Automated Capacity Analysis of Limitation-Aware Microservices Architectures

Rafael Fresno Aranda

Doctoral Thesis

Advised by

Dr. Pablo Fernández Montes

and

Dr. Antonio Ruiz Cortés

Universidad de Sevilla

February 2024

First published in February 2024 by
Rafael Fresno Aranda

rfresno@us.es

Support: Doctoral Thesis supported by the FPU scholarship program, granted by the Spanish Ministry of Science, Innovation and Universities (FPU19/00666).

Dedicado a mis padres, Isabel y Rafael,
por haber estado ahí siempre que lo he necesitado.
Gracias por todo.

Abstract

The appearance of microservice architectures (MSAs) has been an important change in the way that systems and applications are developed. They are an evolution of the service-oriented architecture (SOA) paradigm, and have multiple advantages over traditional monolithic architectures; as an example, we can highlight the agility and speed of deployment, scalability, performance, improved maintenance or flexibility amongst others. This made MSAs more and more popular over the last years, and they have been adopted by many companies such as Netflix, Amazon or Spotify.

In the context of MSAs, it is common to use application programming interfaces (APIs), that serve as a communication mechanism between the services of the architecture. Among the various types of APIs, the most popular are RESTful APIs, which are based on the usage of HTTP requests to manage the state of data or services, known as *resources*. This approach contributes to the decentralization of services, and this is aligned with the essence of MSAs.

In this scenario, many companies have found in APIs the possibility to sell their data and functionality so that they can be used by other businesses. This new paradigm is known as *API economy*, and is defined as the set of business models and practices that revolve around the usage of public APIs. This paves the way for new ways of innovation, collaboration and revenue generation.

One of the elements of the API economy is the definition of pricings and plans. Businesses who wish to use an API must choose between different plans. These plans define a set of usage limitations for a specific price, which is usually a periodic subscription. The most common limitations are the restrictions to the number of requests that can be sent over a certain period of time. Nonetheless, there are many other limitations that depend on the domain of each API.

With the appearance of the API economy, it becomes necessary to analyze how the limitations and the price of the chosen plans have an impact on the capacity of the MSA. That is, calculating the workload that the MSA is able to handle without exceeding the limitations of the consumed APIs, as well as keeping costs within the businesses' budget.

Furthermore, it is not uncommon for businesses to offer their own APIs with their own plans to their customers. In this scenario, businesses act as *prosumers*, because they consume external APIs but also provide their own APIs. The confrontation of these two roles creates an *impedance* problem, where businesses need to carefully manage and balance the costs of the consumed APIs, while simultaneously keeping optimal offerings for their customers.

To the best of our knowledge, this analysis has never been done in the literature, and this fact opens a line of research that is interesting and also useful. Furthermore, the manual analysis of an MSA with external APIs is tedious and error-prone, so it is convenient for businesses to have some automated analysis systems. Given all of this, the main goal of this thesis is *the development of models and techniques to assist in the capacity analysis of microservice architectures that consume external APIs with limitations*. We coin this type of MSAs as *limitation-aware microservice architectures* (or LAMAs). The main results have been the following:

(I) *Definition of a model for the description of LAMAs and pricings*. This comprises: (i) analyzing a representative set of real-world APIs to know their structure and common elements; (ii) extending an existing model with new elements; (iii) proposing a new model for the description of the topology of a LAMA; (iv) defining a catalogue of operations to solve questions about pricings and the capacity of a LAMA.

(II) *Implementation of an ecosystem of tools to support the automated analysis of the capacity of a LAMA*. This includes: (i) developing a set of tools for the automated analysis of the validity of a pricing and the capacity of a LAMA; (ii) implementing a base catalogue of analysis operations; (iii) validating the tools with real-world and synthetic scenarios.

In this thesis, we present a set of models and tools that comprises the following: (i) a model to describe RESTful API pricings, and a serialization that is in line with the OpenAPI standard; and (ii) various tools to validate a pricing, automatically calculate the capacity of a LAMA and provide answers to analysis operations. Therefore, the results of this thesis are oriented to *help businesses using LAMAs in the process of making decissions according to their needs*.

Resumen

La aparición de las arquitecturas de microservicios (MSAs) ha supuesto un cambio importante en la forma en la que se desarrollan sistemas y aplicaciones. Se trata de una evolución del paradigma de las arquitecturas orientadas a servicios (SOAs), y cuenta con diversas ventajas frente a las arquitecturas monolíticas tradicionales. Por ejemplo, podemos destacar la agilidad y rapidez de despliegue, escalabilidad, rendimiento, mejor mantenimiento o flexibilidad. Esto hace que las MSAs hayan ganado popularidad en los últimos años, habiendo sido adoptadas por muchas empresas como Netflix, Amazon o Spotify.

En el contexto de las MSAs, es habitual el uso de interfaces de programación de aplicación (APIs), que sirven como mecanismo de comunicación entre los servicios de la arquitectura. Entre los diversos tipos de APIs, las más populares son las APIs RESTful, que se basan en el uso de peticiones HTTP para controlar el estado de datos o servicios, denominados *recursos*. Esta aproximación permite una mayor descentralización de los servicios, lo que coincide con la esencia de las MSAs.

Ante este escenario, muchas empresas han encontrado en las APIs la posibilidad de vender sus datos y funcionalidades para que puedan ser usados por otras empresas. Este nuevo paradigma se denomina *API economy*, y se define como el conjunto de modelos y prácticas de negocio que se centran en torno al uso de APIs públicas. Esto abre la puerta a nuevas formas de innovación, colaboración y generación de beneficios.

Una de las bases de la API economy es la definición de planes de precios. Las empresas que quieran usar una API deberán elegir entre diversos planes. Estos planes establecen una serie de limitaciones de uso a cambio de un precio específico, habitualmente una suscripción periódica. Las limitaciones más habituales son las restricciones del número de peticiones que se pueden enviar en un determinado periodo de tiempo. No obstante, existen otras muchas limitaciones que dependen del dominio de cada API.

Con la aparición de la API economy, se hace necesario analizar cómo influyen las limitaciones y los precios de los planes elegidos en la capacidad de la MSA. Esto es, averiguar la carga de trabajo que la MSA puede soportar sin exceder las limitaciones de

las APIs consumidas, así como mantener los costes dentro del presupuesto de las empresas. Además, no es poco frecuente que las empresas ofrezcan sus propias APIs con sus propios planes a sus clientes. En este escenario, las empresas actúan como *prosumidores*, porque consumen APIs externas a la vez que proveen sus propias APIs. La confrontación de estos dos roles genera un problema de *impedancia*, porque las empresas necesitan gestionar y equilibrar cuidadosamente los costes de las APIs consumidas, mientras que mantienen condiciones óptimas para sus clientes.

Hasta donde nosotros sabemos, este análisis no se ha hecho nunca en la literatura existente, lo que abre una línea de investigación interesante a la vez que útil. Además, el análisis manual de una MSA con APIs externas es tedioso y propenso a errores, por lo que es conveniente tener algún sistema de análisis automático. Por lo tanto, el objetivo principal de esta tesis es *el desarrollo de modelos y técnicas para asistir en el análisis de la capacidad de arquitecturas de microservicios que consumen APIs externas con limitaciones*. Estas MSAs las hemos denominado *arquitecturas de microservicios conscientes de limitaciones* (LAMAs). Los resultados principales han sido:

(I) *Definición de un modelo para la descripción de LAMAs y los planes de precios.* Esto comprende: (i) analizar un conjunto representativo de APIs reales para conocer su estructura y elementos habituales; (ii) extensión de un modelo existente con nuevos elementos; (iii) proponer un nuevo modelo para la descripción de la topología de una LAMA; (iv) definir un catálogo de operaciones para resolver cuestiones sobre planes de precios y sobre la capacidad de una LAMA.

(II) *Implementación de un ecosistema de herramientas para dar soporte al análisis automático de la capacidad de una LAMA*. Esto incluye: (i) desarrollar un conjunto de herramientas para el análisis de la validez de un pricing y la capacidad de una LAMA; (ii) implementar un catálogo base de operaciones de análisis; (iii) validar las herramientas con escenarios reales y sintéticos.

En esta tesis, presentamos un conjunto de modelos y herramientas que comprende lo siguiente: (i) un modelo para describir planes de precios de APIs RESTful, así como una serialización alineada con el estándar OpenAPI; y (ii) diversas herramientas para validar un pricing, calcular automáticamente la capacidad de una LAMA y dar respuesta a operaciones de análisis. Por lo tanto, los resultados de esta tesis están destinados a *ayudar a las empresas que usen LAMAs en la toma de decisiones en función de sus necesidades.*

Contents

List of Figures

Part I

INTRODUCTION

Introduction

This chapter introduces the research context and outlines the goals and contributions of this thesis. Specifically, Section §1.1 describes the concepts of the research context which frame the scope of the work. Next, Section §1.2 exposes the main problem addressed. Section §1.3 describes the main goals of this thesis as well as the research questions that support our research. In Section §1.4 we present a summary of our contributions and a list of publications, research stays and awards. Next, in Section §1.5 we provide details about the research methodology followed in this thesis. Finally, Section §1.6 describes how the contents of this thesis is organized.

1.1 Research Context

Microservice Architectures Are Blooming

As organizations continue to navigate the complexities of digital transformation, the principles of service orientation stand out as key enablers for building future-proof, adaptable and efficient software systems. For more than a decade, this paradigm change driven by the need for more agile, scalable and resilient systems, has profoundly impacted how software is designed, developed and deployed. Service orientation emphasizes the provision of software as a suite of independently deployable services, each performing a distinct function. This approach contrasts with the traditional monolithic architecture where all functionalities are tightly integrated into a single application.

Originally, service-oriented architectures (SOAs) laid the foundation for this evolution. SOA's primary goal was to break down monolithic applications into interoperable services, promoting flexibility and reuse [1, 2]. However, during the last years, microservice architectures (MSAs), a finer-grained evolution of SOAs, take these principles to the next level by focusing on small, self-contained services that can be developed, deployed and scaled independently. In fact, the current trend in this direction was facilitated by advancements in cloud computing, containerization, and DevOps practices, which provided the necessary tools and methodologies to manage these distributed services efficiently.

There are clear evidences on the benefits of MSAs. In the context of this thesis, we can highlight the following four: (i) agility and speed of deployment, as they allow teams to respond quickly to market changes and customer demands, giving them a competitive advantage [3, 4]; (ii) scalability and performance optimization, as each service can be scaled independently based on its specific resource requirements and efficiently utilizing underlying infrastructure resources [5, 6] (iii) resilience and fault isolation, as failure of one service does not necessarily bring down the entire system and it is easier to identify, diagnose, and rectify issues [3]; or (iv) improved maintenance and update cycles, as smaller codebases and independent services simplify maintenance and updates and teams can implement changes to a single service without impacting others, reducing the risk associated with deployments [7].

As a consequence of those benefits, various industries, from finance to healthcare, have embraced service-oriented architectures, especially microservices, for their digital transformation initiatives. Companies like Netflix, Amazon, and Spotify are notable examples of successful implementations of microservice architectures. These organizations have

demonstrated how service orientation can support massive scale, continuous innovation and high availability.

Moreover, service orientation plays a pivotal role in inter-organization integration, where different organizations need to link their systems and processes. By using service-oriented architectures, companies can expose certain functionalities as services, which can be consumed by other organizations. This approach simplifies integration and fosters collaboration between businesses. Specifically, in Business to Business (B2B) scenarios, where organizations interact with each other, service orientation allows for the creation of seamless, automated workflows. For instance, a supplier's inventory system can be integrated directly with a retailer's ordering system, enabling real-time inventory updates and automated order placement.

Also within the context of large organizations, service orientation aids in breaking down silos between different departments or business units. By decomposing complex systems into microservices, different teams can work on individual services with clear interfaces, enhancing collaboration and reducing dependencies [8]. This modularity is particularly beneficial in very large organizations (such as public administrations), where different departments might have varied technology stacks and development practices. Microservices allow these diverse units to integrate their systems more efficiently, leading to more cohesive and unified intra-organizational processes.

The Consolidation of an API Economy

The concept of APIs (Application Programming Interfaces) is at the heart of MSAs, serving as the fundamental mechanism for interaction and communication between the discrete services that define this architectural style. In a microservice architecture, each service is developed, deployed and operated independently, catering to a specific business function or process. APIs facilitate these services to communicate with each other and with the outside world, acting as well-defined contracts that specify how software components should interact. This design principle allows for a loosely coupled system where services can be updated, replaced or scaled without impacting the overall application. Moreover, APIs enable the MSA to be language-agnostic, allowing different services to be written in the languages best suited for their requirements. Through this central role, APIs not only empower the modularity and flexibility of microservices but also enhance their scalability, resilience and maintainability, which are key advantages that make MSAs highly effective for complex, evolving software applications.

Building upon the foundational role of APIs in microservice architectures, REST

(REpresentational State Transfer) has emerged as the de-facto standard for designing web-based APIs, owing to its simplicity, scalability, and statelessness. RESTful APIs use HTTP requests to manage the state of resources (such as data or services) on the web, making them an ideal fit for the decentralized and distributed nature of microservices. This architectural style's alignment with the principles of the web has fostered an open ecosystem of supporting tools and technologies, ranging from API gateways for managing API requests and responses, to comprehensive service discovery mechanisms that ensure dynamic routing and load balancing. The ubiquity of REST has also led to the widespread availability of developer resources, frameworks and best practices, significantly lowering the barrier to entry for implementing MSAs. Consequently, RESTful APIs not only facilitate the internal workings of microservices but also enable these architectures to seamlessly integrate with external systems and services, further extending their reach and utility in modern software applications.

In this context, the term *API economy* began to gain traction in the early 2010s as companies like Google, Amazon, and Salesforce started to demonstrate the power of APIs in expanding business operations and creating new revenue streams [9]. These companies leveraged APIs not just as a technical interface, but as a strategic asset, enabling them to tap into networks of developers and other businesses to grow their platforms exponentially. In general, during the last decade, The API economy has transformed how businesses operate, fostering new levels of innovation, collaboration, and revenue generation.

Specifically the *API economy* paradigm describes a set of business models and practices centered around the use of APIs to enable software applications to communicate with each other and leverage third-party services efficiently.

Furthermore, the API economy boosted the idea of a marketplace that emerged when companies exposed their internal services and data to external parties through web APIs. Consequently, thanks to the standardization of REST, a consolidated API market has been established composed by a number of services ready to be used with a flexible pricing model. As an example, in Fig. §1.1 [10] we can see the real pricing of the SendGrid API that corresponds to a widely used email service. Specifically, this API offers a suite of features aimed at facilitating robust and scalable email services for applications and businesses; moreover, it is designed to allow businesses to seamlessly integrate email sending capabilities into their applications, enabling automated transactional and marketing email workflows. Concerning its pricing, we can see four distinct tiers adapted to a variety of customer needs: (i) An entry-level tier, labeled *Free*, that offers a no-cost option with a daily cap of 100 emails that could be an attractive choice for individuals or small busi-

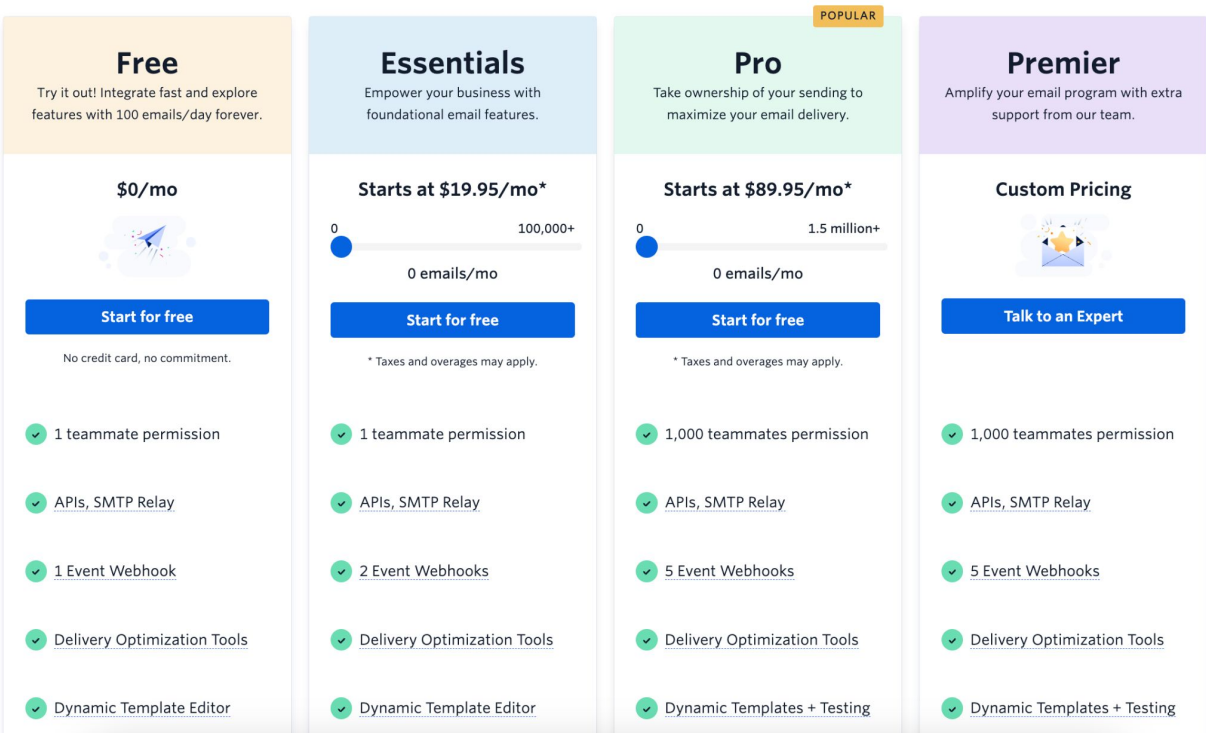

Figure 1.1: SendGrid API plans

nesses taking initial steps towards email integration. (ii) An *Essentials* tier that extends its offerings with a starting price of $19.95 per month, indicating that additional charges may apply based on usage. This plan is designed for businesses seeking more foundational email features, with the provision for over 100,000 emails per month. (iii) The *Pro* tier, highlighted as *POPULAR*, indicates a preferred choice among customers, starting at $89.95 per month with potential taxes and overages. This plan significantly expands the scale of operations, allowing for more than 1.5 million emails per month and broadening collaboration with permissions for up to 1,000 teammates. It also enriches the feature set with five event webhooks and includes dynamic templates alongside testing capabilities, suggesting a comprehensive solution for larger businesses or those with extensive email marketing needs. (iv) The *Premier* tier adopts a tailored approach with custom pricing, implying a personalized service configuration and pricing structure.

The illustrative example of SendGrid showcases how services are available in the global API market as they typically offer multiple plans with different pricing options and limitations; overall, they delineate a clear escalation in service offerings and capabilities, directly correlated with the price, addressing the expected spectrum of customers with their different needs and expectations.

1.2 Problem Statement

The Need for a Better Capacity Management

In general, the API economy brings a transformative impact to organizations by fostering new business models and enhancing customer experiences. On the one hand, it encourages a collaborative ecosystem, where businesses and developers work together, leading to enhanced service offerings and broader market reach. Operational efficiency is another significant benefit, as APIs facilitate seamless integration with other services and partners, automating processes and reducing manual workloads; furthermore these new business models allow a fine-grained outsourcing that can be dynamically adapted to in terms of different pricing types: from single request billing or periodic subscriptions, to a longer period of reserved resources. On the other hand, APIs enable more personalized services by integrating diverse services and data sources, thus responding more effectively to customer needs.

This balance of internal efficiency and external engagement through APIs allows service-based organizations to navigate the digital landscape with agility, ensuring they remain competitive and responsive to market dynamics. In fact, similarly to other industries (such as automotive or manufacturing), this landscape promotes the creation of service chains where the different organizations act as links that both consume and provide services: in this reality, from the perspective of services, organizations become prosumers that should accommodate their operation and also manage their capacity, which represents a cornerstone of their competitiveness. Indeed, the importance of the capacity management is well established by widely used service management frameworks such as ITIL [11].

This capacity management represents an important challenge that follows the pattern of the impedance mismatch problem identified in multiple domains (from the object–relational persistency in relational databases [12] to the electrical engineering circuits [13]). Specifically, in the prosumer capacity management, the potential impedance mismatch is derived from their confronted roles. On the one hand, as consumers, they rely on third party providers with their pricing and limitations. On the other hand, as providers, they deliver services to their customers, for a price and with certain limits. However, the dynamics on those different realities are intrinsically different. As service consumers, they have to select amongst the potential providers that evolve their offering (i.e. pricing and limitations) which one (or ones) is adequate from the perspective of their business model. Conversely, as service providers, they have to design and change their offering

to be competitive and grow while satisfying their customers and augment its capacity if necessary.

Specifically, in the middle of these two realities, the service prosumers need to solve the impedance of pricing and limitations by taking decisions on which is the best offering that they will put on the market to search for customers, while selecting the best plan to use of each provider. Those decision making processes represent a central part of the capacity management.

In many scenarios, organizations delivering services are not actually doing it in the context of market but there will always be a set of operational requirements that will evolve. Those requirements specify the acceptable conditions of load in which the service is required to operate (e.g. how many requests per second, for a certain endpoint, should be accommodated). As a consequence, in this case, the same impedance problem is also apparent.

The Bluejay Case Study

In order to illustrate the impedance mismatch problem of service prosumers, we can analyze the real scenario of the Bluejay framework. This framework can be used in the context of software development teams to describe and monitor best practices and has been successfully applied in undergraduate studies of Software Engineering courses and has proven to be an adequate tool to improve the understating and adherence to best Team Practices (or TPs) within the context of agile development methodologies [14]. Within the context of Bluejay, a TP can involve the analysis of information from a single single tool or the analysis of the correlation of events from multiple tools. In the first case, a typical TP for an agile team might be *over any 2-week iteration, 75% of user stories should be "1-point" stories* that can be analyzed by consuming the event log from the project management tool (such as Github Projects, Pivotal Tracker or ZenHub). In the second case, a good TP could be to check that *every time a user story is moved from the state "To Do" to the state of "In Progress", a new Git branch is created* (for the development of that user story); in this case, the analysis of the TP involves a correlation analysis between the project management tool and the Git management platform (e.g. GitHub or GitLab).

As shown in Fig. §1.2, Bluejay [1] collects evidence from different external sources using APIs (i.e. GitHub, Pivotal Tracker, Heroku, etc.), calculates metrics from them, evaluates the expected TPs, and presents the fulfillment state through visual dashboards. In such

[1] Available at https://docs.bluejay.governify.io/

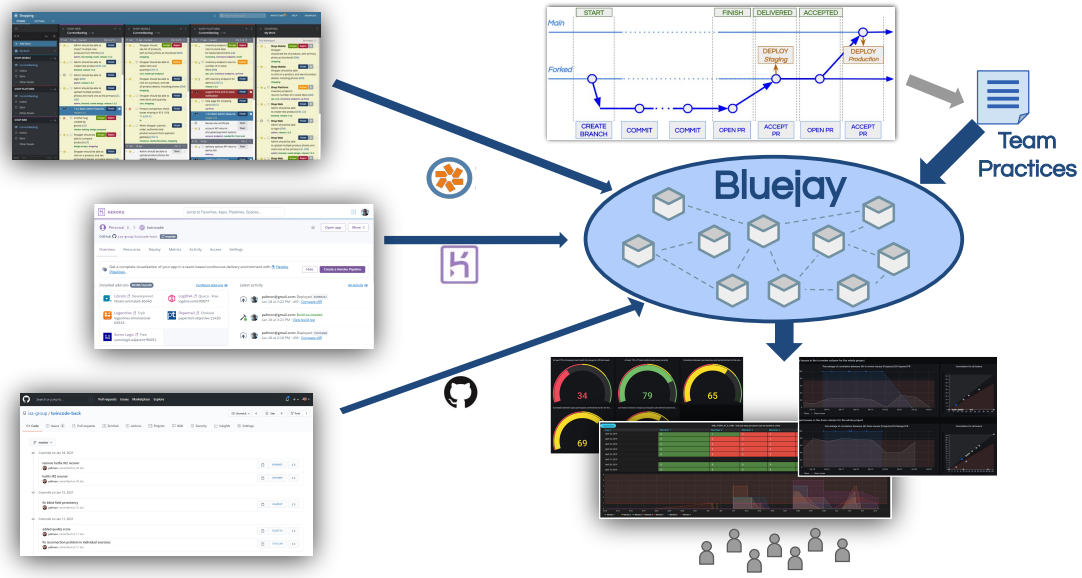

Figure 1.2: Bluejay overview

a context, every team has a dashboard that can be used to assess if the TPs are being followed and, if not, to identify which are the root causes of the unfulfilled TPs.

Consequently, Bluejay corresponds clearly with the stereotype of prosumer since it provides a service to its consumers (i.e. the development teams) while it relies on third party providers (the different third party tools) by consuming their APIs. In this duality we can ground the impedance mismatch problem as Bluejay needs to operate in balance:

- From its role of service provider, Bluejay needs to accommodate a variable workload from the different teams with several variability dimensions: the number of teams, the number of members in each team, the number of TPs to assess per team or member, or the refresh rate to update the TPs fulfillment. Moreover, the dynamics of the load evolution is highly dependant on the domain-specific scenario: e.g. a teaching environment may be organized in 3-months terms and the subjects typically define weekly iterations, while on a software factory in the industry we can find long-term projects with 3-6 week sprints.

- From its role of service consumer, Bluejay consumes the APIs by means of a set of API keys that need to be setup and billed (typically monthly) with different limitations (e.g. rates over the requests per minutes allowed) that can evolve through

time. In this case, Bluejay operators can modify the tool subscriptions at any point but the billing life-cycle will have a recurring period so the budget planning and strategy should take that into consideration.

With this confronting perspective, the impedance mismatch problem is clear as there is a need to optimally align the capacity derived from the tool subscriptions (as consumer) to the expected and current load from its users (as provider). As a consequence, in order to properly drive the capacity management of Bluejay it is paramount to answer questions such as *How many teams of 3 members can I accommodate with a monthly budget of $1,000 with a TP refresh rate of 1 hour?* or *Which is the minimal cost of accommodating 50 teams of 5 members for a 2-term subject?*.

Limitation-Aware Microservice Architectures (LAMAs)

The Bluejay use case has highlighted a common situation in service prosumers and we can coin these cases as Limitation-Aware Microservice Architectures (or LAMAs). Specifically, a LAMA needs to provide a certain offering to its customers (or guarantee certain operational requirements) while consuming external APIs with pricing plans; these plans contain usage limitations, e.g. 1,000 requests per day. If exceeded, external API providers may stop access to their services or impose additional fees for each additional request. Therefore, it is important to analyze the impact of these external limitations in the architecture.

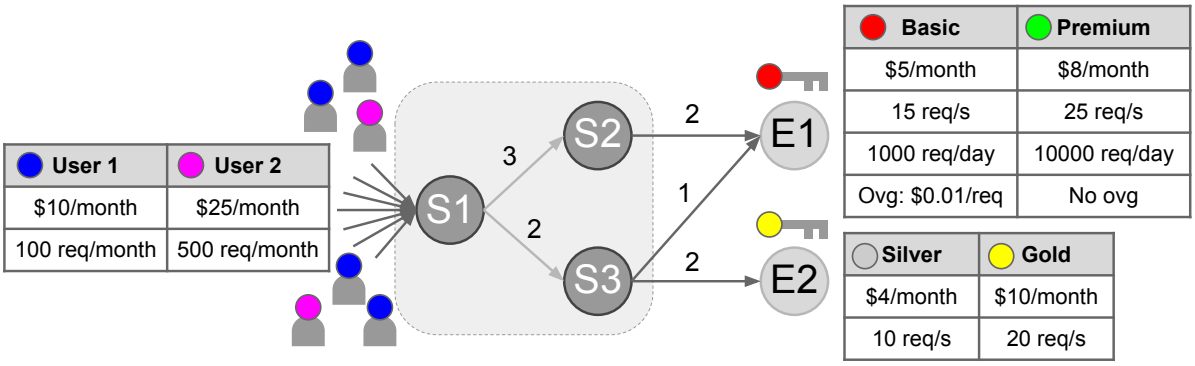

Figure 1.3: Example of LAMA with three internal services and two external APIs

In Fig. §1.3 we can find an example of a simple LAMA with three internal services (S1, S2 and S3) and two external APIs (E1 and E2). Both APIs offer various pricing plans, each of them with different limitations. The LAMA also offers its own plans, which may differ from user to user (we used generic names *User 1* and *User 2*, which can be

used by multiple users). Edge labels are read like this: S1 sends 3 requests to S2 each time S1 is invoked.

Even with a simple LAMA like this, the capacity management is not trivial and answering questions to analyze and optimize the operating expenses (or OpEx) becomes a challenge. In fact, to operate this LAMA, it would be highly beneficial to have an analysis framework that provide answers to questions like the following: *What is the operational cost for my LAMA in order to have X customers of type User 1 and Y of type User 2?* or *Which is the right pricing for my offering in order to have a profitable margin of Z?*.

Unfortunately, current approaches for the capacity management of LAMAs rely on ad-hoc analysis with a set of tools that provide low-level (usually real-time) usage metrics that are designed from a perspective of IT maintenance and are optimized for detecting and solving technical problems (such as performance leaks or bugs). In such a context, answering questions such as the ones presented before implies manual interpretation and analysis of pricings with extensive manipulation of data based on wide margins that are not precise.

1.3 Research Goal and Questions

Given the research context and the identified problem, the primary objective of this thesis can be articulated as follows:

> The main goal of this thesis is **the development of expressive models and techniques to assist in the capacity analysis of Limitation-Aware Microservice Architectures**.

In order to fulfill the objective, we address the following four Research Questions (RQs):

- **RQ1: Is it possible to describe in a structured way the information required for adequate capacity analysis?** This question explores the feasibility of devising comprehensive models that encapsulate all necessary metrics for effective capacity analysis in LAMAs that could be integrated in available standards like the OpenAPI Specification.
- **RQ2: Which analysis operations are required to support the capacity manage-**

ment? The focus here is on identifying and developing the analysis operations and techniques that are essential for informed capacity management decision-making, considering the key characteristics of LAMAs.

- **RQ3: How is it possible to monitor a LAMA, in relation to the capacity analysis?** This investigates the integration of real-time data monitored from the LAMA derived from the interactions between the different components (both internal and external), assessing which live metrics are required and how they can be extracted in a technology-agnostic way.

- **RQ4: Is it possible to improve the current capacity management process by incorporating a hybrid analytic approach (combining with real-time data)?** This question delves into the potential for augmenting static LAMA information with real-time data insights, aiming to develop a more dynamic analytical framework to make more informed and adaptive capacity management decisions for the LAMA.

1.4 Contributions

Summary of Contributions

The main contributions of this thesis are the following:

1. **Extended pricing model for RESTful APIs pricings**. We extended an already existing model to add some relevant elements that were originally missing. We named this new model *Pricing4APIs*. We also extended the SLA4OAI serialization, to include the new elements introduced in our new model. This contribution is related to RQ1, and is currently under revision in [15].

2. **Catalogue of analysis operations for RESTful API pricings**. By devising a catalogue of operations, we pave the way for future automated tools that provide solutions to these operations. Furthermore, the catalogue may be extended with more operations. This contribution addresses part of RQ2, and some operations were presented in [16].

3. **Automated capacity analysis of LAMAs**. This is the main contribution of the thesis. We developed a Constraint Satisfaction and Optimization Problem (CSOP)-based transformation to automate the calculation of the capacity of a LAMA. This transformation is particularly interesting because of the ability of CSOP to be easily

extended by simply adding new variables and constraints. We implemented a tool that automatically transforms a LAMA into a CSOP. This contribution tackles RQ2 and part of RQ4, and it was presented in [17, 18, 19, 20].

4. **Catalogue of analysis operations for LAMAs**. We devised a catalogue of operations that can be solved with the automated tool mentioned before; this catalogue can be easily extended if more variables are added to the CSOP. This contribution completes RQ2, and it was presented in [21] over the scenario detailed in [22, 23].

5. **Monitoring framework for LAMAs**. We developed a monitoring framework that collects traces and metrics from a LAMA during execution time. With this information, the topology of the LAMA can be automatically inferred. Therefore, it is easier to detect possible issues with unexpected or unwanted requests that could be otherwise difficult and tedious to debug. This contribution addresses RQ3 and part of RQ4, and it was presented in [24].

