network_time + inmem_weight \cdot inmem_calls \\ &\quad + db_weight \cdot db_calls + dist_weight \cdot dist_calls \end{aligned} \tag{7.3}$$

Finally, we introduce two additional categorical variables, which describe if request bundling is used or not and which kind of RESTful operation type that is executed, respectively: Get, Update, or Create (see Table 7.1). Please note that Delete could be added here but is not included, as the Lakeside Mutual application does not use it in our scenarios.

7.5 Empirical study

In this section, we describe our empirical study. First, we define the scope of the study. Second, we give an overview of the open source application or *prototype* we have selected for our study, and that executes the *tasks* or functions that we measure in our experiment. Then, we provide details about the used hardware and software configuration of our

Table 7.1: Definition of Parameters

Independent variable	Description
<i>inmem_time</i>	Time spent in milliseconds by in-memory operations when executing a function.
<i>inmem_calls</i>	Number of in-memory operations when executing a function.
<i>db_time</i>	Time spent in milliseconds by database operations when executing a function.
<i>db_calls</i>	Number of database operations when executing a function.
<i>dist_time</i>	Time spent in milliseconds by distributed calls when executing a function.
<i>dist_calls</i>	Number of distributed calls when executing a function.
<i>request_bundle</i>	Categorical variable indicating whether request bundling is used or not.
<i>method</i>	Categorical variable indicating which function is executed.
Dependent variable	Description
<i>total_time</i>	Total round trip time in milliseconds between the API client and backend.

Private Cloud. After that, we describe the developments and deployments realized as well as the *measurement tools*. Finally, we describe how to launch the experiment or the *workload*. Finally, we discuss our experiment design and hypotheses. We have followed the guidelines proposed in [RMR⁺18, Woh12].

The study’s data set and further analyses are provided as an open data set for enabling reproducibility of our results⁶.

7.5.1 Scope definition

The goal of the empirical study is to analyze API request bundling usage for the purpose of predicting its impact on the performance of a Microservice-based application from the perspective of API clients. It is a *multi-test within object study quasi-experiment* where one single *object* is evaluated using a set of *subjects*. The *context* involves as *subjects* the sizes of the requests, which were preselected, the *methods* used, and a flag indicating whether request bundling is used or not. Details are provided in Section 7.5.6. The *object* is a close to real-life microservices-based application, available online, and developed with the goal to mimic professional microservice/API design (details provided in Section 7.5.2).

⁶<https://doi.org/10.5281/zenodo.5107982>

7 Evaluation of API Request Bundling and its Impact on Performance

We have deployed this application on a modern Private Cloud (details are described in Sections [7.5.3](#), [7.5.4](#) and [7.5.5](#)).

7.5.2 Lakeside Mutual application

Lakeside Mutual⁷ is a Java-based open-source system implemented based on experience with real-life applications. It implements an insurance company services to customers and employees. It is composed of four frontend microservices and four backend microservices, and the latter are the focus of our study. We are particularly interested in three functions that are provided by these microservices:

Get customers returns a list of customers using their IDs. It can be used by both customers and employees. However, a customer can only get information using the *customer-self-service-backend* microservice. On the other hand, an employee can get information about all customers using the *customer-management-backend* microservice. Both call the *customer-core* microservice that takes care of the fetch database operations. This function can be invoked with or without using request bundling.

Update customers' addresses changes the addresses of a list of customers. Again, it can be used by both customers and employees under the same conditions listed for the first function. This function did not support request bundling in the original application. Request bundling support for it has been implemented by us.

Create insurance quota requests creates a list of insurance quota requests for a given customer. It can be realized by customers using the *customer-self-service-backend* microservice, which then forwards the request to the *policy-management-backend* microservice. Request bundling for this function had to be implemented by us.

7.5.3 Configuration details

We have executed the experiments on a Private Cloud that consists of 3 Ubuntu⁸ 18.04.5 LTS Virtual Machines (VMs) on top of vSphere⁹ 6.7 environment. Each of these VMs

⁷<https://github.com/Microservice-API-Patterns/LakesideMutual>

⁸<https://releases.ubuntu.com/18.04/>

⁹<https://www.vmware.com/products/vsphere.html>

hosts a Minikube instance version 1.14.2 with 8 dedicated CPU cores Intel Xeon(R) CPU E5-2650 v4 @ 2.20GHz and 20 GB of system dedicated memory. Each Minikube instance hosts a Kubernetes engine version 1.14.2 and Istio version 1.4.3.

The central Minikube instance contains the central *Control Plane*, Kong Ingress Controller^[10] version 0.8.0 and the *customer-self-service-backend* microservice built using Java version 8. The 3 remaining microservices are also built using the same version of Java and accessed using the aforementioned *Ingress Controller* through YAML-defined Kubernetes Endpoints^[11] and Ingress Rules^[12]. On the client side, one Ubuntu 18.04.5 LTS virtual desktop is used to launch HTTP requests into the Private Cloud. The virtual desktop has 2 CPU cores Intel Xeon(R) CPU E5-2650 0 @ 2.00GHz with 8 GB of system memory.

7.5.4 Development and deployment details

To test request bundling in the three functions described in Section 7.5.2, we have implemented two additional methods in the microservices source code, which are the following:

`changeAddresses()`, which takes as input a comma-separated list of IDs and addresses, is added to both *customer-management-backend* and *customer-core* microservices API specification. This method updates those customers' data with the new provided addresses.

`createInsuranceQuoteRequests()`, which takes as input a comma-separated list of IDs and other information about the insurance is added to the *customer-selfservice-backend* microservice API specification. This method creates insurance quota requests for the customers.

Also, the source code is instrumented to compute the independent variables defined in our regression model described in Section 7.4, which are *db_time*, *inmem_time* and *dist_time*, which required only trivial instrumentation.

¹⁰<https://github.com/Kong/kubernetes-ingress-controller>

¹¹<https://kubernetes.io/docs/concepts/services-networking/service/>

¹²<https://docs.konghq.com/kubernetes-ingress-controller/1.1.x/guides/getting-started/>

Each of the microservices is running as a containerized Docker^[13] image deployed on a multi-clustered Istio service mesh with one single shared control plane^[14]. With each of these microservices running on a separate VM, the communication between them is established using *Edge proxies* or *Sidecars* (see Chapter 4) over a private network. An *API gateway* Ric20 or *Front proxy* is responsible for intercepting incoming or ingress communication from API clients. For that purpose, the Kong API Gateway is used by integrating it to Istio KC20.

7.5.5 Launching the experiment

Each experiment consists of executing a Shell^[15] script on the virtual desktop that executes cURL^[16] HTTP requests on the Private Cloud API Gateway and does the following:

- Create a list of preauthorized customers;
- Get customers data with and without request bundle;
- Update customers' addresses with and without request bundle;
- Create insurance quota requests with and without request bundle;

The dependent variable *total_time*, defined in our regression model described in Section 7.4, is recorded in the output file for each of the above operations. Also, *backend_time* is calculated using the HTTP headers *X-Kong-Upstream-Latency* and *X-Kong-Proxy-Latency* provided by Kong API Gateway^[17]. On the server side, the independent variables *db_time*, *inmem_time*, and *dist_time* are collected and computed using the output log file of each microservice.

7.5.6 Experiment design and hypotheses

In our study, we aim to measure the effect of using request bundling on API performance or response time. The size of the request bundle is chosen in a predefined interval setting. We have limited the maximum request bundle size to 50 to be realistic. Our experiment

¹³<https://hub.docker.com>

¹⁴<https://archive.istio.io/v1.4/docs/setup/install/multicluster/shared-vpn/>

¹⁵<https://www.shellscript.sh>

¹⁶<https://curl.haxx.se/docs/manpage.html>

¹⁷<https://docs.konghq.com/gateway-oss/0.10.x/proxy/>

follows the *one factor with two treatments balanced* design since, in each trial, we either use request bundling or not on all the functions described in 7.5.2.

Hence, we define the following experiment hypotheses:

The *null hypothesis* H_0 states that request bundling has negative or no effect on API performance.

The *alternative hypothesis* H_1 states the opposite.

Those hypotheses were checked for each combination of using request bundle (Yes/No), tested function (Get/Update/Create) and request bundle size (5 intervals), which totals 30 tests.

7.6 Data collection and analysis

7.6.1 Multivariate regression analysis

As described in Section 7.5.5, the variables *total_time*, *inmem_time*, *db_time*, and *dist_time* data are directly collected from output log files on client and server sides. The remaining variables *inmem_calls*, *db_calls*, and *dist_calls* are directly counted in the source code. Table 7.2 shows the data collected for experiments using request sizes of 10, 20, 30, 40, and 50.

As a result of our analysis using R language¹⁸, we obtained the four regression models described in Tables 10.2. Notice that the categorical variable *method(Create)* is not shown since it was selected as the reference category, so its coefficient equals zero [Sta18]. The same applies to the categorical variable *request_bundle(N)*, in case no request bundling is used.

As Table 7.4 shows, all the models have very low (F-Statistic) p-values and very high Adjusted R-squared values, which clearly hints that they have a very high *statistical significance*. However, we cannot give a definite conclusion without analyzing the residuals.

To verify the applicability of the linear models described in Table 10.2, we first have to check that the residuals follow a normal distribution. Figures 1 to 4 (provided in our open data set¹⁹) show the Histograms of Residuals of all those models. Except for Model

¹⁸<https://www.r-project.org>

¹⁹<https://doi.org/10.5281/zenodo.5107982>

7 Evaluation of API Request Bundling and its Impact on Performance

Table 7.2: Data collected during the experiments

Request bundle?	Function	Size	Inmem_calls	Inmem_time	Db_calls	Db_time	Dist_calls	Dist_time	Total_time
No	Get	50	2600	55,07355	50	46,597509	50	929,328941	1673
		40	2080	45,576909	40	38,590982	40	1112,832109	1949
		30	1560	40,305603	30	38,190849	30	772,503548	1432
		20	1040	19,344074	20	15,071418	20	380,584508	819
		10	520	11,842218	10	11,648897	10	301,508885	511
	Update	50	1500	15,338561	100	97,869806	50	828,791633	1794
		40	1200	13,156796	80	90,622794	40	722,22041	1710
		30	900	9,710725	60	74,760225	30	581,52905	1330
		20	600	5,964218	40	33,33821	20	343,697572	701
		10	300	3,493898	20	22,846122	10	168,65998	332
	Create	50	1600	491,892331	100	95,342467	50	1721,765202	3088
		40	1280	437,888573	80	82,19147	40	1485,919957	2833
		30	960	327,063494	60	60,522121	30	1143,414385	1999
		20	640	252,230347	40	37,559331	20	856,210322	1422
		10	320	118,169791	20	23,961	10	364,869209	678
Yes	Get	50	241	5,02067	50	17,598524	1	46,380806	97
		40	201	8,473363	40	20,126167	1	43,40047	112
		30	161	7,883633	30	17,62739	1	49,488977	92
		20	121	2,044953	20	6,320683	1	37,634364	70
		10	81	1,563875	10	7,096915	1	23,33921	55
	Update	50	937	14,533892	100	75,450795	1	63,015313	302
		40	757	16,697279	80	84,001458	1	28,301263	330
		30	577	10,811141	60	64,232514	1	17,956345	187
		20	397	5,966065	40	29,861325	1	18,17261	115
		10	217	3,649556	20	15,518327	1	12,832117	70
	Create	50	3488	46,408423	100	120,666903	50	1660,924674	1945
		40	2798	41,364438	80	81,855359	40	1291,780203	1945
		30	2108	36,585093	60	68,053868	30	1414,361039	1600
		20	1418	25,210233	40	38,04053	20	1129,749237	1244
		10	728	19,248005	20	25,998748	10	355,753247	440

Table 7.3: Models' description

Model	Intercept	Inmem_time	inmem_calls	db_time	db_calls	dist_time	dist_calls	request_bundle(Y)	method (Get)	method (Update)
1	-54,585	1,506	X	4,074	X	1,192	X	-312,011	331,758	312,936
2	-94,019	1,795	-0,081	X	X	0,968	17,01	-76,891	226,742	330,915
3	165,215	1,695	-0,077	X	X	0,576	25,034	-75,963	X	X
4	602,925	X	-0,194	X	7,85	X	35,105	-296,56	-386,275	-637,184

Table 7.4: Models' summary

	Adjusted R-squared	F-Statistic: p-value	Jarque-Bera: p-value	Ljung-Box: p-value
Model 1	0.9815	<2.2e-16	0.8407	0.1378
Model 2	0.9835	<2.2e-16	0.7316	0.09669
Model 3	0.9767	<2.2e-16	0.2867	0.03966
Model 4	0.9247	6.338e-13	0.8405	0.1169

2 and 3, the rest of the histograms display approximately a bell curve shape, which suggests that the residuals of Models 1 and 4 are probably normally distributed. Model 2 and 3's Normal Q-Q Plots, in addition, show that most of the points lie straight on the line. So, there is no reason to believe that the residuals are not normally distributed.

Multi-Fit Studentized Residual diagrams are Studentized Residual diagrams with multiple variables as independent variables. Studentized Residuals can detect potential outliers in the models. They are reliable since they consider each observation a potential outlier, remove it, and refit the regression model. This process is repeated until all the observations are tested. As we can see in Figures 1 to 4, very few outliers are observable, with no obvious outliers in Models 2 and 3, and there is an overall clear constant variance exhibited by all the models.

To validate these observations, we have run Jarque-Bera tests on all models, and the results are shown in Table 7.4. The null hypothesis H_0 for these tests tells us that the residuals are normally distributed. With a significance level $\alpha = 0.05$, we fail to reject the null hypothesis for all the models, confirming that these models' residuals are normally distributed.

To check whether these residuals are independent, we use the Ljung-Box tests described in Table 7.4. The null hypothesis H_0 for these tests shows that the residuals are not autocorrelated. With a significance level $\alpha = 0.05$, we fail to reject the null hypothesis for all models except for Model 3, which is thus discarded from further analysis. Therefore, Models 1, 2, and 4 residuals are independent and are selected for *total_time* prediction calculations.

7.6.2 Selected models

Based on the analysis from the previous section, we have selected Models 1, 2, and 4 to produce our prediction data. Model 4 produces two negative values for the predicted

total_time, and so it is discarded from further analysis. Figures 5 to 10 (provided in our open data set²⁰) compare the models' values of predicted *total_time* and its actual measured values when using the different methods with and without using request bundle. A visual inspection confirms that the estimation for most methods with or without request bundling provide reasonably close values. Some notable specific observations are: It is observable that all models work best in case we do not use request bundling and for the *Create* method using request bundle. In the case of the *Get* method using request bundle, the models are best when we have at least 30 requests bundled. For the *Update* method using request bundle, Model 1 makes the best predictions for *total_time*.

To compare the two selected models, we have calculated their prediction errors using the Mean Absolute Percentage Error (MAPE) [TB17]. We found that models 1 and 2 have prediction errors that are equal to 16% and 21% respectively, which are below the target prediction error of up to 30% for Cloud-based architectures [MA02], and thus explainable with network infrastructure imperfections such as latency and unforeseen errors.

7.7 Validity evaluation

As shown in Table 7.4, the models proposed display a very high *statistical significance*, which provides some confirmation for the conclusions from our experiment. Also, in Section 7.6.1, we have accurately verified our *statistical test assumptions* concerning residuals' normality, and the *error rate* is very low since we have conducted the tests using a significance level of 0.05. The *instrumentation* used in the tests and described in Section 7.5 encourages a very high *reliability of the measures*. However, there might be a threat of validity regarding the *reliability of treatment implementation* since we have made some changes to the application as explained in Section 7.5.4. While those were rather small and following the same development methods used in the rest of Lakeside mutual application, an impact of those changes on the results cannot be fully excluded.

On the client side, the dependent variable *total_time* was measured using the Linux-provided *date* utility for calculating the elapsed time between the start and end of an API call. Similarly, on the server side, the independent variables *inmem_time*, *db_time*,

²⁰<https://doi.org/10.5281/zenodo.5107982>

and *dist_time* are measured using JAVA provided *time* utility integrated into the source code. There is no reason to believe that these measurements lack accuracy. The rest of the independent variables *inmem_calls*, *db_calls*, and *dist_calls* are manually counted, which was convenient considering the limited number of calls in the application we used. However, this is regarded as a threat to validity of other large applications. To alleviate this threat, we plan to develop a script that can generate the counting results automatically.

We studied only the Lakeside Mutual open-source application. While it is a realistic business application, realized with state-of-the-art technologies, it cannot be excluded that no results can be generalized to other applications. In particular, generalization to non-business applications might be problematic. In our study, we only considered three methods (although the most common) that might be a threat to validity, especially for other types of applications. Also, the *instrumentation* employed is very specific and technology-dependent. To generalize our approach, we plan to apply our models to other applications and technologies.

7.8 Conclusion and future work

In this chapter, we evaluated the impact of API request bundling on the performance of Cloud-based benchmark based on a realistic application.

To answer **RQ2.3**, we have evaluated the impact of request bundling by empirically calculating the total round-trip time using 30 different *workload* settings. As shown in Table 7.2 and Figures 5 to 10, it is clear that *total_time* decreases considerably when using request bundle and independently of the *workload* setting. As a result, we successfully reject the null hypothesis H_0 described in Section 7.5.6 and confirm that request bundling positively impacts performance independently of the *workload* setting.

Concerning **RQ2.4**, we presented a regression model that can be used to estimate the performance impact of request bundling. This can be easily elaborated by counting *inmem_calls*, *db_calls*, and *dist_calls*, given appropriate mean weights values measured before as described in Section 7.4. We have empirically validated the model using a realistic application and reported four regression models using the data collected. After data analysis, we selected two models that best fit. These models generate prediction

7 Evaluation of API Request Bundling and its Impact on Performance

errors that are substantially below the target prediction error of up to 30% for Cloud-based architectures. Further, it is important to note that a rough estimation is usually good enough for making architectural design decisions, as those are usually decided at an early project stage. In addition to those models, we have defined an approach that not only can be used for request bundling evaluation using a specific *workload* setting and application but can also be applied to other optimization techniques and patterns. Indeed, this approach is supported by a service mesh infrastructure that provides all the functionalities needed including fast deployment and monitoring.

In future work, we plan to validate our model using other *workload* settings, applications, and technologies. For that purpose, we aim at automating the *instrumentation* used for counting *inmem_calls*, *db_calls* and *dist_calls*, in order to be able to use it in larger applications. We also plan to integrate other parameters into our model, especially those related to network latency and error management.

In this chapter, we answered the first part of **RQ2** by evaluating the impact of *Request Bundle* on the performance of a microservices-based benchmark application. In the next chapters, we extend the study by combining *Request Bundle* with *Rate Limit* and *Load Balancing* and evaluate that combination with regard to performance and reliability.

8 Combining API Patterns in Microservice Architectures: Performance and Reliability Analysis

There are many challenges in maintaining the desired quality of service levels in modern microservice and cloud applications. Numerous techniques and patterns, such as API Rate Limit, Load Balancing, and Request Bundle, have been suggested for API services and clients to improve quality properties related to performance and reliability. However, no study has measured the impact of these techniques and their combinations in a specific configuration, especially using a large distributed system workload setting. This chapter experimentally studies the effects of combining the API Rate Limit, Load Balancing, and Request Bundle patterns based on a realistic, third-party microservice-based application deployed in a private cloud and on the Amazon Web Services cloud (AWS) using 130 different configurations. We have run each configuration 500 times in the private cloud, totaling more than 4500 hours of runtime, and 200 times on AWS, totaling more than 3900 hours of runtime. Experimenting with such complex and compound scenarios would not be possible without the support of the employed service mesh-based infrastructure, which provides the tools and functionalities to implement and monitor different combinations of those API patterns. We developed regression models from the collected data to predict the performance and reliability impacts of combining such techniques and patterns. We found that the models provide acceptable prediction errors below 30% on the private cloud and AWS. Further, we found that the models work best in highly reliable environments like AWS. In addition to the concrete analyses provided in our work, we propose a general and largely automated method that can be followed iteratively to evaluate similar techniques and patterns for their quality properties.

8.1 Introduction

In this chapter, we leverage the previous research findings regarding the impact of *Rate Limit* and *Request Bundle* on reliability and performance respectively. Now, we combine those API patterns with *Load Balancing* in a more complex experimentation setup. Then, we evaluate the impact of such a combination on both quality properties, which answers the last part of **RQ2**.

With the emergence of cloud and microservices-based architectures, many challenges and issues have arisen due to these architectures' high versatility and volatility. As an essential aspect of such architectures, API design plays a significant role in having seamless and reliable access from API clients to backends [Sur19]. Hence, it has become crucial to evaluate current techniques, best practices, and their combinations to achieve the desired quality of service properties. This chapter focuses on reliability and performance qualities and the tradeoffs between them. Improving one of these quality properties should not degrade the other. Unfortunately, objective support for choosing the right combination of API patterns to balance such quality properties in distributed systems is lacking so far. Also, no realistic and large-scale datasets exist that can be used as a basis for that support.

