SLA-Driven Governance of RESTful Systems

Antonio Gámez Díaz

Doctoral Thesis

Advised by
Dr. Antonio Ruiz Cortés
and
Dr. Pablo Fernández Montes

Universidad de Sevilla

September 2021

Support: Doctoral Thesis supported by the FPU scholarship program, granted by the Spanish Ministry of Education, Culture and Sports (FPU15/02980).

Dedicado a mi madre, Rosi.
Sin todo su esfuerzo no hubiera llegado nunca hasta aquí.
Gracias por todo.

Abstract

The Software as a Service (SaaS) paradigm has become entrenched in the industry as a deployment model, bringing flexibility to the customers and a recurring revenue to the business. The main architectural paradigm of SaaS systems is the service-oriented one since it provides numerous advantages in terms of elasticity, fault tolerance, and flexible architectural design. Currently, the RESTful paradigm, a layer of abstraction on the server created by defining resources and entities that can be accessed by means of a URI, is the preferred choice for the construction of SaaS, as it promotes the deployment, isolation and integration of microservices through APIs.

Nowadays, APIs are regarded as a new form of business product and ever more organizations are publicly opening up access to their APIs as a way to create new business opportunities. In the same way, other organizations also consume a number of third-party APIs as part of their business.

We henceforth define the concept of a *RESTful System* as an information system following the RESTful paradigm to shape the integration model between both its own components as well as other information systems.

Furthermore, understanding *governance* as the way in which a component is directed and controlled, in RESTful Systems, those components will be the RESTful APIs and what we aim to control or regulate is their behavior (i.e., how an API is being consumed or provided).

As APIs are increasingly regarded as business products, a crucial activity is to describe the set of *plans* (i.e., the *pricing*) that depicts the functionality and performance being offered to clients. API providers usually define certain *limitations* in each instance of a plan (e.g., *quotas* and *rates*); for example, a free plan might be limited to having one hundred monthly requests, and a professional plan to have five hundred monthly requests.

However, although API providers use the *Service Level Agreement* (SLA) concept to delimit the functionality and guarantees to which they commit to their customers, there is no standard model used by API providers for modeling API *pricing* (including the *plans*

and *limitations*). Although some providers do model the information regarding the API *pricing* and API *limitations* with an ad hoc approach, there is no widely accepted model in the industry. Wherefore answering questions regarding API limitations (e.g., determining whether or not a certain *pricing* is valid) is still a manual or non-interoperable process coming along with some inconveniences (being tedious, time-consuming, error-prone, etc.).

Understating *governance* as to *how a system is directed and controlled*, we translate this concept to meet the SLA-driven approach: we consider the SLA (i.e., API *pricing*) as the element that will drive the directions, policies and rules to deliver and maintain the RESTful System. Adding the *SLA* to the idea of *governance* of *RESTful systems* leads to the main hypothesis of this dissertation: there is no well-established model for describing API *pricings*)in RESTful systems, which is hindering the automatic *SLA-Driven governance*.

We claim the main goal of this thesis to be: the *creation of an expressive, fully-fledged specification of SLAs for RESTful APIs endorsed with an open ecosystem of tools aimed at the SLA-Driven Governance of RESTful systems.* The results of this endeavor are twofold:

(I) *Creation of a sufficiently expressive specification for the description of API pricings and the analysis of their validity.* This comprises: (i) conducting an analysis of real-world APIs to evaluate the characteristics of the API *pricings* and limitations; (ii) identifying the relevance of SLAs in APIs in both academic and industrial scenarios; (iii) proposing a comprehensive model for describing API *pricings*; (iv) defining analysis operations for common questions regarding the validity in API *pricings* and limitations; (v) performing an evaluation of the model in real-world APIs.

(II) *Implementation of an ecosystem of tools to support the SLA-Driven governance of RESTful APIs.* This includes: (i) developing a set of API *governance* tools; (ii) implementing a validity analysis operation; (iii) performing a validation of the tools and operations in realistic scenarios.