Publications

Accepted Journal Papers:

- R. Fresno Aranda, J. S. Ojeda Pérez, P. Fernández, A. Ruiz Cortés. **Governify. An agreement-based service governance framework**. *Software Impacts* (Accepted, In press) [22]. JCR IF: 2.1 (Q3 Computer Science).

International Conferences:

- A. Guerrero, R. Fresno, A. Ju, A. Fox, P. Fernández, C. Müller, A. Ruiz Cortés. **Eagle: a team practices audit framework for agile software development**. *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)* [23]. GGS class 1 (A+), ICORE rank A*.

- R. Fresno Aranda. **Automated capacity analysis of limitation-aware microservices architectures**. *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)* [20]. GGS class 1 (A+), ICORE rank A*.

- R. Fresno Aranda, P. Fernández, A. Ruiz Cortés. **SLA4OAI-Analyzer: Automated Validation of RESTful API Pricing Plans**. *Proceedings of the 23rd International*

Conference on Web Engineering (ICWE 2023) [16]. GGS class 3 (B-), ICORE rank B.

- R. Fresno Aranda, P. Fernández, A. Durán, A. Ruiz Cortés. **Semi-automated Capacity Analysis of Limitation-Aware Microservices Architectures**. *Proceedings of the 19th International Conference on the Economics of Grids, Clouds, Systems and Services (GECON 2022)* [19]. GGS class *Work in Progress*.

National Conferences:

- R. Fresno Aranda. **Easy security management over microservices architectures based on OpenAPI Specification**. *Actas de las XV Jornadas de Ingeniería de Ciencias e Ingeniería de Servicios (JCIS 2019)* [25].
- R. Fresno Aranda, P. Fernández, A. Ruiz Cortés. **Towards the automation of design time capacity analysis over microservices architectures**. *Actas de las XVI Jornadas de Ingeniería de Ciencias e Ingeniería de Servicios (JCIS 2021)*. [17].
- R. Fresno Aranda, P. Fernández, A. Ruiz Cortés. **Smart LAMA API: Automated Capacity Analysis of Limitation-Aware Microservices Architectures**. *Actas de las XVII Jornadas de Ingeniería de Ciencias e Ingeniería de Servicios (JCIS 2022)* [18].
- A. Santisteban, P. Fernández, J. M. García, R. Fresno Aranda, A. Ruiz Cortés. **Towards a Telemetry Specification for Capacity Analysis in Limitation-Aware Microservices Architectures**. *Actas de las XVIII Jornadas de Ingeniería de Ciencias e Ingeniería de Servicios (JCIS 2023)* [24]. Not rated.
- R. Fresno Aranda, P. Fernández, A. Ruiz Cortés. **A Catalogue of Analysis Operations for API Pricing Plans**. *Actas de las XVIII Jornadas de Ingeniería de Ciencias e Ingeniería de Servicios (JCIS 2023)* [21].

Journal Papers Awaiting Response:

- R. Fresno Aranda, P. Fernández, A. Gámez Díaz, A. Durán, A. Ruiz Cortés. **Pricing4APIs: A Rigorous Model for RESTful API Pricings**. *Computer Standards & Interfaces* [15]. JCR IF: 5 (Q1 Computer Science).

Research Stays

- University of California, Berkeley (United States). From January 15th, 2023 to July 15th, 2023. Supervised by professor Armando Fox. This stay was funded by the Andalusian Government (Junta de Andalucía) through a Fulbright Predoctoral Research grant.

Awards

- The paper **Easy security management over microservices architectures based on OpenAPI Specification**, presented at the national conference JCIS 2019, received an award for the *Best short paper*.

1.5 Research methodology

The research methodology employed throughout the PhD project was the Design Science Research Methodology (DSRM) [26]. This methodology is focused on creating and improving artifacts such as systems, methods, procedures and tools, with the purpose of solving specific problems. The artifacts are continuously and iteratively evaluated to improve their performance in solving the defined problem. The DSRM consists of a number of steps that must be followed to successfully apply this methodology.

Specifically, the main steps of the design science process are [27]:

Problem identification and motivation: Define the specific research problem to justify the value of developing an effective solution. Conceptually atomizing the problem can be useful so that the solution can capture the complexity of the problem. The problem addressed in this thesis is the lack of approaches and to develop an adequate capacity management analysis of LAMAs. This problem was identified through two means: firstly, based on our experience as software developers, where we were required to analyze the capacity manually; and secondly, through a literature review that confirmed the lack of solutions in the research community. The provision of tools and techniques to assist in the capacity analysis in LAMAs would be highly beneficial to the industry, as it would optimize their operating expenses.

Definition of the objectives for a solution: Infer the objectives of a solution from the problem definition. Objectives can be quantitative, qualitative, or might be rationally deduced from the problem specification. The resources needed for this include

knowledge of the state of the problems and current solutions and their effectiveness. In this thesis, the main objective is to provide techniques and tools for the capacity analysis of LAMAs.

Design and development: Create the solution artifacts, these can be broadly defined as constructs, models, methods, or instances. The desired functionality of the artifact and its architecture are determined, then the actual artifact is created. Resources needed to move from the objectives to design and development include knowledge of the theory that can be offered as a solution. In this thesis, the main artifacts are the extended Pricing4APIs and a comprehensive toolset for the capacity analysis of LAMAs that integrates various techniques and was developed during the PhD.

Demonstration: Demonstrate the effectiveness of the artifact in solving the problem. This could involve its use in experimentation, simulation, a case study, a test, or other appropriate activity. Resources needed for demonstration include effective knowledge of how to use the artifact to solve the problem.

Evaluation: Observe and measure the effectiveness of the artifact in solving the problem. This activity consists of comparing the solution objectives with the actual observed results from using the artifact in the demonstration, using pertinent metrics and analysis techniques. At the end of this activity, researchers may decide to return to *design and development* to try to improve the artifact's effectiveness or proceed with communication and leave improvements for later projects. In our case, we focused on developing and applying the model and tool suite in different real and synthetic scenarios.

Communication: Communicate the problem and its importance, the artifact, its utility and novelty, the rigor of its design and its effectiveness, through academic or professional publications, which can use the structure of this process to structure their empirical research (problem definition, literature review, hypothesis development, data collection, analysis, results discussion, and conclusion). In this phase, we disseminated our research results to the research community and other relevant audiences through presentations at top international conferences.

1.6 Document Structure

This thesis document is organized as follows:

Part I: INTRODUCTION. First, Chapter §1 includes the research context, the problem statements, the list of research goals and questions, contribution details (including publications) and the present section about the structure of this document.

Part II: STATE OF THE ART. It includes the background for the most important concepts of this thesis. In particular, Chapter §2 introduces microservice architectures and RESTful APIs, as well as how to deploy and monitor them. Then, Chapter §3 provides information about the API economy, API pricings and marketplaces. Finally, Chapter §4 defines the concept of capacity and capacity analysis, based on definitions found on ITIL.

Part III: PROPOSAL. This is the main part of the document, as it contains the contributions of the thesis. Chapter §5 introduces the exdended pricing model, as well as the validity operation and the catalogue of analysis operations. Next, Chapter §6 provides details about the concept of limitation-aware microservice architecture, including its elements. Chapter §7 describes the concept of capacity of a LAMA and introduces the CSOP transformation and the catalogue of operations. Lastly, Chapter §8 offers information about the automated monitoring framework.

Part IV: VALIDATION It contains various examples of synthetic and real-world scenarios where our proposal has been validated. Chapter §9 provides some insights on the expressiveness and automated validation of RESTful API pricings. Then, Chapter §10 introduces an automated tool to transform a LAMA into a CSOP and perform various analysis operations. Finally, Chapter §11 describes an experiment for the monitoring framework.

Part V: CONCLUSIONS. It includes the final conclusions and future work in Chapter §12.

Part II

STATE OF THE ART

Microservice Architectures

This chapter provides some background for the concepts of microservice architectures and RESTful APIs. Precisely, Section §2.1 contains an introduction to these architectures. In Section §2.2 we present the elements of RESTful APIs that are typically found in real-world APIs, as well as the de facto standard for the description of these APIs. Then, Section §2.3 introduces the basics for the operation of microservice architectures, including the deployment and monitoring of their services.

2.1 Introduction

Microservice architectures have become more popular over recent years. This is because they offer multiple benefits over other architectures which used to be more common in the past. Some of these advantages are ease of deployment and language independence.

Usually, microservices are pieces of software designed to perform a very specific task. For example, common tasks include accessing a database, communicating with an external API, rendering the user interface of a web page. Because of this, they are more maintainable and scalable than a traditional architecture, since only one specific functionality inside a greater architecture is affected. Additionally, making a change in a service means that only that service needs to be deployed again. Big architectures split their functionality over multiple microservices. To communicate between each other, some means of communication are needed. Commonly, microservices make use of *RESTful APIs* to address this.

Traditionally, systems have been developed using monolithic architectures. This means that a single application handles each and every task of the system. In contrast with microservice architectures, this approach increases maintenance difficulty. Because there is just one artifact, making changes becomes much more complex, and, therefore, adding new features or fixing some bugs mean that a complete deployment of the system is needed. Moreover, by constantly working on the same artifact, there is a high risk of causing side effects which have an impact on unintended areas of the system.

Fig. §2.1 depicts a fictitious e-commerce application that is divided into multiple microservices [28]. Each one of these services provides a solution for the tasks that the system needs to perform. In this case, there is one specific service which provides the web application which includes the user interface that will be seen by the users of the application through a browser. In addition, there are three more services that take care of the different kinds of data that the shop must store: user accounts, product inventory and shipping. As can be seen in the figure, each service communicates exclusively with its own database, and offers the remaining functionality through RESTful APIs. This figure also introduces the concept of an API gateway. Its main purpose is to proxy any request that is sent to the system, so that a user who needs to access one individual service can do so through the use of the gateway. The gateway then redirects the requests to the corresponding services. API gateways are very common in microservice architectures, because, apart from routing requests, they can be set up to implement other elements, such as security and load balancing.

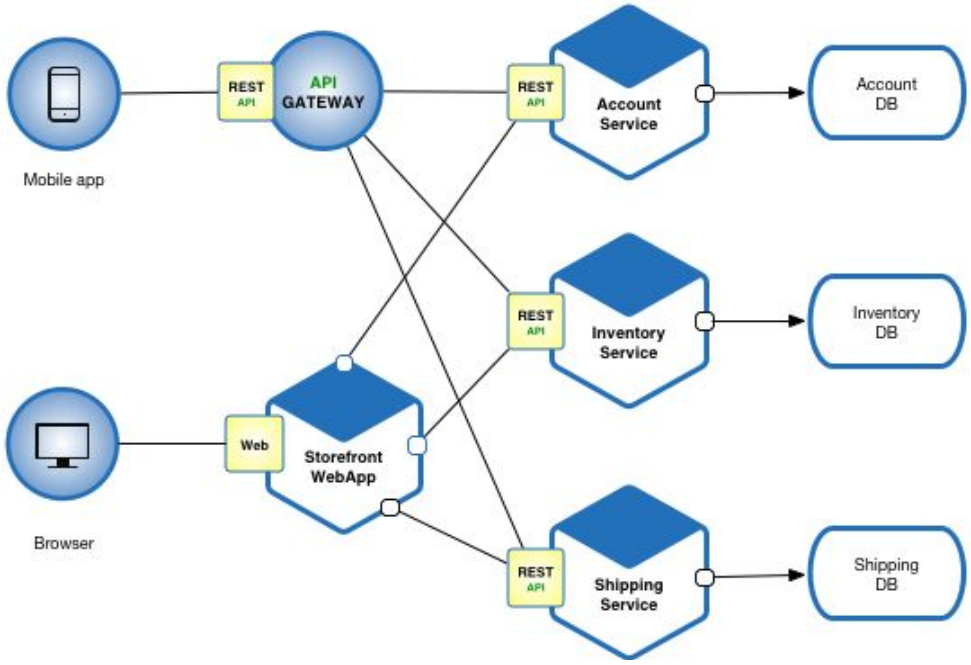

Figure 2.1: Microservice architecture example

Features

All microservice architectures have a series of features in common. These features are the ones which define this architectural style, and make it different from monolithic applications. The following list of features is extracted from an article by Martin Fowler [4], one of the most famous developers and followers of microservice architectures.

- **Componentization via services**: we try to build software so that pieces can be plugged together, just as we do in the physical world. These pieces have usually been named **components**, and they are independent units of software that can be interchangeable. In the context of microservice architectures, services behave as components. They offer some mechanisms so that other services can communicate with them. They can also be individually deployed. One drawback of this approach is that remote calls between services are slower and more expensive, in contrast with in-app operations.

- **Organized around business capabilities**: traditionally, development teams have been split into different technology layers: UI, server and database. This leads to costly and slow processes when dealing with changes, because all teams need to be notified and develop their part. In microservice architectures, each team is organized around **business capabilities**, and as such, they have the skills to take part in any layer of the full stack.

- **Products not projects**: usually, software development revolves around projects: they are devised, developed and delivered, and the development teams has no relevancy in the maintenance. Microservices developers commonly take another approach, where they build the software and also maintain it. This allows them to better understand how the software is being used, and changes can be done faster.

- **Smart endpoints and dumb pipes**: many products put great emphasis on the communication between components, developing mechanisms such as sophisticated buses. The microservices approach in this aspect is quite different. Services are built to be decoupled and cohesive, and they implement all the logic that they need to receive a request, perform a task and produce a response. They usually rely on web protocols to communicate with each other. Also, they use dumb buses, meaning that they only exist to route messages. As mentioned, all the important logic is in the service.

- **Decentralized governance**: centralizing services is not advisable, because it leads to sharing the same technologies across the entire system. Problems are not always the same and thus some tools are more appropriate than others. By decentralizing, each microservice is free to be developed using the most appropriate technology for the task. Moreover, it can also make use of the database that best suits its need. Teams using microservices prefer useful tools rather than sticking with defined standards which may not be optimal for all cases.

- **Decentralized data management**: a system may need to retrieve information including different attributes or attributes with different semantics depending on the request. Microservices let developers choose different data models for each service, so that the specific information can be retrieved without the need to transform and edit information. On top of that, microservices are able to use different databases with different technologies. One drawback is that a change in a database might need to be replicated in others. This needs some time and, depending on the architecture, additional transformations so that data is consistent across the entire system.

- **Infrastructure automation**: automation techniques have come a long way over the last years, making it possible for a piece of software to be automatically built, tested and deployed without any manual intervention. Microservices are able to make use of these continuous integration and delivery tools. Additionally, due to their small nature, they can complete the full automated pipeline much faster than big monolithic applications.

- **Design for failure**: one of the main drawbacks of microservices is that they are prone to errors and failures. A failure in a small service could mean that the complete system will not work correctly. Because of this, microservices are expected to be resilient to failures, and to gracefully recover from any of them. Development teams usually make extensive use of dashboards to control the status of each individual service. This way, they are aware of the requests they are receiving, the computer load, and a number of other relevant metrics.

- **Evolutionary design**: in microservice architectures, changes are easily made because they only affect a reduced number of services and not the entire system. A good method to be able to decide how to split a system into services is checking how many changes they are expected to undergo, and considering whether it would be easy for the service to be completely replaced if needed. Additionally, if developers often find themselves changing multiple services together, that could mean that those services should really be developed as a single microservice. In the same way, services that stay relatively stable and services with constant changes should stay separate. One downside of microservice architectures is that a change in a service might have a negative impact on the services making use of it. For this reason, it is important to develop services so that they are as tolerant as possible to changes in others.

Advantages

Microservice architectures provide an interesting catalogue of advantages over monolithic architectures, some of which were already mentioned in previous paragraphs. In the following list, we are going to cover the most important ones, which in most cases are directly related to the aforementioned features:

- **Language independence**: inside a single system, different microservices may be implemented using different programming languages. This allows each service to use the language which would be most appropriate for the task it needs to perform. For example, one service could be written using JavaScript, while another one can be implemented in Python. Because communication is done through common structures such as JSON or XML, it does not affect this fact.

- **Ease of deployment**: cloud services are greatly growing in popularity. For this reason, easier deployments are an important feature that development teams always look for. Because the small nature of a microservice (hence the 'micro' prefix), it will usually need smaller servers to be hosted. This means that deployment costs

are reduced. Additionally, adding new features or fixing bugs only requires the corresponding service to be deployed again. In contrast, a monolithic application would need to be deployed as a whole, even if the change only affected one line of code.

- **Cross-functional development teams**: when a big system is divided into smaller microservices, each one of them only needs a small development team to maintain it. Nonetheless, each team should have some knowledge about every technology layer, such as user interface, storage and external communication with other services. On the contrary, monolithic applications are managed by big teams which need to be divided according to the various technology layers. This means that a greater coordination is needed, because every change affects all layers, and therefore all teams need to participate in meetings and discussion.

- **Different databases**: instead of having a single shared database for the entirety of the system, each microservice has its own storage. This means that each service can use the type of database that suits its needs. For example, one service might need to store geographical information, so it is able to use a database with good geographical support; at the same time, another service might need to use a non-relational database over a relational one. This approach has one disadvantage, though: some pieces of information might be duplicated and scattered across the different databases.

- **Newer technology**: monolithic application have a clear drawback, which is that any new functionality is restricted to the same and usually old technology that the application uses since it was originally developed. In contrast, a new microservice which is added to an architecture may use newer technologies when needed. Additionally, older services can easily be rewritten and updated if needed.

Many well-known companies have shifted their focus towards microservice architectures, because of the aforementioned advantages. Two of the most important examples of current, known sites using microservice architectures are Amazon and Netflix. Both were early adopters of the architecture back in their day, and they have grown to be great supporters of microservices. Additionally, as an anecdote, Amazon has stated that microservices should be able to be developed by teams that can be fed only on two pizzas.

2.2 RESTful APIs

One of the most important decisions that need to be made when developing a microservice architecture is choosing the communication method between services. There are multiple approaches to do this, but, nowadays, most real-world MSAs rely on RESTful APIs for this communication.

In this approach, each service of the MSA offers a public API that follows the RESTful guidelines. Each feature of the service is provided through a different endpoint, which accepts the appropriate parameters to modify its requests. Then, the services communicate between each other through HTTP requests to these endpoints. These MSAs can be considered *RESTful architectures*.

On modern information systems, APIs (Application Programming Interfaces) can be seen as a fundamental piece of software development. Besides their use for the communication between internal services of an architecture, public APIs offered by third parties allow other businesses to make use of some functionality offered by external systems. For example, a system is able to retrieve forecast information, send emails, or upload a video by using external API requests, without the need to actually develop the corresponding functionality. Currently, there are many types of APIs across the entire range of software types; nevertheless, on the web, the most common ones are SOAP and REST APIs.

Nowadays, SOAP (Simple Object Access Protocol) APIs are usually provided by older, legacy or unmaintained services. In this kind of communication, clients and servers exchange XML messages which contain requests and responses. The use of XML as the communication language makes it harder for humans to understand. Additionally, the purpose of a request is not always clear because of how SOAP requests are structured. For these and other reasons, SOAP-based systems are currently disappearing, with most modern services being developed using REST APIs.

Concepts

In contrast, REST (REpresentational State Transfer) APIs, also known as RESTful APIs, are much more common throughout the web. The term *REST* was first used in the PhD thesis by Roy Fielding [29]. The main characteristic of RESTful APIs is that they make use of standard HTTP methods to handle different functionality. The most common HTTP methods are:

- **GET**: used to retrieve information about a certain resource.

- **POST**: used to create a new resource, with the information included in the request body.
- **PUT**: used to update an existing resource, with the information included in the request body.
- **DELETE**: used to delete an existing resource.

Objects in RESTful APIs are referred to as *resources*, which are defined in a specific domain. For example, a pet store may define a resource named *dog*, which represents a dog in the store. To retrieve information about a dog with ID 123 from the store, the corresponding request would be `GET http://api.samplepetstore.com/dogs/123`. In the same way, adding a new dog to the pet store would be done through a request similar to `POST http://api.samplepetstore.com/dogs`, where the dog information would be included inside the request body.

A URL in a RESTful API is known as an *endpoint*, and it is accessed through HTTP methods. For example, `GET http://api.samplepetstore.com/dogs/123` is an operation of the pet store API. An endpoint might have some *path parameters* to identify specific resources, such as dog with ID 123 in the previous example. Additionally, endpoints might support certain *query parameters* to modify the request. For example, the endpoint `http://api.samplepetstore.com/dogs?limit=10` might represent a list of dogs in the pet store including a maximum of 10 results. Endpoints may also support parameters inside the request headers or the request body.

Unlike SOAP APIs, where XML is used as the communication language, REST APIs usually exchange data using JSON (JavaScript Object Notation), which has a cleaner structure and is therefore easier for humans to read. However, many REST APIs also support XML, CSV or even plain text for both requests and responses. As of today, most companies (including important ones such as Google, Spotify, Instagram or X) offer a wide range of their services through public REST APIs. This way, any business is able to use the data that they provide for their own systems, as well as being able to create or update information (for example, upload a video to YouTube, post a photo in Instagram...). Additionally, many companies also offer different pricing plans. Through these plans, they can make some money by imposing restrictions over several quality-of-service attributes, such as response time, availability or number of requests per period of time. More information about pricing plans can be found in Section §3.2.

Though APIs offer multiple benefits, for both consumers and providers, they are not devoid of problems. The main one is that each API would be documented using their

own specific method or language. Therefore, there are numerous inconsistencies over different APIs, and this makes it hard for developers to create tools that are able to work with any arbitrary API. For this reason, in 2010 a project named Swagger was born. Its goal was to create a standard specification for RESTful APIs, so that every system could be documented using the same standard structure. After the first two versions were released and well received, the specification was donated to a newly created group, named the OpenAPI Initiative (OAI). This new group is sponsored by the Linux Foundation. In 2016, the Swagger specification was renamed to OpenAPI Specification (OAS), to better reflect that it was not tied to Swagger anymore. In 2017, the third version of the specification was released, and, to this day, it remains the latest version, with some minor changes made through the years.

Figure 2.2: OpenAPI Initiative members

Fig. §2.2 shows the current OAI members as of this writing [30]. The OpenAPI Initiative aims to solve the issues mentioned before. Having a standard structure to document APIs, many tools can be now developed so that they can be used with any API. There are many types of tools that have been developed around the OpenAPI Specification. Among others, there are multiple test suites generators, interactive web portals and client and server prototypers. One of the most famous tools is Swagger UI, shown in Fig. §2.3. It is an interactive web portal which lists all endpoints of an API, and allows users to send request with custom parameters and request bodies.

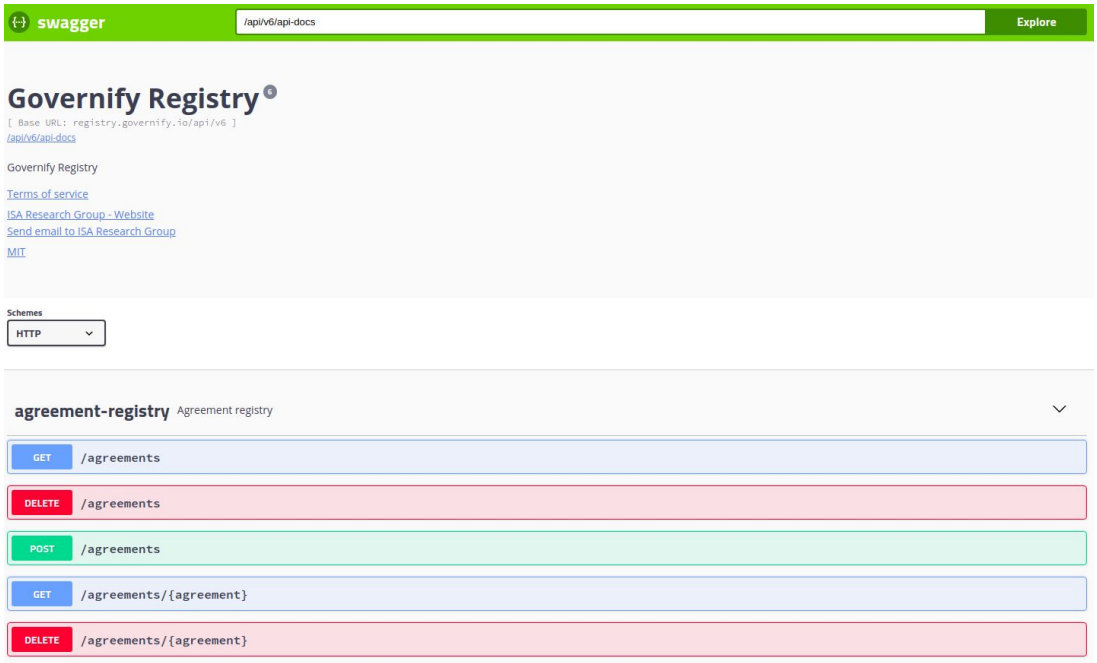

Figure 2.3: Swagger UI example

OpenAPI Specification

The OpenAPI Specification defines a schema to be followed when describing a RESTful API. It is language-agnostic, that is, it does not depend on any specific language. The schema consists of certain fields and attributes that specify general information and explain how the API works. OpenAPI descriptions can be written in JSON or YAML, which are similar and easily translatable to each other. For simplicity and readability, we will use YAML for the following examples. All examples are extracted from the official Swagger website [31], with minor modifications to fit into the listings.

First, every API description must indicate the specific OpenAPI version that is being used. As an example, an API using version 3.0.0 should contain the line in listing 2.1.

```
1 openapi: 3.0.0
```

Listing 2.1: OpenAPI version

Then, additional information about the API itself should be included, similar to the one in listing 2.2. This information may consist of a title (line 2), a description (line 3), a version number (line 4), developer emails, licenses... This is usually displayed on web portals and is not tied to the actual functionality of the API.

```
1 info:
2   title: Sample API
3   description: Optional multiline or single-line description in CM or HTML
```

```
4   version: 0.1.9
```

Listing 2.2: API information

Now, some servers may be specified. These servers refer to the ones where the API is or will be available. Each server consists of an URL and an optional description. It is important to note that, unlike previous Swagger versions, OpenAPI 3.0 does not support a separate *base path* (such as `/api/v1`), and it must be included within the server URL. Listing 2.3 shows an example with two servers.

```
1  servers:
2    - url: http://api.example.com/v1
3      description: Optional server description, e.g. Main (production) server
4    - url: http://staging-api.example.com
5      description: Optional server description, e.g. Internal staging server
```

Listing 2.3: Servers information

Next, the different API paths must be defined. API paths refer to the endpoints that said API supports. These include the admitted HTTP methods, query and path parameters, requests and response bodies, errors, status codes, etc. In listing 2.4, line 2 refers to the endpoint (to be appended to the server URL), and line 3 indicates the HTTP method. Lines 6 and below show how a response is defined.

```
1  paths:
2    /users:
3      get:
4        summary: Returns a list of users.
5        description: Optional extended description in CommonMark or HTML
6        responses:
7          '200':
8            description: A JSON array of user names
9            content:
10             application/json:
11               schema:
12                 type: array
13                 items:
14                   type: string
```

Listing 2.4: API paths

The parameters of an endpoint may be included in the URL path, as query parameters, inside headers or cookies, or as a request body. All of these option can be defined using the OpenAPI Specification. In listing 2.5, line 6 defines the name of a parameter, whose location is specified in line 7. Lines 10 to 13 indicate its type and a range restriction.

```
1  paths:
2    /user/{userId}:
```

```
3     get:
4       summary: Returns a user by ID.
5       parameters:
6         - name: userId
7           in: path
8           required: true
9           description: Parameter description in CommonMark or HTML.
10          schema:
11            type : integer
12            format: int64
13            minimum: 1
14      responses:
15        '200':
16          description: OK
```

Listing 2.5: Path parameters

Serving as a general overview, these are the basic elements to be included in an API description using OpenAPI. Nevertheless, there are many other elements that can be added, such as security measures. Some of these elements can also be parametrized.

Additionally, OpenAPI leaves the standard open, so that anyone who is interested is able to contribute. Developers may add their own fields to the specification. Usually, the convention for these additional attributes is to prepend the prefix `x-` before them.

2.3 Operation

Deployment

While microservices can be deployed in any infrastructure as any other monolithic application, they are particularly suitable for cloud deployment options. Many well-known companies offer their own solutions for the deployment of cloud services, such as Amazon (Web Services) or Microsoft (Azure). Typically, these solutions are Infrastructure as a Service (IaaS), meaning that the provider offers and maintains a cloud server and the MSA developers need to provision and manage all required software. Other platforms, such as Heroku, offer a Platform as a Service (PaaS) solution, where developers only need to upload the code of the service and it is automatically deployed. In this section, we will focus on some options offered by Amazon Web Services (AWS), as well as deployment variations using Docker and Kubernetes.

AWS Elastic Cloud Computing (EC2)[1] offers various tiers with different infrastructure

[1] https://docs.aws.amazon.com/ec2/

features, such as types of CPU and optimization for short bursts of requests. Within each tier, there are multiple options with different types of CPU, storage and RAM, among others. When a developer chooses an option, they get a key to remotely access the server, so that they can install the necessary packages and software to deploy the service. It is the responsibility of the developer to maintain and update the packages, while Amazon takes care of the underlying physical infrastructure. Fig. §2.4 shows the architecture of a sample AWS EC2 instance [32].

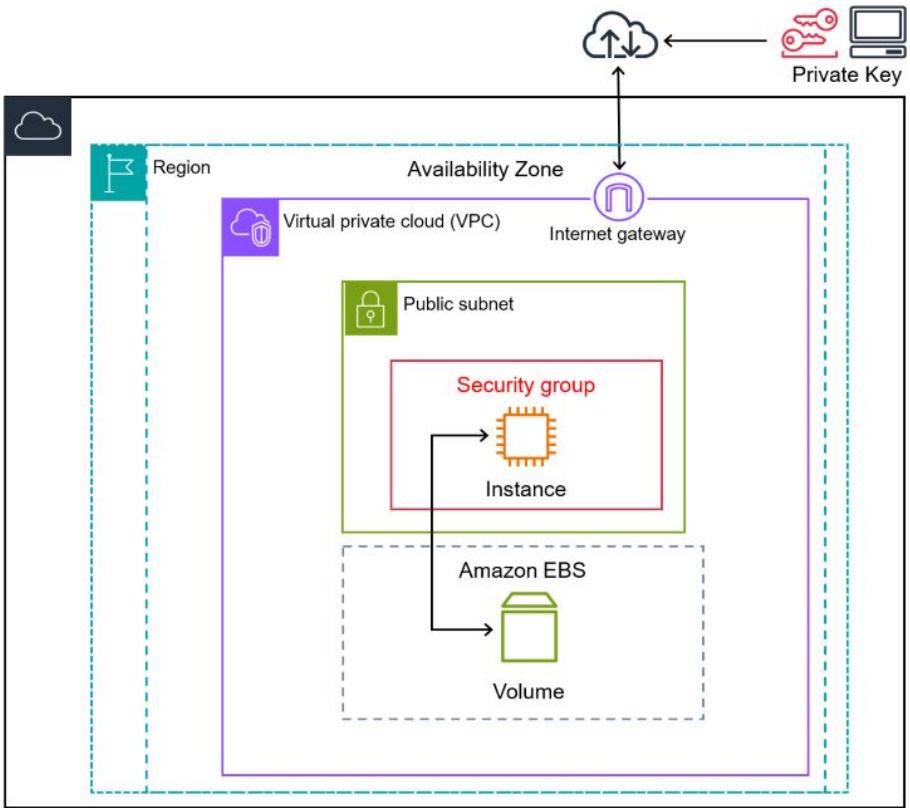

Figure 2.4: Architecture of an AWS EC2 instance deployed on an Amazon Virtual Private Network

Among the available AWS EC2 tiers, one of the most popular ones is T2. These machines are small and cheap, and are optimized for small services which are expected to receive short bursts of requests. Within the T2 tier, the most used option is the *t2.micro* instance. It is free to use for up to 750 hours of computing time, meaning that it is used by many developers who want to try AWS or deploy small services which are not expected to handle too much workload. Another option is the A1 tier, which is relatively new (released in 2018). It offers machines with ARM CPUs, which are becoming more popular over the last few years. These instances are more expensive than traditional instances based on x64 CPUs, but offer more performance in some cases.