Many API patterns and techniques related to microservices APIs are recommended in the literature [ZSZ⁺18, Hig20, Hig21] to achieve desired quality properties such as performance and reliability. Three of the most common API patterns in this context are:

Request Bundle as already described in Chapter 7 is a technique used on both client and server sides to reduce communication overhead caused by exchanging too many messages between both sides. This is usually achieved programmatically on both the API client and backend by adding up the data in those messages to form one block of data to be sent simultaneously. However, some complexity might be introduced since the client and server sides must handle large chunks of data [ZSL⁺21].

Rate Limit (see Chapter 6) can be enabled only on the server side to limit the number of requests from API clients at a specific time. This can be achieved by keeping track of the requests per API client by storing this information in a database. When the maximum limit of requests is reached, the API client is blocked

until the reset time. This pattern is usually realized using *Front* or *Edge Proxies* described in details in Chapter 4, depending on the target scope of the application. As a result, it can prevent excessive and abusive API clients' requests [Mad20]. However, it might also create bottlenecks and traffic congestion.

Load Balancing at the API level is usually realized using *API Gateways* [Ric18, HBTCP20]. It works by distributing the load over multiple node instances to avoid overloading or underloading them [MA19]. This is realized using *Load Balancing* algorithms like round-robin, random, and weighted algorithms that keep track of the API requests and their assigned node instances utilizing a database. However, this might also create bottlenecks and performance degradation, especially when the *Load Balancing* algorithm is not configured correctly.

Those are essential API patterns to study because they are commonly used techniques for optimizing the performance and scalability of APIs. By studying these patterns, developers can learn how to implement them effectively in their APIs and gain a deeper understanding of how they work and how they can be used to improve the performance and scalability of their APIs.

Studying these patterns in combination is particularly useful because they are often used in real-world API architectures and might influence each other's performance and scalability. By understanding how these patterns can be combined and used, developers can understand more comprehensively how to design and build performant and scalable APIs. Additionally, studying these patterns in combination can help developers understand the trade-offs and potential challenges in using them together. It can help them to make informed decisions about which patterns are most appropriate for their specific use cases.

Our study first presents a regression model we have developed to quantify the performance and reliability impacts of such combinations of patterns. Then, we experimentally evaluate the model using a large-scale setup and a realistic, third-party microservices-based application deployed on a service mesh-based infrastructure. Please note that besides providing concrete data and analyses for the studied patterns that can, e.g., be used during early architecting, our approach can also be used as a general and largely automated method that can be applied iteratively to study other combinations of optimization techniques and patterns.

We aim to answer the following research questions:

RQ2.5 What method(s) can be used to experimentally study and evaluate possible combinations of the *Rate Limit*, *Load Balancing*, and *Request Bundle* patterns concerning the performance and reliability impacts on microservices API?

RQ2.6 What are the performance and reliability impacts of possible combinations of those patterns in a realistic application and workload setting?

We have set up a modern cloud-based infrastructure to answer these questions using the Istio¹ service mesh. Istio provides several features natively, like *Rate Limit* and *Load Balancing* as described in Chapter 4, using its integrated Envoy² proxy. We used this infrastructure to measure widely-used industry practices. We have selected the Lakeside Mutual application³ as our benchmark application, which is a realistic, third-party application (i.e., the authors were not involved in its development) developed based on practitioner experiences using several API best practices.

The benchmark tests were realized based on nine different API operations of the benchmark application using or not using *Request Bundle*. Then the other two patterns were added in different combinations for each case. The tests were executed 500 times on a private cloud and 200 times on AWS⁴ for each configuration, equivalent to 130000 repetitions in total and more than 4500 hours of runtime using the private cloud and 52000 repetitions on AWS and more than 3900 hours of runtime, generating a huge data set.

We believe this is a realistic, large-scale experimental study (designed to fulfill the first methodological principle for reproducible performance evaluation in Cloud Computing [PVB⁺19]). Our benchmark is provided as open access⁵ and can be used to run similar tests for other API patterns and combinations of API patterns and reproduce our study.

Next, we developed regression models that could be used for guidance to architect similar types of microservice applications, as in our benchmark application, and present

¹<https://istio.io/>

²<https://www.envoyproxy.io>

³c.f. <https://github.com/Microservice-API-Patterns/LakesideMutual>

⁴<https://aws.amazon.com>

⁵<https://doi.org/10.5281/zenodo.7692007>

a complete and largely automated method to apply iteratively in different contexts and settings. We found that the models provide acceptable prediction errors on the private cloud and AWS since they are lower than 30% for Cloud-based architectures [MA02]. The models work best in highly reliable environments such as AWS.

The chapter is structured as follows. More detailed background about *API Load Balancing*, *Request Bundle*, and *Rate Limit* is provided in Section 8.3. Then we describe our approach in Section 8.4. After that, we present our regression model in Section 8.5. Section 8.6 provides details about our experiments setup and running the experiments. Then, the data generated are collected and analyzed in Section 8.7.1. Section 8.8 thoroughly discusses the results and threats to validity. Finally, we conclude in Section 8.9.

8.2 Related work

Many experimental studies have studied performance and reliability prediction models in software architecture. Wartika et al. [WGNA19] proposed applying a Bayesian algorithm to evaluate the performance of a software complexity prediction model based on three requirements: user, system, and software specifications. Another study [BKR09] performed a similar parametric approach by using the Palladio component model and a simulation tool based on that model to realize performance predictions. In contrast to the previous studies, Wang et al. [WZ18] proposed an approach to defining a reliability prediction model based on a deep learning neural network. The authors also claim that their approach is experimentally proven to be more accurate than the other approaches. A more relevant experimental study [CGJ⁺22] suggested the methodology MIPaRT to evaluate the performance and reliability of microservice systems continuously. The authors use the Markov model to randomly generate valid and invalid requests to the backend to explicitly force failure and ex-vivo testing to generate raw data for performance-reliability analysis. None of these studies have focused on API patterns and their combination impact on performance and reliability as we did in our research.

Many studies have proposed solutions to improve load balancing efficiency in microservices architecture and their future directions [WWL⁺21]. Yipei et al. [NLL18] argue that existing load-balancing strategies lack optimization and suffer from network congestion and complexity. They propose a chain-oriented load-balancing algorithm

(COLBA) to solve these problems and claim it is more efficient than existing load-balancing strategies. Similarly, XiaoDong et al. [LJSZ18] propose a shortest queue-waiting-time load balancing algorithm (SQLB) that outperforms classical load balancing algorithms. In addition, the authors experimentally demonstrate that this algorithm generates a low prediction error. On the other hand, Rusek et al. [RL18] show that decentralized load-balancing algorithms are more efficient than centralized ones by analyzing their time complexity. Another study used scenario calculus to formally analyze load balancing impact on scalability and capability of microservices [ZWB18]. It demonstrates that elastic round-robin scheduling is more scalable than shortest-waiting queue scheduling. Li et al. [LYW⁺22] developed the DWLOAD (Dynamic Weight Loading) algorithm to maintain high efficiency and reliability of load balancing in a microservices environment. This was mainly realized by calculating and recording the real-time weights of the hosting servers. BLOC, a self-managing load balancing system, was also proposed [BW22] to provide a second layer control for service mesh load balancers in microservices architectures using other patterns like *Rate Limit*. Lastly, an online decision algorithm based on Lyapunov optimization techniques was proposed to distribute the load in a hybrid cloud cost-effectively when a flash crowd event occurs [NLL⁺15]. As a result, only a few studies have evaluated the impact of load balancing on reliability. Also, they have yet to consider combining it with other patterns such as *Request Bundle* either.

Although *Request Bundle* and *Rate Limit* API patterns have been studied experimentally in Chapters 6 and 7, their combinations with other API patterns and techniques have not been the subject of any experimental study in the scientific literature. This represents a significant gap in understanding the impact of widely used API practices such as *Rate Limit* and *Load Balancing* and their combinations. Also, there is no way to measure the impact of those API patterns and their combinations on some important quality properties of a specific configuration setting, such as performance and reliability. This chapter closes this gap by presenting an experimentally validated regression model using a real-life microservice-based application and setting. This model could also support the existing simulators and prediction models.

8.3 Background about Load Balancing, Request Bundle and Rate Limit

Load Balancing is a technique to distribute the load evenly between two or more resources to avoid overloading or underloading them. This technique can be performed either on cloud resources level [KK19] or service level [XG20]. Our study mainly focuses on service level load balancing, a feature provided by service meshes like Istio (see Chapter 4). Indeed, Istio provides four methods to achieve load balancing. While the *round robin* method dispatches API requests to replicas using a particular order, the *random* method does it randomly. On the other hand, the *weighted* method explicitly assigns a percentage weight to each node instance. More interestingly, the *least requests* method gives the largest weight dynamically to the node instance with the least number of requests.

Request Bundle (see Chapter 7) generally consists of containerizing multiple messages into one containing additional information related to those messages, such as identifiers. Instead of sending many messages to the API backend, request bundling encapsulates those messages into one, avoiding unnecessary chatty communication. However, some complexity might be introduced on both the API client and server, which might also cause bottlenecks and traffic congestion. In our study, we limited the bundle size to 50 to reduce the likelihood of those issues. *Request Bundle* is a general practice already adopted in many APIs [ZSL⁺21]. Since those APIs and others without request bundling constantly evolve, measuring the impact of using such a practice and its combination with other API patterns regarding performance and reliability is crucial.

Rate Limit has also been the focus of many studies [ZSZ⁺18, SZZ⁺18] (see Chapter 6), but has also not been studied with combinations of other patterns and techniques. It is usually implemented using an API Gateway [Ric20] and ensures that API providers are not overwhelmed by an intentional or unintentional flood of requests from API clients. *Rate Limit* plays an essential role in maintaining the desired level of quality property related to performance and reliability, and combining it with a billing strategy or a *Rate Plan* [ZSL⁺20c] makes it even more effective. However, efficiency drastically decreases by including mandatory patterns like *API key* [ZSL⁺20c] to keep track of API clients. In our study, we use Istio, which supports two types of rate-limiting: global and local rate limiting. While the former is activated globally on the service mesh level, the latter

triggers each service instance of the mesh.

8.4 Approach overview

Figure 10.3 illustrates the patterns and combination scenarios we have considered in our study. Finding an optimal point or area in possible solutions where performance and reliability are maximized is not straightforward without additional information. So, we need first to identify parameters and metrics to estimate the localization of this point or area. Also, this optimal point or area might change in time due to unforeseen circumstances like resource saturation. Thus, we need to identify a model that accurately considers chosen and third-party parameters related to resource management. We have also defined metrics that could estimate the localization of the optimal point or area in the space of possible solutions.

Combining the three patterns and techniques we have considered in our study must be carefully decided. For example, in many situations, it will not make sense to use *Request Bundle* with *Rate Limit*, since otherwise, we would have large API requests that overload the backend before *Rate Limit* is triggered. Also, using *Request Bundle* with *Load Balancing* is a logical choice to not overwhelm one node on the backend with large API requests to handle. However, using *Request Bundle* along with *Rate Limit* might not work correctly since the backend would be already overloaded after only a few API requests. Then, it would be wiser to introduce *Load Balancing*, too, to avoid such a scenario. Also, combining patterns generates costs and complexity, which should be justified in advance by accurately measuring the foreseen improvements of those combinations on performance and reliability. There should be a way to find a balance between high performance and reliability in conjunction with low costs and complexity and, if required, find the best combination to handle a specific workload configuration, which is the aim of our model.

Our proposed regression model employs data extracted from API or API Gateway, API clients, and API backend or microservices. The data extraction is realized by considering each possible combination of the patterns in our study. The iterative process illustrated in Figure 10.3 makes the approach dynamic and adaptive, depending on the usage period and the cloud setting. That means the same regression model could dynamically extract

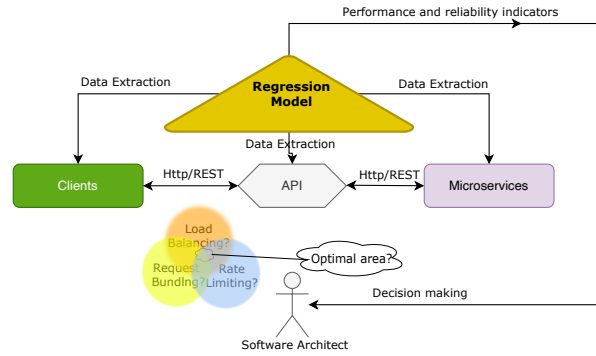


Figure 8.1: Approach overview

the performance and reliability indicators in a completely different cloud environment. Please note that these indicators would only represent a rough estimation to the software architect for decision-making. More accurate predictions, like the usage period, would need more parameters to consider in the regression model.

8.5 Regression model description

8.5.1 Independent and dependent variables

As previously stated in Section 8.3, our work is motivated by the high uncertainty in the scenarios covered in Figure 10.3 to localize the optimal point or area in the combination of the patterns to achieve optimal performance and reliability. In addition, the optimal selected point or area might change in time due to unforeseen circumstances like resource saturation. Hence, given a specific workload configuration, we first need to define a model that could help us measure the performance and reliability impact.

Our study has considered two types of independent variables: numerical and categorical. The first type of variables includes *rpm*, which is the number of requests per minute for a specific workload setting, *api_users* which is the number of API users that generate load for that particular workload setting, *rate_limit*, which is the maximum predefined number of requests allowed per API user per minute, *bundle_size* which indicates the size of the request bundle, in case it is used and *methods_count* which reflects the number of methods that are considered in the study. The second type of variables includes *request_bundle* and *load_balancing*, which indicate whether *Request Bundle*

and *Load Balancing* are used respectively. It also includes *method*, which indicates the API operation executed on the backend: Get, Update, Delete, or Create. (see Table 8.1).

The first dependent variable we have considered is *total_time*, equivalent to the total round trip time spent by API requests to be completely fulfilled. It is defined as:

$$total_time = network_time + backend_time$$

network_time is equivalent to the round-trip time spent by API requests from the API client and the API Gateway, and *backend_time* is the round-trip time spent from the API Gateway backend service(s) and maybe database(s). The prior study needed to include server details concerning program execution and database time. As these details are not required here, we employ *rpm* as an API client parameter that is more straightforward to calculate to define *total_time* as follows:

$$total_time = \sum_{i=1}^{api_users} \frac{bundle_size_i}{rpm_i}$$

where,

$$bundle_size = \begin{cases} 10 \dots 50, & \text{if } request_bundle = Yes. \\ 1, & \text{otherwise.} \end{cases}$$

The second dependent variable is *failure_rate*, which indicates these API requests' failure percentage. Here, we do not distinguish failure rate due to rate limiting and failure rate due to other reasons than rate limiting. We define:

$$failure_rate = \begin{cases} \sum_{i=1}^{methods_count} rpm_i \cdot \frac{load_weight_i}{load_balancers} - rate_limit_i, & \text{if } rpm_i \cdot \frac{load_weight_i}{load_balancers} > rate_limit_i \\ \wedge load_balancers \geq 1 \\ \sum^{n_t} \frac{P_{failure}^t}{api_users} \cdot failure_rate^t, & \text{otherwise.} \end{cases}$$

The load balancing algorithm determines *load_weight_i*. In our study, we exclusively used the *weighted load balancing* algorithm, and thus their values are fixed and known in advance along with *methods_count* and *load_balancers*. Therefore, these variables are not included in the following regression analysis. $P_{failure}^t$ and $failure_rate^t$ denote the probability of failure during each time interval t and the corresponding failure rate as proposed in Chapter 6. This is a Bernoulli distribution composed of n_t trials and is

Table 8.1: Definition of Parameters

Independent variable	Description
<i>rpm</i>	The number of requests per minute for API users.
<i>api_users</i>	The number of API users sending the requests.
<i>rate_limit</i>	The value of the applied rate limit ranging from 100 to 1200 (100 increments).
<i>request_bundle</i>	Categorical variable indicating whether request bundling is used or not.
<i>bundle_size</i>	The request bundle size ranges from 10 to 50 (10 increments).
<i>load_balancing</i>	Categorical variable indicating whether load balancing is used or not.
<i>load_balancers</i>	Number of load balancers.
<i>load_weight</i>	Percentage of requests that is forwarded to a specific backend instance.
<i>method</i>	Categorical variable indicating the method executed by API users.
<i>methods_count</i>	Number of methods executed by API users.
Dependent variables	Description
<i>total_time</i>	The total roundtrip time spent for requests between API client and backend.
<i>failure_rate</i>	The percentage failure rate of API requests.

not considered further in the current study, so these parameters are not included in the regression analysis.

8.6 Experimentation setup

8.6.1 Application under study

We have decided to choose Lakeside Mutual⁶ application, which is a real-life application developed to make in practice various microservice API patterns like *Request Bundle* [ZSL⁺21]. Indeed, we have primarily extended the application to support request bundling on the following functions (provided in the replication package⁷ and designed to fulfill the fourth methodological principle for reproducible performance evaluation in Cloud Computing [PVB⁺19]):

Update customers using their IDs by application customers or administrators. Get customers already supports request bundling.

⁶<https://github.com/Microservice-API-Patterns/LakesideMutual>

⁷<https://doi.org/10.5281/zenodo.7692007>

Create/Get/Delete insurance quota requests using their IDs, which application customers do.

Create/Get/Update/Delete insurance policies done by the application administrator using *policy-management-backend* microservice.

Request Bundle is enabled by implementing a new dedicated chain of methods on the backend supporting a list as an input parameter instead of one parameter at a time for each API operation described above. For example, *getCustomers()* method now gets a list of customers' IDs instead of one ID at a time. *Rate Limit* and *Load Balancing* are both supported by Istio through EnvoyFilter⁸ and virtual services⁹ respectively. The full implementation details are provided in the replication package.

We are executing nine different API functions on the backend with and without using *Request Bundle*. In our study, we could have used an existing benchmark. Other cloud benchmarks exist, like those proposed in the context of Cloudsim [CRB⁺11] and PerfCompass [DNWG14], but they are relatively low-level. They are defined at the network rather than the business application level, which is relevant for request bundling. As a consequence, there are no usable cloud application-level benchmarks. Even if some existed, they likely would not apply to our study, as we would need them to use request bundling. Instead, we defined realistic scenarios based on Lakeside Mutual's request bundling functions described above.

8.6.2 Infrastructure details

We have used a private cloud hosted in our University data center for our experiment. It is composed of 11 Ubuntu¹⁰ 18.04.5 LTS Virtual Machines (VMs) installed on vSphere¹¹ 6.7 environment. Each of these VMs runs a Minikube instance version 1.20.0 with eight dedicated CPU cores Intel Xeon(R) CPU E5-2650 v4 @ 2.20GHz and 20 GB of system dedicated memory. Each Minikube instance executes a Kubernetes¹² engine version 1.20.2 and Istio version 1.10.0.

⁸<https://istio.io/latest/docs/tasks/policy-enforcement/rate-limit/>

⁹<https://istio.io/latest/docs/concepts/traffic-management/>

¹⁰<https://releases.ubuntu.com/18.04/>

¹¹<https://www.vmware.com/products/vsphere.html>

¹²<https://kubernetes.io>

Table 8.2: AWS cost details (in US Dollars)

Service	Cost
EC2-Other	1783,53
EC2-Instances	1369,52
Elastic Container Service for Kubernetes	356,67
EC2-ELB	356,67
Tax	701,98
Total	4211,89

There are two Cluster environments that we have set up to differentiate API requests that are handled with and without load balancing. The first Cluster comprises two replicated central Minikube instances containing the central *Control Plane* and the *customer-self-service-backend* microservice built using Java version 8 each. Replication is also done for Minikube instances running *customer-management-backend* and *policy-management-backend*, which are built using the same version of Java and accessible using *Ingress Controller* YAML-defined Kubernetes Endpoints^[13] and Ingress Rules^[14]. The replicas share one Mongo^[15] (latest version) database instance. The second Cluster does not support load balancing, and thus there is no replication.

On the client side, two Ubuntu 18.04.5 LTS virtual desktops inject HTTP requests into the private cloud. Each virtual desktop has 2 CPU cores Intel Xeon(R) CPU E5-2650 0 @ 2.00GHz with 8 GB of system memory.

We have also run the same experiment on AWS using a four-node Amazon Elastic Kubernetes Services (EKS^[16]) cluster. The costs of using AWS are described in Table 8.2 (designed to fulfill the eighth methodological principle for reproducible performance evaluation in Cloud Computing [PVB⁺19] and not applicable for the private cloud).