In this thesis, we present the *Governify4APIs ecosystem* as the set comprised of (i) a *model* aimed at describing API *pricings* that is closely aligned with industry standards in APIs (OpenAPI Specification) and (ii) a set of companion *tools* for enacting the automatic *governance* using our specification, ranging from low-level validation tasks to SaaS solutions based on our model. *Governify4APIs* is, therefore, a fully-fledged specification, aligned with the mainstream standards and intended to enable an *SLA-Driven Governance of RESTful Systems.*

Resumen

El paradigma del software como servicio (SaaS) se ha afianzado en la industria como modelo de despliegue, aportando flexibilidad a los clientes y unos ingresos constantes a las organizaciones. El principal paradigma arquitectónico de los sistemas SaaS es la arquitectura orientada a servicios, ya que proporciona numerosas ventajas en términos de elasticidad, tolerancia a fallos y diseño flexible. RESTful, una capa de abstracción sobre el servidor creada mediante la definición de recursos y entidades a las que se puede acceder mediante una URI, es la opción preferida para la construcción de SaaS, ya que promueve el despliegue, el aislamiento y la integración de microservicios a través de APIs. Hoy en día, las APIs se consideran una nueva forma de producto empresarial y cada vez más organizaciones abren públicamente el acceso a sus APIs como forma de crear nuevas oportunidades de negocio. Del mismo modo, otras organizaciones también consumen una serie de APIs de terceros como parte de su negocio.

A partir de ahora definimos el concepto de *Sistema RESTful* como un sistema de información que sigue el paradigma RESTful para conformar el modelo de integración tanto entre sus propios componentes como con otros sistemas de información. Además, entendiendo *gobierno* como la forma en que se dirige y controla un componente, en los sistemas RESTful, esos componentes serán las APIs RESTful y lo que pretendemos controlar o regular es su comportamiento (es decir, cómo se está consumiendo o proporcionando una API).

Dado que las APIs están, cada vez más, siendo consideradas como productos comerciales, una actividad crucial es describir el conjunto de *planes* (es decir, el *pricing*) que describe la funcionalidad y el rendimiento que se ofrece a los clientes. Los proveedores de API suelen definir ciertas *limitaciones* en cada instancia de un plan (por ejemplo, *quotas* y *rates*); por ejemplo, un plan gratuito podría estar limitado a tener cien peticiones mensuales, y un plan profesional a tener quinientas peticiones mensuales. Sin embargo, aunque los proveedores de APIs utilizan el concepto de *Acuerdo de Nivel de Servicio* (SLA) para delimitar la funcionalidad y las garantías a las que se comprometen con sus clientes, no existe ningún modelo estándar usado por los proveedores para modelar el *pricing* de las API (incluyendo los *planes* y *limitaciones*). Aunque algunos proveedores modelan la in-

formación relativa a los *pricings* y las *limitaciones* de las APIs con un enfoque ad hoc, no existe un modelo ampliamente aceptado en el sector. Por lo tanto, responder a las preguntas relativas a las *limitaciones* de la APIs (por ejemplo, determinar si un determinado *pricing* es válido o no) sigue siendo un proceso manual o no interoperable, cosa que conlleva algunos inconvenientes (es tedioso, consume tiempo, es propenso a errores, etc.).

Entendiendo el *gobierno* como *la forma de dirigir y controlar un sistema*, podemos traducir este concepto teniendo en cuenta el SLA, esto es, consideramos este elemento como aquel sobre el que se realiza la dirección, políticas y reglas para entregar y mantener el *sistema RESTful*. Añadir el concepto *SLA* a esa idea de *gobierno* de *sistemas RESTful* nos lleva a la hipótesis principal de esta tesis: *no existe un modelo bien establecido para describir los SLAs (o pricing) en los sistemas RESTful, lo que está dificultando el gobierno automático*. Es, por tanto, el objetivo principal de esta tesis *la creación de una especificación expresiva y completa de SLAs para APIs RESTful, respaldada por un ecosistema abierto de herramientas orientadas al gobierno de sistemas RESTful dirigido por SLAs*. Los resultados principales han sido:

(I) *Creación de una especificación suficientemente expresiva para la descripción de los pricings de la API y el análisis de su validez*. Esto comprende: (i) realizar un análisis de APIs del mundo real para evaluar las características de los *pricings* y *limitaciones* de las APIs; (ii) identificar la relevancia de los SLAs en las APIs tanto en escenarios académicos como industriales; (iii) proponer un modelo completo para describir los *pricings* de las APIs; (iv) definir operaciones de análisis para preguntas comunes sobre la validez en los *pricings* y *limitaciones* de las APIs; (v) realizar una evaluación del modelo en APIs del mundo real.

(II) *Implementación de un ecosistema de herramientas para apoyar la gobernanza SLA-Driven de las APIs RESTful*. Esto incluye: (i) desarrollar un conjunto de herramientas de gobierno de APIs; (ii) implementar una operación de análisis de validez; (iii) realizar una validación de las herramientas y operaciones en escenarios realistas.