AWS offers other alternatives to traditional IaaS solutions. In particular, a popular option is AWS Lambda[2], a *serverless* alternative that is based upon the idea of Function as a Service (FaaS). In this approach, the microservice is not a deployed as a regular server; instead, it is deployed as a single function that receives some parameters, performs some actions and, if necessary, returns some data. Furthermore, in contrast to AWS EC2, the developer does not need to manage any packages within the server. Small services deployed as functions typically run considerably faster than the same service in a standard server, making Lambda an ideal solution for microservices that perform a small and very specific task. A service using AWS Lambda may be used together with other AWS components that act as middlewares between the clients and the function. An example of this is the utilization of an API gateway, which AWS also offers (AWS API Gateway).

An alternative to the traditional way of building and sharing applications is Docker[3]. It is a well-known containerization technology that works by encapsulating a service and all of its dependencies into a single *image*, which can then be distributed and shared anywhere. This facilitates the replication of deployment infrastructures, as an image includes all requirements for a service to run and will work the same in any computer. Building and downloading images is easily done through a command-line interface, and it only takes a few seconds to share an application with other developers. There are other similar technologies that work very similarly, such as Podman, but Docker remains the main solution for the containerization of services. Fig. §2.5 depicts the architectural differences of containerized deployments versus other types of deployment [33].

A Docker image can be used to deploy a service to a traditional IaaS platform, such as AWS EC2. Instead of installing all packages and dependencies of the service in the EC2 instance, a developer would only need to install Docker and download the image of its service. If it is an MSA with multiple services, databases, etc., Docker can automatically deploy all components thanks to the use of the docker-compose tool. However, a more interesting approach is also available. Kubernetes, or K8s[4], is a platform that orchestrates containers to automatically manage their resources, balance their loads, scaling, monitoring and many other tasks. It was originally developed by Google, but is now open-source. Kubernetes works as a mix of IaaS and PaaS, giving developers the ability to control some parts of the infrastructure, while also automating many of the most common tasks.

[2] https://docs.aws.amazon.com/lambda/
[3] https://www.docker.com/
[4] https://kubernetes.io/

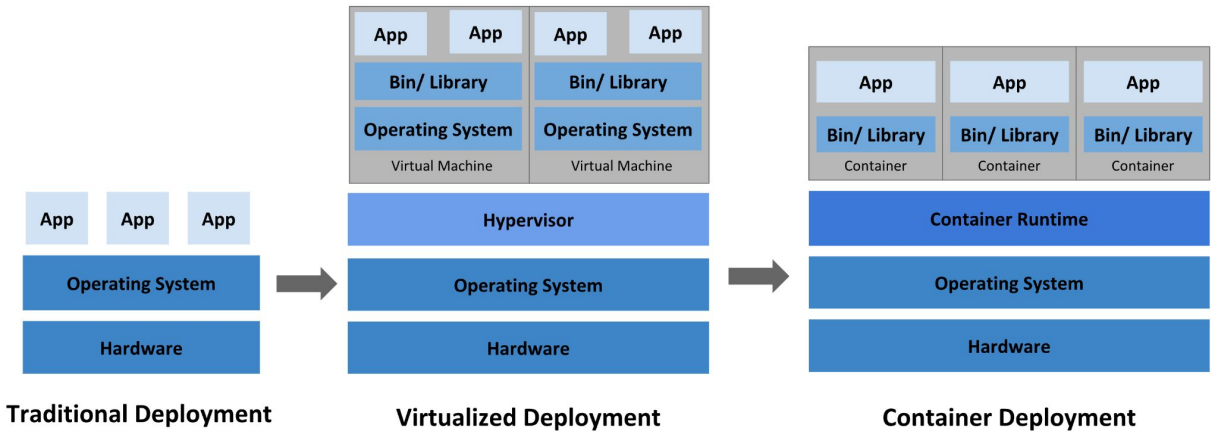

Figure 2.5: Traditional vs virtualized vs containerized deployments

Monitoring

Once the microservices have been deployed, it is necessary to ensure that each service operates and communicates correctly with other services and APIs, as each service functions independently within the MSA [34]. While research on telemetry for the Internet of Things (IoT) field has been widely explored, its application for RESTful architectures is still under-researched. Even in the IoT field, most of the telemetry data extracted from IoT devices comes from the network layer [35], while REST APIs operate and limit requests at the application layer. This is why there is a need to collect data coming from the application layer, which is referred to as *application level telemetry*.

Fortunately, there exist specifications that define application level telemetry in a standardized format, such as OpenTelemetry [36], which provides a vendor-agnostic way to collect telemetry traces from external sources and send them to different targets through collectors for analysis and visualization. A high-level overview of OpenTelemetry is shown in Fig. §2.6 [37], and the collected traces follow the format shown in listing 2.6.

```
{
  "trace_id": "7bba9f33312b3dbb8b2c2c62bb7abe2d",
  "span_id": "086e83747d0e381e",
  "name": "/v1/sys/health",
  "start_time": "2021-10-22 16:04:01.209458162 +0000 UTC",
  "end_time": "2021-10-22 16:04:01.209514132 +0000 UTC",
  "attributes": {
    "net.transport": "IP.TCP",
    "net.peer.ip": "172.17.0.1",
    "net.peer.port": "51820",
    "net.host.ip": "10.177.2.152",
    "net.host.port": "26040",
```

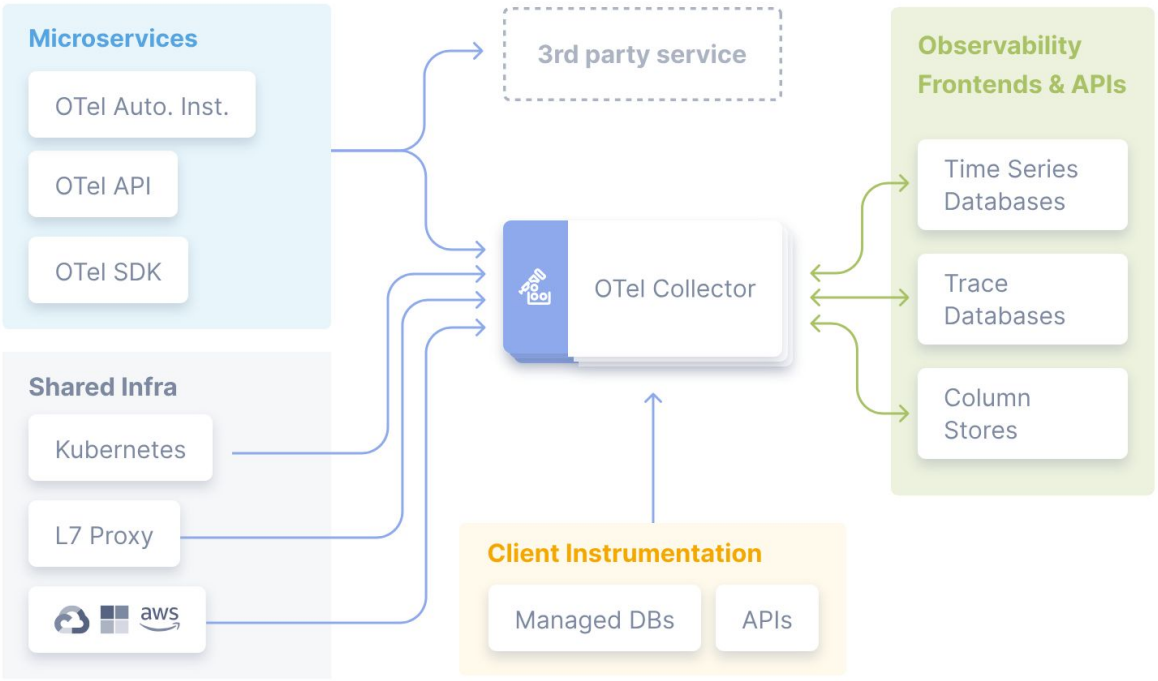

Figure 2.6: OpenTelemetry diagram

```
    "http.method": "GET",
    "http.target": "/v1/sys/health",
    "http.server_name": "mortar-gateway",
    "http.user_agent": "Consul Health Check",
    "http.host": "10.177.2.152:26040"
  }
}
```

Listing 2.6: Fragment of an OpenTelemetry trace

Prometheus [38] is another popular specification used for collecting metrics from remote sources and exposing them in a consistent manner, as shown in listing 2.7. It has gained widespread adoption in the industry due to its ease of use and flexibility in handling a variety of metric types. Fig. §2.7 shows an example of a dashboard with data collected from Prometheus [39]. However, as with OpenTelemetry, it is a general-purpose specification that may be used for different tasks apart from RESTful architectures, such as monitoring storage or detecting system anomalies. As a result, the transmitted packages become quite large due to the increase of attributes, which may introduce unnecessary overhead in the network, ultimately affecting the performance of microservices.

Figure 2.7: Example of Prometheus data visualized in a Grafana dashboard

```
# TYPE assets_parse_errors_total counter
join_parse_errors_total 0
# TYPE join_http_response_time_seconds summary
join_http_response_time_seconds_sum{status="200"} 0.6420005
join_http_response_time_seconds_sum{status="404"} 0.59900004
join_http_response_time_seconds_sum{status="301"} 0.1440001
```

Listing 2.7: Fragment of Prometheus metrics

In the cloud native landscape[5], tools that leverage OpenTelemetry and Prometheus data, such as NewRelic[6] or Honeycomb[7], provide useful information and visual representations of telemetry data.

[5] https://landscape.cncf.io/
[6] https://newrelic.com/
[7] https://www.honeycomb.io/

The increasing adoption of MSAs in modern software development has brought about the necessity of analyzing their *capacity*. Further information and definitions about the capacity of an MSA is available in Chapter §4. While researchers have tackled this issue in the past, their focus has often been on performance analysis rather than capacity, which is a more complex metric that encompasses factors beyond other metrics used in performance analysis. Consequently, the literature on the subject has offered a range of frameworks and methods for measuring Quality of Service (QoS), with most of them relying primarily on performance metrics such as response times, network latency or resource consumption. However, none of these approaches have adequately addressed the problem of capacity.

Telemetry data can provide valuable insights into capacity analysis of RESTful architectures by allowing the inference of their topology through service endpoint call graphs [40]. This can be achieved by analyzing traces containing information about the source and target, as well as the operation being performed. Additionally, this approach can provide the actual number of requests made by each service to an API per operation, which is a crucial aspect of the capacity problem.

Efforts have been made to develop frameworks for extracting telemetry data [41, 42]. However, neither of these frameworks was designed to use the resulting metrics for a specific purpose, nor do they provide a concrete model or specification for application level telemetry. This creates a gap in the monitoring of MSAs, as it limits the ability to measure more complex metrics like capacity.

This same problem arises with specifications like OpenTelemetry and Prometheus. They have not been designed for a specific purpose other than collect telemetry data, why is why they are intended to be used as frameworks that leverage tools implemented over them, such as NewRelic or Honeycomb. However, currently these tools are not able to provide effective means to answer capacity questions, and the underlying technology is prone to input undesired overhead into the network.

In response, researchers have proposed models for measurable QoS goals and frameworks for extracting metrics to evaluate the monitored system against these goals [43]. Tundo et al. [44] proposed a model that contains measurable QoS goals based on the ISO/IEC25011 standard and implemented a framework that extracts key performance indicators (KPIs) based on user needs. Nevertheless, both of these approaches focus on performance analysis since their purpose is to reduce the impact of monitoring overhead on performance and meet monitoring goals based on performance indicators, respectively.

API Economy

This chapter introduces the API economy, a term that has gained traction over the last years. First, Section §3.1 provides an introduction to the concept of API economy. Next, Section §3.2 contains information about the concept of API pricings and their elements and features. Then, Section §3.3 presents the idea of API marketplaces and their benefits for consumers and providers.

3.1 Introduction

The so-called *API economy* refers to an ecosystem of APIs used as business elements, where software developers subscribe to and consume external APIs, while also providing their own APIs with their own pricing plans. WSO2 defines the API economy as *the ability for APIs to create new value and revenue streams for organizations through the use of APIs* [45]. Similarly, Capgemini defines it as *the ecosystem of business opportunities enabled by the delivery of functionality, data, and algorithms over APIs* [46]. Thus, the API economy has popularized the consumption of APIs and their payment through what are known as *pricing plans*, which describe their functionality, their capacity limits (aka limitations) and the price for using them. Fig. §3.1 shows an overview of a company that uses an API to offer its functionality to any customer that wants to use it, in any type of device [47].

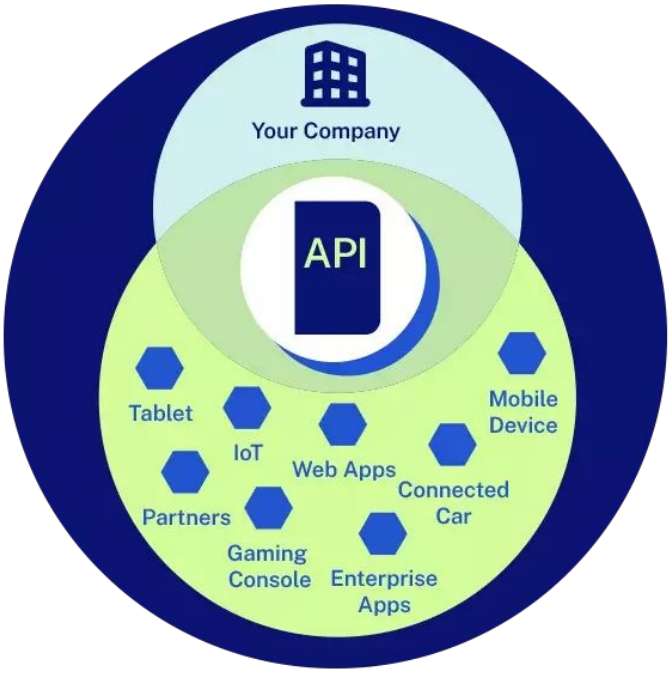

Figure 3.1: Overview of the API economy applied to a company that offers an API that can be used in any device

More and more often, the design of the system is based on the premise that its functionality is offered to its users within certain operating conditions, which implies that its architecture is built and operated taking into consideration the limitations derived from its capacity and usage of external APIs. Note that for the remainder of this thesis we will refer to the microservices of an architecture as **internal services**, even though they

usually offer an API themselves; the term API will be reserved for **external APIs**.

The API economy is very much in line with and particularly beneficial for systems with an MSA that uses external APIs. In an MSA, each microservice should have a well-defined API and make use of a standardized paradigm to integrate with the rest of microservices in the architecture and the external supporting services; usually, the most used paradigm used to define the interface and inter-operate is the RESTful paradigm. The usage of the same communication pattern for all internal and external services makes it easier for developers to reuse the same code to send RESTful requests to any API. When consuming external APIs, businesses need to be aware of their pricings and limitations so that they can choose the plans that suit their needs.

Furthermore, if the MSA is part of a Software as a Service (SaaS), some services may offer APIs to its customers. In this case, it is also usual to offer RESTful APIs. Again, this approach eases the development of the MSA as it is possible to reuse the code that was created for the APIs of the internal services. In such a context, limitations play an even bigger role as the operating conditions would be explicitly grounded in the specific plans agreed with the SaaS customers. As an example, let us recall Fig. §1.3. It shows an MSA composed by three different services (S1, S2, S3) that make use of two external APIs (E1, E2) with different pricing plans. The MSA customers may choose between two options, named *User 1* and *User 2*, with different guaranteed operating conditions on the Requests Per Month and a corresponding monthly price.

3.2 Pricing

API providers usually sell their functionality through multiple *pricing plans*, or simply *pricings* [48]. In the widely used case of RESTful APIs the functionality is defined as a set of endpoints to resources that can be operated (with the HTTP directives). In such a context, a given API could have a number of endpoints and their specific pricing plans would specify the expected operating conditions for each endpoint. For the sake of clarity, from now on, for the APIs in Fig. §1.3 we will assume an important simplification: each API only has a single endpoint and a single operational directive.

In the scenario depicted in Fig. §1.3, E1 has two plans: *Basic*, with a price of \$5/month, a limitation of 15 RPS and another limitation of 1,000 requests per day (RPD); and *Premium*, for \$8/month, 25 RPS and 10,000 RPD; also, E2 has two plans: *Silver*, with a price of \$4/month and a limitation of 10 RPS; and *Gold*, for \$10/month and 20

RPS.

Pricing plans are usually paid through subscriptions, which tend to be billed monthly or yearly. The more expensive plans have less restrictive limitations. Some plans may also include an *overage* cost, that is, they allow clients to exceed their limitations for an additional fee (e.g. \$0.01 per request beyond the imposed limitation in the example of Fig. §1.3).

When clients subscribe to a pricing plan, they usually obtain a personal *API key* to identify their own requests and help providers apply the appropriate limitations. It is possible for a single client to obtain multiple keys for the same plan to overcome its limitations. Nonetheless, providers commonly limit the number of requests that can be sent from the same IP address to prevent a client from obtaining too many keys, especially for free plans.

While pricings from different providers may differ, most of them share a similar structure with multiple common elements. Each pricing comprises one or various *plans*, from which API consumers may chose. Some providers allow consumers to choose multiple plans simultaneously, or even select multiple instances of the same plan. Nonetheless, they usually limit the number of free plans to 1 to prevent consumers from potentially avoiding limitations by getting an infinite number of free plans. If a consumer does use multiple subscriptions, it is their responsibility to send requests using all plans. Each plan contains the following elements:

Price. The price of a plan indicates how much money the consumer has to pay the provider in order to use said plan. Most plans in real-world APIs have a periodic subscription model, where the consumer must pay periodically, e.g. \$50/month. In most cases, this period is monthly, but some providers also offer cheaper options for longer subscriptions, such as yearly. Some plans follow a pay-as-you-go model, where the consumer pays based on their actual usage of the API, e.g. \$0.1/request. Some providers allow consumers to negotiate the price based on their specific needs.

Capacity limitations. Also known as *limitations*, they define how many resources of the API the consumer may use per period of time, e.g. 100 requests/day. Resources may be domain independent (such as requests or credits) or domain dependent (such as emails or weather forecasts). Nonetheless, many domain dependent resources can be mapped to a certain number of requests, e.g. sending an email may involve sending one request. Limitations are usually divided into two groups:

- **Rates** are applied to *sliding* time windows. For example, for a one-week sliding

window, if a request is sent on Wednesday at 15:36:39, the window would close on the following Wednesday at 15:36:38. Rates are usually defined over shorter period of times, such as seconds. They define the *speed* at which the consumer may send requests. They are commonly used to prevent the consumer from sending short bursts of requests that may affect the performance of the API. Many pricings have the same rates for all of their plans.

- **Quotas** are applied to *static* time windows. For example, a one-week static window could always start on Monday at 00:00 and end on Sunday at 23:59, regardless of when the first request is sent. Quotas are commonly defined over longer period of times, such as days or months. These limitations represent how much of the API the consumer may use. Commonly, quotas are the defining factor of a plan, with cheaper plans having lower quotas and viceversa. Some plans may include **overage costs**, which are additional costs billed when the consumer exceeds the quota, e.g. $0.1 per additional request.

Fig. §3.2 illustrates graphically the differences between sliding and static windows in a hypothetical scenario. Considering the instant t when the last request was made, the analysis of the situation is twofold: (i) inspecting 1 second back, i.e., a 1-second sliding window, there exist 4 occurrences; (ii) observing only the 1-second static window elapsed from 0s to 1s and from 1s to t, there only exist two occurrences. In short, depending on whether a sliding (rate) or a static (quota) window is chosen, the observed occurrences may differ.

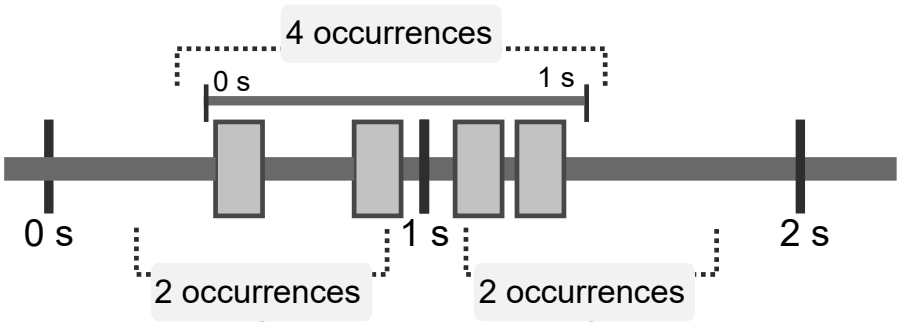

Figure 3.2: Sliding (rates) vs fixed (quotas) windows

Features. Some pricings may include access to extra features in more expensive plans, e.g. an email API might include additional email templates or the ability to add a signature. In some cases, these features are related to SLA terms, such as access to phone support, email support or guaranteed uptime and response time.

Some pricings define capacity limitations in terms of *credits*. Each plan has a maximum number of credits per month, and each operation (e.g. each endpoint) costs a specific number of credits. This way, providers can fine-tune the cost of each individual operation. Moreover, a single operation may have different costs depending, for example, on its parameters.

Besides a pricing, APIs usually have certain terms of use and SLAs (Service Level Agreements), including compensations and penalties for both clients and providers. These elements are out of the scope of this thesis, which focuses on pricings. However, a more detailed analysis of APIs should include all of their elements.

Fig. §1.1 in Chapter §1 shows the pricing of SendGrid, an email RESTful API that contains multiple plans with various capacity limitations and features.

In this point it is important to highlight that, although the SendGrid offering does not explicitly state the nature of those limitations (e.g. 100,000 emails per month in the *Essential* plan), we can foresee that they correspond to quotas while in the technical documentation they mention additional rates [1] that should be taken into consideration when using the API.

Governify4APIs

As discussed, while APIs now represent business product there is an important need to define and regulate API limitations (such as request quotas and rate limits). In fact, limitations are crucial for ensuring the equitable distribution of resources, preventing abuse and maintaining service quality. Yet, the lack of standardization and transparency in the communication of these limitations poses significant challenges. Consumers often find themselves navigating through complex and ambiguous documentation to understand the restrictions applied to their API usage, leading to a manual, tedious, and error-prone process of API management. This scenario underscores a critical gap in the API ecosystem: the need for a structured, standardized approach to modeling, communicating and managing API limitations. In [49], an initial approach for odelling pricing is presented (Governify4APIs); the aim of this model was to provide a framework to study its properties. Specifically, the study seeked to offer a clear, standardized representation that can facilitate better understanding and management of these constraints. Moreover, in this work, it was proposed the alignment with the Open API Specification (OAS) as this foundational standard plays a pivotal role, by providing a common language for describing API behaviors and constraints. The present thesis takes the Governify4APIs model as

[1] https://docs.sendgrid.com/v2-api/using_the_web_api#rate-limits

an starting point and extends it to provide support on the capacity analysis in LAMAs; consequently, section §5.2 present the evolution of the model.

3.3 Marketplaces

Most API providers offer their APIs through their own official websites, where they include all documentation about endpoints and pricings. However, when a business wants to consume multiple APIs in its system, it may be tedious and complicated to find the most suitable APIs, choose the most appropriate plans, subscribe to each one of them and keep track of the limitations to ensure that they are not exceeded. Furthermore, if each provider describes its APIs in its own format, it is difficult to compare different APIs and plans. For this reason, over the last years, there has been an ongoing effort to create API marketplaces where businesses and developers can search, find, compare and subscribe to multiple APIs that share a common documentation format.

In a marketplace, a business looking for an API is able to search through a list of available APIs and filter results based on various categories. This is much easier than looking through the Internet to find an API for a specific purpose, as this can be a daunting and time-consuming task. APIs can be compared to find the one that better suits the needs of the business. Additionally, as all pricings share a common format, it is easier to compare different APIs and choose the one with a better value for its cost. After subscribing to an API, marketplaces offer dashboards to check the consumption and billing information. When subscribed to multiple APIs, these dashboards make it easier to have all relevant information in a single location.

Marketplaces also offer various benefits to API providers. In particular, a provider who chooses to publish its API in a marketplace does not need to create and maintain a website to keep its documentation and pricings. Furthermore, the marketplace takes care of payments and subscriptions, which is specially interesting for small providers who may not have a payment platform. Providers may also have access to their own dashboards to check the usage of their APIs and detect spikes or service disruptions.

One of the most popular API marketplaces is RapidAPI[2]. It provides a wide variety of categories, as well as multiple collections of most popular APIs for different purposes, such as weather or sports. It also has a blog where developers can find tips and tricks, as well as tutorials on how to use the marketplace and consume the APIs. RapidAPI has

[2] https://rapidapi.com/hub

vastly evolved over the last years, as it has absorbed and joined multiple marketplaces and API mashup websites. Fig. §3.3 shows the main page of RapidAPI [50].

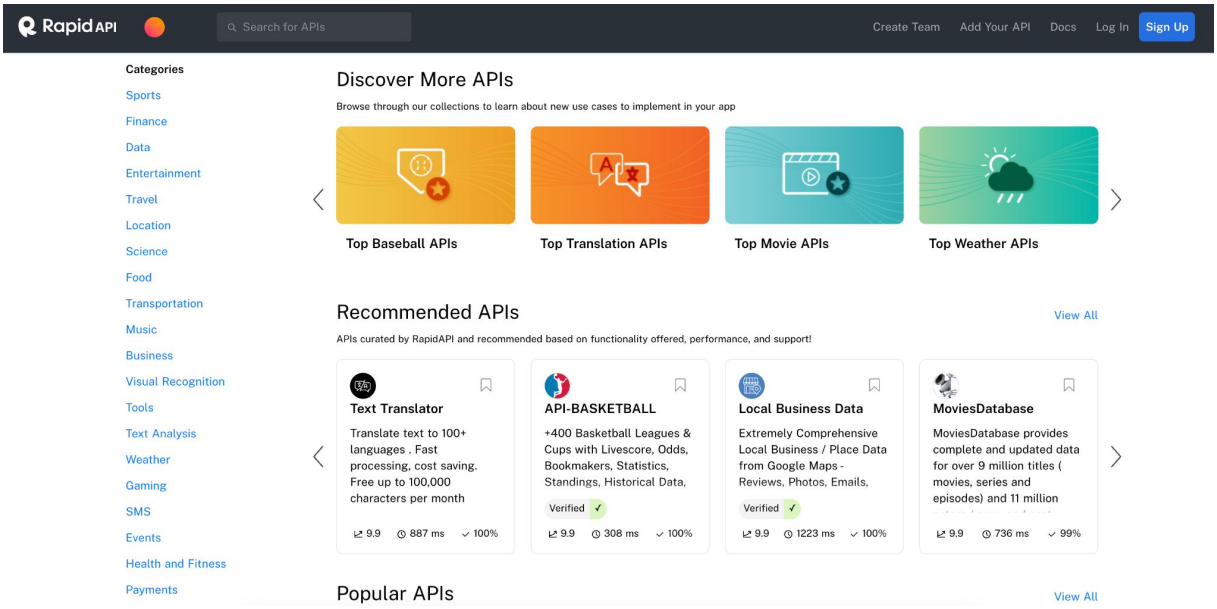

Figure 3.3: RapidAPI main page

Besides RapidAPI, there are other marketplaces such as APILayer[3]. Some major companies also offer their own versions of marketplaces, such as Amazon Web Services.

[3] https://apilayer.com/

Capacity Analysis

This chapter introduces the concept of capacity as it is known in the context of IT service management and engineering.Section §4.1 provides an introduction to the capacity of a system. Then, Section §4.2 describes the concept itself and contextualizes it within ITIL 4 (a widely used management framework), including some insights on the capacity management practice. Next, Section §4.3 introduces the capacity analysis of a microservice architecture, detailing current approaches for this analysis and indicating why they lack support for the specific features of MSAs that consume external APIs with limitations.

4.1 Introduction

The *capacity* of a system is a relatively wide term. In general, the capacity refers to the maximum workload that the system is able to handle, under certain circumstances. Furthermore, the *workload* does not have a standard metric, although most analyses use the number of users or requests. The circumstances depend on the system under analysis. For example, a business might be interested in analyzing the capacity of one of its systems considering the limitations of the deployment infrastructure. If this infrastructure is robust and scalable, the system should be able to handle bigger workloads. Therefore, if the system supports a bigger workload, it means that it can support more users and in turn generate more revenue for the business. This scenario showcases the importance of the analysis of the capacity.

Additionally, costs also play an important role in the capacity analysis of a system. Following the example in the previous paragraph, if the system is not expected to have a high number of users, it is not advisable to invest a disproportionate amount of money in a very expensive deployment infrastructure. This is because the revenue would not be high enough to sustain the infrastructure costs, and the business would lose money. Again, this demonstrates the necessity of analyzing the capacity of a system to ensure that it works under optimal operating conditions, while keeping costs under control.

In the following sections, we provide a comprehensive analysis of the concept of capacity within ITIL 4, a widely used IT management framework where the capacity management plays a very important role. After the importance of this management is stated, we then introduce the concept of capacity analysis to the scope of microservice architectures.

4.2 Capacity Analysis in ITIL

The Information Technology Infrastructure Library (ITIL) emerged in the late 1980s, as a response to the growing dependence on Information Technology (IT) in the business world. Developed by the Central Computer and Telecommunications Agency (CCTA), a government agency in the United Kingdom, ITIL was initially focused on standardizing the processes involved in managing IT services. This standardization was aimed at ensuring efficiency and predictability in IT service management (ITSM), which was becoming increasingly complex.

The first version of ITIL, known as ITIL v1, was a collection of more than 30 volumes,

each detailing various aspects of IT service management. As ITIL gained popularity, it underwent several revisions to keep up with the evolving landscape of IT. The most notable of these revisions were ITIL v2, introduced in the early 2000s; and ITIL v3, which was released in 2007 and later updated in 2011. ITIL v3 expanded the framework to an approach based on the life-cycle of the service management, covering service strategy, service design, service transition, service operation and continual service improvement.

The latest iteration, ITIL 4, was launched in 2019. This version focuses on integrating ITIL guidance with best practices from other methodologies like Agile, DevOps, and Lean, acknowledging the need for a more flexible, collaborative, and customer-centric approach in the digital era.

ITIL has become a global standard in IT service management for several reasons. Firstly, it provides a comprehensive, consistent, and coherent set of best practices, promoting efficiency and effectiveness in IT service management. By adopting ITIL, organizations can improve service delivery, increase customer satisfaction and optimize resource utilization, leading to cost savings. In such a context, ITIL helps in aligning IT services with the overall business strategy. This alignment is crucial as IT plays a pivotal role in the modern business environment. It ensures that IT services support business goals, enhancing the overall value creation of the organization. Additionally, ITIL offers a common language for IT professionals, enabling better communication and understanding within and between organizations. This aspect is particularly important in a globalized business world where cross-functional and cross-organizational collaboration is the norm.

From a high level perspective, ITIL is structured around a service life-cycle composed by five stages (see Fig. §4.1):

- **Service Strategy**: This stage involves the development of policies and strategies to serve the overarching business goals. It includes aspects like service management as a strategic asset, defining the market and financial management.
- **Service Design**: In this stage, ITIL focuses on designing IT services, including architectural, process, policy and documentation design, to meet current and future business requirements.
- **Service Transition**: This involves the transformation of services into a live operational environment, focusing on change management, release and deployment, service validation and testing.
- **Service Operation**: At this stage, ITIL addresses the efficient and effective deliv-

ery and support of services, ensuring that the value is realized in the operational phase. Key processes include incident management, event management and request fulfillment.

- **Continual Service Improvement**: This stage is about aligning and realigning IT services to the changing business needs by identifying and implementing improvements to IT services and processes.