8.6.3 Running the experiment

Each virtual desktop executes a Shell^[17] script with 130 different configurations of bundle sizes ranging from 10 to 50, rate limits ranging from 100 to 1200 set on server-side (∞

¹³<https://kubernetes.io/docs/concepts/services-networking/service/>

¹⁴<https://docs.konghq.com/kubernetes-ingress-controller/1.1.x/guides/getting-started/>

¹⁵<https://www.mongodb.com>

¹⁶<https://aws.amazon.com/eks/>

¹⁷<https://www.shellscript.sh>

if rate-limiting is disabled), and whether request bundling is used or not. The script executes the following depending on the configuration:

- Create a predefined number of API users;
- Get customers' data with or without request bundling;
- Update customers' addresses with or without request bundling;
- Create/get/delete insurance quota requests with or without request bundling;
- Create/get/update/delete policies with or without request bundling;

Those API requests or methods are executed using load balancing in only the first Cluster, as stated in Section 8.6.2. We have run the script 500 times for each configuration in the private cloud, totaling 130000 repetitions and more than 4500 hours of runtime, and 200 times on AWS, counting 52000 repetitions and more than 3900 hours of runtime (designed to fulfill the second methodological principle for reproducible performance evaluation in Cloud Computing [PVB⁺19]). *total_time* is calculated at the end of each trial, and *failure_rate* is calculated when switching from one configuration to another.

8.7 Multivariate regression analysis

8.7.1 Data collection and analysis

All the data are collected from output log files on the client side. Due to the large set of data collected (130 configurations tested), we have decided to include it in an online Appendix¹⁸.

We have realized data analysis using R language¹⁹. Tables 8.3 and 8.4 list the combined models that we generated from the data collection for *total_time* prediction on the private cloud and AWS, respectively. Tables 8.5 and 8.6 list the combined models that we generated from the data collection for *failure_rate* prediction on the private cloud and AWS, respectively. We have only collected data concerning the independent variables that we have experimentally varied as described in Section 8.5.1. Notice that

¹⁸<https://doi.org/10.5281/zenodo.7692007>

¹⁹<https://www.r-project.org>

8.7 Multivariate regression analysis

Table 8.3: Combined models' description on the private cloud: total time prediction

Model	Total time								
	Intercept	Load balancing	Request bundle	Rate limit	API users	RPM	Method delete	Method get	Method update
1	745,1	294,9	-79,37	0,0051	36,78	-0,12	-334,9	-1009	17,53
2	338,4	294,9	-79,73	0,0051	36,78	-0,12	X	X	X
3	297,6	295,2	X	0,0052	36,74	-0,12	X	X	X
4	489,7	X	-80,66	0,0048	36,85	-0,13	X	X	X
5	448,4	X	X	0,0049	36,81	-0,13	X	X	X

Table 8.4: Combined models' description on AWS: total time prediction

Model	Total time								
	Intercept	Load balancing	Request bundle	Rate limit	API users	RPM	Method delete	Method get	Method update
1	8504	620,5	-5872	0,19	310,4	-16,71	-662,4	-1852	-1349
2	7406	592,8	-5845	0,19	310,4	-16,72	X	X	X
3	297,6	295,2	X	0,0052	36,74	-0,12	X	X	X
4	489,7	X	-80,66	0,0048	36,85	-0,13	X	X	X
5	448,4	X	X	0,0049	36,81	-0,13	X	X	X

Table 8.5: Combined models' description on the private cloud: failure rate prediction

Model	Failure rate								
	Intercept	Load balancing	Request bundle	Rate limit	API users	RPM	Method delete	Method get	Method update
1	46,02	-11,63	-38,59	-0,0025	-0,088	0,0058	n/a	n/a	n/a
2	46,02	-11,63	-38,59	-0,0025	-0,088	0,0058	X	X	X
3	26,2	-11,46	X	-0,0024	-0,1	0,0062	X	X	X
4	40,05	X	-38,54	-0,0024	-0,091	0,0059	X	X	X
5	20,34	X	X	-0,0024	-0,11	0,0063	X	X	X

Table 8.6: Combined models' description on AWS: failure rate prediction

Model	Total time								
	Intercept	Load balancing	Request bundle	Rate limit	API users	RPM	Method delete	Method get	Method update
1	-1,76	0,81	3,13	0,00017	0,059	-0,0045	-0,07	n/a	n/a
2	-1,77	0,81	3,13	0,00017	0,059	-0,0045	X	X	X
3	-1,43	0,66	X	0,00024	0,041	-0,0004	X	X	X
4	-1,38	X	3,01	0,00017	0,059	-0,0045	X	X	X
5	-1,11	X	X	0,00024	0,041	-0,0043	X	X	X

Table 8.7: Combined models' summary: total time & failure rate prediction

#	Private Cloud				AWS			
	Total time		Failure rate		Total time		Failure rate	
	Adjusted	F-Statistic	Adjusted	F-Statistic	Adjusted	F-Statistic	Adjusted	F-Statistic
	R-squared	p-value	R-squared	p-value	R-squared	p-value	R-squared	p-value
1	0,2452	<2.2e-16	0,5925	<2.2e-16	0,6438	<2.2e-16	0,09809	<2.2e-16
2	0,1683	<2.2e-16	0,5925	<2.2e-16	0,6383	<2.2e-16	0,09806	<2.2e-16
3	0,1677	<2.2e-16	0,3641	<2.2e-16	0,6113	<2.2e-16	0,05358	<2.2e-16
4	0,1599	<2.2e-16	0,5717	<2.2e-16	0,6374	<2.2e-16	0,08846	<2.2e-16
5	0,1593	<2.2e-16	0,3439	<2.2e-16	0,6094	<2.2e-16	0,04708	<2.2e-16

the *method* variable is not significant using model 1 concerning *failure_rate* prediction but highly significant using the same model concerning *total_time* prediction. Also, the coefficient values relative to *load_balancing* and *request_bundling* reflect the case where they are enabled. In other words, when they are disabled, the values of those coefficients are zero, as described [Sta18]. The same applies to *method* create that gets zero value for its coefficient in model 1.

On the private cloud and as shown in Table [8.7], the F-statistic p-value is very low for all the models, and the Adjusted R-Squared is relatively high, especially for models 1, 2, and 4 regarding *failure_rate* prediction. The latter is even higher on AWS regarding *total_time* prediction but lower regarding *failure_rate* prediction. However, no conclusion can be made without analyzing the models' residuals. We used Anderson-Darling's and Henze- Zirkler's tests to realize residuals' univariate and multivariate normality. As clearly shown in the models (provided in the online Appendix²⁰), the histograms are not curvy-shaped, indicating that the residuals are not normally distributed. This is confirmed by the Chi-Square Q-Q plots and Scatter plots showing that the points are widely spread and do not form any obvious straight line. The perspective 3D plots deny residuals multivariate normality as well. Since the normality of residuals, one of the assumptions of MANOVA (Multivariate Analysis of Variance), is violated, we have decided to perform MANOVA Pillai's Trace test statistics for further analysis [Bob21].

As shown in Table [8.8], all the variables have very high significance since their p-values are very low. As a reminder, this significance increases as a variable's Pillai value gets

²⁰<https://doi.org/10.5281/zenodo.7692007>

8.8 Discussion and threats to validity

Table 8.8: Models summary (on the private cloud): Pillai's Trace test

Variable	Model 1		Model 2		Model 3		Model 4		Model 5	
	Pillai	p-value	Pillai	p-value	Pillai	p-value	Pillai	p-value	Pillai	p-value
<i>rpm</i>	0,15	<2.2e-16	0,15	<2.2e-16	0,14	<2.2e-16	0,16	<2.2e-16	0,15	<2.2e-16
<i>api_users</i>	0,08	<2.2e-16	0,07	<2.2e-16	0,07	<2.2e-16	0,07	<2.2e-16	0,07	<2.2e-16
<i>rate_limit</i>	0,1	<2.2e-16	0,1	<2.2e-16	0,07	<2.2e-16	0,1	<2.2e-16	0,07	<2.2e-16
<i>request_bundle</i>	0,32	<2.2e-16	0,32	<2.2e-16	n/a	n/a	0,32	<2.2e-16	n/a	n/a
<i>load_balancing</i>	0,04	<2.2e-16	0,04	<2.2e-16	0,03	<2.2e-16	n/a	n/a	n/a	n/a
<i>method</i>	0,07	<2.2e-16	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a

Table 8.9: Models summary (on AWS): Pillai's Trace test

Variable	Model 1		Model 2		Model 3		Model 4		Model 5	
	Pillai	p-value	Pillai	p-value	Pillai	p-value	Pillai	p-value	Pillai	p-value
<i>rpm</i>	0,16	<2.2e-16	0,16	<2.2e-16	0,54	<2.2e-16	0,15	<2.2e-16	0,54	<2.2e-16
<i>api_users</i>	0,34	<2.2e-16	0,33	<2.2e-16	0,38	<2.2e-16	0,33	<2.2e-16	0,38	<2.2e-16
<i>rate_limit</i>	0,02	<2.2e-16	0,02	<2.2e-16	0,02	<2.2e-16	0,02	<2.2e-16	0,02	<2.2e-16
<i>request_bundle</i>	0,11	<2.2e-16	0,11	<2.2e-16	n/a	n/a	0,11	<2.2e-16	n/a	n/a
<i>load_balancing</i>	0,01	<2.2e-16	0,01	<2.2e-16	0,01	<2.2e-16	n/a	n/a	n/a	n/a
<i>method</i>	0,01	<2.2e-16	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a

closer to 1. As a result, we notice that *request_bundle* is the best predictor in models 1, 2, and 4 on the private cloud. On the other hand, *load_balancing* has the lowest Pillai value in models 1, 2, and 3. Also, each specific variable's Pillai value remains stable in the models where the variable is used. Concerning AWS, Table 8.9 also shows that all the variables have very high significance. However, we see clearly that *api_users* and *rpm* are the best predictors in all the models, and the rest of the variables have lower significance.

8.8 Discussion and threats to validity

8.8.1 Models comparison

Let us compare the models we have found in detail, first for their performance and then for their reliability impacts. Figure 8.2²¹ shows that *total_time* in model 1 is very sensitive to *bundle_size*, thus a large performance degradation starts very early. In contrast,

²¹All the figures in this section were realized using the Plotly package provided by R language; see <https://plotly.com>

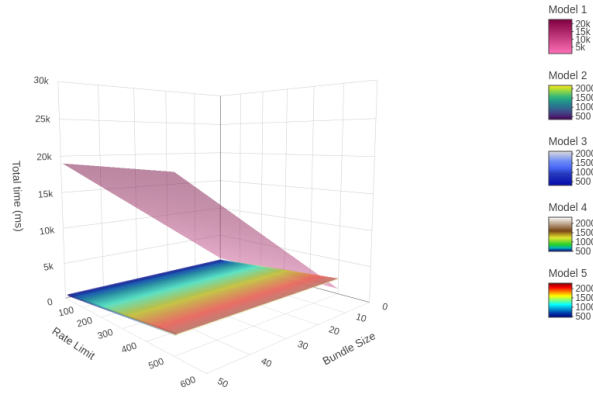


Figure 8.2: Performance analysis (on the private cloud)

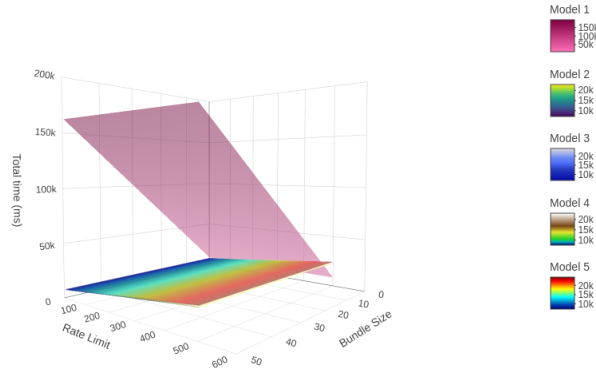


Figure 8.3: Performance analysis (on AWS)

rate_limit does not show any clear impact on performance. All the remaining models overlap, meaning they have similar behavior concerning performance. *bundle_size* does not play a role in models 2 and 4, but increasing *rate_limit* slightly decreases performance. Also, *load_balancing* has no apparent effect on performance. As a result, model 1 is only usable in architectures where *Request Bundle* is not the preferred option and high performance is required. For example, this is where API messages are already large and need a high amount of bandwidth. On the other hand, the remaining models are more suitable when API messages are small and should use *Request Bundle*. This reduces the impact of *Rate Limit* on performance consequently and keeps it manageable, as clearly shown in the figure. Figure [8.3](#), realized using AWS-generated data, confirms

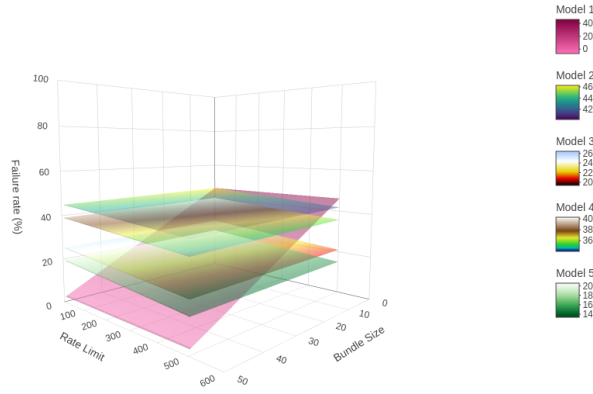


Figure 8.4: Reliability analysis (on the private cloud)

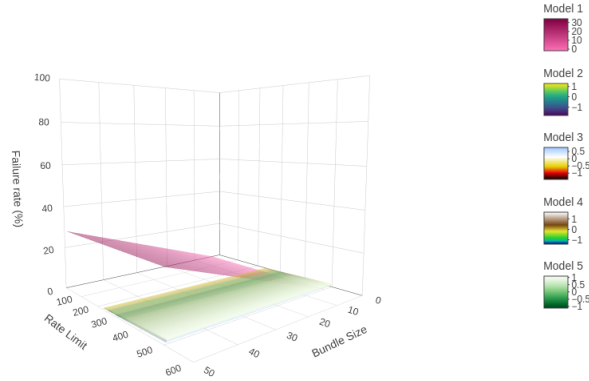


Figure 8.5: Reliability analysis (on AWS)

the conclusions above with a degraded performance overall due to latencies and timeouts usually occurring in public clouds.

In Figure 8.4, we can see that the models diverge but show a similar trend as in Figure 8.2. Model 1 is also susceptible here and witnesses a massive increase in reliability at the highest value of *bundle_size* for all values of *rate_limit*. The remaining models exhibit a gradual reliability increase, with the increase of *rate_limit*. In models 2 and 4, *bundle_size* does not play a role, too. Here, *load_balancing* impacts reliability, especially in model 2. Like the conclusions drawn regarding performance analysis, model 1 is suitable for architecting systems that exchange large API messages where *bundle_size* is less than or equal to 10. Also, the remaining models can be used by systems exchanging

smaller API messages using (or not using) *Request Bundle*. However, at this time, the impact of *Rate Limit* on reliability is influenced by *load_balancing* and illustrated specifically by models 2 and 3 in the figure.

Figure 8.5, realized using AWS-generated data, shows a different trend for the models. The increase in *failure_rate* when increasing bundle size is mainly due to the lack of scalability since we have only used one AWS cluster as described in Section 8.6.2. For models 2 to 5, *failure_rate* is higher in Figure 8.4 due to lower latency, which increases *failure_rate* due to *Rate Limit*. For model 1, as the bundle size increases, more requests get bundled, which produces fewer *rpm* and thus less *failure_rate* due to *Rate Limit*.

8.8.2 Tradeoff analysis

We have investigated different combinations of API patterns like *Circuit Breaker* and *Rate Limit*. We have not considered such combinations since they did not show any interesting dependencies from which design advice can be derived. For instance, *Circuit Breaker* may directly hinder the real impact of *Rate Limit* on reliability. In contrast, the selected combinations have revealed a clear understanding of their impact on reliability and performance. We perform an API patterns tradeoff analysis from the experimental data to validate and complement the models' analysis presented above. We analyze tradeoffs of using *Load Balancing* with firstly *Rate Limit* and secondly *Request Bundle* in terms of performance and reliability impact. Thus we also study tradeoffs between the two investigated quality properties.

Load Balancing vs. Request Bundle. As seen in Figure 8.6, *Load Balancing* has a clear negative impact on performance, especially when *bundle_size* gets higher starting from approximately 30 requests. That is because processing large chunks of data by only one load balancer quickly creates a long waiting queue, which could be resolved by adding more load balancers. This trend is partially confirmed by Figure 8.7 realized using AWS-generated data, with a significant degradation in performance explainable by latency and timeouts that usually occur on public clouds. However, as *bundle_size* gets higher, *Load Balancing* improves performance considerably, unlike in the private cloud. Also, both patterns do not impact reliability, as depicted in Figure 8.8. Models 1 and 2 include *load_balancing* and *request_bundle*, but only Model 2 is recommended in case of *Request Bundle* usage, as stated in the previous section.

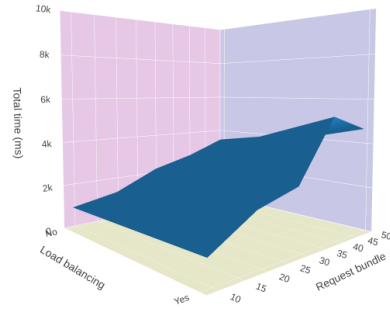


Figure 8.6: Load Balancing vs. Request Bundle: performance perspective (on the private cloud)

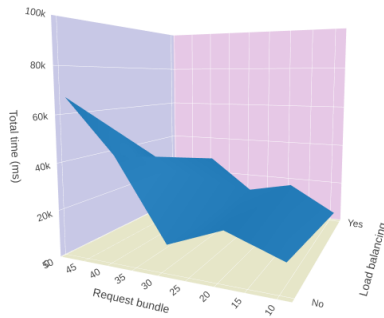


Figure 8.7: Load Balancing vs. Request Bundle: performance perspective (on AWS)

Load Balancing vs. Rate Limit. On AWS, the failure rate remains very low, mainly due to the high reliability provided by public clouds and latency that prevents *Rate Limit* activation. On the other hand, Figure 8.9 shows that *Load Balancing* reduces the failure rate on private cloud, especially when the *rate_limit* value is very high; though, this is hardly noticeable when its value is between 200 and 600 requests per minute. Otherwise, as mentioned earlier, *Load Balancing* reduces the impact of *Rate Limit* on reliability. This could also be reinforced by using additional *load_balancers*. However, this could also lead to bottlenecks and traffic congestion, as depicted in Figures 8.11. Performance degradation is higher on AWS, and the highest *total_time* value is collected at a *rate_limit* of approximately 800 requests per minute, as depicted in Figure 8.12.

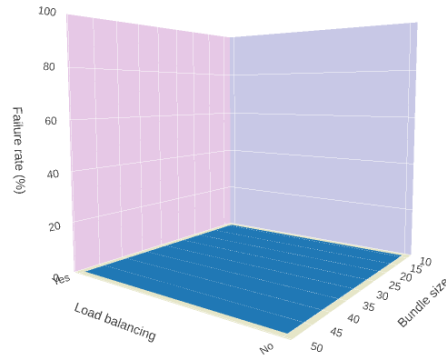


Figure 8.8: Load Balancing vs. Request Bundle: reliability perspective (on private cloud and AWS)

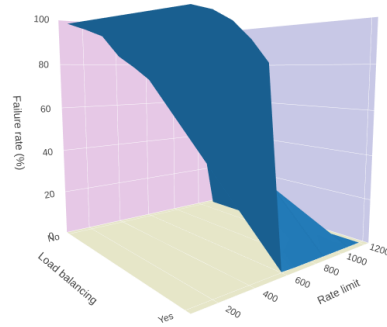


Figure 8.9: Load Balancing vs. Rate Limit: reliability perspective (on the private cloud)

Notice that *Load Balancing* has no obvious impact on performance because, as stated previously, only one cluster is used. Here, we consider Models 2 and 3 since Model 1 is not influenced by the *rate_limit* as far as reliability is concerned, as mentioned in the previous section. Also, model 3 is best where *load_balancing* has the lowest impact on reliability.