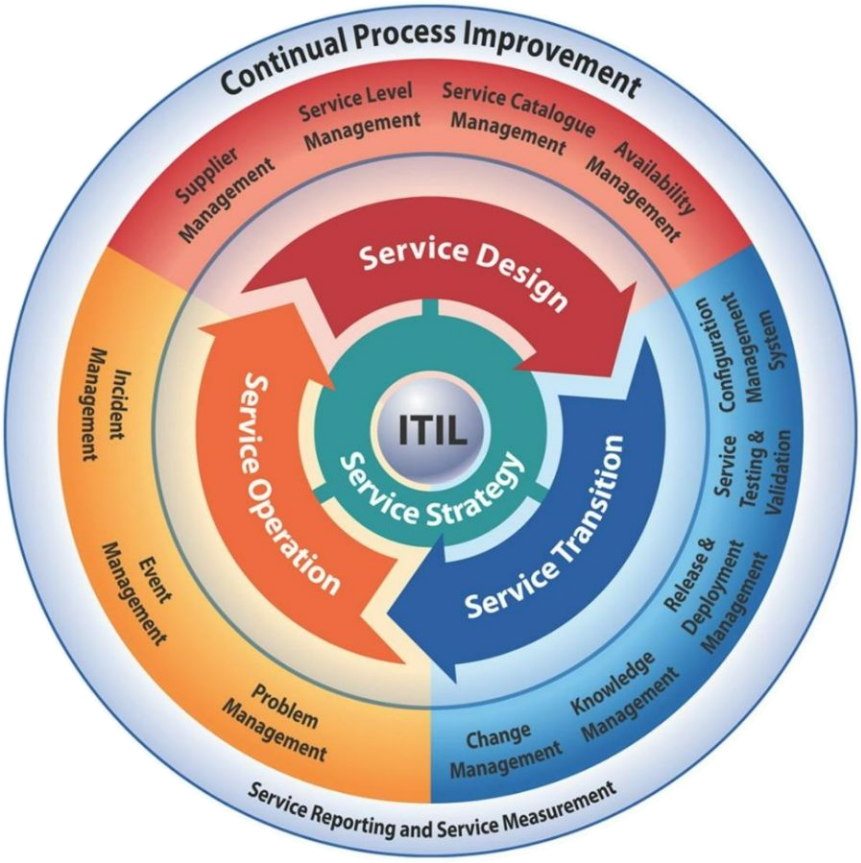

Figure 4.1: ITIL overview

Each stage of the life-cycle includes processes and functions that are guided by ITIL's principles. These principles are designed to be adaptable and flexible, allowing organizations to apply them according to their specific context, needs and maturity level.

In the ITIL framework, the *capacity* appears as a cornerstone of the **Service Design** and **Service Operation** stages and including an specific *capacity and performance management* practice, which is classified under the *service management* category, defined as:

This practice helps organisations ensure that their products and services meet expected performance levels. It also addresses current and future demands, helping organisations identify any changes that could affect their capacity[1]. In a complementary manner, the *capacity and performance management* is defined as: *The purpose of the capacity and performance management practice is to ensure that services achieve agreed and expected performance, satisfying current and future demand in a cost-effective way*[2] or *The objective of ITIL capacity and performance management is to ensure that your IT capacity meets your business needs. Satisfying current and future demand in a timely and cost-effective way is key to this ITIL practice*[3].

As observed in these definitions, the *capacity* of a service is related to the demand and the business needs of an organization. It involves the analysis of the current demand, so that the organization can act accordingly to meet customer needs and also take the corresponding actions to satisfy the expected future demand. The fact that the term *capacity* is always paired with *performance* leads to the conclusion that both concepts are closely related and, therefore, successfully managing the demand of a service also means ensuring that the performance of the service is optimal.

Furthermore, two of the three definitions above talk about *cost-effectiveness*. This means that capacity management directly affects the cost of a service, and thus it is vital for organizations which do not wish to waste any of their capital. In fact, the proper management of the capacity affects both income and expenses. By ensuring that the service meets the customer requirements, organizations can earn their trust and thus continue using the service, while avoiding unwanted service disruptions derived from under-provisioned infrastructure (e.g. cheap servers that allow a lower load than needed). On the other hand, organizations also need to make sure that the capacity of the service is not exceedingly high, so that a considerable part is wasted (e.g. very expensive servers that allow a very high load that is never achieved). Trying to balance costs is a fundamental part of capacity management.

The third definition also mentions the need to satisfy the demand in a *timely* way. This means that the service must meet the capacity requirements when its customers actually need it, and predict potential usage spikes. By correctly analyzing the evolution of demand over time, organizations can adapt their capacity accordingly and, therefore, successfully meet the customer demand. This analysis can also help in the optimization of

[1] https://www.itgovernance.co.uk/blog/what-are-the-itil-4-management-practices

[2] https://wiki.process-symphony.com.au/framework/lifecycle/process/capacity-and-performance-management-itil-4

[3] https://www.smartsheet.com/content/itil-capacity-management

costs (e.g. choosing and changing to a cheaper or more expensive server based on demand predictions).

In the following subsections we explore how the management of Service Level and Capacity in ITIL contextualize the need of an operational analysis that is specially useful in the context of IT infrastructure management based on MSAs.

Service Level

One of the key processes related to capacity within the ITIL framework is the **Service Level Management** (SLM) process. In this context, the SLM process is critical to the success of any IT service organization as it is focused on ensuring that the agreed-upon service levels are met and that customers are satisfied with the services they receive. This process works closely with the **Business Relationship Management** (BRM) process, which is responsible for managing the relationship between the IT organization and its customers.

The SLM process is responsible for understanding the **Service Level Requirements** (SLRs) of customers and translating them into **Service Level Agreements** (SLAs). These agreements serve as the rulebook for measuring and reporting the level of service delivered. The SLRs determine the contingencies to be provided in the architecture, such as high-availability architecture, and the resources required for delivering the service. This, in turn, has a direct bearing on the cost of the services. Moreover, with the rise of paradigms like DevOps, there is a growing need to factor in development measurements during customer discussions and negotiations. DevOps is a methodology that emphasizes collaboration between development and operations teams to deliver service and services more quickly and efficiently. The SLM process needs to be able to work with DevOps teams to ensure that development measurements are taken into account when negotiating SLAs.

In addition to measuring the operational aspects of a service, the SLM process also needs to be able to measure the customer experience. This includes factors such as response times, resolution times and overall satisfaction with the service. By measuring the customer experience, the SLM process can identify areas for improvement and work with other processes within the ITIL framework to address these issues.

Overall, the SLM process is a critical function within the ITIL framework. It is responsible for ensuring that services are delivered to customers at the agreed-upon service levels and that customers are satisfied with the services they receive. With the rise of DevOps and the need to factor in development measurements, the SLM process needs to

be able to adapt to changing circumstances and work closely with other processes within the ITIL framework to deliver high-quality services.

Moreover, the SLM process should be seen as an integral part of the ITIL framework that ensures that services are delivered at the agreed-upon level. This process is responsible for monitoring service performance against SLAs and taking corrective action when necessary. The SLM process also involves regular service reviews with customers to ensure that their changing requirements are met, and service levels are adjusted accordingly. As a consequence, the SLM process is critical for organizations that want to deliver high-quality services to their customers. It helps to establish a clear understanding of the expectations and requirements of the customers and ensures that the services provided meet those expectations. The process also helps to identify any areas where the service may be falling short and take corrective action to improve it.

In this context, the SLM process needs to factor in development measurements during customer discussions and negotiations since current service engineering practices focus on delivering high-quality services at a faster pace. To achieve this goal, ITIL promotes the idea that it is essential to have a clear understanding of the key performance indicators (KPIs) that will enable teams to measure their progress and success. In fact, the SLM process plays a critical role in ensuring that IT service engineers can deliver high-quality service that meets the changing demands of their customers. By incorporating development metrics into the SLM process, organizations can improve their service development and deployment processes and stay competitive in the market. Consequently, the SLM process is a crucial component of the ITIL framework that helps organizations deliver services at the agreed-upon level and ensures that customers' changing requirements are met. Moreover, the SLM process ideally must adapting to incorporate development metrics to enable teams to measure their progress and success. By doing so, organizations can deliver high-quality services that meet the changing demands of their customers and stay competitive in the market.

In the industry, ensuring high-quality performance is essential. However, many times, expected measurements are done at a contractual level without a process defining the ins and outs of the data sources and the measurement of performances. This can lead to a disconnect between the goals of the service provider organization and the actual results achieved. This is where service level management comes in as a crucial element for the service development and operation. Specifically, service level management involves identifying KPIs and keeping track of how things are moving along. By doing so, it helps IT service engineers stay on track and deliver high-quality service at a faster pace. This

is especially important in the current market where speed and efficiency are critical to staying ahead of the competition.

Agreement of service levels happens at multiple levels, including SLAs with customers, operational level agreements (OLAs) within the same organization and SLAs with suppliers. These agreements ensure that everyone involved in the service development and deployment process is on the same page and understands their roles and responsibilities. This not only helps to avoid potential conflicts but also ensures a smooth and efficient workflow.

To enable speed, multiple toolsets or frameworks are employed across various environments, and they are usually managed by separate teams. Therefore, it is imperative that the agreements (either OLA or SLA) exist to ensure guarantees for speedy deliveries. With the right agreements in place, IT teams can work more efficiently and deliver high-quality services faster than ever before.

Another crucial aspect in the context IT services deployed in cloud environments, is the availability. Customers expect seamless access to services they have subscribed to or use, and any downtime can have a significant impact on their satisfaction levels. This is where service level management and availability management come into play.

Service level management involves agreeing on the expected levels of service with the customers. This includes factors such as response times, uptime and other performance metrics. Once these levels are established, availability management takes over to ensure that the service is built to meet and even surpass these expectations.

The level of availability required can vary significantly depending on the nature of the service. For instance, a service that requires 99.95% availability may have a very different architecture compared to one that needs 99.99% availability. Even though the difference is just a few decimals, the impact on the underlying infrastructure can be massive. To achieve a higher availability, the service architects have to factor in multiple layers of contingencies to ensure that even multiple failures do not take down the service. This could involve redundant systems, backups and failover mechanisms to ensure that the service remains available even in the event of a catastrophic failure. The cost of providing services with such high levels of availability can be multiples of those offered at lower levels.

Managing availability in ITIL is highly aligned with the DevOps philosophy. As an addition, DevOps introduce a more concrete operational framework that promotes the usage of automation and continuous monitoring to detect and fix issues before they

impact the service. They may also represent the usage of tools such as load balancers and auto-scaling to ensure that the service can handle spikes in traffic without affecting availability.

In today's fast-paced and highly competitive business environment, organizations are constantly striving to deliver services that meet and exceed customer expectations. This is where perspectives such as DevOps comes into the picture, as they provide a comprehensive framework for managing the entire service development lifecycle, from planning and development to deployment and maintenance in an operational way.

However, its implementation comes with its own set of challenges, one of which is setting availability targets. In order to achieve this, organizations need to have a clear understanding of the different environments that are required for building and testing service. This includes environments that are not customer-facing but are still critical to the overall development process.

Moreover, multiple infrastructures and platforms are used in the service development, deployment and operation, and these elements need to be available as per the team's needs. Any lapse in the availability management of test environments, infrastructures or platforms can result in delayed delivery of services, which can have a negative impact on the overall delivery levels set forth by the service level management process.

Therefore, it is important to carefully analyze the availability requirements to ensure that they are in perfect alignment with the delivery rate and the speed at which the team can deliver. This requires a rigorous approach to availability management, which involves monitoring and tracking the availability of various environments, and taking proactive measures to address any issues that arise.

Another aspect to consider is over-delivering on availability. While this may seem like a good thing, it can actually be detrimental to the business angle of services. Every additional decimal of availability adds exponentially to the cost of services, which can have a significant impact on the bottom line. Therefore, organizations need to strike a balance between availability and cost-effectiveness to ensure that they are delivering services that meet customer expectations while also being financially sustainable.

In conclusion, service level management is an essential aspect from the ITIL Perspective. It helps to ensure that the goals of the project are aligned with the actual results achieved. By identifying KPIs and tracking progress, service level management helps teams stay on track and deliver high-quality services at a faster pace. Agreements at multiple levels ensure that everyone is on the same page and understands their roles and

responsibilities. With the right agreements in place, teams can work more efficiently and deliver high-quality services.

Service Capacity

In general, **capacity management** can be seen as an essential horizontal aspect of the service strategy in the ITIL framework. It should represent a process that encompasses the entire life-cycle of a service or product. The main objective of capacity management is to ensure that the services provided have adequate capacity to deliver value to customers.

Moreover, according to ITIL, capacity management plays a significant role in creating value for customers, and it operates in various levels, including reacting to capacity-related incidents and problems. The process takes control of anything related to capacity throughout the entire lifecycle of a service or product. This is particularly vital for service providers, as they may offer a service with the right scope in terms of functionality but without a capable infrastructure and effective operation, rendering such services useless. In this context, there are three flavors of capacity management in ITIL. The first is **business capacity management**, which is responsible for managing the overall capacity of the organization to meet current and future demands. The second is **service capacity management**, which ensures that the services provided have sufficient capacity to meet the agreed-upon service levels. It collaborates closely with the service level management process to ensure that services meet the required levels of availability, performance, and quality. The third flavor of capacity management is **component capacity management**, which manages the capacity of individual IT components that make up a service. It works closely with the IT service continuity management process to ensure that the components of the service are available when needed.

Efficient business capacity management is essential for ensuring seamless scale-up and scale-down of deliveries. This involves closely monitoring demand and devising plans to support scaling efforts. Effective management of business capacity can also provide valuable guidance on workload expectations.

In the dynamic and rapidly changing business world of today, effectively managing the capacity of your organization is a crucial factor for sustaining success. Capacity management refers to the process of aligning your organization's resources with the demands of your customers, encompassing the management of IT systems, physical infrastructure and human resources. In such a context, one pivotal aspect of capacity management is the ability to promptly and efficiently operate your resources. We can describe this operation as an adjustment of the size of a metaphorical funnel. This funnel denotes the flow of

resources through your organization, starting from the initial demand for your products or services, proceeding through various production stages, and culminating in the delivery of the final product to your customers. If your funnel is too small, you risk losing customers due to long wait times or poor quality. Conversely, if your funnel is too large, you may be squandering valuable resources on excess capacity that is not needed. To ensure that your funnel is always appropriately sized, it is imperative to have a robust business capacity management process in place. In fact, customers provide a list of requirements that must be translated into production, with the size of the funnel determining both the rate of incoming flow and the ultimate outcome. By gaining insights into the requirements coming in, we can adjust the size of the funnel to match our needs. Similarly, scaling down insights can help optimize delivery costs. Moreover, requirements typically arrive in rapid succession, with the quality of the end product or service depending on the size of the funnel. This, in turn, is indicative of the infrastructure, tools, team strength, integration and other factors that must be in place for successful delivery. Ultimately, it is the quality of the product or service that determines its value, making it imperative to ensure that deliveries meet high standards.

The delivery of a service that is plagued with up-time discontinuities or bad performance can result in loss of customers and damage to the company's reputation. Therefore, it is imperative to have a robust business capacity management plan in place to ensure timely, budgeted and high-quality service delivery and operation.

Business capacity management involves comprehending customer needs and aligning project goals with them. From the perspective of ITIL, this can be achieved through market research and customer feedback surveys. The insights derived from these surveys can be utilized to modify project goals and ensure that they are in sync with customer needs both in terms of functional and non-functional requirements. Consequently, business capacity management plays a pivotal role in ensuring project success. It facilitates the management of incoming requirements and ensures timely, budgeted, and appropriate service delivery and operation. It also entails comprehending customer needs and aligning project goals with them.

In the context of IT infrastructures based on an MSA, the existing ITIL process can be readily applied without any modification or enhancement. This is due to the fact that they are typically deployed in a cloud-based environment, which allows for the elasticity of capacity demand. In other words, it is possible to increase capacity as needed and decrease it if demand drops to a certain level. This makes it effortless to ensure that your funnel is always the appropriate size for your customers. Consequently, Service capacity

management is a process that operates at a tactical level and focuses on the specific services offered by your organization.

Effective capacity management is an essential component of any successful organization. It involves implementing a robust business capacity management process, leveraging the ITIL framework, and focusing on service capacity management. By doing so, organizations can ensure that their resources are always aligned with customer demand, which leads to increased customer satisfaction, improved efficiency and, ultimately, greater success. Moreover, managing service capacity is a crucial aspect of ensuring optimal system performance. This is particularly important in regions with moderate numbers of subscribers where resource optimization is key. To achieve this, organizations must adopt service capacity management processes that will allow them to make informed decisions regarding enhancing performance or optimizing resources.

One of the most important sub-processes in service capacity management is component capacity management. This process is different from other sub-processes, as it focuses on the internal configuration items that make up the service, rather than end-to-end performance measurements. It involves analyzing the capacity of the various components leveraged for the delivery of services, usage patterns of the service, and other statistics. By doing so, service providers can maintain performance at the required levels and optimize service elements wherever necessary.

In conclusion, since ITIL promotes the process of service capacity management as essential for any IT system, it is imperative that organizations adopt rigorous procedures to enable them to make informed decisions regarding enhancing performance and optimizing resources. As a consequence, the component capacity management is a crucial subprocess that should provide the necessary data for tactical and strategic decision-making processes. In the context of MSAs, where the service relies on multiple infrastructures and platforms (usually distributed), it is a given that every component of the architecture must be capacity managed. However, it is worth noting that the tools and non-production environments used to manage the quality display must also be capacity managed. This will guarantee that the system functions at an optimal level, and any issues are promptly identified and addressed. Nevertheless, the service capacity management process is particularly pertinent if the organization leverages the process's maturity to keep its ears close to the ground from a capacity perspective. This enables adjustments to be made in an flexible manner when necessary, ensuring that the system remains efficient and effective.

4.3 Capacity Analysis of an MSA

In general, the *capacity* of a microservice architecture refers to the maximum workload that it can handle, although there is no widely accepted definition for the term. Furthermore, the term *capacity* does not have a standard definition that is used by all authors in existing literature. Instead, there are multiple different definitions that share some common details.

In our context, we consider the capacity of a microservice architecture as the capacity of a given *entrypoint*. An entrypoint is the service of the MSA that is invoked first when a customer uses it, and then sends the appropriate requests to other services, which, in turn, may send further requests to more services. Nonetheless, there might be cases where the capacity of the MSA is calculated for a specific internal service, or as the aggregation of all capacities of all services. Additionally, an MSA might have multiple entrypoints; in this case, it could be possible to analyze the capacity for a single entrypoint at a time, or multiple ones simultaneously.

While there is no standard metric for the capacity, existing work in the literature commonly uses the number of requests per unit of time as the metric of choice. User interactions with an MSA through a user interface translate into requests that are sent to internal service endpoints (the entrypoints). Similarly, if the MSA offers a public API to its customers, interactions with it are done through requests to some entrypoints.

In a scenario where the MSA consumes no external APIs, its capacity analysis is relatively simple. The developers of the architecture would need to analyze the maximum load that each service can handle and check whether they can meet all customer demands. The maximum load of a service is typically related to the infrastructure where it is deployed, so cheaper servers (either local machines or cloud providers) allow lower loads and viceversa. In this particular scenario, the business has the ability to scale each service as required, by using more expensive servers or replicating the service in multiple instances. At this point, they need to be aware of the available budget and adapt accordingly, simultaneously meeting the customer demands and trying to keep costs to a minimum (that is, being *cost-effective*). To a certain extent, businesses have full control of their services and can scale them at will, unless they use cloud providers with limitations.

The capacity analysis of an MSA becomes more complicated when it consumes external APIs with limitations. In this case, businesses need to know how many requests will be sent to each external API. Then, based on these numbers, they would need to choose the most appropriate plan in each case. They need to ensure that the chosen

plans allow enough requests to meet the customer demands, while also being as cheap as possible (again, being *cost-effective*). However, in this case, they are bound to a series of specific limitations that restrict the number of requests for a certain period, that is, the rates and quotas. These limitations cannot be exceeded, as doing so could result in service disruptions or even unexpected charges because of overage costs. In this scenario, businesses do not have full control of the situation because the offered pricings are fixed and cannot be changed at will. The only possibility is to choose a different plan (or, in some cases, choose multiple instances of a same plan), but doing so mindlessly could result in either low capacity or high costs. Therefore, it is important to carefully analyze how different choices affect the capacity of the MSA, and act accordingly. Furthermore, rates and quotas are specified for certain periods of time that may not exactly match the usage periods of the MSA (satisfying demand in a *timely* way). This creates a mismatch problem where it might be difficult to choose a plan when the quota resets at a time that does not quite match the time when the demand needs to.

When the MSA offers its own plans to its customers (SaaS scenario), the analysis of its capacity becomes even more important. In order to design the offered pricing plans (from now on, *MSA plans*) and predict the operating conditions, it is necessary to determine the capacity limits that the architecture will offer and the cost of using external APIs, analyzing different scenarios; specifically, as motivating examples, we can identify three common situations: i) in order to articulate a strategic decision to define the MSA plan for a estimated scenario, we can ask about the baseline operational cost for such scenario (e.g. *Q1 - What is the cheapest operational cost for my MSA in order to offer 2 requests per second (RPS) to 20 customers?*); ii) given a fixed relationship or a pre-existing ongoing contract with an external API, we could ask about the expected maximal operating conditions (e.g. *Q2 - Assuming we have a Basic plan and a Gold plan already contracted what is the maximum number of requests per minute (RPM) I can guarantee to all my 20 customers?*); iii) in case we have a pre-existing budget limit for a given scenario, we could ask about the optimal combination of plans to be subscribed and the potential limitations I could guarantee to my customers with this combination (e.g. *Q3 - Assuming we have a monthly budget limit of $120 in my MSA, which is the maximum RPS to each of 20 customers?*).

Beyond SaaS, it is important to note that those motivating examples can also be adapted for any MSA, even if they represent ad-hoc systems inside a single organization and the number of customers is not relevant for the calculations, for example: *Q4 - What is the cheapest operational cost to guarantee a global operating condition of 50 RPS?*, *Q5 - Assuming we have a Basic plan and a Gold plan already contracted what is the*

maximum RPS I can guarantee as operating condition? or, *Q6 - Assuming we have a monthly budget limit of $120 in my MSA, which is the maximum RPS I can guarantee as operating condition?*. In fact, guiding the strategic decision of contracting external APIs and anticipating the different options of operating conditions of the system depending on the cost are critical aspects that could help software architects and DevOps of any MSA.

These analysis questions deal with computer-aided extraction of useful information from a MSA, which helps DevOps teams make certain decisions and detect potential issues. Trying to answer these questions even in a simple scenario leads to the conclusion that it is a tedious, time-consuming and error-prone activity. In addition, as the MSA has some complexity, performing these analyses manually will be neither reliable nor cost-effective, and its automation would be of great value to software architects.

The capacity analysis of an MSA with external APIs is closely related to the *QoS-aware composition* problem. It tackles the selection of the best *providers* for different *tasks* in an architecture, based on QoS attributes offered by these providers that need to be optimized depending on user needs. This problem can be solved using search based techniques, using either Integer Linear Programming [51] or non-deterministic approaches [52, 53]. Drawing inspiration from these proposals, we could adapt their approaches to our problem, interpreting the MSA as a composite service where the QoS attributes of each provider correspond to the attributes of a pricing plan. Nevertheless, this interpretation presents some limitations that does not allow a complete capacity analysis of a MSA, in particular:

- No proposal considers capacity limitations of external providers, instead focusing on other attributes such as availability or response time.
- Each attribute needs an aggregation function that is used to select the best provider for each task. No aggregation function can model the exact semantics of rates and quotas, especially considering that they are defined over different time windows.
- No proposal defines analysis operations about capacity, time windows and cost.
- In previous work, a task could only be associated with a single provider. In our approach, there may be a need to use multiple API keys from the same provider to perform the same task.
- In real-world systems, a single provider can define multiple pricing plans for the same task. These plans differ in their cost, quotas, rates and other attributes. This

was not the case in the previous approach, where a provider only had one plan for a task. Additionally, an MSA might send multiple requests to the same provider.

- There might be cases where there is no solution to the problem, e.g. the MSA is not able to serve enough requests to meet user requirements given the subscribed pricing plans. In previous approaches, this aspect was not taken into account.

We are not aware of any existing proposal which analyses the capacity of an MSA with external APIs regulated by pricing plans. The most similar proposal, which has also been a major inspiration for this thesis, is ELeCTRA by Gamez–Diaz et al. [54]. Based on the limitations of an external API (specified in its pricing) and the topology of an MSA with a single entrypoint, ELeCTRA computes the maximum values of the limitations that the entrypoint of the MSA will be able to offer to its customers. Assuming that the topology of the MSA does not vary, these maximum values are determined solely by the values of the external API limitations, i.e., they are induced by them. This analysis of the limitations induced in an MSA is performed by ELeCTRA by interpreting the problem as a CSOP and using MiniZinc [55] as a solver.

Unfortunately, ELeCTRA's capabilities are insufficient to automatically analyze the capacity of a MSA. Its main limitation is to consider that a pricing consists of a single constraint, or a single quota or a single limit. Thus, it is not possible to model prices, overage cost, or specify several limits (quotas and rates) in the same pricing.

Part III

PROPOSAL

Extended Pricing Model

This chapter presents our proposal for a pricing model that includes the required elements for the capacity analysis of MSAs. In particular, Section §5.1 contains a short introduction. Next, Section §5.2 introduces the pricing model itself, which extends an existing model to add some elements that were originally missing. Then, Section §5.3 includes the serialization of the model, aligned with the OpenAPI Specification. Finally, Section §5.4 contains a catalogue of analysis operations for API pricings, classified by different scenarios and with some examples.

5.1 Introduction

As discussed, the non-functional description of an API (e.g. its pricing) has not been addressed in the most used standard in the industry, the Open API Specification (OAS); consequently, the actual pricing definition depends on ad-hoc approaches by the API provider [56]. This makes it harder to leverage the information included in a pricing, because each organization may describe it in a completely different way. These differences include different formats in the definition of the billing conditions or the usage limitations, mixing and confusing terms such as *rates* and *quotas* that do not have specific definitions. Furthermore, it is difficult to check the validity and correctness of a pricing when its description is not standardized, as it is not possible to have an exact definition of the *validity* of a pricing.

In previous works, Gamez-Diaz et al. presented Governify4APIs [49], a standardized model for RESTful API pricings that was devised after an exhaustive analysis of real-world APIs in the industry. Based on this model, the same authors created SLA4OAI [57], a proposed extension of OAS that includes non-functional information and aims to overcome the aforementioned issues. It is a work-in-progress specification, written in JSON or YAML, that is under discussion by organizations and practitioners [58]. The formal description of an API pricing is useful, for example, to check for its correctness or automatically choose the best plans based on user needs [19]. Consequently, in this chapter we present the *Pricing4APIs* model, an extension of the Governify4APIs model that includes some missing elements, along with its serialization in *SLA4OAI*. Furthermore, we present a first iteration of a catalogue of analysis operations for API pricings, including various examples to demonstrate how to solve them.

5.2 Pricing4APIs

In order to address a more capable capacity analysis in LAMAs, the proposed *Pricing4APIs* model takes the model presented in [49] as starting point and extends it. As a high-level overview, the *Pricing4APIs* model allows to define a set of plans with its associated cost; for each plan, a set of limitations (i.e. quotas and rates) over the potential API operations can be defined. In the context of the RESTful paradigm, those operations are bounded to an HTTP path and method.

Fig. §5.1 illustrates the complete Pricing4APIs model. To enhance clarity, we have divided it into three distinct areas: (i) the yellow elements, which are related to *pricing,*

plans and cost; (ii) the pink elements area, which addresses *limitations and limits*; and (iii) the blue area, which concerns *capacity*. In the subsequent sections, we will elaborate on each aspect of the model, utilizing examples from the external API E1 in Fig. §1.3. To do so, we will consider each part individually: the plan area, the limitations area, and the capacity area.

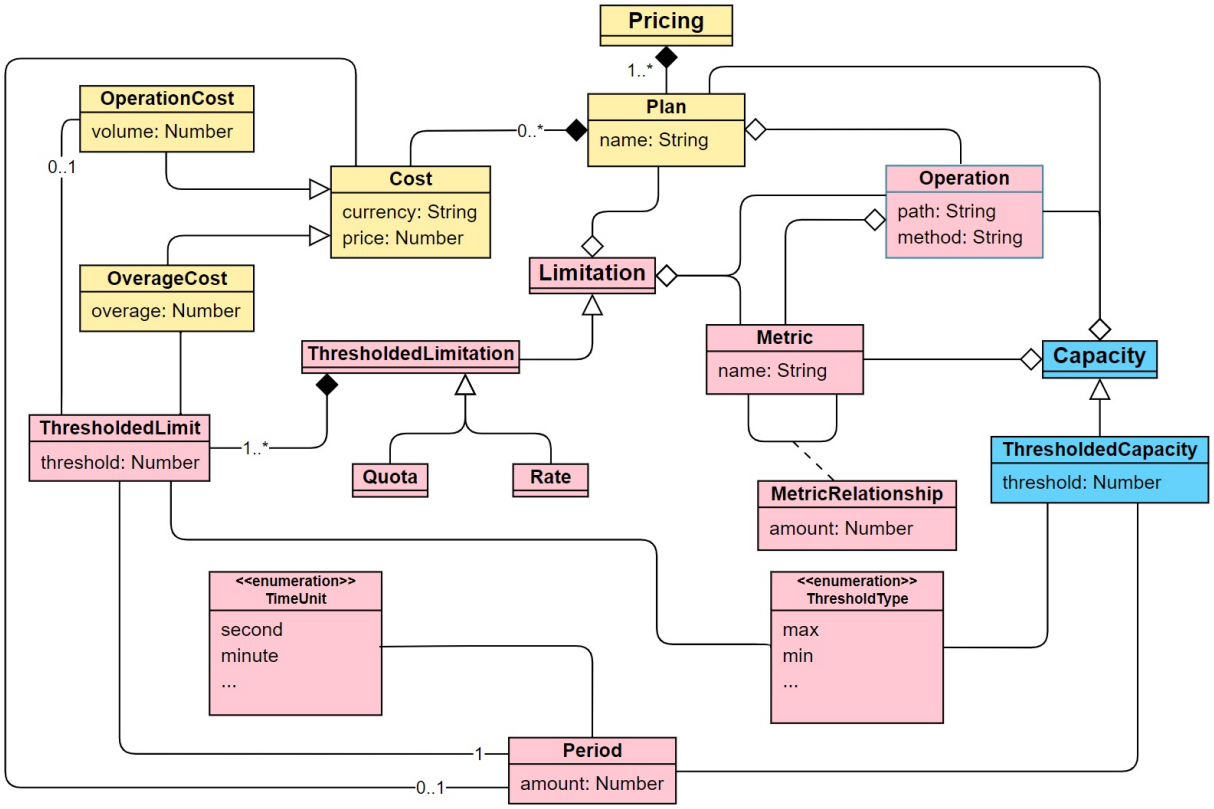

Figure 5.1: *Pricing4APIs* model for API pricings

Pricing, Plans, and Cost

As illustrated with yellow elements in Fig. §5.1), it is shown that `Pricing` is structured around various `Plans`, each labeled with a unique name and associated with a specific `Cost` that defines the charge for accessing the service. For instance, API E1 offers two types of `plans`: a *Basic* plan and a *Premium* plan.

The `Cost` associated with a `Plan` might be straightforward, such as a set fee of *$5* or *$8* in the given scenario. However, the cost can also vary based on additional parameters. In scenarios where the cost is contingent upon a `Limitation`, it bifurcates into two distinct categories: `OperationCost`, applied each instance an `Operation` is executed,

and `OverageCost`, applied after surpassing a predetermined threshold of the `Limitation`, with fees calculated based on the excess usage.

These `Cost` categories may adopt a periodic structure, articulated through a `Period` that specifies the duration and the `TimeUnit` associated with it. For example, the charge for the *Basic* plan is set at *$5* monthly, indicating a `Period` of 1 `TimeUnit` (month).

According to the model, the fee tied to performing a service operation hinges on the relevant `Limitation`. This could lead to differing plans, such as `Plan A`, which levies a $0.10 charge per request, and `Plan B`, which offers a package of 1,000 requests at $75. This arrangement allows clients to select the option that aligns with their usage patterns and budgetary constraints.

Should a client's usage exceed the predefined quota, an `OverageCost` may be applied, ensuring continued service access rather than termination. For example, surpassing the limit of 1,000 requests per day in our scenario results in a $0.01 fee for each additional request.

Limitations and Limits

As illustrated (highlighted by pink elements in Fig. §5.1), for the effective regulation of API usage, each `Operation` within a `Plan` is subject to `Limitations` based on a specified `Metric`. The prevalent form of `Limitation` is the `ThresholdedLimitation`, which sets one or several `ThresholdLimits` for the quantity of `Metric` units within a `Period`, typically adhering to a `max ThresholdType`. This signifies the *maximum* allowable `Metric` units. Through the imposition of `Limitations`, providers can tailor API consumption to align with the overarching `Capacity` of the platform.

An `Operation` is characterized by a combination of an HTTP method and path, essentially defining an endpoint. Metrics such as the *number of requests* are commonly used, but others like *storage*, *bandwidth*, or *CPU usage* are also viable. Metrics can interrelate, for instance, where each API request consumes 2KB of bandwidth, thereby establishing a `MetricRelationship` of *2* between the *number of requests* and *bandwidth per request* metrics.

The methodology for updating `Limitations Metric` distinguishes two categories of `ThresholdedLimit`: a `Quota`, calculated over a *static* time frame, and a `Rate`, which employs a *sliding* time window relative to the initiation of the metric count. For instance, a *one-week sliding window* starts and ends based on the timing of the initial metric unit, unlike a *one-week static window* which operates on a fixed schedule from Monday to

Sunday.

The distinction between sliding and static windows is visually represented in the bottom right of Fig. §5.2, particularly for the *number of requests* metric. This visual aids in understanding how occurrences differ based on the type of window applied: a 1-second sliding window shows four occurrences, whereas a 1-second static window records only two. This demonstrates the choice between sliding (rate) and static (quota) windows affects the tally of observed events.

To prevent more than 4 requests per second, two strategies are viable: a 1-second sliding window or a 1-second static window, each with its limits to control request frequency. This dual approach showcases the flexibility in managing API usage limits.

Industry practices often adhere to defined patterns [56] for setting `ThresholdedLimits`, with `Quotas` spanning longer periods (daily to yearly) and `Rates` focusing on the *number of requests* over shorter intervals. The challenge lies in the ambiguity of `ThresholdLimit` types in documentation, necessitating empirical testing by API users for clarification.

In this scenario, the *Basic* plan incorporates both a `Rate` and a `Quota`, demonstrating a common strategy for defining usage limitations. The model differentiates between `ThresholdedLimitation` and `ThresholdedLimit`, with the former setting a portion of the service's `Capacity` for a specific metric and operation. A `ThresholdedLimitation` might be detailed as a series of `ThresholdedLimits`, like *30 requests every 1 week* and *1 request every 1 second*, offering a structured limit on capacity usage. Alternative representations for `Limitations`, such as frequency distributions [59], provide further flexibility, although this discussion primarily focuses on `ThresholdedLimits` due to their prevalence in the industry. This example illustrates the typical application of rates for immediate capacity limits and quotas for long-term business considerations (as identified in [56]).

Capacity

A vital detail often omitted from public documentation regarding pricing or plans is the concept of `Capacity`. This element is intrinsic to the service provider's infrastructure and encapsulates the various limitations imposed by the platform or system's capabilities. These limitations are inherently tied to both technical specifications and financial considerations, such as processing power, memory allocation, and the architecture's scale.

The process of determining a service's `Capacity` is essential for establishing `Pricing` strategies and evaluating `Limitations`. Specifically, the service's `Limitations` should

align with its `Capacity` to ensure they are not exceeded, maintaining operational integrity.

Illustrated in the framework (marked by blue elements in Fig. §5.1), identifying the `Capacity` leads to its categorization akin to a `Limitation`, specifying the allowable quantity of a particular `Metric` within a predetermined `Period`. Consequently, similar to `Limitation`, `ThresholdedCapacity` is defined by a threshold value and a `ThresholdType` (typically `max` refering to the maximum function) for a specified `Period` and `TimeUnit`.

Expressing `Capacity` in terms of the *requests per second (RPS)* metric for each operation and plan is common. For instance, a *10,000 RPS* limit for the *GET /contacts* operation under the *free plan* implies that all users under this plan can collectively issue up to 10,000 requests per second. Notably, `Capacity` may vary across different plans to reflect the diverse levels of service provided through varied infrastructure setups.

For instance, based on rigorous performance and stress testing, an organization may determine that its system can support up to 10 000 RPS. If there were a restriction of 10 requests per second per client, the maximum number of clients that could be served concurrently would be calculated as $10,000/10 = 1,000$.

When evaluating `Limitations`, the *percentage of capacity utilization* or the *percentage of utilization (PU)* becomes an invaluable metric. This figure crucially influences the feasibility of imposing a `Limitation`, as setting one becomes untenable if the PU exceeds 100%.

The PU will depend on how a consumer consumes the API. There are two interpretations given a *Limitation*: uniform and burst. Therefore, the PU can be calculated in two different ways. To illustrate this idea, let us consider a `ThresholdedLimitation` with a single `ThresholdedLimit` of *43,200 requests every 1 day*: An API user might infer that, because a day encompasses 86,400 seconds, the rate per second is computed as $43,200/86,400 = 0.5$ requests. This estimation is predicated on a *uniform distribution*, gradually enabling the user to exhaust the daily quota of 43,200 requests. This illustration typifies the *minimum PU*. On the contrary, the `ThresholdedLimitation` clarifies that up to 43,200 requests are permissible within a single day, without any restrictions on executing all these requests instantaneously at the start of the day. Consequently, a user is theoretically capable of deploying all 43,200 requests in a singular second. This situation denotes a *burst distribution* and is indicative of the *maximum PU*.

Consequently, the PU must take both these models into account, so that we define the *bounded PU (BPU)* as this range:

1. The lower bound is the *minimum PU*, in which a *uniform* distribution of utilization over the period is assumed.

2. The upper bound is the *maximum PU*, which assumes the utilization of the maximum allowed in a single *burst*.

Fig. §5.2 illustrates different consumption scenarios for the same `ThresholdedLimitation` of *60 requests every 60 seconds*.

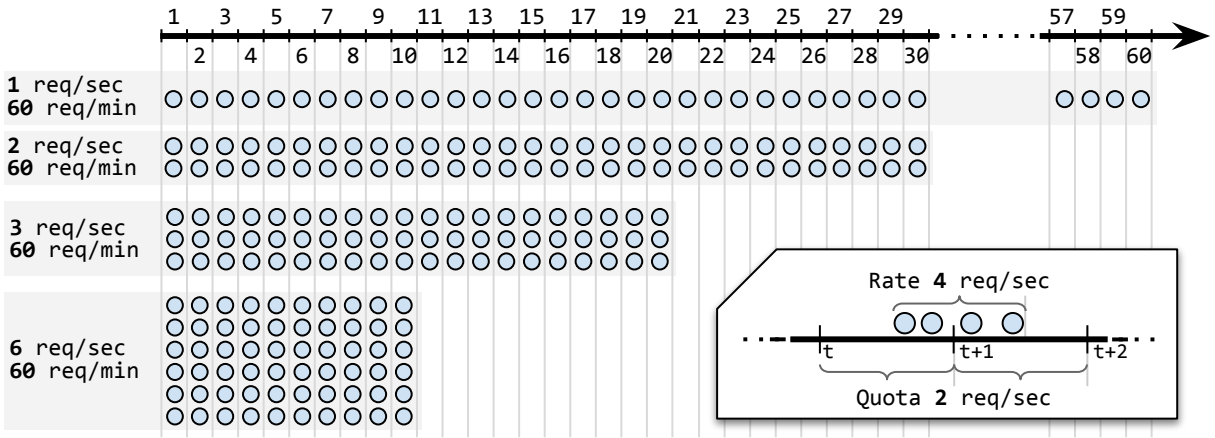

Figure 5.2: Examples of different consumption scenarios for the same `ThresholdedLimitation`

If we consider an *uniform* rate of consumption, where 60 requests are made in 60 seconds, this would be the same as making 1 request per second. However, in a *burst* scenario, the number of requests made in 1 second can range from 2 to a maximum of 60. So, when calculating the BPU (Burst Per Unit) within the limitation of *60 requests every 60 seconds*, we need to consider the minimum value of 1 request per second for a *uniform* distribution, and the maximum value of 60 requests in 1 second for a *burst* within a 1-minute timeframe.

It is crucial to emphasize that an accurate evaluation of capacity plays a vital role in ensuring reliable service and efficient resource utilization in the ever-changing API economy. When dealing with a diverse range of clients, each with their own unique plans and service requirements, the challenges are multiplied. For example, consider multiple consumers, each with different limits on the number of requests they can make per second. The provider must not only guarantee that each consumer receives their entitled

level of service, but also effectively manage backend resources to maintain service quality for everyone. Achieving this delicate balance requires a deep understanding of capacity. Incorrect estimations of capacity can result in overprovisioning, leading to wasted resources and increased costs, or underprovisioning, resulting in service disruptions and potential loss of revenue. Furthermore, in cloud-native environments where elasticity is highly valued, accurate capacity analysis is essential for driving elasticity rules. By accurately determining the capacity and distribution of workloads across different service tiers, elasticity rules can be established to dynamically scale resources up or down.

Accurate capacity analysis is essential for providers to maintain their service commitments and operate efficiently. It plays a crucial role in the API service paradigm, ensuring that providers can deliver their services in a cost-effective manner while optimizing their resources.

5.3 SLA4OAI

The *Pricing4APIs* model can be serialized to be aligned to a variety of API description specifications. Specifically, we propose SLA4OAI, an extension of the OpenAPI Specification (OAS), as it is currently the *de facto* industrial standard for describing APIs. Nevertheless, our model could easily be serialized to other API description languages (e.g., RAML, API Blueprint, I/O Docs, WSDL or WADL). SLA4OAI was initially created by Gamez-Diaz et al. [48, 60], but we extended the serialization with some additional elements.

It is important to highlight that, in the course of the last years, we have led an interest group in the OAI Consortium, to recommend a first simplified version of SLA4OAI[1] with the collaboration of 11 companies involving 22 practitioners. Gamez-Diaz then extended this simplified version to create an advanced specification, and we incorporated a set of key novel features that allow the seralization of the full Pricing4APIs model: the globbing mechanism and extended costs models such as overage costs (present in 11.9% of analyzed real-world pricings)[2].

In SLA4OAI, the original OAS document is extended with an optional attribute, `x-sla`, with a URI pointing to the JSON or YAML document containing the SLA definition. The SLA4OAI metamodel contains the following elements: *context information*, holding

[1] https://github.com/isa-group/SLA4OAI-Specification
[2] https://github.com/isa-group/SLA4OAI-ResearchSpecification

the main information of the SLA context; *infrastructure information* providing details about the toolkit used for SLA storage, calculation, governance, etc.; *pricing information* regarding the billing; and a definition of the *metrics* to be used. The main part of an SLA4OAI document is the *plans* section. This describes different service levels, including the limitations set in the *quotas* and *rates* sections. In what follows, we shall detail some of the fields in an SLA4OAI file. Nevertheless, for a comprehensive description of the syntax, a JSON Schema document is available[61]. Further information is also available in the the specification's GitHub page.

As depicted in listing 5.1, for the SLA4OAI model, starting with the top-level element, one can describe basic information about the `context`, the `infrastructure` endpoints that implement the Basic SLA Management Service [60] (i.e., a protocol as part of the SLA4OAI proposal, beyond the scope of the present thesis), the `availability`, the `metrics` and, inside `plans`, an entry defining `quotas`, `rates`, and `pricing`. Note that, in the model, the `pricing` of a `plan` is related to its cost and billing information.

```
context: ...
infrastructure: ...
availability: ...
metrics: ...
plans:
  MyPlan:
    pricing: ...
    quotas: ...
    rates: ...
```

Listing 5.1: Main elements in SLA4OAI

Specifically, as depicted in listing 5.2, the `context` contains general information, such as the `id`, the `version`, the URL pointing to the `api` OAS document, the `availability` of the document, and the `type` (this field can be either `plans` or `instance`). The `infrastructure` contains the endpoints that implement the Basic SLA Management Service, i.e., the `monitor` and `supervisor` services.

```
context:
  id: E1Example
  sla: '1.0'
  type: plans
  api: ./e1-oas.yaml
  provider: E1
infrastructure:
  supervisor: https://...
  monitor: https://...
availability: '2009-10-09T21:30:00.00Z'
```

Listing 5.2: Context, infrastructure and availability details in SLA4OAI

In the `metrics` field, as depicted in listing 5.3, it is possible to define the metrics that will be used in the limitations, such as the number of requests or the bandwidth used per request. For each metric, the `type`, `format`, `unit`, `description`, and `resolution` (when the metric will be resolved, e.g., `check` or `consumption` to indicate that it will be sent before of after its consumption, respectively) can be defined.

```
metrics:
requests:
  type: integer
  format: int64
  description: Number of requests
  resolution: consumption
```

Listing 5.3: Metric details in SLA4OAI

The `plans` section, as depicted in listing 5.4, has the elements that will describe the plan-specific values – `quotas`, `rates`, and `pricing`.

In this context, it is important to stress that the `plans` section maps the structure in the OAS document so as to attach the specific limitations (quotas or rates) for each path and method. In particular, the limitations are described with a `max` value that can be accepted and a `period` with an `amount` and a time `unit`. Furthermore, the `cost` section defines the `overage` (including the `overage` threshold and `cost` per extra unit) and the `operation` (including the `volume` and the `cost` per unit) costs.

The SLA4OAI model supports globbing to simplify pricings where the same limitation applies to multiple paths and/or methods. The character `*` can be used as a wildcard, so that, for example, limitations attached to `'/v3/*'` apply to all paths starting with `'/v3/...'`, but not to `/api/v3/...`. For methods, limitations attached to method `all` will apply to all methods. It is worth noting that more restrictive globbed paths have higher priority than less restrictive paths, meaning that if they have limitations for the same metrics and methods, the limitations in the former will override the limitations in the latter. For example, `'/v3/operation/*'` has higher priority than `'/v3/*'`.

```
plans:
  Basic:
    pricing:
      cost: 5
      currency: USD
      period:
        amount: 1
        unit: month
    quotas:
      'v3/example':
        get:
          requests:
```

```
13              - max: 1000
14                period:
15                  amount: 1
16                  unit: day
17                cost:
18                  overage:
19                    overage: 1
20                    cost: 0.01
21      rates:
22        'v3/example':
23          get:
24            requests:
25                - max: 15
26                  period:
27                    amount: 1
28                    unit: second
```

Listing 5.4: Plans details in SLA4OAI

5.4 Analysis Operations

The concept of a catalogue of analysis operations is not new. The creation of a catalogue paves the way for future research on automated tools that provide answers to the operations. There have been multiple proposals of catalogues for various elements related to service-oriented computing. Nonetheless, to the best of our knowledge, there are no existing catalogues for API pricings.

While we focus primarily on consumer-oriented operations, it is worth noting that API providers also benefit from their own operations, the main one being *Is my pricing valid?*.

In the field of RESTful API pricings, the work by Gamez-Díaz et al. [48, 56] includes an analysis of API offerings in the industry and a list of all elements commonly found in real-world API pricings. These papers serve as a foundation for our catalogue, as the analysis operations revolve around these elements.

In [62], Molino-Peña et al. propose a catalogue of operations for terms of use in customer agreements (CA). The authors divide a CA into three sections: terms of use, pricing and SLA. They only focus on terms of use as they found little research in the literature about them. Therefore, they do not include any operation for the other two sections, including pricing.

In [63], García et al. present a model for pricing and billing information in the context

of CA. However, this model does not account for multiple plans, different limitations (such as rates and quotas) or features. Therefore it is not suitable for API pricings, which include the aforementioned elements.

Classification

For the classification of the analysis operations, we focus on situations that may happen when an API consumer is searching for a plan or is already subscribed to one or multiple ones. This way, we reflect the goals of these operations. Usually, consumers already have a fixed subscription, a fixed set of features that they want, or a specific number of requests that they need to send to the API. Given the corresponding scenario, they then ask questions that involve optimizing one of three main variables cost, time or number of requests. Therefore, the classification that we present in this thesis is as follows [21]:

- **Given a specific set of subscriptions**. In this scenario, the consumer is already subscribed to the API. They may have multiple subscriptions to the same plan, and/or simultaneous subscriptions to different plans.

- **Given a maximum budget**. In this case, the consumer is not yet subscribed to the API, and has a maximum budget that they are willing to spend. Based on their needs, they will need to choose between the available plans.

- **Given a desired capacity**. The consumer has specific capacity needs, usually based on the needs of their own customers. This capacity is translated into a minimum number of requests that they need to send to the API.

- **Given a desired set of features**. In this scenario, the consumer wants a specific series of features, which may or may not be available for all plans.

- **Given a desired pricing**. This case corresponds to Software as a Service (SaaS) scenarios, where the API consumer is also offering its own pricing to their customers. The offered pricing needs to be aware of the external API pricing.

These categories are not mutually exclusive, and one analysis operation may be classified into more than one category. Nonetheless, we consider that one of the categories is always the *most appropriate* for a specific operation.

For API providers, we include an additional category:

- **Validity of the pricing**. This category includes one main operation, which is checking whether a pricing is valid or not, according to a set of validity criteria.

Catalogue

Following the classification in the previous section, we now enumerate the different analysis operations under each category [21]. Operations will be identified as O_i, where i is the operation number. This makes it easier to refer to specific operations. For each operation, we include some examples using an extended version of the SendGrid API, found on the RapidAPI marketplace and expanded with some features. This pricing is depicted in Fig. §5.3. We will consider that sending one email requires sending one request.

Basic	Pro	Ultra	Mega
$0/month	$9.95/month	$79.95/month	$199.95/month
10 req/s	10 req/s	10 req/s	50 req/s
50 emails/day	40000 emails/month	100000 emails/month	300000 emails/month
Ovg: $0.001/email	Ovg: $0.001/email	Ovg: $0.00085/email	Ovg: $0.00005/email
No support	No support	24/7 phone support	24/7 phone support

Figure 5.3: Extended RapidAPI SendGrid pricing. Each plan has a price, rate, quota, overage cost and, in some cases, a feature

Given a Specific Set of Subscriptions. Analysis operations classified under this category are as follows:

O_1. **How much time do I need to send N requests?** With their current subscriptions, the consumer wants to know how long it will take to send some requests, taking into account the capacity limitations of the subscribed plans.

Example: let us assume that we want to know how long it will take to send 50,000 emails and we have one subscription to plan *Pro*. In this scenario, we would need 1 month and 1,000 seconds to send all emails. If we were to upgrade our subscription to plan *Ultra*, we would only need 5,000 seconds. Furthermore, if we are subscribed to *Pro* and are willing to use overage costs, then we could also send all emails in 5,000 seconds. This would come with an additional cost, but it would be less than upgrading to *Ultra*.

O_2. **Can I send N_1 requests per second (RPS) and N_2 requests per day (RPD)?**

If the answer is negative, a useful auxiliary operation is **Why not?**. This would help the consumer pinpoint the specific rate or quota that is acting as a bottleneck.

Example: we have one subscription to plan *Pro* and want to send 10 RPS and 2,000 RPD. This would not be possible, as 2,000 RPD is equivalent to 60,000 requests per month (RPM), which exceeds the quota of *Pro*. We would either need to upgrade to *Ultra* or use overage costs.

O_3. **What is the maximum speed at which I can send requests, and for how long?** The rates of the subscribed plans will define the maximum speed at which the consumer may send short bursts of requests, while the quotas will determine how many bursts can be sent.

Example: we have one subscription to plan *Pro*. Therefore, the maximum speed at which we can send requests is 10 RPS, which is the rate of *Pro*. At this speed, we would consume the entire quota in 4,000 seconds. We would then need to wait until next month for the quota to be reset. If we are willing to use overage costs, we could extend this time.

O_4. **What is the maximum time that I will need to wait after consuming all quotas?** If the consumer uses all of their available quotas and there are no overage costs (or the consumer does not want to use them), they will need to wait until the quotas are reset. Some APIs actually return the waiting time when a consumer runs out of requests, but only for a single subscription. If the consumer has multiple subscriptions, this information might not be as useful.

Example: let us take the example in O_3 as a starting point. At full speed, we would consume the quota of *Pro* in 4,000 seconds. After that, we would need to wait until next month for the quota to be reset. If we consumed the quota during the first 4,000 seconds (which is a bit more than 1 hour) of a 30-day month, we would need to wait 29 days and a bit less than 23 hours for the quota to be reset.

Given a Maximum Budget. For this category, all operations O_1 through O_4 still apply. However the first step before solving these operations is determining which subscriptions to get for the given budget. This leads to an additional operation:

O_5. **What is the optimal set of subscriptions to get in a specific scenario?** Depending on the circumstances and the consumer needs (e.g. the scenarios in the previous 4 operations), the optimal set will vary. Nonetheless, this set must always take the maximum budget into account.

Example: let us assume that we want to send 50,000 RPM and have a maximum

budget of $100. At a first glance, we could simply get one subscription to *Ultra*, which is $79.95 and has a quota of 100,000 RPM. Nonetheless, we could potentially get many *Basic* subscriptions for $0, enough to allow our desired 50,000 RPM. However, let us assume that SendGrid limits *Basic* subscriptions to 1 per client. In that case, we could get 2 subscriptions to *Pro*, which would be $19.8 and have a total quota of 80,000 RPM.

Given a Desired Capacity. Under this category, the analysis operations are the following:

O_6. **What is the cheapest set of subscriptions that meets the desired capacity requirements?** This operation would return the set of subscriptions with the lowest total cost that allows the consumer to fulfill their needs.

Example: assuming that we want a capacity of 50,000 RPM, the example in O_5 would still apply. The difference is that there is no maximum budget in this operation.

O_7. **What percentage of the desired capacity is met with a specific set of subscriptions?** If the consumer already has a set of subscriptions or a budget constraint, they may want to know how much of the desired capacity is actually able to be fulfilled.

Example: we want a capacity of 50,000 RPM, and, for any specific reason, we want to use one subscription to *Pro*, which has a quota of 40,000 RPM. In this scenario, we are meeting 80% of the desired capacity.

Given a Desired Set of Features. This category contains the following operations:

O_8. **What is the cheapest set of subscriptions that meets the desired feature requirements?** Similar to O_6, this operation would return the optimal set of subscriptions that meets the consumer needs, but focusing on features instead of capacity.

Example: let us assume that we want to have phone support from SendGrid. Even if we just wanted to send 1,000 RPM, neither the *Basic* or *Pro* plans include phone support. Therefore, the cheapest option in this case is one subscription to the *Ultra* plan.

O_9. **What amount of desired features is met with a specific set of subscriptions?** Similar to O_7, the consumer may want to know how many of the desired features can be fulfilled with a given set of subscriptions or a budget constraint.

Example: if we want to have phone support and we are subscribed to the *Pro* plan, then we are meeting 0 of the desired features.

Given a Desired Pricing. In a similar way to the previous two categories, this category

includes two analysis operations:

O_{10}. **What is the cheapest set of subscriptions that meets the desired pricing requirements?** Similar to O_6 and O_8, the consumer wants to know the cheapest subscriptions to get in order to meet the needs of their own customers.

Example: let us assume that we want to offer a pricing with 10 RPS and 50,000 RPM. In this scenario, the cheapest option would be to get two subscriptions to *Pro*. However, if we want to offer this pricing to multiple simultaneous customers, we would need to multiply the limitations by the number of customers, e.g. for 10 customers we would need 100 RPS and 500,000 RPM. This would increase the required cost to subscribe to the SendGrid API.

O_{11}. **What percentage of the desired pricing is met with a specific set of subscriptions?** Again, similar to O_7 and O_9, the consumer already has a set of subscriptions or a budget constraints and may want to know how much of their needs can be fulfilled.

Example: if we want a pricing with 10 RPS and 50,000 RPM and we only get one subscription to *Pro*, then we are meeting 100% of the desired rate but only 80% of the quota. These percentages could be aggregated into one, depending on the consumer needs and preferences.

Validity of a Pricing. The **validity** of a pricing is defined as the absence of conflicts between the different elements of the pricing [16]. Let us recall the basics of a pricing. A *pricing* consists of a series of *plans*, each of them having a series of *limitations*, and each limitation having a series of *limits*. A pricing is therefore valid when all of its elements are valid. For example, a pricing is not valid if a single limitation is not valid and has a conflict. An API provider would benefit from the automated validation of pricings, in order to ensure the coherence of its elements and, potentially, prevent users from exploiting the API.

Another element of an API worth mentioning is its **capacity**. To sum up the definition in the previous section, the capacity of an API is the maximum workload that it can handle over a specific period of time. It usually depends on the internal architecture and the deployment setup of the API. For example, a capacity of 1000 requests per second (RPS) means that the API can serve up to 1000 requests each second to its users.

In the following paragraphs, we present the validity criteria in a hierarchy. We start from fine-grained elements (limits and limitations) to coarse-grained ones (plans and pricing). Each criterion has multiple subcriteria, and all of them must be valid for the criterion

to be valid.

VC1 - Valid limit A *limit* is valid if its threshold is a natural number (VC1.1).

VC2 - Valid limitation A *limitation* is valid if: all its *limits* are valid (VC2.1); there are no *limit consistency conflicts* between any pair of its *limits*, i.e. a limit over a longer period of time has a lower threshold than a limit over a shorter period (VC2.2); there are no *ambiguity conflicts* between any pair of its *limits*, i.e. multiple limits with different thresholds over the same period of time (VC2.3); and there is no *capacity conflict*, i.e. a limit is less restrictive than the *capacity* of the API (VC2.4).

VC3 - Valid plan A *plan* is valid if: all its *limitations* are valid (VC3.1); and there are no *limitation consistency conflicts* between any pair of its limitations, i.e. a limitation over a metric allows another limitation over a related metric (by a certain factor) to be exceeded (VC3.2).

VC4 - Valid pricing A *pricing* is valid if: all its *plans* are valid (VC4.1); and there are no *cost consistency conflicts* between any pair of its plans, i.e. a limitation in a plan with a lower cost is less restrictive than the equivalent limitation in a plan with a higher cost (VC4.2).

Listing 5.5 shows a simplified example of a pricing with a limit consistency conflict (VC2.2). In this example, there is a conflict because there are two quotas defined over the same metric (requests) and the quota with the longest time unit (week) is more restrictive than the quota with the shortest time unit (day).

```
1  Limitations:
2    Quota: 100 requests / 1 day
3    Quota: 10 requests / 1 week
```

Listing 5.5: Example of validity criterion VC2.2 (limit consistency conflict).

Thus, the main operation under this category would be O_{12}, **Is my pricing valid?**.

Limitation-Aware Microservice Architectures

This chapter introduces the concept of limitation-aware microservice architectures or LAMAs. First, Section §6.1 presents some background and the elements that differentiate a LAMA from a regular MSA. Next, Section §6.2 provides further details about the impedance mismatch problem that was originally introduced in Chapter §1. Then, Section §6.3 contains a description of the elements of a LAMA, as well as a domain-specific language for their formal definition.

6.1 Introduction

As discussed in Chapter §3, it is common for MSAs to consume external APIs offered by third party providers. In this scenario, where the developers of the MSA are aware of the limitations of these external APIs, we coin the term *Limitation-Aware Microservice Architectures*, or *LAMA*. In short, a LAMA is an MSA that consumes external APIs with limitations. The LAMA might offer its own plans to its customers in a SaaS scenario.

Let us recall the example in Fig. §1.3. The scenario depicted in the figure is a LAMA, and, from now on, we will refer to it as such. The LAMA consumes two external APIs, each of them with two different plans and various limitations. Furthermore, the LAMA offers two plans to its customers, from which they are able to choose. To simplify calculations in the following chapters and sections, we use the same time units for all rates, quotas and billing periods. However, in real-world LAMAs that consume multiple external APIs, this is not always true, thus complicating the analysis of the aggregated impact of all limitations.

Additionally, the internal services of the LAMA might also have their own limitations. They are usually derived from the deployment infrastructure of the services, and they may have a considerable impact on the overall ability of the LAMA to accept requests from its customers. For example, a service that is deployed in a small server (either locally or in the cloud) might have a relatively high response time. If many requests are sent in a short period of time, they may take too long to be fulfilled and therefore create a queue. Depending on the available memory, CPU or storage, these queued requests could potentially result in a disruption of the service, making the LAMA unavailable. Analyzing the internal limitations of a LAMA is an interesting topic, but it adds a whole new level of complexity to the capacity analysis of the LAMA, so we let it out of the scope of this thesis.

6.2 Impedance Mismatch

As introduced in Section §1.2, potential *impedance mismatch* is derived from the confronted roles of providers and consumers in a prosumer scenario. On the one hand, as consumers, a business relies on third party providers with their pricing and limitations. On the other hand, as providers, they deliver services to their customers, for a price and with certain limits. However, the dynamics on those different realities are intrinsically different. As service consumers, they have to select amongst the potential providers that

evolve their offering (i.e. pricing and limitations) which one (or ones) is adequate from the perspective of their business model. Conversely, as service providers, they have to design and change their offering to be competitive and grow while satisfying their customers and augment its capacity if necessary.

Specifically, in the middle of these two realities, the prosumers need to solve the impedance of pricing and limitations by taking decisions on which is the best offering that they will put on the market to search for customers, while selecting the best plan to use of each provider.

In many scenarios, organizations delivering services are not actually doing it in the context of market but there will always be a set of operational requirements that will evolve. Those requirements specify the acceptable conditions of load in which the service is required to operate (e.g. how many requests per second, for a certain endpoint, should be accommodated). As a consequence, in this case, the same impedance problem is also apparent.

The following subsections provide a deeper insight on the impedance mismatch in LAMAs. The capacity analysis problem will be tackled later in Chapter §7.

Dynamic Impedance Mismatch

The first problem that we identify is the **dynamic impedance mismatch** problem. In some cases, the LAMA developers notice that their demand within the period of a quota (e.g. a month) is either too high or too low, meaning that they are close to exceeding the quota limitation or wasting too many requests. However, they cannot do anything to solve this issue as the quota resets at a specific time (e.g. at the end of the month), so they are unable to change the subscribed plan until the next period. When a business subscribes to an API, they commonly choose a plan based on their quotas. In most cases, quotas reset at a certain time independently of when clients send their requests. For example, a quota may reset at 12 AM on the first day of each month, even if the LAMA sent all its requests the last day of the previous month. This means that the requests sent to the external APIs (the *demand*) do not always align with the periods of the quotas. This may create a misalignment between the billing lifecycle and the fluctuations in the demand.

As an example, we will use Bluejay, a real-world IS that works as an auditor framework for agile software development and was presented in Section §1.2. It collects information from various external sources (GitHub, Pivotal Tracker, etc.) and draws multiple graphs in a dashboard that show the evolution of the developers' adherence to a series of best

practices, known as *team practices*. Bluejay has been used to audit multiple software engineering courses at the University of Seville and the University of California, Berkeley. In the context of academic experiments, there are two types of users: students, which work on their projects and trigger new updates; and teachers, which check the dashboards and do not typically do any changes.

When a student updates their repository, either by pushing some code or updating an issue, this change triggers a new calculation in Bluejay. To check the adherence to the team practices, the system needs to check the status of the project in each one of the different external services used in the project. With that information, Bluejay then computes a series of metrics for each team practice and stores them in a database. These metrics are then used to create graphs that students can use to evaluate their progress, and are useful for teachers too.

In terms of a LAMA, this means that each new change implies a request to the entrypoint of the LAMA, that then sends the appropriate requests to the external APIs. Therefore, the demand to these external APIs varies depending on the amount of changes that students do. Usually, these changes are mostly concentrated towards the end of a sprint or a deliverable, and are more spread out at other times. Consequently, there are periodic activity spikes at these times (e.g. weekly, every two weeks, monthly...). Furthermore, there may be an even higher spike when the course is near its end and students are finishing their projects. However, the quotas of the external APIs used to compute the metrics are reset monthly, independently of the students' activity. For this reason, we need to monitor the demand to know when these spikes happen, and adapt the subscriptions accordingly. This task is not always easy, as we need to wait until the next quota period to be able to increase or decrease said quota. Thus, we need to anticipate to the demand spikes.

Static Impedance Mismatch

A second problem is the **static impedance mismatch** problem, which occurs when the initially expected demand to the external APIs does not match the actual demand.

When designing a LAMA, the system architects or developers may obtain an initial calculation of the number of requests sent to each external API, that is, the demand to these APIs. Taking the API pricings into account, they may adapt the demand to make the most out of each subscribed API plan. Usually, this initial analysis assumes a worst-case scenario, that is, all requests are always sent and all users send their requests at the same time. Thus, the business may get the appropriate number of API keys to support

this *maximum* demand. In other cases, the analysis assumes an expected demand based on initial calculations of expected users and workload. This is a *static demand analysis*.

Nonetheless, in the real world, the system load commonly fluctuates and is not always the same. The LAMA customers send their requests when they need to do so, each one of them at a different time, following a *chaotic load*. Furthermore, the specific functionality used by each customer may be different, thus following different paths within the internal services of the LAMA and, in turn, having a different demand to the external APIs.

Additionally, it is possible to accidentally alter the demand by adding a new feature to the system or by releasing a new version which contains a bug that sends more requests than expected. These situations may lead to one of two different scenarios: (i) the system sends less requests than initially expected, thus wasting available requests and losing money by assuming a worst-case scenario; (ii) the system sends more requests than expected, so the external API limitations are exceeded and a service disruption occurs. Therefore, the initially calculated demand does not match the real demand. This is what we call the *static impedance mismatch* problem.