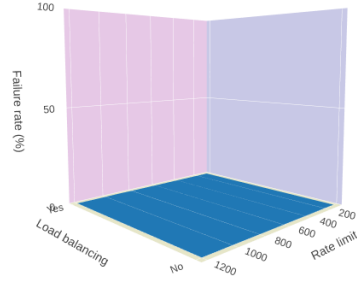


Figure 8.10: Load Balancing vs. Rate Limit: reliability perspective (on AWS)

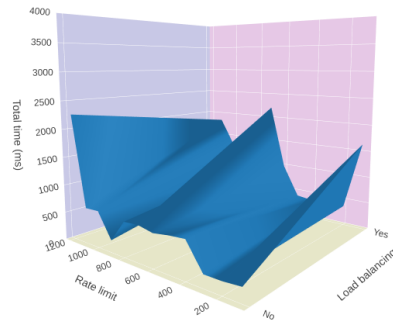


Figure 8.11: Load Balancing vs. Rate Limit: performance perspective (on the private cloud)

8.8.3 Models evaluation

To evaluate the accuracy of the models, we have used the Mean Average Error (MAE) calculations for *failure_rate* prediction errors using the private cloud and AWS data. We have also used the Mean Absolute Percentage Error (MAPE) estimates for *total_time* prediction errors. The results are reported in Table 8.10. Usually, it is advised to use MAPE for calculating model prediction error [All21]. However, as *failure_rate* usually equals 0, MAPE is not applicable, and MAE is the next preferred choice. We see that all models' prediction errors on both clouds are lower than 30%. This is often considered to be acceptable for Cloud-based architectures [MA02]. These numbers are good enough,

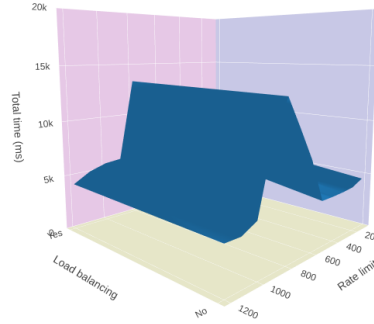


Figure 8.12: Load Balancing vs. Rate Limit: performance perspective (on AWS)

Table 8.10: Calculated MAE and MAPE for *failure_rate* prediction errors and *total_time* prediction errors respectively

Model	Private Cloud		AWS	
	MAE%	MAPE%	MAE%	MAPE%
1	19,72	28,51	1,56	24,83
2	19,72	27,99	1,56	24,83
3	27,25	28,44	1,33	24,58
4	19,67	28,97	1,48	24,82
5	27,95	29,51	1,29	25,3

especially when used for early architecting, where orders of magnitude are needed rather than precise predictions. We also observe that the prediction errors are lower on AWS compared to the private cloud, especially regarding *failure_rate*. This is explainable by the large latency gap that diverges the two environments and that we have not included in the models. Indeed, in the private cloud, the latency is very low, and thus *failure_rate* due to *Rate Limit* is high compared to AWS. It demonstrates that the models work best in high-latency environments like public clouds.

8.8.4 Threats to validity

In the model presented, we have carefully selected the independent variables and varied them in such a way as to comprehend their causal effect relationship to performance and reliability. We have also generated five models of all possible combinations of those independent variables. As described in Table [8.8](#), all the models present very high

statistical significance, which emphasizes the causal effect relationship, and thus mitigates many related internal threats to validity. However, we could have unwillingly made errors in the experiment’s selection and implementation of some scenarios. The workloads could also not be representative of a real-world benchmark. We have mitigated this issue by covering several widely used scenarios in the experiment.

As described in Table 8.8, all the models present very high *statistical significance* but with different Pillai values. We have also found that these values for specific variables remain stable from one model to another. This could be biased because we only used a unique setup for the experiment described in Section 8.6.2, leading to a construct threat of validity. To mitigate this threat, we have run the same experiment in the public cloud AWS.

In our experiment, we have considered the API patterns *Request Bundle* and *Rate Limit* and their combinations with *Load Balancing*. We have used a limited and predefined range for *bundle_size*, *rate_limit*, and *load_balancers*. Also, we have used only local rate-limiting and weighted load balancing and ran the experiment 500 times on the private cloud but only 200 times on AWS. This is mainly due to the limited scope and time of the experiment. However, this could lead to an external threat to validity regarding the generalizability of the results. We plan to mitigate this issue by extending our research to other types of *Rate Limit* and *Load Balancing*. We also plan to generalize our results by increasing the *load_balancers*, *bundle_size* values, and the number of experiment runs on AWS.

A similar threat to validity could be caused by using only one application deployed on one private cloud and one public cloud for the experiment. However, the application is a large open-source application developed using industry best practices by experienced professionals. Also, we plan to further mitigate this issue by running the experiment in other infrastructure environments and settings.

In Section 8.6.1, we have explained that extended modifications of the Lakeside mutual application have been made to enable request bundling. This might engender the *reliability of treatment implementation* thread of validity. We will validate our models using other applications and settings to waive this threat.

8.9 Conclusion and future work

In this chapter, we evaluated the impact of combining the API patterns *Request Bundle*, *Rate Limit*, and *Load Balancing* on the performance and reliability of a cloud-based benchmark based on a realistic application and setting.

To answer **RQ2.5**, we have first presented a regression model to predict the impact of combining the API patterns above on performance and reliability. Then, we validated that model by experimenting with a real-life application and cloud setting. We repeated the experiment of 130 configurations 500 times in the private cloud, which took more than 4500 hours of runtime and 200 times on AWS, totaling more than 3900 hours of runtime, generating a vast dataset. The data collected are then used to create five distinctive and highly significant models. Besides providing general experimental insights, our models can be used during early architecting. In cases where an environment needs concrete numbers, our approach presents a complete and largely automated method applicable iteratively in different contexts and settings with the support of highly configurable service mesh-based infrastructure.

As of answering **RQ2.6**, we used the experiment-generated data to present five models with very high statistical significance. As described in Section 8.8, we have found a distinctive impact of those API patterns on performance and reliability. These models could be used in the early phases of architecting distributed systems to evaluate the impact of combining such patterns on those quality properties. As discussed in Section 8.8, Model 2 is recommended when combining *Request Bundle* and *Load Balancing*, and Model 3 is best when combining *Rate Limit* and *Load Balancing*. Also, Model 1 is only usable when we have large API messages and *Request Bundle* is not used. The remaining models are suitable otherwise. All the models provide the lowest prediction errors on AWS. For future work, we plan to add more variables to our model, including those already defined in Section 8.5.1 and fixed in advance due to the time and scope of the experiment and include other *Load Balancing* algorithms and API patterns.

In this chapter, we generated benchmark data from a complex experimentation setup that combines *Rate Limit*, *Request Bundle*, and *Load Balancing*. We also provided strong prediction models for performance and reliability impact measurement when combining such patterns. In the next chapter, we use the benchmark data generated to realize a complete and automated pipeline for API reconfiguration.

Part VI

Towards API Self-adaptation

9 Automated Pattern-Based Recommendation for Improving API Operation Performance and Reliability in Cloud-Based Architectures

The extensive use of APIs as the entry point to many Cloud-based applications has created challenging problems, especially concerning API quality properties such as performance and reliability. API best practices and patterns, such as bundling requests, rate limiting, or load balancing, have been proposed to solve these challenges. Unfortunately, no study investigating the impact of existing API practices and patterns on such quality properties exists beyond informal recommendations. In this chapter, we fill this gap by proposing a pattern-based, automated recommendation approach to improve the performance and reliability of API operations. We provide a benchmark suite based on a realistic open-source microservice application deployed on the service mesh-based infrastructure already described in the previous chapter to enable the automatic generation of comprehensive decision tree models. These models are then processed to generate API design recommendation algorithms to improve API operations regarding performance and reliability stored in catalogs for reuse. We validate our algorithms using extensive data sets generated by running the benchmark on a private cloud and AWS. For both environments, based on the decision tree models automatically generated from the measured data, API design recommendation algorithms have been calculated using our approach.

9.1 Introduction

The current chapter is a follow-up work of Chapter 8. We use the benchmark data already generated in this chapter to build decision tree models. From these models, we develop algorithms that are the essence of the self-adaptation approach we present in the current chapter and which answers the first part of **RQ3**.

The widespread use of cloud-based applications accessible through APIs has led to many challenges regarding the quality of these APIs, especially when deployed on a large scale. To cope with such challenges, API design has become an essential building block in achieving the desired level of quality properties for the APIs. API design prevents degrading such quality properties and concerns refactoring APIs based on runtime usage observations. While already an extensive set of API patterns and best practices has been proposed [ZSL⁺22, ZSZ⁺18, Hig20, Hig21], today, no support beyond informal guidance on API quality exists. This is particularly unsatisfactory for measurable properties such as performance and reliability. In addition, it is often unclear which effect a *combination* of different patterns would have when just following such informal guidance. Thus, API designers need to improve their APIs based on these informal recommendations by trial and error, which is costly and likely does not lead to optimal results.

This chapter aims to study the possible combinations of *Rate Limit*, *Request Bundle* and *Load Balancing* and, based on the gathered data, automatically generate recommendations to enable refactoring or adaptations of API design to optimize these combinations concerning their performance and reliability impact. This work is based on Chapter 8 in which possible pattern combinations have been studied by providing an initial service mesh-based empirical study resulting in a regression model, which had the aim to quantify the impact of combining such API patterns on API performance and reliability. However, such a model is static and does not consider parameters like the application type and usage period. For this reason, in this chapter, we have substantially extended these empirical studies to provide a novel automatic and dynamic API design recommendation approach based on decision tree modeling, which is the core contribution of this chapter. We aim to answer the following research questions:

RQ3.1 What are the different decision criteria that should be considered to evaluate the impact of combining *Request Bundle*, *Rate Limit*, and *Load Balancing* patterns

on API performance and reliability?

RQ3.2 How can we select the best combinations of these patterns and API operations in terms of performance and reliability impacts?

We provide a benchmark suite based on a real-life application deployed on modern cloud-based infrastructure. In the prototype implementation for experiments, the Istio¹ service mesh and the Envoy² proxy were used as a widely used combination of representative modern infrastructure tools that enable to combine features like *Rate Limit* and *Load Balancing* (see Chapter 4).

We use the data generated in the experiment to automatically construct decision tree models for each combination of those API patterns to answer those questions. Then, we develop API design recommendation algorithms to select the best combination of API patterns and operations in terms of performance and reliability impacts using these decision trees. These algorithms are stored in catalogs for reuse. Please note that such reuse of our empirical results makes sense in similar settings (business application on a similar private cloud or AWS configuration) to get initial estimates. Also, we have derived generic recommendations on pattern combinations that were yet unknown or not yet empirically proven (see Section 9.6). We propose an API design recommendation pipeline that essentially re-runs our benchmark in such a new setting or configuration to get exact recommendation algorithms or estimates in vastly different configurations. Our approach automatically generates and validates the necessary decision trees and API design recommendation algorithms in this context.

The chapter is organized as follows. Our research methods are summarized in Section 9.3. Next, we describe our approach in Section 9.4. In Section 9.5, we analyze the data and develop the API design recommendation algorithm. Then, we provide a discussion and threats to validity in Section 9.6. Finally, we conclude and describe future work in Section 9.7.

¹<https://istio.io/>

²<https://www.envoyproxy.io>

9.2 Related work

Many studies have suggested different approaches for dealing with self-adaptive systems to improve quality properties like performance and reliability. Self-adaptation was a topic that emerged in early Service-oriented Architecture (SOA) research as a way to increase resiliency [dHS11]. Garlan et al. [GCH⁺04] presented Rainbow as a reusable infrastructure for system developers to realize self-adaptation cost-effectively. Layaida et al. [LH05] presented PLASMA as a component-based framework for building self-adaptive multimedia applications. The framework is based on a hierarchical composition and reconfiguration model. Adapt! [JRSG21] plug-in architectures were proposed as an external adaptation solution to achieve full self-adaptation functionality. The middleware approach [GEL⁺06] was introduced to enable this externalization through application and service mirroring. However, all of these approaches are general and not API-centric. They are also not applicable to highly versatile environments like Cloud-based architectures.

The decision tree modeling approach has also been the subject of many studies in the literature. Rathore et al. [RK16] presented a decision tree-based approach to predict software faults. Another similar military-based research developed a decision tree model to identify fault-prone software modules [KAB⁺96]. Chui et al. [CCW98] provided an approach based on decision tree modeling considering the time of data collection to improve performance prediction rate without affecting prediction accuracy. Decision tree modeling has also been combined with other techniques, like Bayes in the transportation world, to build a lane-changing assistance simulation model [HES14] to increase the safety of passengers. These extensive use cases and others for decision tree modeling to predict the performance and reliability of diverse applications show that the approach followed in this chapter is relevant.

Lawin et al. [LPY21] evaluated the effect of using GraphQL and REST technologies on the performance of massively used API. They figured out that the first technology is the better choice when data frequently change, while the second stands out when there is a need for multiple requests. While these studies and others have proposed robust models and interesting results to derive API configuration recommendations based on specific settings and technologies, those models and recommendations lack important properties like self-adaptation and easy reuse. Also, they treat each API pattern individually, not in combination with others, and deal with specific technologies. In this chapter, we try to fill

Table 9.1: Definition of Parameters

Independent variable	Description
<i>rpm</i>	The number of requests per minute for API users.
<i>api_users</i>	The number of API users sending the requests.
<i>rate_limit</i>	The value of the applied rate limit ranges from 100 to 1200 (in increments of 100).
<i>request_bundle</i>	Categorical variable indicating whether request bundling is used or not.
<i>bundle_size</i>	The request bundle size ranges from 10 to 50 (in increments of 10).
<i>load_balancing</i>	Categorical variable indicating whether load balancing is used or not.
<i>method</i>	Categorical variable indicating the method executed by API users (get, delete, update, or create).
Dependent variables	Description
<i>total_time</i>	The total roundtrip time spent for requests between API client and backend.
<i>failure_rate</i>	The percentage failure rate of API requests.

this gap by proposing a complete self-adapted and agnostic API design recommendation pipeline.

9.3 Research methods

Our approach is based on a dataset generated in a prior study which we describe first. Then, we provide details about the benchmark used to evaluate the model.

9.3.1 Regression model

The study considered two types of variables: numerical and categorical. The first type of variables includes *rpm*, *api_users*, and *bundle_size*, collected at the client-side, *rate_limit*, which (if enabled) is set at the server-side, and *total_time* and *failure_rate* which are collected at the server-side. The second type of variables includes *request_bundle* and *load_balancing*, which indicate whether *Request Bundle* and *Load Balancing* are used respectively. It also includes *method*, which indicates the API operation executed on the backend (see Table 9.1).

9.3.2 Benchmark description

As no benchmark for studying API operations existed, a novel benchmark was developed for the study based on the open-source Lakeside Mutual³ application that is realistic and developed based on real-life experience and makes use of several microservice API patterns such as *Request Bundle* [ZSL⁺21] described in details in Chapter 7 (already implemented for the API operation: *get* customers).

Based on a manual analysis of which other API operations could benefit from *Request Bundle*, the application was extended to support *Request Bundle* in the following API operations: *create*, *delete*, and *update*. The implementation follows the same style as the existing bundle implementation in Lakeside Mutual. The application is composed of four Java-based microservices packaged using Maven⁴, containerized using Docker⁵ and deployed on the Istio service mesh. Containers orchestration is realized using Kubernetes⁶. Database per service model is used when *Load Balancing* is inactive. Otherwise, the database is shared by the service and its replica. Istio is used since it provides various functionalities out of the box like *Rate Limit* and *Load Balancing* (see Chapter 4). On the client side, the workload is defined using Shell⁷ script covering all possible combinations of the API patterns under study.

To evaluate the impact of the combination of those API patterns on performance and reliability, we present the benchmark described in Table 9.2 generated from earlier experimental data. In the benchmark, the authors of this study define *bundle_size* ranging from 10 to 50, which reflects the bundle size, in case *Request Bundle* is active. They also define *rate_limit* ranging from 100 to 1200, in case *Rate Limit* is active. Otherwise, *rate_limit* = ∞ . *load_balancing* is a categorical value indicating whether *Load Balancing* is used or not.

The benchmark was run on a private Cloud provided by the University Data Center and AWS⁸. Infrastructure details are provided in Table 9.3. Both experiments were executed covering 130 different configurations 500 times on the private cloud and 200

³<https://github.com/Microservice-API-Patterns/LakesideMutual>

⁴<https://maven.apache.org>

⁵<https://hub.docker.com>

⁶<https://kubernetes.io>

⁷<https://www.shellscript.sh>

⁸<https://aws.amazon.com>

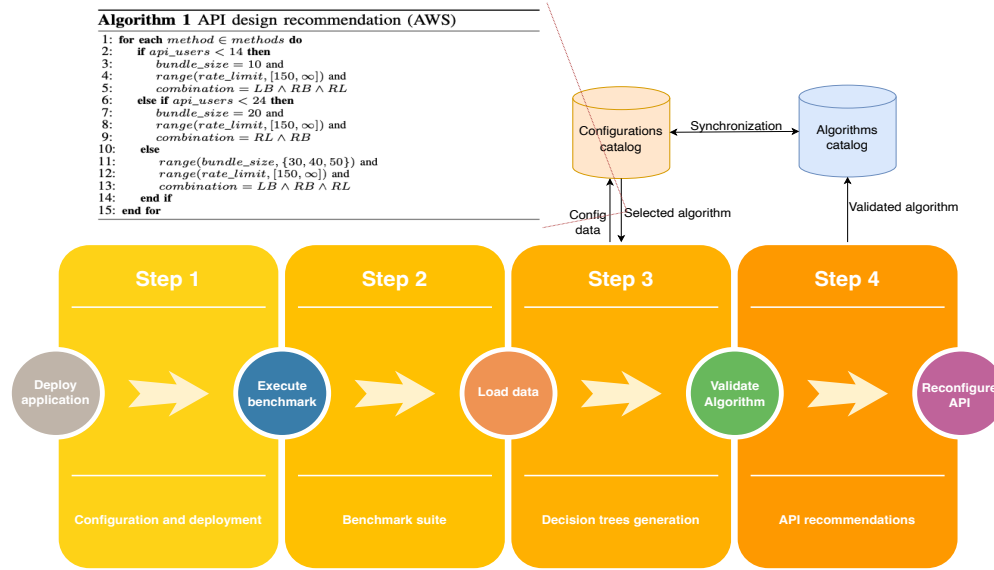


Figure 9.1: API design recommendation pipeline

times on AWS, totaling 130000 repetitions, more than 4500 hours of runtime in the private cloud, and 52000 repetitions on AWS, and more than 3900 hours of runtime. A small portion of the data collected is summarized by calculating the mean and shown in Table 9.2. The full dataset is provided in an online appendix⁹.

9.4 Approach description

Modern applications expose many APIs accessible by API clients and interoperable with other third-party applications. However, part of these APIs is rarely (and sometimes never) exploited by the API clients and applications. The regression model described briefly in Section 9.3.1 could be used to find the right combination of API patterns to improve performance and reliability. However, this approach is static and does not include parameters like the application type and usage period. Since this kind of parameter is hard to consider in the model, we further developed an approach introducing decision trees using the model parameters and the data generated in the experiment described in Section 9.3.2. We then automatically generate an API design recommendation algorithm from the decision trees that could enable APIs to adapt at runtime. Details are provided

⁹<https://doi.org/10.5281/zenodo.7691350>

in the next section.

Figure 9.1 represents the API design recommendation benchmark pipeline that continuously delivers API improvement recommendations to a target platform, described in Step 1. The benchmark is configured using the target cloud setting during this initial step, and the application is deployed. In Step 2, the benchmark suite is executed to generate data which are then collected along with this target configuration to verify whether it exists in the configurations catalog in Step 3. If that is the case, the benchmark selects the corresponding algorithm with the collected data to generate API recommendations in Step 4. Otherwise, data are extracted and analyzed in Step 3 to create decision trees, which are then used to develop new algorithms with the collected data. After validation, these algorithms are used to reconfigure API and added to the algorithms catalog in Step 4. The configurations and algorithms catalogs are continuously synchronized and updated for each newer version of the target application and cloud setting.

9.5 API analysis and recommendations

In this section, we provide details of our API analysis approach. Then, we use this analysis approach to generate API design recommendation algorithms from the private cloud and AWS data collected using the proposed benchmark.

9.5.1 API analysis

The data generated using the experimentation described in Section 9.3.2 are analyzed utilizing the package `rpart`¹⁴ provided by R language¹⁵. This package builds classification and regression models using a two-phase procedure. First, it selects the variables that best split the data into two groups and recursively repeats this until the subgroup is too small or there is no further improvement. Secondly, the resulting tree is reevaluated by selecting the subtree with the lowest risk. We have generated 14 decision trees covering all API patterns and method combinations for each cloud type (private cloud and AWS). The decision trees in Figures 9.2 to 9.7 are used to generate the algorithms for AWS. The remaining figures are for the private cloud.

¹⁴<https://cran.r-project.org/web/packages/rpart/rpart.pdf>

¹⁵<https://www.r-project.org>

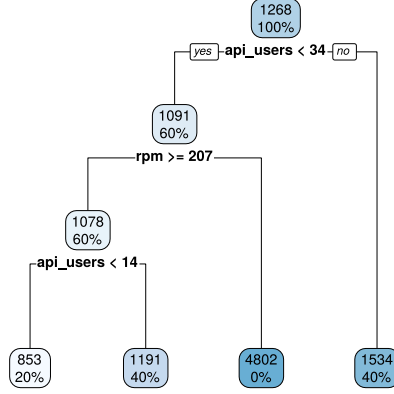


Figure 9.2: Load balancing, request bundle & rate limit combined: Performance decision tree on AWS (milliseconds)

Let's take as an example the case when combining *Rate Limit*, *Request Bundle*, and *Load Balancing* on AWS in Figures 9.2 to 9.7. The first figure shows the case where all the patterns are combined. The performance here is best when $api_users < 14$ with $total_time = 853ms$. Regarding reliability, Figure 9.3 adjusts the value of $rate_limit$ to ≥ 150 for maximum reliability. When $api_users < 24$, the best combination is *Request Bundle* and *Rate Limit* in Figure 9.6, where $total_time$ ranges between $730ms$ and $1270ms$ depending on the value of rpm . The range values of $rate_limit$ do not change as described in Figure 9.7. The remaining and extreme case with $api_users \geq 24$ is illustrated by the first figure with a maximum value of $total_time = 1534ms$ (not considering the case where $rpm < 207$ with the probability of occurrence $\approx 0\%$). Those conditions are generated for specific applications and cloud settings in a certain period. To do that in real-time, we need to construct algorithms based on those decision trees that must be run continuously, as explained in Section 9.4. The following Section describes our algorithms in more detail.

9.5.2 API design recommendation algorithms

Let LB , RB , and RL denote *Load Balancing*, *Request Bundle*, and *Rate Limit*, respectively. Each possible *combination* is described as:

$$combination \in \{LB \wedge RB, LB \wedge RL, RL \wedge RB, LB \wedge RB \wedge RL\}$$

Algorithms 1 and 2 are generated from the data collected on both AWS and the private

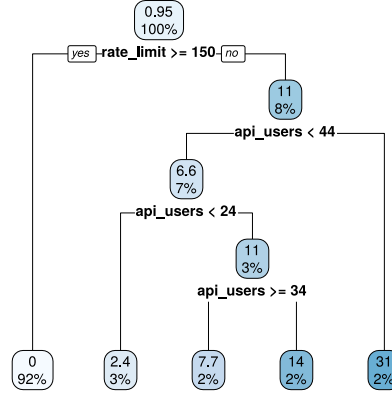


Figure 9.3: Load balancing, request bundle & rate limit combined: Reliability decision tree on AWS (% failure)

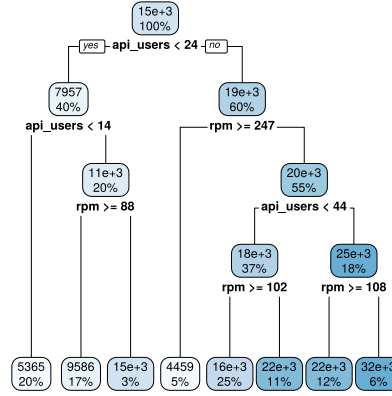


Figure 9.4: Load balancing & rate limit combined: Performance decision tree on AWS (milliseconds)

cloud, respectively, to determine the best API patterns and methods combinations. The values corresponding to *rate_limit*, *rpm*, and *api_users* are generated from the collected data. Each new data collection will yield values specific to the application settings and usage period. In Algorithm 1, in case *api_users* < 14 or *api_users* ≥ 24, then the conditions that are displayed are extracted from the decision trees depicted in Figures 9.2 and 9.3, as described in the last section. The case in the middle is illustrated by Figures 9.6 and 9.7. Algorithm 2 follows the same logic using different decision trees depicted in Figures 9.8 to 9.13.

```

foreach method  $\in$  methods do
  if api_users < 14 then
    bundle_size = 10 and
    range(rate_limit, [150,  $\infty$ ]) and
    combination = LB  $\wedge$  RB  $\wedge$  RL
  else if api_users < 24 then
    bundle_size = 20 and
    range(rate_limit, [150,  $\infty$ ]) and
    combination = RL  $\wedge$  RB
  else
    range(bundle_size, {30, 40, 50}) and
    range(rate_limit, [150,  $\infty$ ]) and
    combination = LB  $\wedge$  RB  $\wedge$  RL
  end
end

```

Algorithm 1: API design recommendation (AWS)

```

foreach method  $\in$  methods do
  if api_users < 14 then
    bundle_size = 10 and
    range(rate_limit, [650,  $\infty$ ]) and
    combination = LB  $\wedge$  RL
  else if api_users < 24 then
    bundle_size = 20 and
    combination = LB  $\wedge$  RB
  else if api_users < 34 then
    bundle_size = 30 and
    if rpm  $\geq$  1900 then
      range(rate_limit, [150,  $\infty$ ]) and
      combination = RL  $\wedge$  RB
    else
      range(rate_limit, [650,  $\infty$ ]) and
      combination = LB  $\wedge$  RL
    end
  else
    range(bundle_size, {40, 50}) and
    if rpm  $\geq$  2071 then
      range(rate_limit, [150,  $\infty$ ]) and
      combination = RL  $\wedge$  RB
    else
      range(rate_limit, [250,  $\infty$ ]) and
      combination = LB  $\wedge$  RB  $\wedge$  RL
    end
  end
end

```

Algorithm 2: API design recommendation (private cloud)

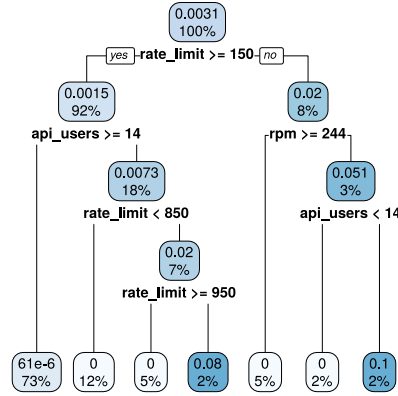


Figure 9.5: Load balancing & rate limit combined: Reliability decision tree on AWS (% failure)

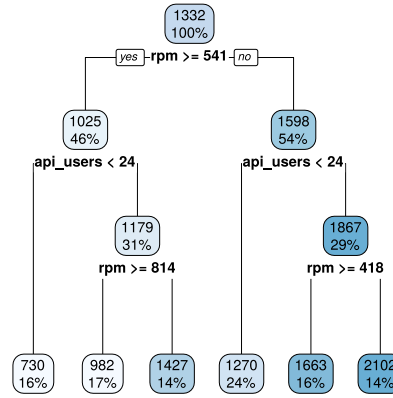


Figure 9.6: Rate limit & request bundle combined: Performance decision tree on AWS (milliseconds)

9.6 Discussion and threats to validity

In this section, we first discuss the generalizability of the results and the possible design advice. Then, we provide some threats of validity that we may encounter.

9.6.1 Discussion

The following results are observed from both generated algorithms [1](#) and [2](#). While the outcomes of these algorithms differ in several cases, we also observe some similarities, which are discussed in the following.

Load balancing & request bundle combined: This combination is beneficial when

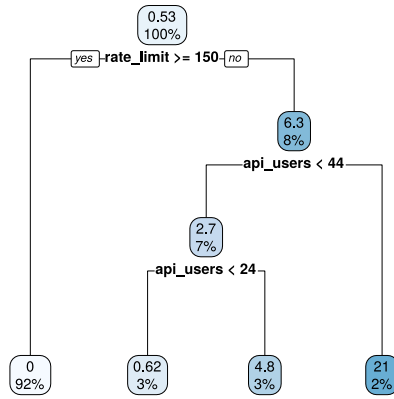


Figure 9.7: Rate limit & request bundle combined: Reliability decision tree on AWS (% failure)

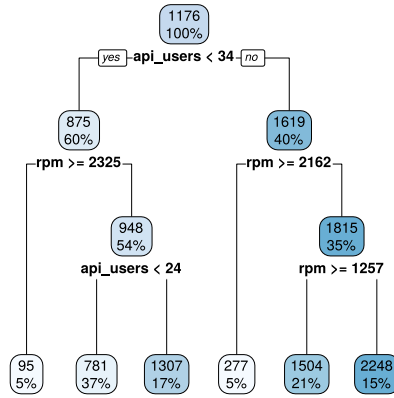


Figure 9.8: Load balancing, request bundle & rate limit combined: Performance decision tree on the private cloud (milliseconds)

rpm is not very high, if at all since else it would overload the API backend with large messages and create congestion in the load balancer. It is illustrated in Algorithm 2 that this combination is only recommended with relatively low *api_users* and *bundle_size* in the private cloud. However, for AWS, our algorithms do not recommend this combination. The combination should be used in applications with a limited or dedicated number of users, like a private company website.

Rate limit & request bundle combined: When using the private cloud, this combination might be used in highly demanding environments with high values of *api_users* and *rpm*. *Rate Limit* plays a significant role in preventing the backend from

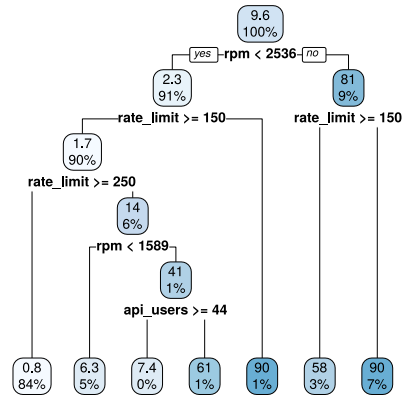


Figure 9.9: Load balancing, request bundle & rate limit combined: Reliability decision tree on the private cloud (% failure)

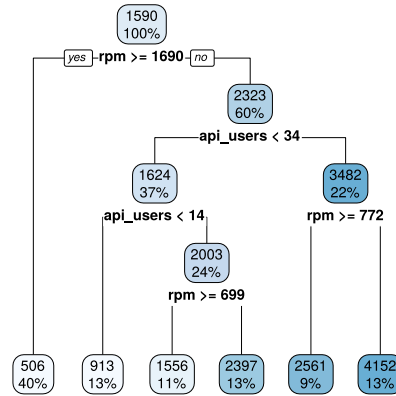


Figure 9.10: Load balancing & rate limit combined: Performance decision tree on the private cloud (milliseconds)

being overwhelmed. Also, when combined with *Request Bundle*, *Rate Limit* does not cause a high failure rate, increasing reliability from the API clients' perspective. However, *Request Bundle* has to be supported on both the client and server side, increasing complexity and costs.

Load balancing & rate limit combined: The combination is to be used when *api_users* and *rpm* are relatively low due to using one load balancer. Adding more load balancers will make the combination more scalable. It should also be used with relatively high values of *rate_limit* (≥ 650). This shows that *Load Balancing* plays a significant role in distributing the load and alleviating the role of *Rate Limit* in preventing server

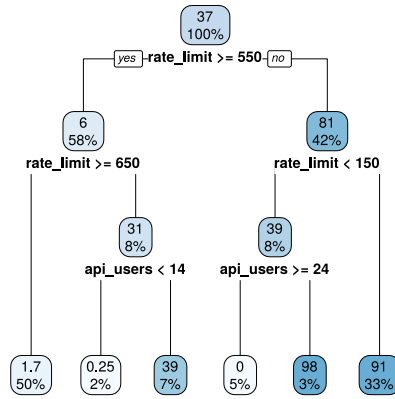


Figure 9.11: Load balancing & rate limit combined: Reliability decision tree on the private cloud (% failure)

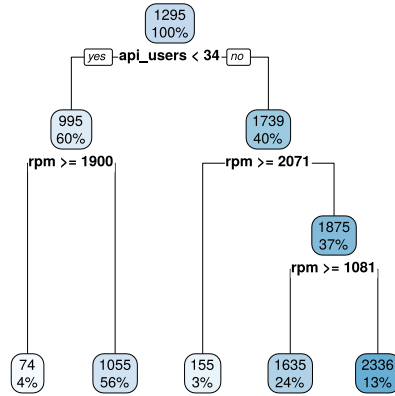


Figure 9.12: Rate limit & request bundle combined: Performance decision tree on the private cloud (milliseconds)

overload and decreasing both performance and reliability. The combination should not be used in highly versatile environments like AWS.

Load balancing, request bundle & rate limit combined: The combination is recommended in relatively extreme situations when either `api_users` or `rpm` are relatively high. *Request Bundle* plays a crucial role, especially on AWS, by preventing the load balancer from overloading when `api_users` and `rpm` are very high. The combination is the most flexible and should be used in highly versatile applications like Social Media and E-commerce.

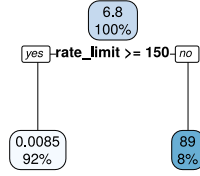


Figure 9.13: Rate limit & request bundle combined: Reliability decision tree on the private cloud (% failure)

9.6.2 Threats to validity

In any modeling study, a couple of threats to validity need to be considered and mitigated by workaround solutions or future improvements.

The first threat to validity is regarding the algorithms presented in the last section. We considered all *methods* to be the same because the type of the method had a negligible effect on the *total_time* calculated. However, we do not exclude the general case where in some situations, some types of *methods* might take much longer execution times than others. To mitigate this issue, we plan to add a *method_time* variable to our model in future work.

Also, the API pattern implementations used in our model are particular to the configuration used in a prior study. For instance, only one load balancer and one type of rate limiting are used. This could lead to an external threat of validity regarding the general applicability of our model. While the dataset used for different configurations might be completely different, we believe that the automation and cyclic process of the tree decision modeling to generate best-case scenarios is still valid. Nevertheless, we must validate that using different datasets from different configurations and settings.

The proposed benchmark might be biased because it is based on only one open-source application. We tried to mitigate this issue by selecting a realistic open-source application (based on real-life experience in the domain) by former practitioners. As the authors were not involved in the application writing, the bias that we could have

influenced the experiment outcomes this way is mitigated. We selected diverse operations from the application to represent a realistic operation set in the benchmark. However, the application is a business system. Systems with vastly different loads might have different characteristics and need other benchmarks. For these reasons, we cannot claim generalizability beyond the system represented well by this benchmark.

Our results might not be generalizable beyond the environments and technologies used in our study. We tried to mitigate this threat by using two very different environments, the private cloud and AWS. We only tested one configuration on each, but as our approach requires rerunning the benchmark on a new environment to provide accurate recommendations, dealing with this threat is part of our approach. We believe that for similar private or public cloud scenarios, our generated algorithms will likely give a reasonable estimate but no precise recommendations. This would be good enough for design advice but not for automatic adaptations based on our approach.

9.7 Conclusion and future work

To answer **RQ3.1** and **RQ3.2**, we have developed algorithms based on decision trees generated for each combination of the studied API patterns.

Regarding **RQ3.1**, we have found that the *api_users* and *rpm* values are the main criteria for each decision on those algorithms. We also conclude that each recommended combination of API patterns is relevant to a specific configuration setting. Specifically to the configuration in this chapter, *Load Balancing* and *Request Bundle* combined should only be used with a limited number of *api_users*, preferably in a dedicated environment. Similarly, the combination *Load Balancing* and *Rate Limit* should be avoided in public clouds, for example. Instead, all the patterns should be combined unless the cost and complexity of *Request Bundle* is an issue.

Concerning **RQ3.2**, not only have we developed algorithms from decision tree models to generate the best combinations of these patterns in terms of performance and reliability impacts, but we also propose an entire API design recommendation pipeline that generates those algorithms automatically by re-running our benchmark using new settings and configurations with the support of the service mesh-based infrastructure. The validated algorithms are then stored in a catalog for re-use if the same configuration setting is

triggered.

Our generated algorithms can provide for private or public cloud scenarios similar to our experiment setting a good estimate which would be good enough for design advice. Also, using our approach in a self-adaptation scenario is possible, i.e., to run automatic adaptations based on our approach. But for this, we need to re-run our benchmark on the target environment to provide precise recommendations.

In future work, we plan to run the API design recommendation pipeline for new configurations and settings. We also aim to integrate new parameters into our model, like the methods execution time.

The approach presented in this chapter answers the first part of **RQ3**. It has also made it possible to combine the dynamic approach for API pattern evaluation we have utilized so far in the dissertation with the static analysis, which is the work suggested in the next chapter.

9.7 Conclusion and future work

Table 9.2: Sample data: LB=Load Balancing, RB=Request Bundle, OP=Operation, SZ=Bundle Size, TT=Total Time(ms), FR=Failure Rate(%)

				Rate limit														
				100		200		300		400		500		600		700		∞
LB	RB	OP	SZ	TT	FR	TT	FR	TT	FR	TT	FR	TT	FR	TT	FR	TT	FR	TT
Yes	Yes	Get	50	173,32	31,91	222,56	15,96	239,01	10,64	249,17	7,98	386,46	12,76	588,87	10,64	814,30	9,12	280,98
			40	100,60	49,66	161,85	24,83	182,33	16,55	194,88	12,42	235,96	19,87	446,65	16,55	683,52	14,19	235,89
			30	79,00	53,54	127,72	26,77	145,45	17,85	155,48	13,39	249,52	21,42	507,21	17,85	725,92	15,30	204,03
			20	94,84	29,97	109,34	21,78	116,87	14,52	122,38	10,89	317,50	14,71	541,16	14,52	746,28	12,44	164,43
			10	64,18	31,20	75,23	15,60	79,02	10,40	81,55	7,80	277,31	12,48	501,92	10,40	690,79	8,92	125,88
		Update	50	1694,05	47,87	2347,08	23,93	2829,87	15,96	3202,48	11,97	3417,89	9,57	3440,80	7,98	3062,27	13,68	2206,88
			40	690,41	74,17	1501,51	37,25	2054,19	24,83	2412,76	18,62	2616,78	14,90	2686,67	12,42	2333,14	21,24	1557,92
			30	443,77	80,31	1237,47	40,16	1659,41	26,77	1998,13	20,08	2182,29	16,06	2227,49	13,39	1925,88	22,95	1264,42
			20	917,49	44,96	1227,73	22,48	1404,52	21,78	1689,10	16,33	1826,23	13,07	1841,12	10,89	1621,98	15,76	836,63
			10	510,90	46,81	686,79	23,40	873,89	15,60	1035,38	11,70	1131,87	9,36	1180,39	7,80	1032,16	13,37	451,94
		Delete	50	1878,47	47,87	2802,29	23,93	3008,76	15,96	3039,55	11,97	3041,84	9,57	3021,96	7,98	2685,27	13,68	2152,24
			40	544,65	74,17	1611,92	37,25	2084,14	24,83	2218,97	18,62	2262,23	14,90	2268,20	12,42	1980,38	21,24	1484,82
			30	280,74	80,31	1340,56	40,16	1600,04	26,77	1674,18	20,08	1687,21	16,06	1693,75	13,39	1471,86	22,95	1316,66
			20	823,02	44,96	1194,01	22,48	1037,57	21,78	1108,92	16,33	1140,91	13,07	1148,58	10,89	1033,16	15,76	934
			10	516,88	46,81	691,40	23,40	742,61	15,60	734,47	11,70	721,58	9,36	707,89	7,80	633,57	13,37	459,28
		Create	50	1804,28	47,87	2541,28	23,93	2676,87	15,96	2702,85	11,97	2700,02	9,57	2677,38	7,98	2505,30	13,68	2686,54
			40	763,51	74,17	1625,92	37,25	1934,97	24,83	2019,48	18,62	2060,32	14,90	2065,04	12,42	1862,34	21,24	1867,13
			30	413,74	80,31	1292,95	40,16	1548,42	26,77	1584,90	20,08	1609,71	16,06	1609,33	13,39	1439,68	22,95	1580,91
			20	905,41	44,96	1196,48	22,48	1033,29	21,78	1086,63	16,33	1123,11	13,07	1115,16	10,89	1056,33	15,76	1084,51
			10	456,18	46,81	670,42	23,40	733,00	15,60	735,22	11,70	723,65	9,36	700,85	7,80	652,40	13,37	551,97
	No	Get	50	689,96	64,67	855,37	71,25	1206,95	49,61	1406,60	37,39	1275,43	42,84	1397,88	48,68	1788,78	42,63	1643,10
			40	563,35	64,15	703,43	69,93	985,75	48,07	1137,54	36,34	1039,13	41,90	1182,27	47,54	1530,61	41,37	1302,57
			30	463,46	64,03	574,90	65,59	777,26	44,37	883,25	33,75	812,65	39,80	922,42	44,36	1224,45	38,30	1034,79
			20	270,45	96,51	348,13	81,03	487,17	54,30	575,15	40,73	559,09	51,88	671,44	54,16	955,22	46,54	719,87
			10	148,16	95,85	224,22	75,96	279,79	50,64	314,54	37,98	299,71	49,56	437,46	50,64	689,05	43,41	388,14
		Update	50	918,42	48,69	1068,77	71,84	1459,03	71,25	2212,97	55,82	2662,99	44,65	2758,25	37,39	2469,20	39,00	2923,57
			40	789,48	48,77	936,57	71,34	1253,94	69,93	1831,40	54,08	2172,24	43,26	2239,12	36,34	2010,26	38,12	2198,13
			30	696,78	48,49	757,38	71,10	1017,43	65,59	1455,35	49,91	1708,43	39,93	1742,78	33,75	1570,58	35,86	1767,02
			20	429,95	97,55	491,03	95,25	668,80	81,03	966,40	61,09	1147,01	48,87	1227,86	40,73	1091,59	48,84	1182,39
			10	205,06	96,96	263,27	93,86	431,99	75,96	597,72	56,97	690,20	45,58	710,35	37,98	628,98	46,41	597,28
		Delete	50	895,41	48,69	824,58	71,84	1482,69	71,25	3084,93	55,82	4308,49	44,65	4349,28	37,39	3810,93	39,00	3528,01
			40	821,04	48,77	722,60	71,34	1225,45	69,93	2583,96	54,08	3486,26	43,26	3492,44	36,34	3063,40	38,12	2633,99
			30	744,96	48,49	606,85	71,10	1213,10	65,59	2252,40	49,91	2846,37	39,93	2785,86	33,75	2448,73	35,86	2363,27
			20	240,22	97,55	257,92	95,25	676,70	81,03	1169,42	61,09	1554,57	48,87	1730,57	40,73	1519,18	48,84	1577,66
			10	131,79	96,96	133,15	93,86	344,33	75,96	600,10	56,97	791,03	45,58	878,57	37,98	771,18	46,41	786,28
		Create	50	895,41	48,69	824,58	71,84	1482,69	71,25	3084,93	55,82	4308,49	44,65	4349,28	37,39	3810,93	39,00	4486,12
			40	821,04	48,77	722,60	71,34	1225,45	69,93	2583,96	54,08	3486,26	43,26	3492,44	36,34	3063,40	38,12	3137,15
			30	744,96	48,49	606,85	71,10	1213,10	65,59	2252,40	49,91	2846,37	39,93	2785,86	33,75	2448,73	35,86	2801,52
			20	240,22	97,55	257,92	95,25	676,70	81,03	1169,42	61,09	1554,57	48,87	1730,57	40,73	1519,18	48,84	1815,61
			10	131,79	96,96	133,15	93,86	344,33	75,96	600,10	56,97	791,03	45,58	878,57	37,98	771,18	46,41	899,42