Demand Analysis

In both previous problems, it is necessary to obtain the actual demand of the LAMA at any given point in time so that the business can adapt accordingly. Consequently, a third problem is the *demand analysis*, that is, analyzing the current demand of a LAMA.

The simplest way to perform this analysis is to monitor the internal services of the LAMA and check the requests that they are sending to each other and to the external APIs. To do this, the services need to be instrumentalized in order to collect information about their usage. By collecting traces of the various services, it is possible to infer the demand and even the internal topology of the LAMA, which may differ from the expected topology if an implementation bug exists (thus creating a static impedance mismatch). Besides using this information to adapt the subscribed plans as required, the business is also able to leverage the collected traces to modify or correct the internal topology if needed.

Section §8 provides further information about the requirements of a monitoring framework for LAMAs, and shows a first implementation that can be used to automatically draw the topology of the LAMA and also answer multiple capacity analysis operations described in Section §7.4.

Predictive Analysis

In some LAMAs, the demand might be periodic, e.g. depending on seasons, school years, months, etc. For example, in the case of Bluejay, students have activity spikes when they are close to a deadline, and an even higher spike closer to the end of the term. Through a predictive analysis, the LAMA developers could detect these activity periods and anticipate to them by subscribing to the most appropriate plans in each case. They would still be subject to the dynamic impedance mismatch problem, but they could minimize the impact of the issue.

In some cases, this analysis reveals that it is not possible or convenient to support the whole demand that is expected to be sent during an activity period. For example, it may happen that, in order to support every expected request, it is necessary to subscribe to a plan that is so expensive that its cost cannot be assumed by the business that creates the LAMA. In this scenario, it may be more convenient to *lose* some requests and, in turn, subscribe to a cheaper plan whose cost is actually acceptable for the business. This means that the LAMA would assume some *risks*, by acknowledging the possibility that, by choosing a cheaper plan, a part of the expected demand may not be satisfied. The developers need to evaluate whether it is recommended to lose these requests as this means that some users will get a degraded experience. The balance between cost and service disruptions can be analyzed through multiobjective optimization, where one of the two variables has more importance depending on the specific needs of each particular LAMA.

6.3 Topology

In general terms, a LAMA is an MSA with at least one external API that is regulated by a pricing, which includes, among other things, capacity limitations and usage price.

As shown in Fig. §1.3, the topology of a LAMA can be represented as a DAG (directed acyclic diagram), where, in this particular diagram, each dark node corresponds to an internal service in the LAMA and a light node corresponds to an external API. Each directed edge between nodes represents a consumption from one node to another: e.g. *microservice S1 consumes microservices S2 and S3, microservice S2 consumes external API E1, and microservice S3 consumes external APIs E1 and E2.* Each edge is labeled with the number of requests that are derived from the invocation of the consumer service: e.g. *each time the microservice S1 is invoked, the microservice S2 is consumed 3 times and microservice S3 is consumed 2 times.*

It is worth noting that, for simplicity, we are using a maximal consumption modelling of the LAMA, assuming that every request is always necessary, which is not always true. Sometimes, sending certain requests depends on some conditions that must be met, and this fact could be considered through statistic and probabilistic analysis. Furthermore, we assume that requests do not consume any time and are immediate.

Elements

All LAMAs share a common set of elements, which are the following:

- **Internal services**: these are the internal microservices of the LAMA. They are developed by the LAMA developers, and who have control over their implementation, deployment and, in some cases, maintenance. These services might have internal limitations derived from their deployment infrastructure; however, this information is out of the scope of this thesis. A LAMA should have at least 1 internal service, but most of them have multiple ones. In fact, a LAMA with a single service would actually be a monolithic application.

- **External APIs**: these are the external services consumed by the LAMA. In many cases, these APIs are backed by a MSA or even a LAMA, but this information is not relevant to the businesses who consume them, as they see the APIs as black boxes. For this reason, it is not necessary to know the internal architecture of the external APIs. A LAMA should have at least 1 external API. Otherwise, we would have a regular MSA without limitations.

- **Entrypoint**: the internal service of the LAMA that is invoked first when a customer performs some action. Then, this service sends other requests to other services, which, in turn, may send further requests to other services. A LAMA might have multiple entrypoints, each one of them corresponding to a different action that a customer may perform. Nonetheless, for the analysis in this thesis, we will consider that there is only one entrypoint.

- **Relationships**: the number of requests sent between a pair of internal services, or a pair of an internal service and an external API. Within a LAMA, there should be at least one request from an internal service to an external API. Otherwise, we would have a regular MSA. Furthermore, every external API should receive at least 1 request (if not, the external API would be irrelevant for the LAMA). However, not every single pair of services needs to have a relationship.

- **Pricings**: the pricings of each external API. These pricings should contain all relevant information included in each plan, including the elements described in Section §3.2, such as quotas, rates or usage price. Not all pricings need to have all elements, but they require at least a price or one limitation. Otherwise, they would be irrelevant for the LAMA. Every external API should have a pricing.

Description Language of a LAMA

To make it easier to leverage the information about the topology of a LAMA and the pricings of its external APIs, we propose the use of a formal description language that includes all relevant information described in the previous subsection. This language, which we named *LAMA Description Language* (LDL), contains the internal and external APIs, its relationships (requests sent between them), and all the information regarding pricing plans. Note that LDL is still a work in progress and may be extended in the future if needed. As such, all limitations currently refer to the number of requests. However, for the scope of this thesis, it provides all the data we need. Listing 6.1 shows the LDL of the LAMA in Fig. §1.3.

```
Services S1, S2, S3
External E1, E2
Entry S1
Relationships S1<3>S2, S1<2>S3, S2<2>E1, S3<1>E1, S3<2>E2

Pricings E1->P1, E2->P2
Pricing P1: Basic, Premium
Plan Basic: $5/month
    rate: 15/s
    quota: 1000/day
    ovg: $0.01
Plan Premium: $8/month
    rate: 25/s
    quota: 10000/day
Pricing P2: Silver, Gold
Plan Silver: $4/month
    rate: 10/s
Plan Gold: $10/month
    rate: 20/s
```

Listing 6.1: LDL of the LAMA in Fig. §1.3

While there exist some languages to describe a DAG, we found it easier to create our own language that includes exactly the information that we need. This way, we have control over each field and we can add, modify or remove parts of the language as we see fit. Furthermore, while we have created and extended the SLA4OAI serialization to describe API pricings, we decided to use a simpler notation within LDL. SLA4OAI

includes many fields that are irrelevant to the analysis of a LAMA, such as context information or details about SLA management infrastructure. This makes SLA4OAI too complex and cumbersome for this task, where we simply need the basic information of a pricing. Nonetheless, it should be possible to develop a tool that extracts the relevant information within an SLA4OAI specification file and translate it into LDL.

Capacity Analysis

This chapter presents the concept of capacity analysis of LAMAs. Precisely, Section §7.1 provides an introduction to contents of this chapter. Section §7.2 introduces the definition of the capacity of a LAMA and the main elements involved in its analysis. Then, Section §7.3 includes information about the automatization of this analysis. Finally, Section §7.4 lists a set of analysis operations classified in a catalogue and with various examples.

7.1 Introduction

In Chapter §4, we introduced the concept of capacity and the capacity analysis of a regular MSA, showcasing its necessity. In the following sections, we will ground these concepts within the scope of LAMAs, considering their particular features. Besides the definition of the capacity analysis problem, we provide further details about the elements that are involved in it.

Furthermore, we now introduce a first approach to automate the capacity analysis of a LAMA. This approach is based on a transformational model, which converts the topology of a LAMA and the external API pricings into a constraint satisfaction and optimization problem (CSOP). Each element of the LAMA is transformed into a parameter within the CSOP, while the topology is represented as a set of constraints. Then, an analysis operation would be converted into additional constraints so that the solution of the CSOP provides the solution to the operation. As far as we know, this transformation has never been explored in the literature for the particular features of LAMAs. Furthermore, the declarative nature of a CSOP makes it easy to further expand the model if new elements are added to the LAMA, or if new operations are introduced. We also introduce a catalogue of three basic analysis operations, from which a wide variety of other operations can be defined.

7.2 Capacity of a LAMA

The capacity of a LAMA refers to the maximum workload that it can handle over a given period of time and at a maximum cost, without exceeding any of the external limitations derived from subscribed pricing plans. This definition is in line with the *capacity and performance management practice* in ITIL 4 [64].

The capacity analysis of a LAMA should provide answers to the software architects and DevOps to make decisions over the subscribed external APIs and the potential operating conditions for the LAMA users. In particular, this analysis should take into account three dimensions that are intertwined:

- *Metrics.* This dimension addresses the metrics (bounded to a scale) that have an impact on the capacity or are constrained by external APIs. In this thesis we focus on a single metric, *number of requests*, that is the most widely used metric in the industry [56] and is constrained and limited in most commercial API pricing plans.

It is important to note that the metric should always be bounded to a particular scale. In the case of *number of requests*, we could have different time scales such as Requests Per Second (RPS) or Requests Per Hour (RPH).

- *Temporality.* This dimension represents the temporal boundaries for the capacity to be analyzed. In this context, the same LAMA could have different capacities depending on the time period when it is calculated. These boundaries are typically linked with the desired operating conditions or, in case of a SaaS, the defined pricing plans. In a realistic setting, there could be scenarios where different external APIs have different plan periods and consequently, the capacity analysis should combine multiple temporal perspectives involved.

- *Cost.* This dimension takes into account the derived costs from the infrastructure operation and the cost derived from the contracted plans with the different external APIs. In the example of Fig. §1.3, multiple options are possible, depending on the number of plans contracted; we assume that it is possible to contract multiple times a particular plan as this is the norm in the real API market.

For example, given the LAMA in Fig. §1.3, the capacity can be analyzed by manual calculations. In 1 second, using the cheapest plans (*Basic* and *Silver*) and no overage cost, the capacity of the LAMA is 1 RPS, because 1 request to S1 results in 8 requests to E1, and one more request to S1 would result in 16 requests to E1, thus exceeding the limitation of the *Basic* plan. The cost is a fixed value, $9 in this case. In 2 seconds, the maximum number of requests allowed to E1 using the *Basic* plan is 30; therefore, the capacity is 3 RPS, resulting in 24 requests to E1 and 12 to E2. The cost, however, remains the same.

When dealing with real-world architectures, the number of internal services and external APIs is considerably high, and thus there is a great number of plans and possible combinations. Additionally, when defining the pricing plans to be offered to the LAMA customers, it is fundamental to know the limitations derived from the usage by the external APIs together with its associated cost. In fact, these costs will be part of the operational costs of the LAMA, and are essential when analyzing the OpEx (Operational Expenditures) [65] for the desired operating conditions in general, and to have profitable pricing plans in the case of a SaaS LAMA.

7.3 Automated Capacity Analysis

Automated LAMA capacity analysis deals with extracting information from the model of a LAMA using automated mechanisms. Analyzing LAMA models is an error-prone and tedious task, and it is infeasible to do it manually with large-scale and complex LAMA models. In this thesis we propose a similar approach to that followed in other fields, i.e., to support the analysis process from a catalogue of analysis operations (analysis of feature models [66, 67], service level agreements [68, 69, 70] and Business Process [71]).

In this sense, all the analysis operations we have faced so far can be interpreted in terms of optimal search problems. Therefore, they can be solved through Search Based Software Engineering (SBSE) techniques, similarly to other cloud engineering problems [72]. We tackle this problem as a Constraint Satisfaction and Optimization Problem (CSOP), where, *grosso modo*, the search space corresponds to the set of tuples $(Requests, Time, Cost)$ that conform valid operating conditions of the LAMA. The objective function is defined on the variable that needs to be optimised in each case: requests, time or cost.

Formal Description of LAMAs

The primary objective of formalizing a LAMA is to establish a sound basis for the automated support. Following the formalization principles defined by Hofstede et al. [73], we follow a transformational style by translating the LAMA specification to a target domain suitable for the automated analysis (*Primary Goal Principle*). Specifically, we propose translating the specification to a CSOP that can be then analyzed using state-of-the-art constraint programming tools.

A CSOP is defined as a 3-tuple (V, D, C) composed of a set of variables V, their domains D and a number of constraints C. A solution for a CSOP is an assignment of values to the variables in V from their domains in D so that all the constraints in C are satisfied.

Table §7.1 describes the mapping of a LAMA to a CSOP. Because of its complexity, we describe each transformation from the table in the following paragraphs. We recommend that readers get familiarized with this mapping. Table §7.2 summarizes the meaning of each of the abbreviations used in the mapping.

- **Positive number of requests**. All internal services and external APIs must serve a positive number of requests. Therefore, all variables req_{S_i} and req_{E_i} that denote the

request served by internal services and external APIs respectively must be greater than or equal to 0.

- **Requests served by internal services**. Each internal service in the LAMA S_i must serve all requests sent to it by every other service S_j, denoted as $req_{S_jS_i}$. Thus, for each internal service there is a constraint $req_{S_i} = \sum_{j=1}^{n} req_{S_jS_i} \cdot req_{S_j}$.

- **Requests served by external APIs**. Each external API E_i must serve all requests sent to it by the internal services S_j, denoted as $req_{S_jE_i}$. External APIs do not send requests between them. Thus, for each external API there is a constraint $req_{E_i} = \sum_{j=1}^{n} req_{S_jE_i} \cdot req_{S_j}$. Additionally, the total number of served requests is the sum of the requests sent to each plan below its limitations, $limReq_{ij}$, and the requests sent over the limitations, $ovgReq_{ij}$. This differentiation in two variables helps us obtain the number of overage requests more easily. Thus, for each plan P_{ij} of external API E_i there is a constraint $req_{E_i} = \sum_{j=1}^{n} limReq_{ij} + ovgReq_{ij}$. Furthermore, no requests can be sent using a plan with no keys, so for each plan there is a constraint $limReq_{ij} > 0 \rightarrow keys_{ij} > 0$. Also, no overage requests can be sent if there are no requests below limitations, so for each plan there is another constraint $ovgReq_{ij} > 0 \rightarrow limReq_{ij} > 0$.

- **Quota of each pricing plan**. The number of requests served by each external API E_i must not exceed any quota q_{ij}, defined over a time unit qu_{ij}. Multiple keys $keys_{ij}$ for each plan may be obtained. For each external API E_i and each of its respective plans P_{ij}, there is a constraint $limReq_{ij} <= keys_{ij} \cdot q_{ij} \cdot \lceil time/qu_{ij} \rceil$.

- **Rate of each pricing plan**. The number of requests served by each external API E_i must not exceed any rate r_{ij}, defined over a time unit ru_{ij}. Note that rates need to account for the time unit of the quota, as the rate is reset at the beginning of each unit. Therefore, for each external API E_i and each of its respective plans P_{ij}, there is a constraint $limReq_{ij} - qu_{ij} \cdot \lfloor time/qu_{ij} \rfloor <= keys_{ij} \cdot r_{ij} \cdot \lceil time \mod qu_{ij}/ru_{ij} \rceil$. If a plan has no no quota, the constraint is simplified to $limReq_i <= keys_{ij} \cdot r_{ij} \cdot \lceil time/ru_{ij} \rceil$.

- **OpEx of each external API**. The cost of each external API E_i is the sum of the subscriptions to each plan P_{ij} plus overage costs. For each external API E_i, there is a constraint $OpEx_i = \sum_{j=1}^{n} keys_{ij} \cdot cost_{ij} + ovg_{ij} \cdot ovgReq_{ij}$.

- **Total OpEx**. The total cost of the LAMA is the sum of the cost of each external API. There is a constraint $OpEx = \sum_{i=1}^{n} OpEx_i$.

Table 7.1: LAMA to CSOP mapping.

LAMA Services and APIs		CSOP Mapping
	Services S	$\forall\, S_i \text{ in } S, \begin{cases} V \leftarrow V \cup req_{S_i} \\ D \leftarrow D \cup domain(req_{S_i}) \\ C \leftarrow C \cup req_{S_i} >= 0 \end{cases}$
	External E	$\forall\, E_i \text{ in } E, \begin{cases} V \leftarrow V \cup req_{E_i} \\ D \leftarrow D \cup domain(req_{E_i}) \\ C \leftarrow C \cup req_{E_i} >= 0 \end{cases}$
	Pricings $E_i \rightarrow P_i$ Pricing P_i: $P_{i1}, ..., P_{in}$	$\forall\, E_i \text{ in } E \,\forall\, P_{ij} \text{ in } P_i, \begin{cases} V \leftarrow V \cup limReq_{ij} \cup ovgReq_{ij} \cup keys_{ij} \\ D \leftarrow D \cup domain(limReq_{ij}) \cup domain(ovgReq_{ij}) \\ C \leftarrow C \cup limReq_{ij} > 0 \rightarrow keys_{ij} > 0 \cup \\ ovgReq_{ij} > 0 \rightarrow limReq_{ij} > 0 \end{cases}$ $\forall\, E_i \text{ in } E, C \leftarrow C \cup req_{E_i} = \sum_{j=1}^{n} limReq_{ij} + ovgReq_{ij}$
LAMA Elements		CSOP Mapping
Entry	Entry S_i	$V \leftarrow V \cup reqL$ $D \leftarrow D \cup domain(reqL)$ $C \leftarrow C \cup req_{S_i} >= reqL$
Consumption	[API] Rltshp $S_1 < req_{S_1 E_i} > E_i, ...,$ $S_n < req_{S_n E_i} > E_i$	$\forall\, E_i \text{ in } E,\ C \leftarrow C \cup req_{E_i} = \sum_{j=1}^{n} req_{S_j E_i} \cdot req_{S_j}$
	[Service] Rltshp $S_1 < req_{S_1 S_i} > S_i, ...,$ $S_n < req_{S_n S_i} > S_i$	$\forall\, S_i \text{ in } S,\ C \leftarrow C \cup req_{S_i} = \sum_{j=1}^{n} req_{S_j S_i} \cdot req_{S_j}$
Limitations	Plan P_{ij} rate: r_{ij}/ru_{ij} quota: q_{ij}/qu_{ij}	$V \leftarrow V \cup time$ $D \leftarrow D \cup domain(time)$ $\forall\, E_i \text{ in } E \,\forall\, P_{ij} \text{ in } P_i,\ C \leftarrow C \cup limReq_{ij} <= keys_{ij} \cdot q_{ij} \cdot \lceil time/qu_{ij} \rceil \cup$ $limReq_{ij} - qu_{ij} \cdot \lfloor time/qu_{ij} \rfloor <= keys_{ij} \cdot r_{ij} \cdot \lceil time \mod qu_{ij}/ru_{ij} \rceil$
Individual cost	Plan $P_{ij} cost_{ij}$ ovg: ovg_{ij}	$\forall\, E_i \text{ in } E \,\forall\, P_{ij} \text{ in } P_i, C \leftarrow C \cup ovg_{ij} = 0 \rightarrow ovgReq_{ij} = 0$ $\forall\, E_i \text{ in } E, \begin{cases} V \leftarrow V \cup OpEx_i \\ D \leftarrow D \cup domain(OpEx_i) \\ C \leftarrow C \cup OpEx_i = \sum_{j=1}^{n} keys_{ij} \cdot cost_{ij} + ovgReq_{ij} \cdot ovg_{ij} \end{cases}$
Total cost	Plan $P_{ij} cost_{ij}$ ovg: ovg_{ij}	$V \leftarrow V \cup OpEx$ $D \leftarrow D \cup domain(OpEx)$ $OpEx = \sum_{i=1}^{n} OpEx_i$

Table 7.2: Parameters and variables glossary.

Parameters	**Definition**
S_i	An internal service of the LAMA.
E_i	An external API consumed by the LAMA.
P_i	Pricing offered by E_i
P_{ij}	j-esim pricing plan of P_i.
$req_{S_iS_j}$	Number of requests served by service S_j each time S_j is invoked.
$req_{S_iE_j}$	Number of requests served by external API E_j each time S_i is invoked.
r_{ij}	Rate of plan P_{ij}.
ru_{ij}	Time unit of the rate of plan P_{ij}.
q_{ij}	Quota of plan P_{ij}.
qu_{ij}	Time unit of the quota of plan P_{ij}.
$cost_{ij}$	Subscription cost of plan P_{ij}.
ovg_{ij}	Overage cost of plan P_{ij}.
Variables	**Definition**
$reqL$	Number of requests served by the LAMA, equivalent to the number of requests of the entrypoint.
req_{S_i}	Number of requests served by service S_i.
req_{E_i}	Number of requests served by external API E_i.
$limReq_{ij}$	Number of requests served using plan P_{ij} within its limitations.
$ovgReq_{ij}$	Number of requests served using plan P_{ij} beyond is limitations.
$keys_{ij}$	Number of keys subscribed for plan P_{ij}.
$time$	Time period.
$OpEx_i$	Total cost of external API E_i.
$OpEx$	Total cost of the entire LAMA.

7.4 Analysis Operations

We propose a catalogue of three analysis operations that leverage the formal description of LAMAs as a CSOP to automatically extract helpful information. Analogous analysis operations have been defined in the context of the automated analysis of feature models [66], service level agreements [68, 69, 70] and in the area of MSAs [54]. We may remark that it is not our intention to propose an exhaustive set of analysis operations, as an unbounded number of operations could be potentially defined by adding or removing constraints and parameters.

For the description of the operations as CSOPs, we will refer to the input specification of a LAMA `L` and a variable `v`. Additionally, we will use the following auxiliary operations:

- `map(L)`. This operation translates a LAMA specification `L` to a CSP following the mapping described in the previous section.
- `minimize(CSP, v)`. This standard CSOP-based operation returns a solution for the input `CSP` (if any) with the minimum value of variable `v`.
- `maximize(CSP, v)`. Same that prior operation but with the maximum value of variable `v`.

In what follows, we present three basic analysis operations, and, for the first operations identified in Section §4.3 (now replacing the term *MSA* with *LAMA*), and using the LAMA in Fig. §1.3, we provide an explanation of how it is mapped to a CSOP from the corresponding basic operation.

Maximum number of requests. This operation returns the maximum number of requests that a LAMA L is able to serve, over a specific time window t and for a maximum total cost c. This operation can be translated to a CSOP as follows:

$$\text{maxRequests}(\text{L}, \text{t}, \text{c}) \iff \text{maximize}(\text{map}(\text{L}) \ \wedge \text{time} = \text{t} \wedge \text{OpEx} <= \text{c}, \text{reqL})$$

With this operation we can answer question Q2 (*Assuming we have a Basic plan and a Gold plan already contracted what is the maximal RPM I can guarantee to all my 20 customers?*) in Section §4.3 resulting in 5.6 RPS to each customer:

$$\text{Q2} \iff \text{maxRequests}(\text{L}, 60\text{s}, 15)/20 = 5.6\,\text{req}$$

Similarly, question Q3 (*Assuming we have a monthly budget limit of $120 in my LAMA, which is the maximum RPS to each of 20 customers?*) is translated into the expression $\text{maxRequests}(\text{L}, 1\text{s}, 120)/20$, resulting in 1.35 (that is, 1) RPS to each customer:

$$\text{Q3} \iff \text{maxRequests}(\text{L}, 1\text{s}, 120)/20 = 1.35\,\text{req}$$

Minimum cost. This operation returns the minimum cost of the LAMA L, so that it can serve a minimum of RL requests over a time window t. From the result of this operation we can obtain the optimum (cheapest) plan combination (including the number of keys to be subscribed for each plan and possible overage requests). The translation of this operation to a CSOP is as follows:

$$\text{minCost}(\text{L}, \text{RL}, \text{t}) \iff \text{minimize}(\text{map}(\text{L}) \wedge \text{reqL} = \text{RL} \wedge \text{time} = \text{t}, \text{OpEx})$$

The question Q1 (*What is the cheapest operational cost for my LAMA in order to offer 2 RPS to 20 customers?*) is translated into $\text{minCost}(\text{L}, 2 \cdot 20, 1\text{s})$, resulting in a total cost of $174:

$$\text{Q1} \iff \text{minCost}(\text{L}, 2 \cdot 20, 1\text{s}) = \$174$$

Minimum time. This operation returns the minimum time that a LAMA L needs to serve at least RL requests, given a maximum total cost c. This operation can be translated to a CSOP as follows:

$$\text{minTime}(\text{L}, \text{RL}, \text{c}) \iff \text{minimize}(\text{map}(\text{L}) \wedge \text{reqL} = \text{RL} \wedge \text{OpEx} <= \text{c}, \text{time})$$

Monitoring Model

This chapter presents a monitoring framework to automatically collect traces and metrics from a LAMA and infer its topology. First, Section §8.1 gives an overview of the framework and its usefulness for the capacity analysis of a LAMA. Next, Section §8.2 introduces a list of requirements that need to be met by a framework to be useful for the capacity analysis of a LAMA. Then, Section §8.3 details the framework itself and its components.

8.1 Introduction

As previously discussed in Section §6.2, the problem of impedance mismatch in LAMAs is an important issue related to the capacity analysis of the architecture. It is useful for businesses to find out whether the expected consumption of the external APIs is actually correct when the system is in production and being used by its customers. If it is not, the situation can result in unexpected service disruptions or charges derived from overage costs. For these reasons, it is necessary for businesses to use some tools to analyze the real consumption of the APIs so that they can take action if they need.

In this chapter, we present an initial implementation of a monitoring framework for LAMAs. First, we describe a list of requirements that this framework must meet in order to be useful for the analysis of a LAMA. For example, it should be able to automatically identify the internal services and the external APIs, as well as detect all intermediate requests and keep track of their order and hierarchy. Then, we introduce our framework, which consists of an agent and a collector. The agent is an installable software that can be added to the internal services of the LAMA, which then reports all collected traces and metrics to the collector. Next, the data can be extracted from the collector to process it and automatically infer the topology of the LAMA. Moreover, this inferred topology could then be translated into LDL, the domain specific language for LAMAs introduced in Section §6.3.

8.2 Requirements for LAMA Demand Analysis

In Section §6.2, we discussed how understanding the need for resources in a LAMA involves tracking how much communication happens between its own services and with outside web services. Often, these actual communication numbers do not align with what the LAMA developers initially thought they would need, leading to problems such as system crashes or unexpected bills.

To address these issues, we suggest using telemetry, a method to gather important data, such as traces and metrics, which helps in figuring out how much capacity a LAMA has. The field of telemetry is broad, with many different standards that cover a wide range of data points and metrics, like how long it takes for data to travel across the network. Many of these details are not necessary for understanding a LAMA's capacity and only end up using more resources (like taking up more network bandwidth or filling up storage space) without being helpful. That is why it is important to focus on a telemetry standard

that only collects the exact information needed for analyzing a LAMA's capacity. This means the standard should:

1. Collect only the essential traces and metrics to avoid using too much bandwidth and storage.
2. Only include reports from the LAMA's own services to keep the data relevant.
3. Ensure that services share identifiable information, such as their names, IP addresses, or the processes they are running, to make it easier to tell them apart.
4. Require that any data shared by a service gives enough context to understand how it fits into the overall flow of requests, with each action in this flow called a *span*.
5. Make sure each span within a trace is unique and gives details about an action taken between two services at a specific time, mainly through HTTP requests, but be open to including other communication methods if necessary.
6. Link spans and metrics from the same service by time, to keep the data organized.
7. Allow services to skip reporting a metric if it is not helpful, to avoid gathering unnecessary information.
8. Ensure that the metrics shared are useful for figuring out how to adjust the capacity as needed, especially by showing how much of the system's resources are being used.

By following these guidelines, a telemetry system should be able to collect all the important information needed to map out how the LAMA's internal services and external web services interact. This can also help in spotting potential problems before they cause the system to crash, by analyzing the data and making predictions.

8.3 Monitoring Framework

This section introduces the monitoring framework, designed to fulfill the requirements outlined previously. At its core, the monitoring framework is comprised of an *agent* responsible for exporting telemetry data and a *collector* that archives this data in a non-relational database. Before expanding on these components, we provide an overview of the underlying technology. The inception of the monitoring framework traces back to [74], aimed at automating the collection of network traces to deduce the topology of a LAMA,

thereby facilitating capacity analysis. Given its relevance to this thesis, we summarize pivotal aspects of this work, noting our collaborative efforts and joint publication at the conference JCIS 2023 [24].

Technological Foundation

The framework's agent and collector are both developed in Node.js, utilizing TypeScript and JavaScript. This setup restricts monitoring to Node.js-based services for the agent, while the collector remains language-agnostic, capable of integration with any service, provided an agent is present for data export. The agent leverages JavaScript libraries from OpenTelemetry.

OpenTelemetry, as briefly mentioned in Section §2.3, serves as a vendor-neutral framework for monitoring and trace collection. It offers SDKs, APIs, and tools for data collection, transformation, and delivery to monitoring solutions. The Node.js SDK from OpenTelemetry simplifies exporting traces, metrics, and managing trace contexts.

The concept of a *collector* in OpenTelemetry is crucial, acting as an intermediary that processes and forwards data from agents to monitoring backends. While it supports numerous monitoring systems like Jaeger and Zipkin, integration with traditional databases like MongoDB requires custom collector implementations using *gRPC*.

gRPC, developed by Google, is an open-source remote procedure call (RPC) system that enhances the development of distributed services by enabling seamless server-client interactions over a unified TCP/IP connection. Utilizing HTTP/2, gRPC offers reduced latency, supporting efficient, continuous data transmission and facilitating asynchronous, bidirectional communication.

Framework Architecture

Aligned with the specified requirements in Section §8.2, the Monitoring Framework features two principal components: an agent and a collector. Its architecture, designed to monitor RESTful Microservice Architectures (MSAs) not limited to LAMAs, employs OpenTelemetry and gRPC, with development in Node.js. Consequently, only Node.js services are currently monitorable, and the data is stored in a non-relational database, suitable for handling vast datasets.

The agent collects data from an MSA's internal services, transmitting it to the collector through a process known as *instrumentation*. Installation within a service is requisite for data collection, facilitated by the agent's modular design and configurable parameters.

The collector embodies the telemetry model, receiving, processing, and storing data from agents. It comprises a gRPC server for data reception and a database driver for storage. Internally, it modifies the data for efficient database storage. The collector data model is based on the document oriented persistency with MongoDB connecting via the *mongoose* driver. Metric and trace models are defined using mongoose schemas, incorporating essential details like name, version, and IP address. Traces, comprising transaction series within the same context, are central to the model, with the OpenTelemetry Node.js SDK automatically associating spans by context.

Agent. As a Node.js module, the Monitoring Framework agent monitors system events, reporting traces and metrics to the collector. Adhering to the requirements from Section §8.2, the agent utilizes the OpenTelemetry SDK, enhanced for specific needs, including configurable parameters like collector URL and service name, essential for service identification within a LAMA. Data transmission to the collector is facilitated through OpenTelemetry's HTTP and Express.js SDKs.

Collector. The collector manages the reception and database storage of collected data, incorporating a gRPC server for agent communication. It processes and stores traces and metrics, additionally offering a RESTful API for data retrieval in JSON or CSV formats. This data can then be analyzed to deduce the LAMA's topology, with the process and inference details provided in an online Jupyter notebook. This notebook demonstrates the methodology using real-world APIs, translating the inferred topology into LAMA-DL for Smart LAMA application, and presents various analysis examples.

Part IV

VALIDATION

Pricing Model

This chapter describes the validation of our extended pricing model to determine its expressiveness and usefulness. Section §9.1 provides an overview of the validation process. Section §9.2 presents a detailed analysis of the model in an extensive set of real-world APIs, including various metrics and statistics. Then, Section §9.3 introduces an automated tool for the validation of a pricing according to a set of validity criteria.

9.1 Introduction

In order to validate the usefulness of our Pricing4APIs model, and the SLA4OAI serialization in particular, we carried out two different experiments.

First, we analyzed a representative set of real-world and publicly available APIs found in the industry to check whether their elements can be modeled using Pricing4APIs. We provide a detailed analysis including various metrics and statistics to showcase the most common elements in these pricings. Then, we manually modeled a subset of these real-world API pricings to validate if our serialization is actually able to cover all of their aspects. We concluded the experiment with a series of remarks and a clear description of the elements that, as of today, are not supported by Pricing4APIs or SLA4OAI.

Next, we developed a tool to automatically validate an API pricing, that is, automatically answering the analysis operation O_{12} in Section §5.4. This tool is publicly available as a command-line tool and as a RESTful API, and can be used to validate any pricing modeled using SLA4OAI. We validated the tool with the subset of modeled APIs from the previous experiments. Initially, thanks to this tool, we found out some modeling mistakes that we made ourselves. When corrected, we observed that none of the validated pricings had any conflict, which is to be expected because real-world API providers try to carefully craft their pricings so that their customers do not have any issues or doubts when choosing between plans.