Table 9.3: Infrastructure details of the experiments

	Private cloud	AWS
Cluster	11 Ubuntu ¹⁰ 18.04.5 LTS Virtual Machines (VMs) installed on vSphere ¹¹ 6.7 environment.	Amazon EKS Cluster ¹² composed of four EC2 ¹³ instances pool.
Clients	Ubuntu 18.04.5 LTS virtual desktops are used to inject HTTP requests into the private cloud. Each of the virtual desktops has 2 CPU cores Intel Xeon(R) CPU E5-2650 0 @ 2.00GHz with 8 GB of system memory.	
Minikube	1.20.0	
Kubernetes	1.20.2	
Istio	1.10.0	
Number of clusters	2	1

10 Microservices API Analytics: Bridging the Gap Between Runtime and Static Analysis for Enhanced Software Quality

Building the API recommendation pipeline in the previous chapter has sparked the possibility of integrating that dynamic approach with the static analysis to answer the last part of **RQ3**. The combined approach has emerged to take advantage of both strategies' strengths and overcome their weaknesses.

Microservice architectures are widely used for creating highly scalable and adaptable systems. However, managing the complexity of distributed services in cloud-based environments can be challenging. Traditional methods for analyzing software architecture are not well-suited for microservice systems. We suggest Microservice API Analytics as a new approach to address this challenge, combining static code analysis, runtime behavior analysis, and tracking system development and evolution. A holistic quality model is proposed to evaluate and assess microservices APIs throughout their lifetime, enabling better decision-making for service architects and self-adaptation of the microservice system.

The widespread adoption of microservice architectures has led to the development of highly scalable systems in which it is possible to react promptly to new and changing requirements. Managing distributed services, particularly in cloud-based environments, can be challenging and create high maintenance efforts [STVDH18, MBMR21]. While the initial promise was to reduce overall complexity by breaking down monolithic systems into smaller, more manageable microservices, this has not always been the case. Microservice architectures can sometimes increase complexity by introducing new challenges, such as data consistency or complex interactions across services [STVDH18].

A vital challenge in a service architect's daily work is taming the complexity of

these distributed microservice systems and better understanding the outcomes of their architectural decisions. However, existing methods for analyzing the qualities of software architecture are only of limited use when it comes to microservice systems. Established software quality metrics like coupling or cohesion may work sufficiently for monolithic applications but are not well suited to track the interactions between distributed services using asynchronous communication protocols. Also, most microservice systems evolve organically, and changes affecting their API design are often more feature-driven than based on improving architectural qualities [SZ21]. Consequently, the overall architecture of these systems often exists only in an implicit form without clear visualization or understanding of the whole system [BDC22]. What is still understandable in cohesive monolithic designs can become very complex when the system size reaches several hundreds of dynamically interacting services. Analyzing this behavior during runtime is a very challenging task.

There have been efforts to fill this analytical gap regarding microservice systems. Most of these approaches focus only on single and isolated attributes. Some studies analyze static artifacts such as Docker configuration files or source code [BDC22]. Others examine the behavior of a microservice system during runtime [TRA⁺17] or investigate its evolution over time [TAS⁺20]. These are all significant contributions, but they only illuminate a partial aspect of the whole. To understand highly dynamic and complex systems such as microservice architectures, a novel approach is needed that combines all existing practices into a new and holistic analysis strategy.

The work we present in this article is a first step in this direction. Our research on *Microservice API Analytics* aims at defining such a holistic quality model. We aim to evaluate and assess microservices APIs' essential and critical aspects throughout their lifetime, including static code analysis, runtime behavior checking, and tracking these systems' whole development and evolution cycles as a basis of self-adaptation of the microservice system.

10.1 Challenges in Microservice API Analysis

When analyzing microservices, we must consider the central part of their communication, their APIs. The API of a service is its only public interface that clients (or other services)

can access to establish a connection with the service and request information. As such, many properties of a service architecture, like its dependencies, security aspects, or data payloads, highly depend on the design and interactions of its APIs. There are different approaches to evaluating the structure and properties of a microservice system via its APIs, with static analysis often being one of the first methods used.

Static analysis generally refers to investigating only assets available without executing the system. Relying solely on the OpenAPI¹ specification is often not sufficient, as not all systems use this technique to document their APIs, and while there also exists a counterpart for asynchronous communication (AsyncAPI²), it is not yet widely adopted. Instead, we focus on reconstructing a system's architectural model. While this can include every asset part of the development process, the vast majority are source code and configuration artifacts [BDC22]. Such a static API analysis has several advantages, especially regarding microservice systems. First, setting up and starting a system comprised of several dozens or hundreds of services can be time-consuming and require considerable computational and network-related resources. Second, to run the system, all artifacts must be well-formed and compileable (ideally without runtime errors), a constraint not always guaranteed during development. In contrast, a static analysis relies only on file artifacts, requiring less setup time. It can be done in much shorter intervals, making it well-suited for integration into a continuous development process. Since the system does not have to run for the analysis, one can also investigate all possible call paths between services, not only those invoked during runtime.

And yet, despite its advantages, this approach is not free of challenges: Due to the polyglot nature of many microservice systems, extracting all relevant information from the underlying source code artifacts is not trivial. To gather all relevant data and analyze their semantics, one has to parse and understand the structure of the code. A standard way of doing this is by reconstructing an abstract syntax tree (AST) that hierarchically describes an application's implementation structure. Unfortunately, this method is language-specific, i.e., a parser that can construct an AST from a Java program cannot be reused to analyze an application written in Go or Rust.

In contrast to monoliths, microservices systems often use various programming lan-

¹<https://www.openapis.org/>

²<https://www.asyncapi.com/>

guages and many technologies with different kinds of configuration files (e.g., Docker, Kubernetes, API Gateway, and so on). Even if two services are implemented in the same language, they could still vary in the used frameworks or technologies [Thö15]. While this may sound odd, having such a variety is a crucial characteristic and advantage of microservice development. Teams are no longer bound to a specific language. Instead, they can choose the language they see as the best fit for their task and preferences. The ability to choose from several languages makes it also easier for companies to find experienced developers, enabling them to recruit from a larger pool of experts. However, this language and technology variety makes AST generation and analysis for microservices rather costly. Reconstructing a complete system’s architecture would require configuring and maintaining several different language parsers at once – not an easy task regarding the high update frequency of some languages like, for instance, C# or Typescript. This, and probably their popularity, might be the main reason that, so far, most research in this area focuses only on Java-based microservices (see, for instance [BDC22, FS16]).

Complementary to the static analysis, to *analyze runtime behavior*, we realize a *benchmark-based model* by executing a benchmark application representative of real-world microservices-based systems. By generating relevant data after a long benchmark run, we can build models to help us understand its behavior. Also, this can solve issues of the static analysis described above, like the validation of configuration artifacts used and the burden and costs of writing different parsers for different programming languages.

Our research evaluates the impact of API patterns on the quality properties of microservice systems, including reliability and performance [MZ21, MZP22, MZ23] using analytical and regression modeling. We have proven that the *Request Bundle* generally has a positive impact on performance and provided robust models that can be customized to predict its performance impact on different applications and infrastructure settings. We have also proposed an analytical model that predicts the impact of *Rate Limit* on reliability by calculating the ratio of the requests’ failure rate over the success rate. In our latest research, we further combined the two patterns above with *Load Balancing* and analyzed the impact of that combination on the performance and reliability of microservice systems. We produced highly significant regression models and a completely automated method that can be used iteratively and in multiple configurations and application environments.

Like the static perspective, the runtime analysis perspective also presents many challenges to getting accurate results acceptable to developers. First, a benchmark microservices-based application has to be selected to be a functional representative system that meets the experimentation requirements. This can be challenging, especially when the application needs to be customized to meet particular needs. Thus, the developer must also acquire the proper programming language skills, which could make the benchmark selection much more difficult. Second, that perspective requires running the microservices-based system for a considerable time to generate datasets that are representative enough, which consumes a lot of hardware and memory resources and higher costs. Of course, while ongoing, the experiment has to be continuously monitored to ensure its smooth execution. This requires the availability of the developer, who must solve technical problems that usually occur promptly to ensure the experimentation is on time. Last but not least, collecting the needed data and analyzing them requires considerable effort and high proficiency in automation and scripting. This is a big challenge as the dataset generated is usually huge, and analyzing it would also require a lot of hardware and memory resources, which are costly. These actions would need to be realized continuously and iteratively to adapt to changing requirements and configurations of the benchmark, which refers to self-adaptation. The self-adaptation itself presents a significant challenge as it requires the development of robust algorithms with the appropriate set of parameters and conditions.

10.2 Microservice API Analytics

Overcoming these limitations when analyzing and evaluating the quality of microservices is the primary goal of our research. Our new approach focuses on the public API of each service. It consists of several new analysis strategies for each stage of the development cycle, including the planning, implementation, runtime, and maintenance phases. Each phase consists of different artifacts providing a specific view of certain aspects of a service's quality. By combining all our findings, we aim to form a holistic analytical tooling to support architects in their decision-making process and to help prevent any potential architectural problems in advance.

Our approach is based on two main concepts: (1) The analysis of static development

and configuration artifacts, such as the source code or Docker configuration files, and (2) the analysis of the benchmark execution results. That includes all parameters and values measurable during the operation of the benchmark, including traffic data and log files. The following sections provide an overview of the different approaches and how we plan to combine them into a unified analysis method.

10.2.1 Source Code Detector-based Analysis

In contrast to the static analysis approaches explained in the previous section, our work follows a different approach. We developed a technology called source code detectors – lightweight parsers that search the underlying source code artifacts only for specific keywords and structures. Our detectors rely on the Python library `pyparsing`³, which in turn relies on the concept of *Parsing Expression Grammars* (PEGs) [For04], a technique similar to regular expressions or context-free grammars but easier to implement and use. Since the effort required to write a single detector is very small, extending the analysis process to additional languages is relatively easy.

We successfully applied our detector approach to several microservice reference systems and reconstructed a model of the communication structures down to single API operations. Compared to conventional, parser-based techniques, our approach is much more language-agnostic and can significantly reduce the effort of maintaining different tools per language. In contrast to many existing methods, our approach also detects many important technical details of microservice architectures, such as synchronous communication and asynchronous interactions within the various services.

Going further, by interpreting our communication model as a graph structure, we formally defined a set of architectural metrics we used to measure and evaluate specific characteristics of a system, like dependencies between APIs that may increase the coupling between services or even result in cyclic communication calls [GZ21].

In another work, we applied our model to estimate the level of data excessively exposed by a single microservice API [GZ22]. We identified the data transferred between APIs and analyzed whether the calling APIs processed all requested data attributes or whether data was transferred unnecessarily (see Figure 10.1 for an example of this behavior). This unwarranted transfer can be problematic in many ways, as it increases the message

³<https://pypi.org/project/pyparsing/>

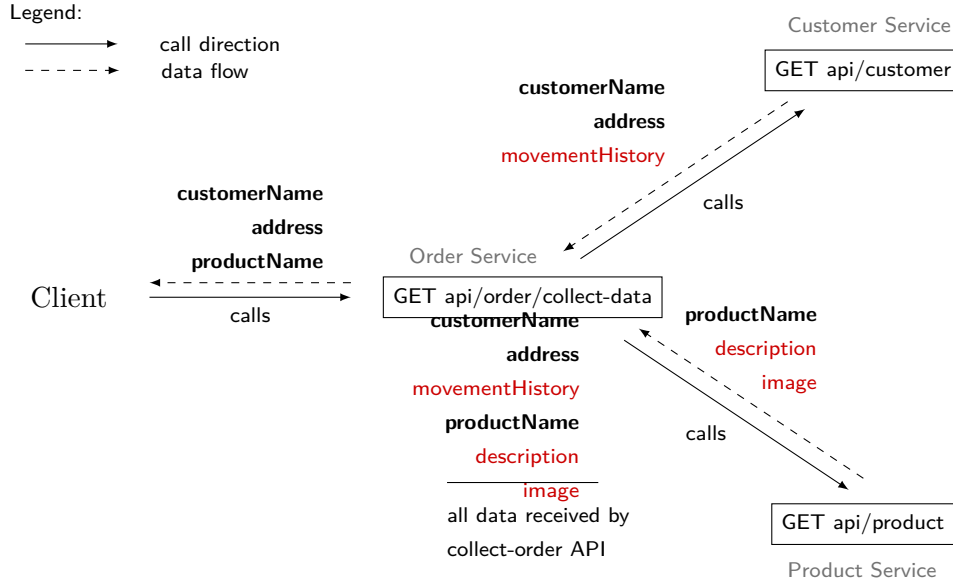


Figure 10.1: Example of Excessive Data Exposure: The *collect-data* API request more data from the *customer* and *product* APIs than required by the client. This creates an avoidable security risk [GZ22].

payload and presents an additional security risk in case of a potential data breach. We could measure the level of data exposed per API based on our findings. Our approach allows software architects to identify and reorganize data-excessive APIs to minimize data exposure. (See the *Static Analysis* section of Table 10.1 for a selection of the metrics we were able to derive from our model [GZ21, GZ22]). Of course, these metrics are only exemplary and are more of a starting point to show the possibilities of purely static analysis methods.

However, even if our detector approach might overcome some of the limitations of existing analysis methods, there are scenarios where more than static analysis of code artifacts is required to create a model expressive enough to cover all aspects of a microservice architecture. Implementations can sometimes be very complex, and although our technique can detect and follow call chains through several object hierarchies, some code constructs are out of our scope. For example, our detectors might identify a strong communication cycle between two APIs – in most cases, a severe issue as these cycles could result in a potential communication deadlock. However, the implementation may contain conditional checks that avoid these scenarios (e.g., by aborting the communication attempt after several tries). Detecting such branches in the code would require more sophisticated analysis than we currently use. It may even require the examination of

Table 10.1: Metrics description (selected metrics from MZ21, MZP22, MZ23, GZ21, GZ22)

Perspective	Metric		Description
Static Analysis	<i>cycRatio</i>		The <i>Cycle Ratio</i> for a specific API operation measures the count of cyclic communication paths originating from the API. A high value (>1) suggests strong coupling, indicating a need for further investigation.
	<i>domainCycRatio</i>		The <i>Domain Cycle Ratio</i> . Cyclic calls can either happen through direct calls between two APIs or by transferring domain events asynchronously. Asynchronous transfers are harder to track as they're implicit in domain logic. This metric helps detect such cycles.
	<i>ede</i>		<i>Excessive Data Exposure</i> is a metric to calculate the amount of data attributes an API receives unnecessarily, i.e., the API receives data but does not process it in any form.
	<i>edes</i>		<i>Excessive Data Exposure Severity</i> extends the previous metric. Each exposed attribute is ranked as low, medium, or high severity. High severity exposes in a breach could lead to attackers accessing sensitive data.
Runtime Analysis	Input Metrics	<i>rpm</i>	The number of requests per minute for API users.
		<i>api_users</i>	The number of API users sending the requests.
		<i>request_bundle</i>	Categorical variable indicating whether request bundling is used or not.
		<i>inmem_calls</i>	Number of in-memory operations when executing a function.
		<i>db_time</i>	Time spent in milliseconds by database operations when executing a function.
		<i>dist_calls</i>	Number of distributed calls when executing a function.
		<i>method</i>	Categorical variable indicating the method executed by API users.
	Derived Metrics	<i>total_time</i>	The total roundtrip time spent for requests between API client and backend.
		<i>failure_rate</i>	The percentage failure rate of API requests.
		<i>NRL_FR</i>	Per-minute number of requests failed for other reasons than rate limiting.
		<i>SR</i>	Per-minute number of requests succeeded.
		<i>RL_FR</i>	Per-minute number of requests failed due to rate limiting.

different artifacts like source code, container compositions, or other configuration files at once. Also, more than static analysis is needed to track the frequency of a specific API call. Suppose our investigation revealed that a single API operation uses a large payload that could significantly affect the system's performance. During production, it could still turn out that, since clients call this API only occasionally, the problem would be negligible. These and other cases show scenarios where static analysis methods can be insufficient and are better combined with additional strategies, such as runtime analysis, to achieve optimal results.

10.2.2 Runtime Benchmark-based Model for Self-adaptation

Analyzing microservices API at runtime is a big challenge, especially when hundreds or thousands of them are involved⁴. Service meshes [MZ19] are notable solutions used by major cloud providers like Google⁵ to support analytics. They provide monitoring tools and dashboards like Grafana⁶ and tracing tools like Zipkin⁷ to improve the observability of such complex environments. Unfortunately, these tools need to be extensively configured, and dedicated teams have to extract the necessary data and analyze them using a third-party machine learning tool, which is costly and error-prone. Also, microservices API usually evolve and frequently change due to new business needs or infrastructure requirements. This has to happen smoothly without degrading service availability and performance. So, there is a need for a continuous process that takes care of such sudden changes and evolutions with minimum human intervention.

In our approach, we have defined various exemplary, widely-used API patterns and measured their impact on quality properties like performance and reliability. The *Request Bundle* [MZ21] pattern consists of grouping several API requests into a single one. This would avoid chatty communication between clients and servers. However, this could also generate a large payload that could quickly cause traffic congestion. Also, clients and servers must support the *Request Bundle* pattern, which adds up a layer of complexity. To accurately measure the performance impact of the pattern, we developed a regression model using predefined parameters (some of them are listed in Table 10.1). We validated

⁴<https://www.cloudzero.com/blog/netflix-aws>

⁵<https://cloud.google.com/anthos/service-mesh>

⁶<https://grafana.com>

⁷<https://zipkin.io>

the model using 30 different workload settings on our University’s private cloud. We have also studied *Rate Limit* [MZP22] pattern and its impact on reliability from the API client perspective. The pattern protects API services from unwanted bursts of requests, which could increase their reliability. However, selecting the right value of rate limiting is not trivial, as simply reducing it too much might be too restrictive and thus unreliable. To find a balanced value of *Rate Limit*, we developed an analytical model based on the workload on the client side and Bernoulli distribution of success and failure rates on the server side. We validated the model using 100 configurations on the private cloud and Google cloud (GCP⁸).

Combining the two previously described patterns with *Load Balancing* to evaluate both the performance and reliability impact on Microservice systems was the subject of one of our latest research [MZ23]. In this study, we generated several robust regression models based on parameters shown in Table 10.1. Each of these models is specifically adapted to a particular pattern combination scenario. We validated those models on the private cloud and Amazon cloud (AWS⁹) and provided a completely automated method that can be applied to various application and configuration settings.