9.2 Expressiveness

Analyzing API Limitations and Pricing

For this analysis, we considered three different sources: (i) the previous work in Gamez-Diaz et al. [56] in which the authors analyzed a set of 69 APIs from two of the largest API directories; (ii) the work of Neumann et al. [75] in which the authors analysed a set of 500 APIs from the top most popular 4,000 websites in the Alexa ranking [76]; (iii) the 27 most popular APIs from RapidAPI[1].

We adapted and applied the process described in contribution [56] (i), screening the API repositories and applying the inclusion criterion described by the authors (*which*

[1] https://rapidapi.com/collection/popular-apis. Accessed on January 2023. Note that this list is regularly updated, and some APIs may be added, deprecated or removed. Some of the APIs included in our analysis are no longer available.

includes more than 5,000 APIs with a last update in 2020). The result was the selection of one source: ProgrammableWeb. We extracted the most popular API categories (97*th* percentile, i.e., 14 categories selected, with more than 16,500 APIs). We filtered this dataset by removing duplicates (only one API per company was chosen at random). As a result, we had 2,966 potential APIs to study. Out of them, 30 APIs were selected.

In contribution [75] (ii), the authors analyzed a set of attributes of 500 APIs by focusing on their general features such as their fit to REST best practices and design decisions rather than on their specific pricing aspects. Nevertheless, this dataset is interesting as a starting point for our analysis since it includes a variety of APIs and provides a comprehensive analysis of certain attributes. From this dataset, we filtered out any rows which were not RESTful APIs, leaving a subset of 499 unique APIs. First, we selected those APIs with a *Payment Plan*, as specified in a column in the dataset, obtaining 55 APIs, which represents the 11.02% of the total 499 APIs. We noted some errata in the classification of some APIs that we are very familiar with (e.g., GitHub was wrongly classified in the *not having plans* section). The reason behind these errors might be that the APIs were analysed some time ago, when they did not have plans at the time; alternatively, some APIs might have their pricing plans *hidden* within their documentation (e.g., we found that Yelp has an implicit VIP plan). This led us to analyze the rest of the dataset (444) manually to check whether the API still existed and whether it had API limitations. This analysis resulted in 162 APIs to be included (67 with pricing plans and 95 without but with API limitations, which represent 15.09% and 21.4% respectively of these 444 APIs originally classified as "not having plans"). Adding these to the first set of 55 APIs, led to 217 APIs to be analyzed.

The list of most popular APIs from RapidAPI (iii) includes 27 different RESTful APIs. RapidAPI acts as a gateway to these APIs and provides simple pricing options. Some of these pricings differ from the pricings offered by the API providers in their official websites, which are often more complex. After a manual analysis of all of them, we found that 22 APIs had pricing plans and limitations, while 5 had no plans or limitations.

Combining the 30 APIs extracted according to [56], the 217 from the dataset in [75] and the 22 from RapidAPI and removing duplicates left a dataset of 268 APIs – the *Pricing4APIs dataset*. Table §9.1 presents the overall picture of the analysis that was carried out. Out of more than 17 027 APIs, we manually modeled 54 of them, having a 90% confidence level and an 11% margin of error [77]. The full Pricing4APIs dataset, including details about their attributes, is available at [78] as part of Dataset D01.

We analyzed 268 APIs in regard to two main types of attributes: *limitations* and

Table 9.1: Main numbers of our Pricing4APIs dataset

APIs from Gamez-Diaz (N=2966)	30
APIs from Neumann (N=217)	217
APIs from RapidAPI (N=27)	22
Total APIs after removing dups.	268
APIs having pricing	176 out of 268
Manually modeled APIs (N=266)	54

pricing. Both types include a wide range of other attributes, some supported by our model but others not. For example, our model does support overage costs (e.g., $0.1 per exceeded request), but it does not support complex metrics based upon HTTP protocol-related aspects (i.e., headers, parameters, etc.).

Although the Pricing4APIs dataset comprises 268 APIs, only 176 of them, the 65.7%, present a pricing or a plan. Consequently, the analysis of pricing is limited to this reduced dataset. Table §9.2 presents some results of the analysis.

Table 9.2: Results of the analysis in real-world APIs

	N=268	N=176
Has limitations	95.9%	94.9%
Has quotas	59.7%	72.2%
Has rates	78.7%	69.9%
Has quotas and rates	42.5%	46.6%
Simple cost (e.g., monthly price)	60.8%	92.6%
Has a pay-as-you-go cost model	9.3%	14.2%
Includes overage cost	11.9%	18.2%

Limitations analysis: Most APIs (95.9%) have limitations in terms of quotas (59.7%) or rates (78.7%). Almost half use a combination of the two (42.5%). These limitations are usually rather simple (such as monthly requests for quotas and secondly requests for rates). However, a minority tend to have a higher level of expressiveness. For example, they use the information from the HTTP request – from query parameters (2.2%) to other low-level aspects of the HTTP message such as headers, body, etc. (9.3%). A marginal number of APIs allow consumers to exceed the limitation value once or many times per month (1.1%).

Pricing analysis: the vast majority of the APIs (92.6%) include a simple (e.g., monthly) cost. Nonetheless, they may have operation costs (14.2%) or include overage costs (18.2%). Finally, a minority have purchasable add-ons or extras (5.7%) or their pricing is calculated based on the number of users (10.2%).

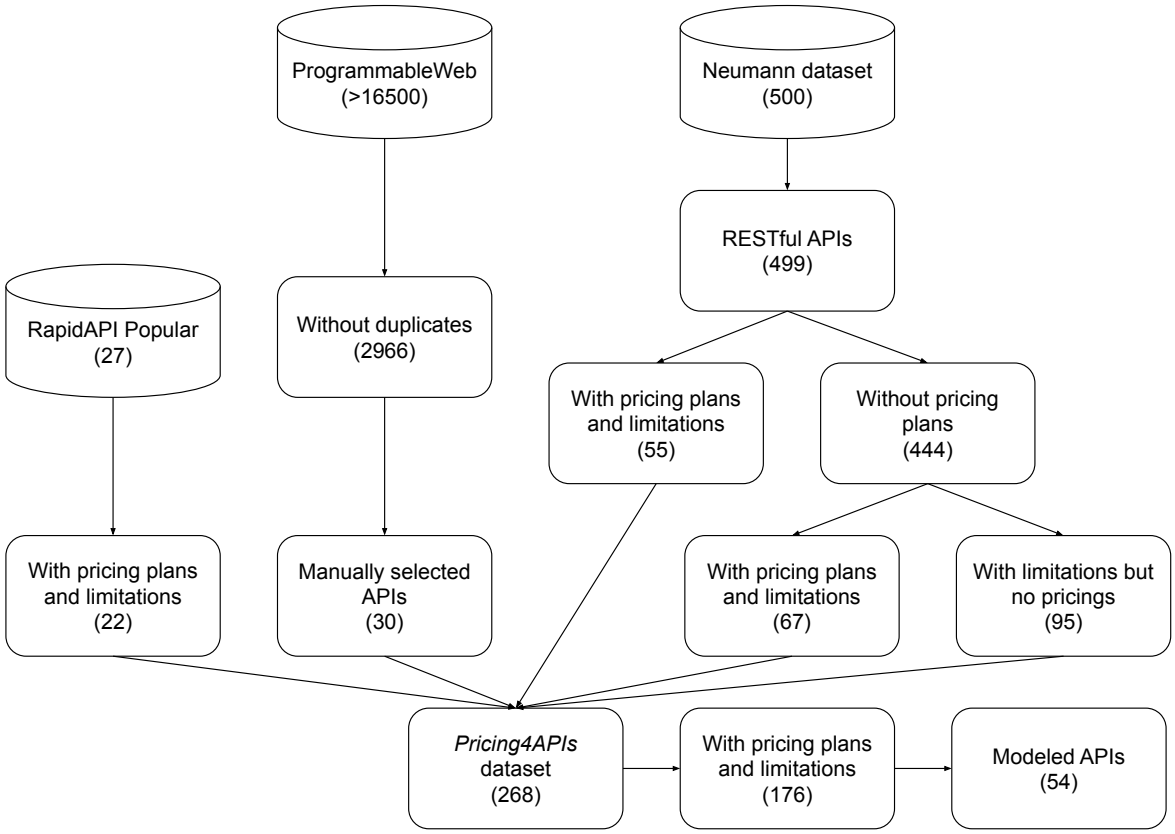

Figure 9.1: Diagram of the database filtering process, starting from the Gamez-Diaz and Neumann datasets and the RapidAPI most popular list

Given the 54 modeled APIs, we analyzed the different metrics included in the documentation of each of them. We found a total of 145 metrics, although different providers may name a same metric with different names (e.g. *requests* and *transactions*). 61.38% of these metrics are domain independent (such as requests, storage or users), while 38,62% are dependent (such as emails, documents or invoices). We grouped the 145 metrics in different categories based on their similarity, resulting in 14 categories. The most populated one is *requests*, including 58 metrics. The second one is *AI* (Artificial Intelligence), with 19 metrics, as some of the analyzed APIs are related to artificial intelligence and include a considerable amount of metrics. Many categories only include a few metrics

because they are difficult to group together. This analysis is available at [78] as part of Dataset D02.

Modeling API Pricings

This subsection describes a validation of our Pricing4APIs model by modeling a number of real-world APIs, first describing the modeling process and then the issues that arise during this process. This process included the construction of a curated list of 54 API pricings with the model in Section §5.3, which represents the variability found in the industry.

As noted above, we analyzed different attributes of the Pricing4APIs dataset of 268 APIs. The next step would be to write the SLA4OAI specification of every API so that it can be passed to the automated analyzer. However, since this is a time-consuming task, we decided to follow a hybrid sampling approach, using purposive and probabilistic sampling [79], to obtain a subset of APIs. With the former, we wanted to ensure that our model covers the most representative elements of API pricings, so we modeled all 22 APIs from RapidAPI's most popular list. With the latter, we aimed to reduce the threats of purposive sampling by modeling 32 additional pricings. According to [79], purposive sampling is the most common approach in software engineering research, used in 76% of studies. The resulting subset includes 54 APIs.

Note that the process of modeling a single API pricing consists of (i) reading and understanding the entire API documentation, (ii) extracting the API endpoints and methods (skipped if OAS documentation is available), (iii) reading and understanding every limitation in every plan of the API pricing, and (iv) specifying the metrics and API limitations in accordance with our proposed model for each API path and method. The process of modeling the API itself is tedious, which is why APIs having a public OAS documentation greatly facilitate the subsequent modeling task. With the introduction of globbing, step (ii) is simplified when the same limitation applies to multiple endpoints, and even completely unnecessary when the limitation applies to all endpoints. The 22 APIs from RapidAPI were modeled using globbing, which resulted in much simpler files.

In the following sections, we determine the issues found during this OAS modeling process.

In the process of modeling the pricings of this subset of 54 APIs, we encountered several issues. We have classified them into two categories: *modeling issues* and *open issues*, depending on whether they are issues that can be partially modeled with SLA4OAI or issues that need changes that will be taken into consideration when establishing future

work.

Modeling issues:

MI-01 In *pay-as-you-go* plans, users are only charged with the requests that they actually consume (e.g., *FacePlusPlus*). This situation was modeled as a quota, with no *max* field (or *max: unlimited*) with its corresponding *OverageCost*. As an alternative, we could also have modeled this as an *OperationCost*.

MI-02 In some APIs (e.g., *FacePlusPlus*), the *operation cost* depends on the HTTP status code that is returned to the consumer. Hence, the same request to the same endpoint might well be billed differently with regard to the status code (e.g., $0.01 if 200 OKs and $0.005 if 400 Bad Requests). We modeled this situation as a new metric for each status code. For example, in *FacePlusPlus*, the *QPS* metric has been split into *QPS_OK*, *QPS_timeout* and *QPS_invalidParam*.

MI-03 If a certain plan explicitly denies access to certain API operations (e.g., *Azure Search*), those operations are not included in the model.

MI-04 If the actual value for a quota or rate is unknown (e.g., *Accuweather*), we omit this rate/quota. For example, a number of APIs explicitly mention that they apply some rate-limiting value, but they do not mention what the actual value is.

MI-05 Some metrics are dependent on some aspect of the HTTP request (body, parameters, etc.) and do not have any associated period (e.g., *FacePlusPlus*). In this case, the *period* property is removed.

MI-06 There are also pricings with unknown cost (such as educational plans, non-profit organizations, enterprise, etc.). These are modeled with *custom: true* (e.g., *GeoRanker*). Additionally, if a limitation has a custom value to be negotiated with its provider, it is also modeled with *custom: true* (e.g., *Yelp*).

MI-07 In plans whose billing depends on the number of users (e.g., *Box*) or on other variables affecting the cost (number of organizations, consumers, accounts, etc.), we considered the minimum number allowed. For example, plan *Business Starter* of *Box* requires a minimum of 3 users, and it becomes more expensive if more users join the plan. Therefore, for the sake of simplicity, we consider the cost of *Business Starter* to be the cost for 3 users.

MI-08 Finally, in APIs whose documentation does not specify whether the time window in which limits are calculated is fixed or sliding, we assumed that limits with longer

periods (e.g. years or months) use fixed windows, and limits with shorter periods (e.g. seconds or minutes) use sliding windows. This decision is based on the research in [56].

Open issues:

OI-01 Some HTTP query parameters are limited to a certain range of allowed values instead of a maximum value (e.g., *Scopus*). Despite the fact that we modeled some parameters as a metric (e.g., number of results), parameters within a range were not modeled. In the Scopus case, *Scopus Search* limits the number of results to 25 in the *non-subscriber* plan, whereas this number rises to 200 in the *subscriber* plan. Nevertheless, it also limits the parameter *view* to *STANDARD* in the former case and allows *COMPLETE* only in the latter.

OI-02 Another open issue arises when the overage cost is also limited (e.g., *Georanker*). Some providers force one to move to another plan if one surpasses a certain value of the overage cost. This situation has not been modeled. For example, the *small* plan includes 300,000 requests, with an overage cost of $0.001 per request. However, this overage cost goes up to 750,000 requests. Once this amount is reached, one has to move to the *medium* plan.

9.3 Automation

In this section, we focus on automating the validation of a pricing, that is operation O_{12} in Section §5.4. This operation, in order to be useful for practitioners, needs to be automated by means of a specific tool. To this end, we have developed *SLA4OAI-Analyzer*[2] , a publicly available command-line tool prototype [80]. Once installed, given a SLA4OAI file, the command *sla4oai-analyzer -o <operation> -f <myFile.yaml>* will initiate the validity analysis for this file.

For example, for the validity operation, *sla4oai-analyzer* first checks the syntax validity according to the JSON Schema defined in the repository, and then checks each validity criterion in each part (pricing, plan, limitation, and limit). Fig. §9.3 depicts a consistency conflict detected by this tool, caused by a modeling mistake.

As illustrations of some outputs of the tool, Fig. §9.2 shows a pricing with *syntax errors* and Fig. §9.3 a *consistency conflict*.

[2] https://github.com/isa-group/sla4oai-analyzer

```
 > sla4oai-analyzer -o validity -f '.\yelp.yaml'
------ BEGIN CHECKING FILE: .\yelp.yaml ------
CHECKING SYNTAX...
SYNTAX ERRORS in yelp.yaml
  SYNTAX ERROR: in path "#/":
    Missing required property: metrics
------ END CHECKING FILE: .\yelp.yaml ------
```

Figure 9.2: Tool running a syntax check

```
 > sla4oai-analyzer -o validity -f '.\inconsistent-ex.yaml'
------ BEGIN CHECKING FILE: .\inconsistent-ex.yaml ------
CHECKING SYNTAX...
SYNTAX OK
CHECKING VALIDITY...
  USING DEFAULT CAPACITY
    LIMIT CONSISTENCY CONFLICT:
      in Plan1>/method1>get>requests
      ('60 per 60/second' and '1 per 1/second')
VALIDITY ERROR
------ END CHECKING FILE: .\inconsistent-ex.yaml ------
```

Figure 9.3: Tool running the validity operation with errors

Pricing Validation API

The SLA4OAI-Analyzer is also available as an API at [81]. Its code is publicly available at [82]. The API takes a URL that points to an SLA4OAI specification file as an input and checks for all conflicts that were presented in the previous subsection. The API returns a Boolean value that indicates whether the file is valid, and a detailed log showing the validity of each individual element of the pricing.

To validate the API, we provide an online Jupyter notebook that is available at [83]. Note that the notebook is shared with *execute* access, meaning that the cells can be executed but not modified. Nonetheless, they may be modified if the notebook is duplicated into your own Deepnote account. The notebook includes two examples for each validity subcriterion: one example has a conflict and the other does not. Because the full SLA4OAI specification files are complex and contain additional information that is not relevant for the detection of conflicts, each example includes a short pseudocode fragment, that shows the pricing elements that are actually relevant for the corresponding example.

Alternatively, the API also accepts a full SLA4OAI file directly within the request body. There is a cell at the end of the notebook that contains an example of this.

Furthermore, we provide a Postman documentation[3] with 54 examples of invocations of the validity operation using this API. Fig. §9.4 is a screenshot of an invocation and response of the API analyzing the validity of the *Accuweather* pricing.

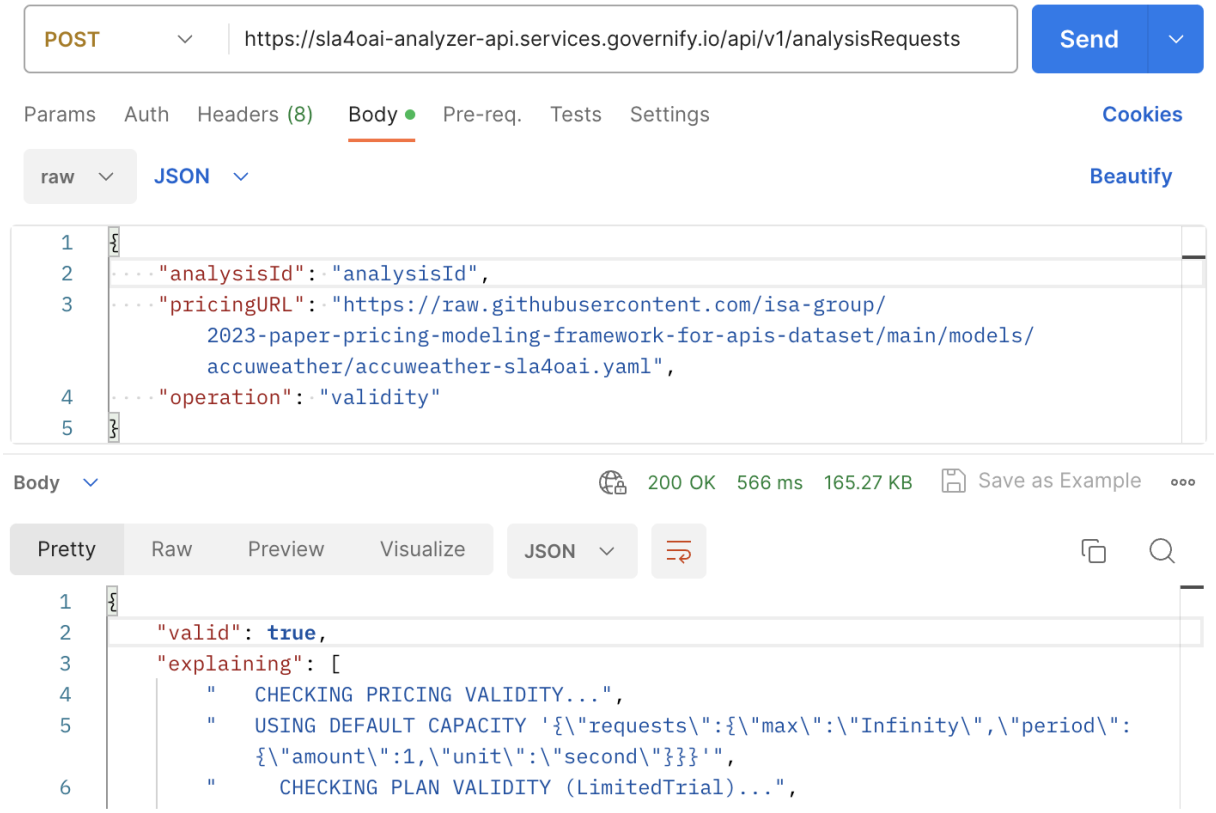

Figure 9.4: Simple UI for the *sla4oai-analyzer* API

[3] https://documenter.getpostman.com/view/683324/TVKEYHv8

Capacity Analysis of a LAMA

This chapter presents the validation of our proposal for the automated capacity analysis of a LAMA. First, Section §10.1 gives some details about the validation process. Next, Section §10.2 introduces a public API that transforms a LAMA into its corresponding CSOP and solves various analysis operations. Then, Section §10.3 contains an example of a real-world LAMA that was used to validate a subset of analysis operations.

10.1 Introduction

After defining the CSOP-based transformational model in Section §7.3, we wanted to automate the process of performing said transformation. This would greatly increase its usefulness for businesses who do not really need the details about how the CSOP works, as the transformation of the LAMA to a CSOP would be transparent. The idea behind our proposal is that a business simply describes a LAMA in a stardardized way, including its topology and the pricings of the external APIs. Then, they use that description to ask and solve a catalogue of analysis operations.

First, we introduce our automated tool, that is offered as a public API. Using the tool, it is possible to automatically obtain answers to three basic analysis operations, from which a wide variety of other operations could be potentially defined. Using our tool on the LAMA in Fig. §1.3, we provide examples for some operations that were originally formulated in Section §4.3. This showcases the usefulness of the tool to solve various operations that any business using a LAMA could ask.

Next, we recover the case study introduced in Section §1.2, in which we presented Bluejay, an automated framework for auditing software development teams. We modeled the LAMA of Bluejay and then used our tool to ask various questions about the ability to audit a number of projects with a given budget or time constraint. This experiment further supports that our tool can be used for real-world LAMAs and that businesses can benefit from it to make informed decisions about the usage of external APIs.

10.2 Smart LAMA

In order to verify that our proposal for the capacity analysis of a LAMA can be exploited in a useful way, we have developed a tooling support that partially supports it. Specifically, we have developed a RESTful API that provides our 3 basic capacity analysis operations, and an online Jupyter notebook that shows the Python implementation of the 6 analysis questions posed in Section §4.3. This tooling support can and should be understood as a minimal but solid proof of concept.

Smart LAMA [18][1], is a public RESTful API that supports, among others, various endpoints which transform a LAMA into a CSOP using the MiniZinc modeling language [55]. In particular, for the scope of this thesis, we will focus on the three main endpoints

[1] Available at https://smart-lama.services.governify.io/api/v2

that provide solutions to the three basic analysis operations.

All endpoints start with the base URL `/api/v2/operations`. They support the POST method and require the formal description of the LAMA to be included in the request body. The response includes the result of the operation (a number) and the MiniZinc output (a string containing the final values of all variables used to solve the CSOP).

- `/maxRequests`. This operation returns the maximum number of requests that the LAMA is able to serve per unit of time without exceeding any external limitation. It supports some query parameters: `OpEx` is used to specify a maximum total budget that can be spent to subscribe to the different pricing plans; `time` can be used to specify the unit of time in which the operation is calculated (indicated as the number of seconds, e.g. a minute is represented as 60); and `K-<API>-<Plan>` is used to indicate a specific number of subscriptions to a plan (e.g. `K-E1-Basic=1` means that there is 1 subscription to plan *Basic* of API E1). Note that there will be no restriction to the total cost of the LAMA if no OpEx or specific subscriptions are indicated.

- `/minCost`. This operation returns the minimum cost to serve a certain number of requests, which is specified using the `reqL` query parameter, over a certain time window, specified through the `time` query parameter. Both parameters are required. Obtaining the minimum cost implies obtaining the optimum combination of subscriptions to the pricing plans, which is also included within the MiniZinc output and may be extracted if needed.

- `/minTime`. This operation returns the minimum time (in number of seconds) in which the LAMA can serve a certain number of requests, which is specified using the `reqL` query parameter. This endpoint also supports the `OpEx` and `K-<API>-<Plan>` parameters, which work exactly as described above.

Alternatively, it is possible to register a LAMA if its topology is not expected to change between operations. The LAMA is transformed into its corresponding CSOP and stored in memory, which reduces the time to execute the analysis operations. To register a LAMA, its LDL description (explained in Section §6.3) should be sent through a POST request to `/api/v2/lamas`. Once a LAMA has been registered, the same three analysis operations can be performed, but the endpoints start with `/api/v2/lamas/<id>` to specify the LAMA under analysis and only support the GET method.

Note that, by default, the API is set up to assume that no overage requests should

be used. To enable the use of overage requests, all operations support the `useOvg` query parameter, which should be explicitly set to `true`. Only pricing plans with overage costs may have overage requests.

A known issue of the transformation into a MiniZinc model is that it generates a considerable amount of internal variables that, in some situations, may have very high values and cause *out of bounds* errors. To minimize these errors, we decided to limit the maximum number of subscriptions to each plan to 10. This workaround has proven to be useful based on our own experience. Furthermore, it is very uncommon to obtain that many subscriptions to a single plan, as API providers usually have limitations on the number of subscriptions per client or IP address.

To validate Smart LAMA, we developed an online Jupyter notebook through the Deepnote website [84]. It contains wrappers that take a formal description of a LAMA as input, send the appropriate request to the API using the corresponding query parameters, and return the solution provided by MiniZinc, including the final values of all internal variables after solving the CSOP. The notebook includes a complete example based on the LAMA in Fig. §1.3 and shows how to use the API to solve each of the 6 different analysis questions introduced in Section §4.3. Note that the notebook has *execute* access, meaning that its cells can be executed but not edited. However, it can be duplicated and then edited.

Some examples of API calls included in the notebook [84] are the following:

- **Q1. What is the cheapest operational cost for my LAMA in order to offer 2 RPS to 20 customers?** In this operation, the total number of requests that the LAMA should serve is $2 \cdot 20 = 40$. This operation can be solved using the endpoint `/api/v2/operations/minCost?reqL=40&time=1`.
- **Q5. Assuming we have a Basic plan and a Gold plan already contracted what is the maximal RPS I can guarantee as operating condition?** Using the endpoint `/api/v2/operations/maxRequests?K-E1-Basic=1&K-E1-Premium=0&K-E2-Silver=0&K-E2-Gold=1&time=1` it is possible to obtain the solution to this operation. Note that we are assuming that we do not want any additional subscriptions besides one *Basic* and one *Gold*. Therefore, the number of subscriptions to *Premium* and *Silver* must be set to 0. Otherwise, there would be no limitation to the number of subscriptions to these two plans.
- **Q6. Assuming we have a monthly budget limit of $120 in my LAMA, which is the maximum RPS I can guarantee as operating condition?**. The endpoint

`/api/v2/operations/maxRequests?OpEx=120&time=1` provides a solution to this operation.

10.3 Real-World Example

To further evaluate and validate our proposal, we used it to model and analyze the LAMA of Bluejay, already presented in Section §1.2. Following the same notation used in Fig. §1.3, we created a diagram of the architecture of Bluejay, which is shown in Fig. §10.1. Note that the number of requests that are sent between nodes varies depending on several variables, such as the number of analyzed metrics and the number of members working on each project. For this evaluation, we assume that we have 1 period, 1 scope and 2 pages (the meaning of these parameters is not relevant for this thesis); for the specific case of S1 we have 11 metrics, 5 guarantees, 12 projects and 2 members per project; for S2 we have 10 metrics, 4 guarantees, 23 projects and 6 members per project. The differentiation between the two subjects will be useful to determine the capacity of the LAMA in each separate case.

After the topology of the architecture is described, we can then model the LAMA using LDL. Because all external APIs have a free plan, it could be possible to get an infinite amount of keys to overcome all limitations, but, as explained in the previous section, we added additional constraints to limit the number of keys per API to a maximum of 10. Note that the number of requests served by the LAMA is equivalent in this case to the number of projects that can be analyzed, as the analysis of 1 project begins with 1 request to the *Reporter*. The results of some analysis operations are the following:

What is the maximum temporal resolution I can use for S1 if I am only using free plans? For this operation, we need to know how many teams can be analyzedd in 1 day using maxRequests, which is the longest time period of all external limitations. Then, this number is divided by the number of teams in S1, and 1 day is divided by this resulting number:

$$1\,\text{day} / (\text{maxRequests}(\text{L}, \$0, 1\,\text{day}) / 12) = 4690.76\text{s} = 82.68\,\text{min}$$

How many teams can be analyzed with a resolution of 5 minutes using free plans? This operation is solved by obtaining the maximum number of teams that can be analyzedd in 1 day using maxRequests and dividing the resulting number by the number of 5 minute

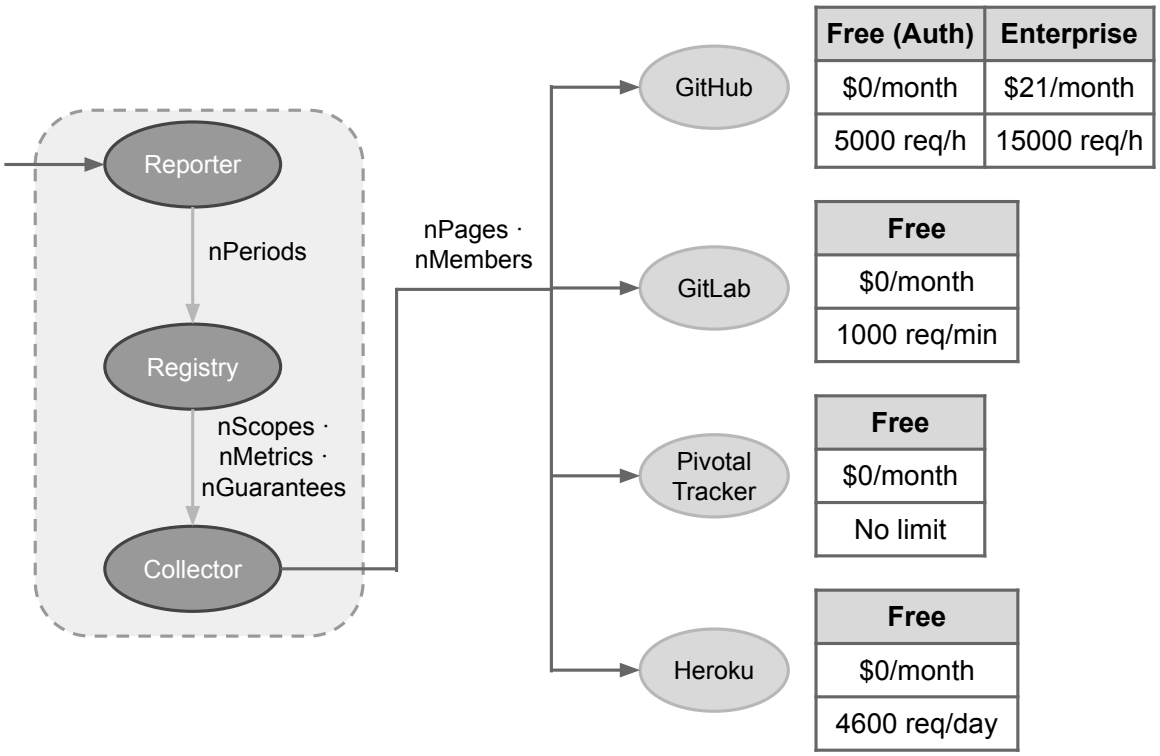

Figure 10.1: The LAMA of Bluejay, including its internal services and external APIs. Each API is depicted along its plans and limitations

periods in 1 day (288):

$$\text{maxRequests}(\text{L}, \$0, 1\text{ day})/288 = 0.73\text{ projects}$$

What is the cheapest combination of plans to analyzed the projects in S2 with a resolution of 30 minutes? This operation is solved by using minCost for the number of projects in S1 (23) multiplied by the number of 30 minute periods in 1 day (48). Once the minimum cost is obtained, we can induce how many keys are needed for each API. This information can be directly included in the output of MiniZinc together with the total cost:

$$\text{minCost}(\text{L}, 23 \cdot 48, 1\text{ day}) = \text{Unsatisfiable}$$

There is no combination of plans (keeping in mind the limitation of 10 keys per API) that is able to analyze all 23 projects each 30 minutes.

11

Monitoring Model

This chapter showcases the applicability and utility of the proposed monitoring framework. Section §11.1 provides an overview of the validation process. Section §11.2 explains the design of the validation experiment, including the description of a synthetic LAMA. Next, Section §11.3 introduces the experiment itself and the results of the process to automatically infer the topology of the LAMA, and also performs various analysis operations using Smart LAMA.

11.1 Introduction

After stating the importance of the impedance mismatch problem and introducing our first implementation of a monitoring framework in Section §8.3, we now validate said framework with a sample LAMA. As our current implementation of the framework requires all services to be implemented using Node.js and also to be instrumented with the agent, we decided to use a synthetic LAMA for this experiment, based on Fig. §1.3. Nonetheless, any real-world LAMA using Node.js can be monitored using our framework.

The first step towards performing this experiment is creating the synthetic LAMA and simulating the requests between the internal services and the external APIs. To make this task easier, we created a simple, generic API whose only purpose is sending requests to other APIs. Then, the LAMA is deployed using Docker and the traces are stored through a locally deployed instance of the collector.

Next, we take the collected traces and clean their data to facilitate their analysis. Using these traces, we can automatically draw various diagrams to visually show the chain of requests and, more importantly, the topology of the LAMA. With this inferred topology, we are able to automatically obtain the LDL description of the LAMA and use it with the tool presented in the previous chapter. This demonstrates that the monitoring framework is useful for the capacity analysis of a LAMA and can be easily used together with other tools.

11.2 Experiment Design

For the validation of the monitoring framework, we created the synthetic LAMA shown in Fig. §11.1, that is a simplified version of the one in Figure §1.3. The services are configured and deployed using a docker-compose file.

The following fragment of code shows the configuration of one of the services, in particular, the Microservice 2. The Docker image named *isagroup/api-requester* is a custom-made RESTful API whose only purpose is sending requests to other services. The remaining lines show how to set up the name of the service within the LAMA, as well as the agent to collect traces and metrics.

```
1 microservice-two:
2     container-name: microservice-two
3     image: isagroup/api-requester
4     environment:
5         RESTSENSE_SERVICE_NAME: microservice-two
```

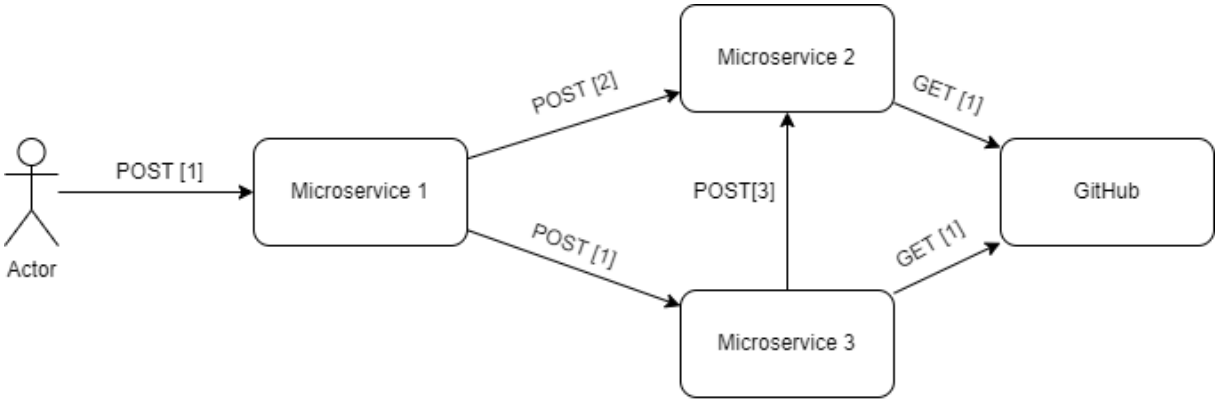

Figure 11.1: Simplified LAMA for the monitoring experiment

```
        RESTSENSE_COLLECTOR_URL: host.docker.internal:4317
    command:
        - /bin/sh
        - -c
        - |
          npm i @restsense/agent
          node -r @restsense/agent/tracing index.js
```

Listing 11.1: Configuration for the deployment of Microservice 2 using docker-compose

Microservices 2 and 3 need to send requests to the GitHub API. Note that, for the design of this experiment, the limitations are irrelevant as our only goal is monitoring the LAMA to obtain its topology. As long as Microservices 2 and 3 consume the GitHub APIs, it is enough. External APIs do not neet do be instrumented.

To specify the number of requests between each service, the *api-requester* service has a RESTful API with a single endpoint. This endpoint receives information about how many requests will be sent and where. To start the chain of requests, we must send one request to the entrypoint (in our case, Microservice 1), describing the entire chain of requests between the other services and the external APIs. Then, Microservice 1 will send further requests to Microservices 2 and S with information about their corresponding requests. Finally, Microservices 2 and 3 will send their requests to the GitHub API. Fragment 11.2 shows the structure of a sample request for a service using *api-requester*.

```
[{
    "url": "http://microservice-two/api/v1/request",
    "method": "POST",
    "body": {
        "url": "ANOTHER_URL",
        "body": { "..." },
        "headers": { "..." }
    }
```

```
9 }]
```

Listing 11.2: Request example for a service using *api-requester*

This way, any arbitrary LAMA can be easily described and simulated. Note that *api-requester* was designed for the only purpose of simulating requests between services and, as such, it is necessary to know the number of requests beforehand in order to define the chain of requests. This means that we already know the topology of the LAMA, so we do not really need to infer it. Nonetheless, this is only for experimentation and in the case of a real-world LAMA we would not now its topology.

11.3 Experiment Execution

Having already defined the LAMA using docker-compose and *api-requester*, we can now obtain information about its topology using traces and metrics. As seen in the previous section, the services were already instrumented to include the monitoring agent as part of the docker-compose file. The collector was also running in the background (it can be included in the docker-compose file or deployed on its own) and, in order to extract the collected data from the MongoDB database, we can use MongoDB Compass[1]. Alternatively, the collector also provides some endpoints to extract the traces and metrics without manually querying the database.

The collected data, extracted in CSV format, was then uploaded to an online Jupyter notebook for further processing. This notebook is available at [85], and it is shared with *execute* access. The notebook also contains examples of analysis operations using Smart LAMA. The analysis of the data in the notebook follows three steps: data extraction, exploratory analysis and capacity analysis.

Data Extraction

The data extraction process comprises two steps: data loading and preprocessing. The CSV files are loaded into the notebook using the pandas library[2], and are stored in *dataframes*. Then, during the preprocessing step, the dataframes are cleaned through three actions:

1. Information that will not be used in the notebook is removed. This information may be useful for other types of analysis, though.

[1] https://www.mongodb.com/products/compass
[2] https://pandas.pydata.org/

2. Some columns are renamed for better readability, as they originally followed the OpenTelemetry naming conventions.

3. Columns that refer to the same concepts are merged, such as columns including information about response status codes and response text (e.g. 200 and OK).

Exploratory Analysis

This analysis includes the automated extraction of the topology of the LAMA. For this part, the notebook includes various functions and methods to draw diagrams to visualize the data. For example, there is a function to draw a cascade representation of a trace, which includes all of its spans and some information about them. Hovering the mouse over the spans will show this information. However, the most important diagram in this analysis is the one that shows the topology of the LAMA. The notebook includes an example that shows how to draw a DAG from the information of a trace. This DAG looks somewhat similar to the diagram in Fig. §11.1, and it does not include information about pricings, because they are not part of the topology itself. Fig. §11.2 shows the resulting diagram in the notebook.

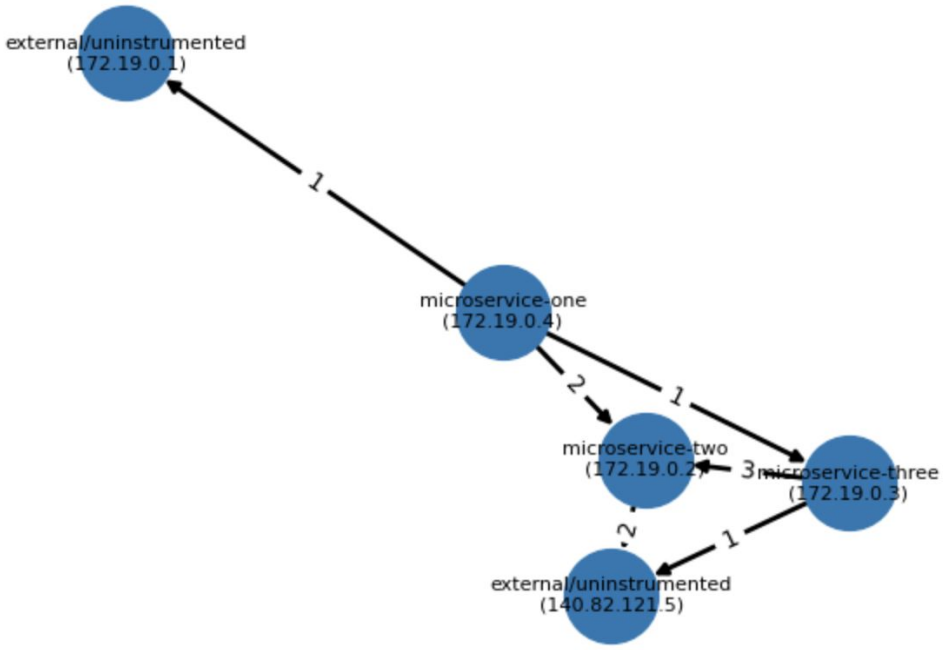

Figure 11.2: Inferred topology of the LAMA under analysis

Capacity Analysis

Once we know the topology of the LAMA and the pricings of the external APIs, we can perform various capacity analysis operations. Although we did not need the GitHub pricing for the topology, we now need it for the capacity analysis of the LAMA. To solve these analysis operations, we used Smart LAMA, already presented in Section §10.2. However, Smart LAMA requires the LAMA to be described using LDL. For this reason, we included a function in the notebook that transforms the inferred topology to LDL, and we manually included the GitHub API pricing.

We can automatically detect whether any service in the LAMA is an internal service or an external API by analyzing its IP address. Furthermore, we know which service is the entrypoint because its span happens first and has no parents. Using this information, along with the number of each edge, we can automatically generate the LDL of the LAMA. Listing 11.3 shows the resulting LDL of our example. Notice that the services and APIs are identified through their IP address and not actual names.

```
Entry: 172_19_0_1
External services: ['140_82_121_5']
Internal services: ['172_19_0_4','172_19_0_1','172_19_0_2','172_19_0_3']
Relationships: [{'from':'172_19_0_4', 'to':'172_19_0_1', 'value':1}, ...]
```

Listing 11.3: Automatically generated LDL of our example

With the topology of the LAMA described using LDL and the external pricing manually added, we can now send requests to Smart LAMA to perform analysis operations. For example, some operations include:

1. What is the maximum number of requests that can be sent in 1 minute?
   Answer: 4,545 requests.

2. What is the maximum monthly quota we can offer to our customers without exceeding the limitations of the external APIs?
   Answer: 136,363 requests per month.

3. What is the minimum cost to send a certain number of requests?
   Answer: $0.00, because the GitHub API has no cost.

4. What is the minimum time to send 5 requests?
   Answer: 1 second, because the GitHub API does not specify any rate.

5. What is the minimum time to send 300 requests with a maximum budget of $20?
   Answer: 1 second, because the GitHub API has no cost and the quota is not exceeded.

While the analysis with the GitHub API may be somewhat limited (because it has no cost and no rates), it is enough to showcase that our monitoring framework is able to automatically infer the topology of the LAMA and then perform capacity analysis operations. Thanks to this automation, we can get the topology at any point during execution time, so we can be aware of unexpected changes in the topology that may produce service disruptions or unwanted charges.

Part V

CONCLUSIONS

Conclusions and Future Work

This chapter concludes the thesis by providing some final remarks. First, Section §12.1 presents a list of conclusions for all contributions and proposed tools. Finally, Section §12.2 contains information about potential lines of work that could be followed in the future to further improve and expand the research of this thesis.

12.1 Conclusions

In this thesis we presented a set of models and tools that guide the capacity analysis of LAMAs; this approach helps in the decision-making process to optimize the operational expenditures (OpEx) of a LAMA and also detects different types of conflicts that could be present in the LAMA offering.

Specifically, we first presented an extension of an existing model for RESTful API pricings, as well as a serialization for said model. We validated the model by analyzing a representative set of real-world APIs, and we modeled 54 of them using our serialization. We devised a first iteration of a catalogue of 11 consumer-oriented analysis operations for API pricings plus an additional validity operation, along with a classification based on different scenarios. This catalogue will be extended with new operations in the future, and we aim to make it as exhaustive as possible. The creation of a catalogue paves the way for automated tools that provide answers to these operations. This way, consumers who use external APIs can make decisions based on their specific scenarios and needs.

Moreover, we presented the problem of automating the capacity analysis of microservices architectures in situations where the MSA consumes external APIs that define pricing plans. We introduced the concept of *limitation-aware microservice architecture* (LAMA) and explored the different dimensions involved in the capacity analysis of a LAMA. We identified three basic analysis operations that allow the definition of any number of other operations. We presented a public API that transforms a LAMA into a proof-of-concept implementation of these operations using MiniZinc, and validated it in a synthetic LAMA example and a real-world application. We are confident that our proposal will prove useful to DevOps teams who need to deal with issues related to capacity analysis of LAMAs.

Regarding the automated extraction of the topology of a LAMA, we presented a first version of a monitoring framework. Telemetry traces can be leveraged to dynamically infer the topology of a LAMA and the requests made to each service. Furthermore, metrics can be used to calculate optimal capacity limitations for a LAMA. We validated the framework using a synthetic LAMA. We also established the requirements that should be met by any new specification for telemetry that relates traces and metrics towards the capacity analysis process. The new specification can enable not only the prediction of LAMA capacity but also the dynamic adjustment of capacity limits for a LAMA to prevent service costs overrun, representing a critical step towards improving the reliability and efficiency of MSAs and ultimately benefiting end-users and organizations alike.

12.2 Future Work

The proposed model and serialization for RESTful API pricings have some limitations that were already mentioned in their corresponding sections. Additionally, the current implementation of SLA4OAI-Analyzer has some limitations. It follows a strict syntax-guided approach, so small changes in the specification derive in important updates to the API. In the future, we aim to transform a pricing into a constraint satisfaction problem (CSP), which should allow for better maintainability and scalability of the API. Furthermore, we did not create any tool to support the remaining 11 analysis operations included in our proposed catalogue. We plan to develop a tool that takes an SLA4OAI specification file as input and automatically answers the operations in the catalogue. Our approach would follow the one presented in Section §7.3 for LAMAs, where we transformed a LAMA into a CSOP. In this case, we would transform the elements of a pricing into a CSOP, and create the corresponding constraints for each analysis operation. The declarative nature of a CSOP makes it easier to further extend the catalogue by simply adding new constraints and variables as needed. The next step would be creating an interface so that the consumer could easily run the various operations. Following the recent popularity of artificial intelligence, we could program a chatbot that takes a query in natural language and translates it into a specific operation. It would then run the operation and return the answer in a easily readable format for the consumer.

For the automated capacity analysis of LAMAs, we would like to improve and extend our proposal in order to support more complex operations. We want to consider the addition of limitations in internal services, which usually have restrictions from their deployment infrastructures. Additionally, we need to support multiple entrypoints, as it is uncommon for LAMAs to only have a single operation.

We are aware that our tooling support for both pricings and LAMAs is partial and therefore incomplete, but it shows the real possibility of answering questions in less time than if it were done manually. In this sense, we could work on using notation to describe both the topology and the pricing closer to some of the available technology. Moreover, the current implementation has some limitations related to how MiniZinc handles variables and their domains. In the future, we want to change the CSOP solver being used by MiniZinc to overcome these issues, or even move to another CSOP modeling language. Either of these changes would be transparent to API users.

About the monitoring framework, while we believe that it is a very useful tool, it is missing some key elements to provide full support for the capacity analysis of LAMAs.

First, it cannot obtain the pricings of the external APIs, which remains a difficult task. Additionally, the framework assumes that all internal services are in the same IP range and also share the same programming language (Node.js), which greatly limits its ability to be used in real-world LAMAs. The first issue could be solved by automatically obtaining the pricings from a centralized source, although they would need to be described using a standardized format (for example, SLA4OAI). The second problem could be solved by creating new agents for different languages, and introducing some configuration parameters that allow a LAMA developer to specify which services (based on IP address or other parameters) are considered internal services.

Bibliography

[1] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice architecture: aligning principles, practices, and culture.* " O'Reilly Media, Inc.", 2016. (page 4).

[2] Eberhard Wolff. *Microservices: flexible software architecture.* Addison-Wesley Professional, 2016. (page 4).

[3] Sam Newman. *Building microservices - designing fine-grained systems, 1st Edition.* O'Reilly, 2015. ISBN 9781491950357. URL https://www.worldcat.org/oclc/904463848. (page 4).

[4] Martin Fowler - Microservices, . URL https://martinfowler.com/articles/microservices.html. (pages 4, 23).

[5] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)*, pages 583–590. IEEE, 2015. (page 4).

[6] Mark Richards. *Microservices vs. service-oriented architecture.* O'Reilly Media Sebastopol, 2015. (page 4).

[7] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *Ieee Software*, 33(3): 42–52, 2016. (page 4).

[8] Microservices: An application revolution powered by the cloud. URL https://azure.microsoft.com/en-us/blog/microservices-an-application-revolution-powered-by-the-cloud/. (page 5).

[9] Daniel Jacobson, Greg Brail, and Dan Woods. *APIs: A strategy guide.* O'Reilly Media, Inc., 2012. (page 6).

[10] SendGrid API Pricing. URL https://sendgrid.com/en-us/pricing. (page 6).

[11] AXELOS. *ITIL Foundation: ITIL 4 Edition*. PeopleCert, 2022. ISBN 978-9-92-560000-7. (page 8).

[12] Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. A classification of object-relational impedance mismatch. In *2009 First International Confernce on Advances in Databases, Knowledge, and Data Applications*, pages 36–43. IEEE, 2009. (page 8).

[13] A. Caronti, G. Caliano, A. Iula, and M. Pappalardo. Electrical impedance mismatch in capacitive micromachined ultrasonic transducers. In *2000 IEEE Ultrasonics Symposium. Proceedings. An International Symposium (Cat. No.00CH37121)*, volume 1, pages 925–930 vol.1, 2000. (page 8).

[14] César García, Alejandro Guerrero, Joshua Zeitsoff, Srujay Korlakunta, Pablo Fernandez, Armando Fox, and Antonio Ruiz-Cortés. Bluejay: A cross-tooling audit framework for agile software teams. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 283–288, 2021. (page 9).

[15] Rafael Fresno-Aranda, Pablo Fernandez, Antonio Gamez-Diaz, Amador Duran, and Antonio Ruiz-Cortes. Pricing4APIs: A Rigorous Model for RESTful API Pricings. *Computer Standards & Interfaces (Under review)*, 2023. URL https://doi.org/10.48550/arXiv.2311.12485. (pages 13, 15).

[16] Rafael Fresno-Aranda, Pablo Fernández, and Antonio Ruiz-Cortés. SLA4OAI-Analyzer: Automated Validation of RESTful API Pricing Plans. In *International Conference on Web Engineering*, pages 363–366. Springer, 2023. (pages 13, 15, 80).

[17] R. Fresno-Aranda, P. Fernández, and A. Ruiz-Cortés. Towards the automation of design time capacity analysis over microservices architectures. In *Proc. XVI Jornadas de Ingeniería de Ciencia e Ingeniería de Servicios (JCIS 2021)*. SISTEDES, 2021. URL https://hdl.handle.net/11705/JCIS/2021/041. (pages 14, 15).

[18] Rafael Fresno-Aranda, Pablo Fernández, and Antonio Ruiz-Cortés. Smart LAMA API: Automated Capacity Analysis of Limitation-Aware Microservices Architectures. In *Proc. XVII Jornadas de Ingeniería de Ciencia e Ingeniería de Servicios (JCIS 2022)*. SISTEDES, 2022. URL http://hdl.handle.net/11705/JCIS/2022/032. (pages 14, 15, 122).

[19] Rafael Fresno-Aranda, Pablo Fernández, Amador Durán, and Antonio Ruiz-Cortés. Semi-automated capacity analysis of limitation-aware microservices architectures. In *International Conference on the Economics of Grids, Clouds, Systems, and Services*, pages 75–88. Springer, 2022. (pages 14, 15, 66).

[20] Rafael Fresno-Aranda. Automated capacity analysis of limitation-aware microservices architectures. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1780–1784, 2022. (page 14).

[21] R. Fresno-Aranda, P. Fernández, and A. Ruiz-Cortés. A Catalogue of Analysis Operations for API Pricing Plans. In *Proc. XVIII Jornadas de Ingeniería de Ciencia e Ingeniería de Servicios (JCIS 2023)*. SISTEDES, 2023. URL https://hdl.handle.net/11705/JCIS/2023/2322. (pages 14, 15, 76, 77).

[22] Rafael Fresno-Aranda, Juan Sebastian Ojeda-Perez, Pablo Fernandez, and Antonio Ruiz-Cortes. Governify. An agreement-based service governance framework. *Software Impact (In press)*, 2024. (page 14).

[23] Alejandro Guerrero, Rafael Fresno, An Ju, Armando Fox, Pablo Fernandez, Carlos Muller, and Antonio Ruiz-Cortés. Eagle: A team practices audit framework for agile software development. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1139–1143, 2019. (page 14).

[24] A. Santisteban, P. Fernandez, J. M. García, R. Fresno Aranda, and A. Ruiz Cortés. Towards a Telemetry Specification for Capacity Analysis in Limitation-Aware Microservices Architectures. In *Proc. XVIII Jornadas de Ingeniería de Ciencia e Ingeniería de Servicios (JCIS 2023)*. SISTEDES, 2023. URL https://hdl.handle.net/11705/JCIS/2023/2641. (pages 14, 15, 106).

[25] Rafael Fresno. Easy security management over microservices architectures based on OpenAPI Specification. In *Proc. XV Jornadas de Ingeniería de Ciencia e Ingeniería de Servicios* . SISTEDES, 2019. URL http://hdl.handle.net/11705/JCIS/2019/020. (page 15).

[26] R. H. Von Alan, S. T March, J. Park, and S. Ram. Design science in information systems research. *MIS quarterly*, 28(1):75–105, 2004. (page 16).

[27] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007. (page 16).

[28] Microservice Architecture pattern, . URL https://microservices.io/patterns/microservices.html. (page 22).

[29] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000. (page 27).

[30] Current Members. URL https://www.openapis.org/membership/members. (page 29).

[31] Basic Structure. URL https://swagger.io/docs/specification/basic-structure/. (page 30).

[32] AWS EC2 Architecture. URL https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html. (page 33).

[33] Containerized vs Other Deployments. URL https://kubernetes.io/docs/concepts/overview/. (page 34).

[34] Sina Niedermaier, Falko Koetter, Andreas Freymann, and Stefan Wagner. On observability and monitoring of distributed systems – an industry interview study. In Sami Yangui, Ismael Bouassida Rodriguez, Khalil Drira, and Zahir Tari, editors, *Service-Oriented Computing*, pages 36–52, Cham, 2019. Springer International Publishing. ISBN 978-3-030-33702-5. (page 35).

[35] Cheng-Fu Huang, Ding-Hsiang Huang, and Yi-Kuei Lin. System reliability analysis for a cloud-based network under edge server capacity and budget constraints. *Annals of Operations Research*, 312:217 – 234, 2020. (page 35).

[36] OpenTelemetry. URL https://opentelemetry.io/docs/reference/specification/. (page 35).

[37] OpenTelemetry Diagram. URL https://opentelemetry.io/docs/. (page 35).

[38] Prometheus. URL https://prometheus.io/docs/concepts/data_model/. (page 36).

[39] Prometheus data in a Grafana dashboard. URL https://prometheus.io/docs/visualization/grafana/. (page 36).

[40] Mia E. Gortney, Patrick E. Harris, Tomas Cerny, Abdullah Al Maruf, Miroslav Bures, Davide Taibi, and Pavel Tisnovsky. Visualizing microservice architecture in the dynamic perspective: A systematic mapping study. *IEEE Access*, 10:119999–120012, 2022. (page 38).

[41] Jhonny Mertz and Ingrid Nunes. Tigris: a DSL and framework for monitoring software systems at runtime. *CoRR*, abs/2103.15986, 2021. URL https://arxiv.org/abs/2103.15986. (page 38).

[42] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, page 247–248, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312028. (page 38).

[43] Jhonny Mertz and Ingrid Nunes. On the practical feasibility of software monitoring: a framework for low-impact execution tracing. In *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 169–180, 2019. (page 38).

[44] Alessandro Tundo, Marco Mobilio, Matteo Orrù, Oliviero Riganelli, Michell Guzmàn, and Leonardo Mariani. Varys: An agnostic model-driven monitoring-as-a-service framework for the cloud. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 1085–1089, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450355728. (page 38).

[45] How to Make the API Economy a Reality - WSO2. URL https://wso2.com/choreo/resources/how-to-make-the-api-economy-a-reality/. (page 40).

[46] How to build an API economy for your enterprise - Capgemini. URL https://www.capgemini.com/2020/12/how-to-build-an-api-economy-for-your-enterprise/. (page 40).

[47] Overview of the API Economy. URL https://rapidapi.com/blog/api-glossary/what-is-the-api-economy-api-economy-definition-explained/. (page 40).

[48] 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 27th ACM Joint*

Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pages 1006–1014, 2019. (pages 41, 72, 75).

[49] Antonio Gámez Díaz. *SLA-Driven Governance of RESTful Systems*. PhD thesis, University of Seville, 2021. URL https://idus.us.es/handle/11441/130656. (pages 44, 66).

[50] RapidAPI Main Page. URL https://rapidapi.com/hub. (page 46).

[51] Liangzhao Zeng, Boualem Benatallah, Anne HH Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. QoS-aware middleware for web services composition. *IEEE Transactions on software engineering*, 30(5):311–327, 2004. (page 61).

[52] José Antonio Parejo, Sergio Segura, Pablo Fernandez, and Antonio Ruiz-Cortés. QoS-Aware web services composition using GRASP with Path Relinking. *Expert Systems with Applications*, 41(9):4211–4223, 2014. (page 61).

[53] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. A lightweight approach for QoS-aware service composition. In *Proc. 2nd International Conference on Service Oriented Computing (ICSOC'04)-short papers*. Citeseer, 2004. (page 61).

[54] Antonio Gamez-Diaz, Pablo Fernandez, Cesare Pautasso, Ana Ivanchikj, and Antonio Ruiz-Cortes. ELeCTRA: Induced Usage Limitations Calculation in RESTful APIs. In *16th International Conference on Service-Oriented Computing (ICSOC)*, pages 435–438. Springer, 2018. (pages 62, 100).

[55] MiniZinc constraint modeling language. URL https://www.minizinc.org/. (pages 62, 122).

[56] Antonio Gamez-Diaz, Pablo Fernandez, and Antonio Ruiz-Cortes. An analysis of RESTful APIs offerings in the industry. In *15th International Conference on Service-Oriented Computing (ICSOC)*, pages 589–604. Springer, 2017. (pages 66, 69, 75, 94, 112, 113, 118).

[57] SLA4OAI Research Specification, . URL https://github.com/isa-group/SLA4OAI-ResearchSpecification. (page 66).

[58] SLA4OAI Technical Committee, . URL https://github.com/isa-group/SLA4OAI-TC. (page 66).

[59] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. 9781491929124, 2016. ISBN O'Reilly Media, Inc. (page 69).

[60] Antonio Gamez-Diaz, Pablo Fernandez, and Antonio Ruiz-Cortes. Automating sla-driven api development with sla4oai. In *International Conference on Service-Oriented Computing*, pages 20–35. Springer, 2019. (pages 72, 73).

[61] Rafael Fresno-Aranda, Pablo Fernandez, Antonio Gamez-Diaz, Amador Duran, and Antonio Ruiz-Cortes. Sla4oai json schema, . URL https://doi.org/10.5281/zenodo.5118599. (page 73).

[62] Elena Molino Peña, José María Garcia, and Antonio Ruiz Cortés. Operaciones de Análisis sobre los Términos de Uso en Customer Agreements. In *Proc. XVII Jornadas de Ingeniería de Ciencia e Ingeniería de Servicios*. SISTEDES, 2022. URL http://hdl.handle.net/11705/JCIS/2022/041. (page 75).

[63] José María García, Octavio Martín-Díaz, Pablo Fernandez, Carlos Müller, and Antonio Ruiz-Cortés. A flexible billing life cycle for cloud services using augmented customer agreements. *IEEE Access*, 9:44374–44389, 2021. (page 75).

[64] ITIL Capacity Management. URL https://www.smartsheet.com/content/itil-capacity-management. (page 94).

[65] Sebastien Andreo, Ambra Calà, and Jan Bosch. OpEx Driven Software Architecture A Case Study. In *Proceedings of the 15th European Conference on Software Architecture (ECSA) (Companion)*, 2021. (page 95).

[66] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information systems*, 35(6):615–636, 2010. (pages 96, 100).

[67] Amador Durán, David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. Flame: a formal framework for the automated analysis of software product lines validated by automated specification testing. *Software & Systems Modeling*, 16 (4):1049–1082, 2017. (page 96).

[68] Octavio Martín-Díaz, Antonio Ruiz-Cortés, Amador Durán, and Carlos Müller. An approach to temporal-aware procurement of web services. In *3rd International Conference on Service-Oriented Computing (ICSOC)*, pages 170–184. Springer, 2005. (pages 96, 100).

[69] Carlos Müller, Manuel Resinas, and Antonio Ruiz-Cortés. Automated analysis of conflicts in WS-Agreement. *IEEE Transactions on Services Computing*, 7(4):530–544, 2013. (pages 96, 100).

[70] Carlos Müller, Antonio M Gutierrez, Pablo Fernandez, Octavio Martín-Díaz, Manuel Resinas, and Antonio Ruiz-Cortés. Automated validation of compensable SLAs. *IEEE Transactions on Services Computing*, 14(5):1306–1319, 2018. (pages 96, 100).

[71] Adela del Río-Ortega, Manuel Resinas, Cristina Cabanillas, and Antonio Ruiz-Cortés. On the definition and design-time analysis of process performance indicators. *Information Systems*, 38(4):470–490, 2013. (page 96).

[72] Mark Harman, Kiran Lakhotia, Jeremy Singer, David R White, and Shin Yoo. Cloud engineering is search based software engineering too. *Journal of Systems and Software*, 86(9):2225–2241, 2013. (page 96).

[73] Arthur HM ter Hofstede and Henderik Alex Proper. How to formalize it?: Formalization principles for information system development methods. *Information and Software Technology*, 40(10):519–540, 1998. (page 96).

[74] Alejandro Santisteban. Telemetry-based capacity analysis for limitation-aware microservices architectures, 2022. (page 105).

[75] Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino. An Analysis of Public REST Web Service APIs. *IEEE TSC*, 2018. ISSN 19391374. doi: 10.1109/TSC.2018.2847344. (pages 112, 113).

[76] Alexa. The top 500 sites on the web. https://www.alexa.com/topsites. (page 112).

[77] L. Isserlis. On the value of a mean as calculated from a sample. *Journal of the Royal Statistical Society*, 81(1):75–81, 1918. ISSN 09528385. (page 113).

[78] Rafael Fresno-Aranda, Pablo Fernandez, Antonio Gamez-Diaz, Amador Duran, and Antonio Ruiz-Cortes. isa-group/2023-paper-pricing-modeling-framework- for-apis-dataset, . URL https://doi.org/10.5281/zenodo.4697208. (pages 113, 116).

[79] Sebastian Baltes and Paul Ralph. Sampling in software engineering research: A critical review and guidelines. *Empirical Software Engineering*, 27(4):94, 2022. (page 116).

[80] Rafael Fresno-Aranda, Pablo Fernandez, Antonio Gamez-Diaz, Amador Duran, and Antonio Ruiz-Cortes. isa-group/sla4oai-analyzer, . URL https://doi.org/10.5281/zenodo.5146288. (page 118).

[81] SLA4OAI-Analyzer API, . URL https://sla4oai-analyzer-api.services.governify.io. (page 119).

[82] SLA4OAI-Analyzer API Repository. URL https://github.com/isa-group/sla4oai-analyzer-api. (page 119).

[83] SLA4OAI-Analyzer Validation, . URL https://bit.ly/phd-sla4oai-validation. (page 119).

[84] Smart LAMA Validation. URL https://bit.ly/phd-lama-validation. (page 124).

[85] Monitoring Framework Validation. URL https://bit.ly/phd-monitoring-validation. (page 130).