In our experimentation, we used a modern infrastructure Istio¹⁰ service mesh to deploy and analyze our selected benchmark application. A service mesh is a dedicated network layer that aims to facilitate developing and maintaining services and microservices. Indeed, the developers do not need to care about the network complexity, which is the responsibility of the service mesh. It provides resiliency features like timeouts and retries, enforcing security using mutual TLS, A/B testing, and canary deployments using traffic shifting. Service meshes are composed of *service* or *edge* proxies called *sidecars* that implement those features out of the box. One notable proxy is Envoy¹¹, which Istio uses. The proxy can also play the role of *front proxy* or *API gateway*. Those proxies are managed by a central *control plane* or *data plane* that distributes configurations and security policies, collects telemetry for observability and monitoring, and so on. There exist several flavors of service meshes with different settings and philosophies. Some can

⁸<https://cloud.google.com>

⁹<https://aws.amazon.com>

¹⁰<https://istio.io/docs/concepts/what-is-istio/>

¹¹<https://www.envoyproxy.io>

be installed on-premise like Istio and Linkerd¹² and others in public clouds like Anthos¹³ on GCP.

We can understand a system's behavior by running it for a long time and generating relevant data. We developed models based on selected data generated by infrastructure logging on both the client and server sides. These data were then analyzed using the R language¹⁴. We used Bernoulli distribution and time series to develop an analytical model to measure the impact of *Rate Limit* on the reliability of microservices from a client perspective [MZP22]. The model is described using equations (10.1) and (10.2). Analytics tools support provided time series data to build that model. We deployed a benchmark e-commerce application on top of Istio¹⁵ service mesh to generate the data. The experiment was executed on the private cloud and the GCP cloud. Some defined input metrics in Table 10.1 were considered in the model's development, like *rpm* and *api_users*. Different values for those metrics were set as input for the empirical validation of this model to cover a wide range of possible workload scenarios. Derived metrics like *NRL_FR* help users evaluate our reliability approach, for example. The model's prediction error stands at the maximum of 17% in both environments, which is reasonable in highly versatile environments like the cloud.

$$\begin{aligned}
 & \overbrace{\sum_{t=1}^{n_t} \frac{P_{failure}^t}{TU} \cdot FR_{failure}^t}^{\text{non-rate-limiting failure rate}} = \\
 & \text{workload} \left\{ \sum_{i=0}^{\infty} SU_i \cdot RPM_i \cdot \frac{\frac{LOAD_i}{RPM_i}}{\frac{LOAD_i}{RPM_i} + SLEEP_i} \right. \\
 & \left. - \underbrace{\sum_{t=1}^{n_t} \frac{P_{success}^t}{TU} \cdot SR_{success}^t}_{\text{success rate}} - \underbrace{\sum_{t=1}^{n_t} \frac{P_{ratelimit}^t}{TU} \cdot FR_{ratelimit}^t}_{\text{rate-limiting failure rate}} \right. \quad (10.1)
 \end{aligned}$$

Such that:

$$\begin{aligned}
 NRL_FR &= \sum_{t=1}^{n_t} \frac{P_{failure}^t}{TU} \cdot FR_{failure}^t \\
 SR &= \sum_{t=1}^{n_t} \frac{P_{success}^t}{TU} \cdot SR_{success}^t \\
 RL_FR &= \sum_{t=1}^{n_t} \frac{P_{ratelimit}^t}{TU} \cdot FR_{ratelimit}^t
 \end{aligned} \quad (10.2)$$

We also developed a regression model to evaluate the impact of *Request Bundle* on the performance of microservices this time [MZ21]. We implemented the pattern using

¹²<https://linkerd.io>

¹³<https://cloud.google.com/anthos/service-mesh>

¹⁴<https://www.r-project.org>

¹⁵<https://istio.io/docs/concepts/what-is-istio/>

Table 10.2: Models' description

Model	Intercept	Inmem_time	inmem_calls	db_time	db_calls	dist_time	dist_calls	request_bundle(Y)	method (Get)	method (Update)
1	-54,585	1,506	X	4,074	X	1,192	X	-312,011	331,758	312,936
2	-94,019	1,795	-0,081	X	X	0,968	17,01	-76,891	226,742	330,915

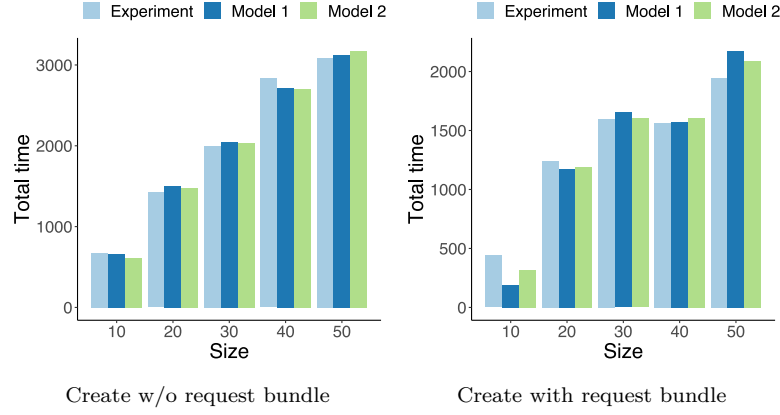


Figure 10.2: Models (1 & 2) vs experiment using Create method

a real-life application called Lakeside Mutual¹⁶. We defined parameters relevant to performance measures, including *db_time*, *inmem_calls*, and some categorical variables like *request_bundle* (see Table 10.1). The best models, described in detail in Table 10.2, perform well with less than 21% *total_time* prediction error. Figure 10.2 compares the models' predicted performance data with the experimental data when triggering the *Create* method.

For example, using Model 1 from Table 10.2, *total_time* for the method *Get* using *request_bundle*(Y) can be calculated as:

$$\begin{aligned}
 total_time &= -54,585 + 1,506 \times inmem_time \\
 &\quad + 4,074 \times db_time \\
 &\quad + 1,192 \times dist_time \\
 &\quad - 312,011 + 331,758
 \end{aligned} \tag{10.3}$$

As in real life, those patterns are not employed individually and are usually combined, so we are building a more general approach that combines the abovementioned patterns, which can also be applied to other patterns and aspects. The approach is based on an API recommendation pipeline based on a benchmark suite, which will help to alleviate

¹⁶<https://github.com/Microservice-API-Patterns/LakesideMutual>

¹⁶X means that the corresponding variable is omitted

the challenges discussed in the previous section. That self-adaptive system takes care of the whole process, from extracting and analyzing data to generating relevant information for early architecting decisions. This system is based on a benchmark suite executed against a given configuration and setting and generates relevant data. These data are then analyzed using the modeling approach described above to generate API recommendation algorithms, which are then stored for reuse. More detailed information is provided in our study [MZ23](#).

10.3 The combined approach for microservices API-analytics

As discussed in the previous section, coping with microservices architectures' challenges is not trivial. From a runtime perspective, we propose a self-adaptive system based on a benchmark suite that automatically generates new configuration data from the real-time usage data and logs. The benchmark suite is executed using a provided configuration and setting to generate a large dataset from which relevant data can be extracted. These data are then analyzed using the modeling approach described in the previous section to generate new API recommendation algorithms. Those algorithms are then mapped to the specific configuration and settings provided and stored in a repository for reuse. The benchmark suite is installed on top of a service mesh or similar framework that provides several functionalities, like observability and monitoring tools. Running the benchmark suite will generate the data and logs needed for the analysis phase using different applications and configuration settings. We develop regression and analytical models that analyze these data and provide real-time API reconfiguration recommendations. This approach would not only alleviate several challenges concerning the runtime perspective by omitting the burden of installation and configuration, for instance. It will also make valid configuration files and logs easily accessible for static analysis.

Figure [10.3](#) illustrates the combination of the data generated by the benchmark suite experiment and metrics from static analysis to define the model. The combined model is then executed using the target configuration parameters to generate indicators of quality properties like performance and reliability. The software architect could finally use those indicators to fine-tune API configuration continuously until reaching his performance or

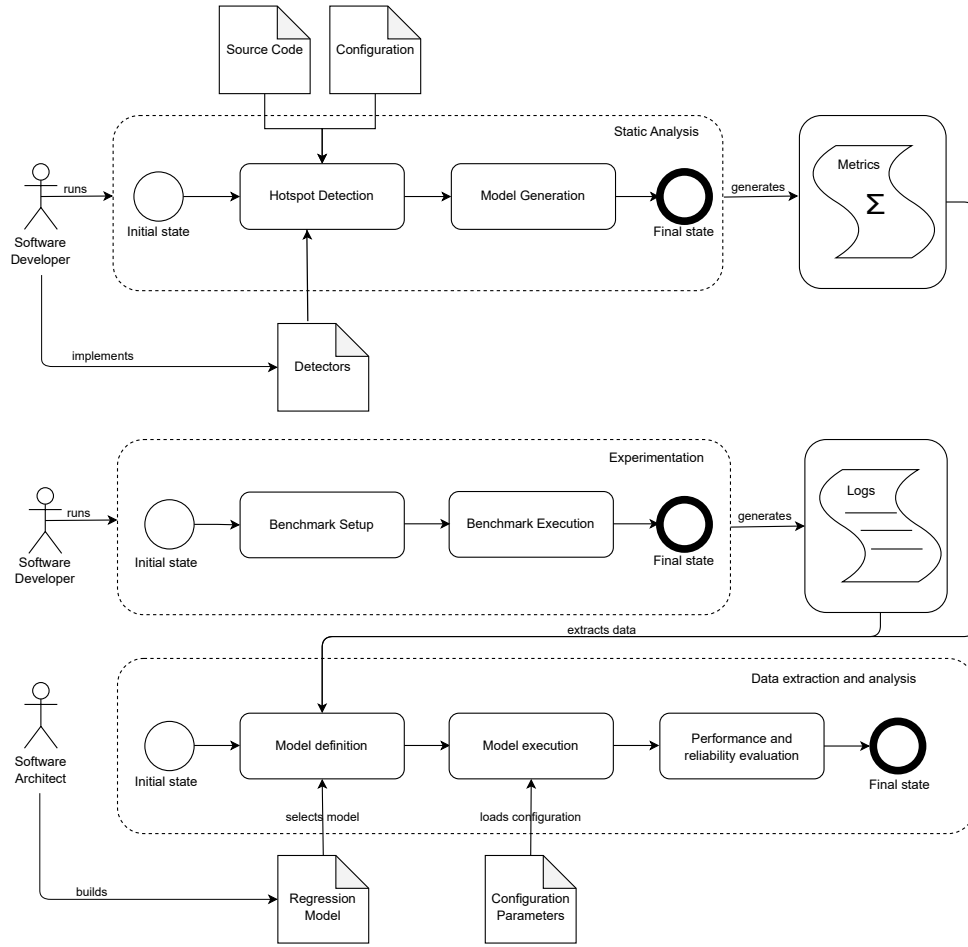


Figure 10.3: Approach Overview: The insights from the static analysis process are combined with the results from various benchmark runs to form a holistic quality model, allowing software architects to make more qualified architectural decisions.

reliability target.

As a concrete example for combining the static and runtime analysis perspectives, consider a microservice system with rigid constraints on performance and payload – e.g., an online social media platform that has to serve several thousand or even millions of user requests per second. In the first approach, our static analysis technique could identify the data excessively exposed between APIs. Stripping all these data attributes away from the payload would narrow the transfer to only the information strictly necessary for the concrete use cases. But an integral aspect of network communication – the message overhead, like protocol and header information – is currently not included in our analysis.

This payload may depend on the used protocol and its concrete implementation in lower abstraction levels. Thus, its impact is not trivial to determine by reading only the source code on the application level.

From the runtime analysis perspective, we define the *Request Bundle* pattern, already described in Table 10.1 as a categorical variable along with *bundle_size*. The pattern, which can be detected using static analysis, brings up the abstraction needed to alleviate the abovementioned issues by providing a platform-agnostic model to evaluate the impact of using or not using *Request Bundle* on the performance of a microservice-based system. In addition, this would exhibit a clear tradeoff between the transmitted data's security, avoiding unnecessary data transfer, and its actual impact on the backend's performance so that software architects have enough accurate information for better decision-making.

By combining architectural metrics from static analysis to runtime data, we would further reduce the uncertainty of using or not *Request Bundle* since we could measure its impact on performance in advance concerning the security of the exchanged data.

10.4 CONCLUSION

Microservice architects face many challenges and complexity that they must deal with promptly. Technologies like service meshes significantly overcome most of the daily issues encountered by providing advanced resilience, security, and observability features. However, they still need tremendous work and human intervention to keep everything running smoothly, which is costly and error-prone. This chapter proposes a self-adaptive approach with a human architect in the loop that combines run-time and static analytics to provide software architects with API reconfiguration recommendations for better and faster decision-making, which answers the last part of **RQ3**.

In future work, we plan to build a fully automated API recommendation pipeline based on a benchmark suite and a knowledge catalog. This catalog is reconstructed when a new configuration or application is set in the benchmark suite. Otherwise, it is reused for faster output and higher efficiency.

Part V

Conclusions

11 Conclusion

In this doctoral thesis, we have addressed all the problems described in Section 3.1 by answering research questions detailed in Section 1.3. Here, we are wrapping up all our contributions (see Section 3.2) that have built up the major parts of the dissertation.

In Part II, we conducted a qualitative investigation focusing on service mesh established practices and introduced a formally defined ADD (Architectural Design Decision) model. Our study involved analyzing 14 architectural design decisions, encompassing 47 decision outcomes and 77 decision drivers based on our research findings. Through our uncertainty reduction assessments, we demonstrated that organizing knowledge within our ADD model can significantly decrease uncertainty. Our future endeavors include integrating our ADD model with other aspects of microservice design and DevOps practices and empirically validating a theory derived from our initial uncertainty reduction estimates. Additionally, we aimed to validate our ADD model through real-life case studies involving field practitioners. Furthermore, our investigation revealed various architectural design decision possibilities in implementing service meshes, particularly highlighting the robust support these technologies offer for API pattern implementation, which answers RQ1.

In Part III, we distributed valuable insights to system engineers and architects into selecting an appropriate rate limit under various workload conditions. Our findings suggest that implementing *Rate Limit* can enhance the reliability properties of APIs under specific workload scenarios. However, achieving the optimal balance between improving success rates and maintaining failure rates at an acceptable minimum level poses a non-trivial challenge. We have taken the initial step toward addressing this challenge by developing and empirically testing an analytical model capable of accurately predicting the impact of configurations and workloads on these variables. Moreover, our model offers a robust approach for dynamically fine-tuning rate limits. Additionally, we have established a readily deployable service mesh-based infrastructure for evaluating

11 Conclusion

Rate Limit along with its various implementations. This work answers RQ2 for the *Rate Limit* specific part.

On the other hand, we introduced a regression model designed to gauge the performance impact of the *Request Bundle* pattern and a regression model to predict the performance and reliability implications of combining those API patterns with *Load Balancing*, which completes answering RQ2. Subsequently, we validated these models through experimentation with a real-world application deployed in a cloud environment. In addition to offering valuable experimental insights, our models are applicable during the early stages of system architecture design. Our approach presents a comprehensive and largely automated methodology for environments requiring precise numerical values facilitated by a highly configurable service mesh-based infrastructure, which can be iteratively applied across various contexts and settings.

It is worth noting that while these models offer accurate estimations, they may not fully account for network infrastructure imperfections such as latency and unforeseen errors, which were not explicitly considered during the development of the analytical and regression models. However, these estimations typically suffice for architectural design decisions made during the early stages of a project. We have also devised an approach to evaluating these API patterns and their combinations within specific workload settings, applications, and other optimization techniques and patterns. This approach is bolstered by a service mesh infrastructure that furnishes essential functionalities, including rapid deployment and monitoring, thereby supporting comprehensive evaluation and implementation of various optimization strategies.

In Part IV, we further developed that approach to answer the last RQ3 by deriving algorithms from decision tree models to determine the optimal combinations of these patterns regarding their performance and reliability impacts and proposing a comprehensive API design recommendation pipeline. This pipeline automates the generation of these algorithms by iteratively re-running the benchmark generated in Part III using new settings and configurations supported by the service mesh-based infrastructure. Once validated, these algorithms are stored in a catalog for future reuse should the same configuration setting be encountered. Additionally, our generated algorithms can provide reliable estimates for both private and public cloud scenarios, akin to our experimental setup, thus offering valuable design guidance. Furthermore, our approach

can be employed in self-adaptation scenarios, enabling automatic adjustments based on our recommendations. Last but not least, leveraging a combination of this approach and static analysis would further enhance the speed and quality of decision-making processes in API reconfiguration for software architects.

Bibliography

- [AGM21] Sachin Ashok, P. Brighten Godfrey, and Radhika Mittal. Leveraging service meshes as a new network layer. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, HotNets '21, page 229–236, New York, NY, USA, 2021. Association for Computing Machinery.
- [All21] Stephen Allright. Mae vs mape, which is best?, June 2021.
- [AP20] Scott Addie and David Pine. Bundle and minify static assets in asp.net core, Februar 2020.
- [AY16] David Ebo Adjepon-Yamoah. cloud-atam: Method for analysing resilient attributes of cloud-based architectures. In Ivica Crnkovic and Elena Troubitsyna, editors, *Software Engineering for Resilient Systems*, pages 105–114, Cham, 2016. Springer International Publishing.
- [Bag20a] Harry Bagdi. Getting started with kong ingress controller, 2020.
- [Bag20b] Harry Bagdi. Integrate kong ingress controller with prometheus/grafana, 2020.
- [BBd⁺16] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, May 2016.
- [BCIEDC20] Steven Bucaille, Javier Luis Cánovas Izquierdo, Hamza Ed-Douibi, and Jordi Cabot. An openapi-based testing framework to monitor non-functional properties of rest apis. In Maria Bielikova, Tommi Mikkonen, and Cesare Pautasso, editors, *Web Engineering*, pages 533–537, Cham, 2020. Springer International Publishing.

Bibliography

- [BDC22] Vincent Bushong, Dipta Das, and Tomás Cerný. Reconstructing the holistic architecture of microservice systems using static analysis. In *CLOSER*, pages 149–157, 2022.
- [BGDR17] Luciano Baresi, Martin Garriga, and Alan De Renzis. Microservices identification through interface analysis. In *European Conference on Service-Oriented and Cloud Computing*, pages 19–33. Springer, 2017.
- [BKBR10] Franz Brosch, Heiko Kozirolek, Barbora Buhnova, and Ralf Reussner. Parameterized reliability prediction for component-based software architectures. In George T. Heineman, Jan Kofron, and Frantisek Plasil, editors, *Research into Practice – Reality and Gaps*, pages 36–51. Springer Berlin Heidelberg, 2010.
- [BKR09] Steffen Becker, Heiko Kozirolek, and Ralf Reussner. The palladio component model for model-driven performance prediction. *The Journal of systems and software*, 82(1):3–22, 2009.
- [BLM21] Gloria Bondel, Andre Landgraf, and Florian Matthes. Api management patterns for public, partner, and group web api initiatives with a focus on collaboration. In *26th European Conference on Pattern Languages of Programs (EuroPLoP)*, pages 1–17, 2021.
- [Bob21] Zach Bobbitt. What is pillai trace? (definition & example), 01 2021.
- [bra21] brainCloud. Making s2s api calls from postman, 2021.
- [BW22] Ratnadeep Bhattacharya and Timothy Wood. Bloc: Balancing load with overload control in the microservices architecture. In *2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 91–100, 2022.
- [CCW98] Bark Cheung Chiu and Geoffrey I. Webb. Using decision trees for agent modeling: Improving prediction performance. *User Modeling and User-Adapted Interaction*, 8(1):131–152, 1998.

- [CGJ⁺22] Matteo Camilli, Antonio Guerriero, Andrea Janes, Barbara Russo, and Stefano Russo. Microservices integrated performance and reliability testing. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, AST '22, pages 29–39, New York, NY, USA, 2022. Association for Computing Machinery.
- [Cha14] K. Charmaz. *Constructing ground theory*. Sage, 2014.
- [Cop96] J. Coplien. *Software Patterns: Management Briefings*. SIGS Books & Multimedia, New York, 1996.
- [CRB⁺11] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, 41(1):23–50, 2011.
- [dHS11] Pieter den Hamer and Torbjorn Skramstad. Autonomic service-oriented architecture for resilient complex systems. In *2011 IEEE 30th Symposium on Reliable Distributed Systems Workshops*, pages 62–66, 2011.
- [DMM06] Antinisca Di Marco and Raffaella Mirandola. Model transformation in software performance engineering. In *Quality of Software Architectures*, volume 4214 of *Lecture Notes in Computer Science*, pages 95–110. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
- [DNWG14] Daniel J. Dean, Hiep Nguyen, Peipei Wang, and Xiaohui Gu. Perfcompass: Toward runtime performance anomaly fault localization for infrastructure-as-a-service clouds. In *Proceedings of the 6th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'14, page 16, USA, 2014. USENIX Association.
- [DP06] Nurzhan Duzbayev and Iman Poernomo. Runtime prediction of queued behaviour. In Christine Hofmeister, Ivica Crnkovic, and Ralf Reussner, editors, *Quality of Software Architectures*, pages 78–94, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

Bibliography

- [EKG⁺18] Simon Eismann, Jóakim Kistowski, Johannes Grohmann, André Bauer, Norbert Schmitt, Nikolas Herbst, and Samuel Kounev. Teastore: A micro-service reference application for cloud researchers. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 11–12. IEEE, 2018.
- [Env19] Envoy. Envoy is an open source edge and service proxy, designed for cloud-native applications, 2019.
- [FML17] Paolo Francesco, Ivano Malavolta, and Patricia Lago. Research on architecting microservices: Trends, focus, and potential for industrial adoption. In *Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption*, pages 21–30, 04 2017.
- [For04] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122, 2004.
- [FS16] Jaroslav Fowkes and Charles Sutton. Parameter-free probabilistic api mining across github. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 254–265, 2016.
- [GCH⁺04] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.
- [GDFMRC18] Antonio Gámez Díaz, Pablo Fernández Montes, and Antonio Ruiz Cortés. Fostering sla-driven api specifications. *JCIS 2018: XIV Jornadas de Ciencia e Ingeniería de Servicios (2018)*,, 2018.
- [GDFRC⁺19] Antonio Gamez-Diaz, Pablo Fernandez, Antonio Ruiz-Cortés, Pedro J Molina, Nikhil Kolekar, Prithpal Bhogill, Madhurranjan Mohaan, and Francisco Méndez. The role of limitations and slas in the api industry. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software*

- Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1006–1014, 2019.
- [GEL⁺06] Eli Gjørven, Frank Eliassen, Ketil Lund, Viktor S. Wold Eide, and Richard Staehli. Self-adaptive systems: A middleware managed approach. In Alexander Keller and Jean-Philippe Martin-Flatin, editors, *Self-Managed Networks, Systems, and Services*, pages 15–27, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- [GGLS67] Barney G. Glaser and Anselm Leonard Strauss. *The Discovery Of Grounded Theory: Strategies For Qualitative Research*, volume 3. Aldine de Gruyter, 01 1967.
- [GKN15] Ian Gorton, John Klein, and Albert Nurgaliev. Architecture knowledge for evaluating scalable databases. In *Proc. of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA 2015)*, pages 95–104, 2015.
- [Goo20] Google. Techniques for enforcing rate limits, 2020.
- [GPHP05] K Goseva-Popstojanova, M Hamill, and R Perugupalli. Large empirical case study of architecture-based software reliability. In *Large empirical case study of architecture-based software reliability*, pages 10 pp.–52. IEEE, 2005.
- [Gup17] Arun Gupta. Microservice design patterns. <http://blog.arungupta.me/microservice-design-patterns/>, 2017.
- [Gys19] Kim Gysen. Bundle / code splitting revised, May 2019.
- [GZ21] Patric Genfer and Uwe Zdun. Identifying domain-based cyclic dependencies in microservice apis using source code detectors. In *European Conference on Software Architecture*, pages 207–222. Springer, 2021.
- [GZ22] Patric Genfer and Uwe Zdun. Avoiding excessive data exposure through microservice apis. In *European Conference on Software Architecture*, pages 3–18. Springer, 2022.

Bibliography

- [HBTCP20] Zeina Houmani, Daniel Balouek-Thomert, Eddy Caron, and Manish Parashar. Enhancing microservices architectures using data-driven service discovery and qos guarantees. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 290–299, 2020.
- [HES14] Yi Hou, Praveen Edara, and Carlos Sun. Modeling mandatory lane changing using bayes classifier and decision trees. *IEEE Transactions on Intelligent Transportation Systems*, 15(2):647–655, 2014.
- [Hig20] James Higginbotham. Cloud native cloud native api management, 2020.
- [Hig21] J. Higginbotham. *Principles of Web API Design: Delivering Value with APIs and Microservices*. Addison-Wesley Signature Series. Pearson Education (US), 2021.
- [HS17] Wilhelm Hasselbring and Guido Steinacker. Microservice architectures for scalability, agility and reliability in e-commerce. In *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*, pages 243–246. IEEE, 2017.
- [HWB17] Stefan Haselböck, Rainer Weinreich, and Georg Buchgeher. Decision guidance models for microservices: Service discovery and fault tolerance. In *Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems*. ACM, 2017.
- [Ist19] Istio. What is Istio?, 2019.
- [JPG19] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. Performance modeling for cloud microservice applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, ICPE ’19*, pages 25–32, New York, NY, USA, 2019. Association for Computing Machinery.
- [JPM⁺18] Pooyan Jamshidi, Claus Pahl, Nabor C Mendonça, James Lewis, and Stefan Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.

- [JRSG21] Sharmin Jahan, Ian Riley, Alonzo Sabino, and Rose Gamble. Towards a plug-in architecture to enable self-adaptation through middleware. In *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*, pages 214–219, 2021.
- [KAB⁺96] T.M. Khoshgoftaar, E.B. Allen, L.A. Bullard, R. Halstead, and G.P. Trio. A tree-based classification model for analysis of a military software system. In *Proceedings. IEEE High-Assurance Systems Engineering Workshop (Cat. No.96TB100076)*, pages 244–251, 1996.
- [KBBML16] Hamzeh Khazaei, Cornel Barna, Nasim Beigi-Mohammadi, and Marin Litoiu. Efficiency analysis of provisioning microservices. In *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 261–268, 2016.
- [KC20] Kong Kevin Chen. Kong ingress controller and service mesh: Setting up ingress to istio on kubernetes, March 2020.
- [KK19] Pawan Kumar and Rakesh Kumar. Issues and challenges of load balancing techniques in cloud computing: A survey. *ACM Comput. Surv.*, 51(6), February 2019.
- [KZH⁺20] Alexander Krause, Christian Zirkelbach, Wilhelm Hasselbring, Stephan Lenga, and Dan Kröger. Microservice decomposition via static and dynamic analysis of the monolith. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 9–16. IEEE, 2020.
- [LF04] James Lewis and Martin Fowler. Microservices: a definition of this new architectural term. <http://martinfowler.com/articles/microservices.html>, March 2004.
- [LH05] Oussama Layaida and Daniel Hagimont. Designing self-adaptive multimedia applications through hierarchical reconfiguration. In Lea Kutvonen and Nancy Alonistioti, editors, *Distributed Applications and Interoperable Systems*, pages 95–107, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

Bibliography

- [Li17] Shanshan Li. Understanding quality attributes in microservice architecture. In *2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*, pages 9–10, 2017.
- [LJSZ18] XiaoDong Liu, Yan Jin, YongHao Song, and XiaoFang Zhao. Queue-waiting-time based load balancing algorithm for fine-grain microservices. In João Eduardo Ferreira, George Spanoudakis, Yutao Ma, and Liang-Jie Zhang, editors, *Services Computing – SCC 2018*, pages 176–191, Cham, 2018. Springer International Publishing.
- [LLGH16] D. Li, Y. Lyu, J. Gui, and W. G. J. Halfond. Automated energy optimization of http requests for mobile applications. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 249–260, 2016.
- [LPY21] Armin Lawi, Benny L. E. Panggabean, and Takaichi Yoshida. Evaluating graphql and rest api services performance in a massive and intensive accessible information system. *Computers*, 10(11), 2021.
- [LSL19] Christoph Laaber, Joel Scheuner, and Philipp Leitner. Software microbenchmarking in the cloud. how bad is it really? *Empirical Software Engineering*, 24(4):2469–2508, 2019.
- [LSZ12] Ioanna Lytra, Stefan Sobernig, and Uwe Zdun. Architectural Decision Making for Service-Based Platform Integration: A Qualitative Multi-Method Study. In *Proc. of WICSA/ECSA*, pages 111–120, 2012.
- [LYW⁺22] Jianting Li, Guohong Yi, Bingqian Wu, Zhicao Cao, and Xiaodong Xu. Research and improvement of dynamic highly available load balancing algorithm for microservices. In *2022 International Conference on Artificial Intelligence and Computer Information Technology (AICIT)*, pages 1–6, 2022.
- [LZJ⁺21] Shanshan Li, He Zhang, Zijia Jia, Chenxing Zhong, Cheng Zhang, Zhihao Shan, Jinfeng Shen, and Muhammad Ali Babar. Understanding and

- addressing quality attributes of microservices architecture: A systematic literature review. *Information and Software Technology*, 131:106449, 2021.
- [MA02] Daniel A. Menascé and Virgílio A. F. Almeida. Capacity planning for web services; metrics, models, and methods. *Prentice Hall PTR*, 26(1), 2002.
- [MA19] G. Justy Mirobi and L. Arockiam. Dynamic load balancing approach for minimizing the response time using an enhanced throttled load balancer in cloud computing. In *2019 International Conference on Smart Systems and Inventive Technology (ICSSIT)*, pages 570–575, 2019.
- [Mad20] N. Madden. *API Security in Action*. Manning Publications, 2020.
- [MBMR21] Nabor C Mendonça, Craig Box, Costin Manolache, and Louis Ryan. The monolith strikes back: Why istio migrated from microservices to a monolithic architecture. *IEEE Software*, 38(05):17–22, 2021.
- [Mor] William Morgan. The history of the service mesh.
- [Mor17] William Morgan. What’s a service mesh? and why do i need one?, 4 2017.
- [MZ19] Amine El Malki and Uwe Zdun. Guiding architectural decision making on service mesh based microservice architectures. In *13th European Conference on Software Architecture (ECSA)*, September 2019.
- [MZ21] Amine El Malki and Uwe Zdun. Evaluation of api request bundling and its impact on performance of microservice architectures. In *IEEE International Conference on Services Computing (SCC 2021)*, September 2021.
- [MZ23] Amine El Malki and Uwe Zdun. Combining api patterns in microservice architectures: Performance and reliability analysis. In *IEEE International Conference On Web Services (ICWS 2023)*, July 2023.
- [MZP22] Amine El Malki, Uwe Zdun, and Cesare Pautasso. Impact of api rate limit on reliability of microservices-based architectures. In *16th IEEE International Conference on Service-Oriented System Engineering (SOSE2022)*, 2022.

Bibliography

- [New15] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly, 2015.
- [NLL⁺15] Yipei Niu, Bin Luo, Fangming Liu, Jiangchuan Liu, and Bo Li. When hybrid cloud meets flash crowd: Towards cost-effective service provisioning. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 1044–1052, 2015.
- [NLL18] Yipei Niu, Fangming Liu, and Zongpeng Li. Load balancing across microservices. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 198–206, 2018.
- [NZP⁺20] Evangelos Ntontos, Uwe Zdun, Konstantinos Plakidas, Sebastian Meixner, and Sebastian Geiger. Metrics for assessing architecture conformance to microservice architecture patterns and practices. In *18th International Conference on Service Oriented Computing (ICSOC 2020)*, December 2020.
- [Pap17] Robert Paprocki. How to design a scalable rate limiting algorithm, 2017.
- [PH15] Marek Polák and Irena Holubová. REST API management and evolution using MDA. In *Proceedings of the Eighth International C* Conference on Computer Science & Software Engineering*, pages 102–109, 2015.
- [PJ16] Claus Pahl and Pooyan Jamshidi. Microservices: A systematic mapping study. In *6th International Conference on Cloud Computing and Services Science*, pages 137–146, Rome, Italy, 2016.
- [PJJ14] Jong-Hyuk Park, Hwa-Young Jeong, Young-Sik Jeong, and Min Choi. Rest-mapreduce: An integrated interface but differentiated service. *Journal of applied mathematics*, 2014:1–10, 2014.
- [PP18] K. J. P. G. Perera and I. Perera. Thearchitect: A serverless-microservices based high-level architecture generation tool. In *2018 IEEE/ACIS 17th International Conference on Computer and Information Science (ICIS)*, pages 204–210, 2018.

- [PVB⁺19] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. Von Kistowski, A. Ali-eldin, C. Abad, J. N. Amaral, P. Tũma, and A. Iosup. Methodological principles for reproducible performance evaluation in cloud computing. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.
- [PZA⁺17] Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai Josuttis. Microservices in practice, part 2: Service integration and sustainability. *IEEE Software*, 34(2):97–104, 2017.
- [PZL08] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. RESTful Web Services vs. Big Web Services: Making the right architectural decision. In *Proc. of the 17th World Wide Web Conference (WWW)*, pages 805–814, April 2008.
- [Rad11] Łukasz Radliński. A framework for integrated software quality prediction using bayesian nets. In Beniamino Murgante, Osvaldo Gervasi, Andrés Iglesias, David Taniar, and Bernady O. Apduhan, editors, *Computational Science and Its Applications - ICCSA 2011*, pages 310–325. Springer Berlin Heidelberg, 2011.
- [Ric16] Chris Richardson. Microservice architecture patterns and best practices. URL: <http://microservices.io/index.html> [accessed: 2018-03-17], 2016.
- [Ric17] Chris Richardson. A pattern language for microservices. <http://microservices.io/patterns/index.html>, 2017.
- [Ric18] C. Richardson. *Microservices Patterns: With examples in Java*. Manning, 2018.
- [Ric20] Chris Richardson. Pattern: Api gateway / backends for frontends, 2020.
- [RK16] Santosh Singh Rathore and Sandeep Kumar. A decision tree regression based approach for the number of software faults prediction. *SIGSOFT Softw. Eng. Notes*, 41(1):1–6, feb 2016.
- [RL18] Marian Rusek and Joanna Landmesser. Time complexity of an distributed algorithm for load balancing of microservice-oriented applications in the cloud. *ITM web of conferences*, 21:18, 2018.

Bibliography

- [RMR⁺18] Reginaldo Ré, Rômulo Manciola Meloca, Douglas Nassif Roma, Marcelo Alexandre da Cruz Ismael, and Gabriel Costa Silva. An empirical study for evaluating the performance of multi-cloud apis. *Future Generation Computer Systems*, 79:726–738, 2018.
- [SCASA⁺17] Mennan Selimi, Llorenç Cerdà-Alabern, Marc Sánchez-Artigas, Felix Freitag, and Luís Veiga. Practical service placement approach for microservices architecture. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2017.
- [Sch15] Marius Schulz. Bundling and minification: an introduction, January 2015.
- [SF21] Salah Sharieh and Alexander Ferworn. Securing apis and chaos engineering. In *2021 IEEE Conference on Communications and Network Security (CNS)*, pages 290–294. IEEE, 2021.
- [Sim19] J. Simpson. Everything you need to know about api rate limiting, 2019.
- [Sin18] Anurag Sinha. Batch: An api to bundle multiple rest operations, Nov. 2018.
- [Smi18] Floyd Smith. What is a service mesh?, 04 2018.
- [SMPZ23] Souhaila Serbout, Amine El Malki, Cesare Pautasso, and Uwe Zdun. Api rate limit adoption ? a pattern collection. In *28th European Conference on Pattern Languages of Programs*, July 2023.
- [Sta18] Jonathan Starkweather. Reference category and interpreting regression coefficients in r, Nov. 2018.
- [STVDH18] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215–232, 2018.
- [Sur19] Chellammal Surianarayanan. *Essentials of Microservices Architecture : Paradigms, Applications, and Techniques*. Taylor & Francis, [Place of publication not identified] :, first edition.. edition, 2019.

- [SZ21] Mirko Stocker and Olaf Zimmermann. From code refactoring to api refactoring: Agile service design and evolution. In *Symposium and Summer School on Service-Oriented Computing*, pages 174–193. Springer, 2021.
- [SZZ⁺18] Mirko Stocker, Olaf Zimmermann, Uwe Zdun, Daniel Lübke, and Cesare Pautasso. Interface quality patterns: Communicating and improving the quality of microservices apis. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, pages 1–16, 2018.
- [TAS⁺20] Leonardo P Tizzei, Leonardo Azevedo, Elton Soares, Raphael Thiago, and Rodrigo Costa. On the maintenance of a scientific application based on microservices: an experience report. In *2020 IEEE International Conference on Web Services (ICWS)*, pages 102–109. IEEE, 2020.
- [TB17] Kishor Shridharbhai Trivedi and Andrea Bobbio. *Reliability and Availability Engineering: Modeling, Analysis and Applications*. Cambridge University Press, 2017.
- [TGH18] Hong-Linh Truong, Lingfan Gao, and Michael Hammerer. Service architectures and dynamic solutions for interoperability of iot, network functions and cloud resources. In *12th European Conference on Software Architecture: Companion Proceedings*, page 2. ACM, 2018.
- [Thö15] Johannes Thönes. Microservices. *IEEE software*, 32(1):116–116, 2015.
- [TRA⁺17] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. Sieve: Actionable insights from monitored metrics in distributed systems. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pages 14–27, 2017.
- [vKES⁺18] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In *Proceedings of the 26th IEEE International Symposium*

Bibliography

- on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '18, September 2018.
- [VKZ04] Markus Voelter, Michael Kircher, and Uwe Zdun. *Remoting Patterns - Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware*. J. Wiley & Sons, 2004.
- [WCD⁺19] Erik Wittern, Alan Cha, James C. Davis, Guillaume Baudart, and Louis Mandel. An empirical study of graphql schemas. In Sami Yangu, Ismael Bouassida Rodriguez, Khalil Drira, and Zahir Tari, editors, *Service-Oriented Computing*, pages 3–19, Cham, 2019. Springer International Publishing.
- [WGNA19] Wartika Wartika, Ford Lumban Gaol, Ariadi Nugroho, and Bahtiar Shaleh Abbas. Application of bayesian regularization algorithm for evaluation of performance software complexity prediction model based on requirement. *International journal of recent technology and engineering*, 8(3):2530–2535, 2019.
- [Woh12] Claes Wohlin. *Experimentation in software engineering*. Springer, Berlin Heidelberg, 2012.
- [WWL⁺21] Hao Wang, Yong Wang, Guanying Liang, Yunfan Gao, Weijian Gao, and Wenping Zhang. Research on load balancing technology for microservice architecture. *MATEC web of conferences*, 336:8002, 2021.
- [WZ18] Jinyong Wang and Ce Zhang. Software reliability prediction using a deep learning model based on the rnn encoder–decoder. *Reliability engineering & system safety*, 170:73–82, 2018.
- [XG20] Xiaojing XIE and Shyam S. Govardhan. A service mesh-based load balancing and task scheduling system for deep learning applications. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 843–849, 2020.
- [Zim17] Olaf Zimmermann. Microservices tenets. *Computer Science - Research and Development*, 32(3):301–310, Jul 2017.

- [ZKL⁺09] Olaf Zimmermann, Jana Koehler, Frank Leymann, Ronny Polley, and Nelly Schuster. Managing architectural decision models with dependency relations, integrity constraints, and production rules. *J. Syst. Softw.*, 82(8):1249–1267, 2009.
- [ZNL17] Uwe Zdun, Elena Navarro, and Frank Leymann. Ensuring and assessing architecture conformance to microservice decomposition patterns. In *International Conference on Service-Oriented Computing*, pages 411–429. Springer, 2017.
- [ZSL⁺20a] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Uwe Zdun, and Cesare Pautasso. Microservice api patterns: Api key, 2020.
- [ZSL⁺20b] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Uwe Zdun, and Cesare Pautasso. Microservice api patterns: Rate limit, 2020.
- [ZSL⁺20c] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Uwe Zdun, and Cesare Pautasso. Microservice api patterns: Rate plan, 2020.
- [ZSL⁺21] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Uwe Zdun, and Cesare Pautasso. Microservice api patterns: Request bundle, 2021.
- [ZSL⁺22] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Cesare Pautasso, and Uwe Zdun. Microservice api patterns. <https://microservice-api-patterns.org/>, 2022.
- [ZSZ⁺18] Uwe Zdun, Mirko Stocker, Olaf Zimmermann, Cesare Pautasso, and Daniel Lübke. Guiding architectural decision making on quality aspects in microservice apis. In *International Conference on Service-Oriented Computing (ICSOC)*, pages 73–89. Springer, 2018.
- [ZWB18] Hong Zhu, Hongbo Wang, and Ian Bayley. Formal analysis of load balancing in microservices with scenario calculus. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, volume 2018-, pages 908–911. IEEE, 2018.

Bibliography

- [ZZZ⁺19] Tianlei Zheng, Xi Zheng, Yuqun Zhang, Yao Deng, ErXi Dong, Rui Zhang, and Xiao Liu. Smartvm: a sla-aware microservice deployment framework. *World Wide Web*, 22(1):275–293, 2019.