En esta tesis, presentamos el ecosistema *Governify4APIs* como el conjunto compuesto por (i) un modelo destinado a describir los *pricings* de las APIs y alineado estrechamente con los estándares de la industria (OpenAPI) y (ii) un conjunto de *herramientas* complementarias para el gobierno automático utilizando este modelo, que van desde tareas de validación hasta soluciones SaaS. Por lo tanto, *Governify4APIs* es una especificación acompañada de todo lo necesario, alineada con los estándares industriales y destinada a permitir un *gobierno de sistemas RESTful dirigidos por SLAs*.

Contents

List of Figures

Part I

PRELIMINARIES

Introduction

This chapter introduces both the research and the thesis context and outlines the goals that have led this thesis. Specifically, Section §1.1 describes the concepts of the research context which frame the scope of the work. Next, Section §1.2 details the relationship of this thesis to broader research projects. Then, Section §1.3 exposes the main goals aimed at this dissertation as well as the hypothesis that supports our research. Finally, Section §1.4 describes how the content of this dissertation is organized.

1.1 Research Context

The *Software as a Service* (SaaS) paradigm has become entrenched in the industry as a deployment model, bringing flexibility to the customer in pay-for-use models and making it unnecessary to maintain its own infrastructure. In particular, software delivered with the SaaS paradigm presents business models that depend on the expected benefits for certain infrastructure costs.

The main architectural paradigm of the SaaS deployment model is service-oriented architecture (SOA) [1], [2]. They have a number of characteristics such as low coupling, abstraction, reusability, autonomy, and interoperability. These architectures have traditionally relied on a set of specifications (SOAP, WSDL) that allow varying degrees of granularity and provide complex interfaces that often require heavy deployment infrastructures.

In recent years, there has been a trend towards a new architectural style that has been called microservice. This style requires that each component (a microservice) can evolve, scale and be deployed independently from the rest, increasing the flexibility of the system as a whole. This architectural style is employed by high-performance commercial systems architectures such as eBay, Amazon and Netflix [3].

A common element of these architectures in particular, and, in general, when defining and implementing microservices is that they follow the RESTful paradigm. This paradigm provides a unified approach to identify the granularity and operational interface of microservices that have a high degree of extensibility. Microservices provide numerous advantages in terms of elasticity, fault tolerance and flexible architecture design [4]. Both industry [5] and academia [6] agree to identify the management and evolution of services as a key element to achieve a more agile integration of systems and to have a technological infrastructure that responds quickly to the business needs. These challenges are magnified in the context of microservice architectures since the independent life cycle of each microservice must be coordinated as part of a service catalog that has a higher growth rate than that of classical architectures [7].

Currently, the RESTful paradigm is the preferred choice for the construction of Software as a Service: as these architectures promote the deployment, isolation and integration of components (i.e., microservices) typically by means of RESTful APIs, they pave the way to easily connect to external APIs (as service consumers) or expose internal APIs to the market (as service providers). From now on, these microservices interacting with each other by means of RESTful APIs will be designated as RESTful systems. In these systems, we usually find an API provider exposing a set of APIs that is totally or partially consumed by one or many API clients.

The term of *API Economy* is being increasingly used to describe the movement of the industries to share their internal business assets as APIs [8] not only across internal organizational units but also to external third parties; in doing so, this trend has the potential of unlocking additional business value through the creation of new assets [9, 10]. In fact, we find several examples in the industry that are deployed solely as APIs (such as Meaningcloud, Flightstats or Twilio, amongst many others).

In this *API Economy* context, APIs frequently define their *pricing* by means of *plans* which defines the *cost* and the API *limitations*, such as *quotas* and *rates*. For instance, Meaningcloud's *pricing* has a *Start-Up plan* whose *cost* is 99$ and whose limitations are: a *quota* of 120K requests monthly and a *rate* of 5 requests per second.

Additionally, a frequent problem is the description of the functional elements of each microservice. In the case of RESTful systems, the resources and the available operations have to be defined. In this context, during the last years, a successful standardization effort to have a specification for the functional part of the APIs has been developed; it is the OpenAPI Specification[1].

This specification has pushed forward the creation of a rich open ecosystem of tools[2] and techniques to help the industry in the development lifecycle and evolution of APIs and microservice architectures. This set of tools includes code generators, visual editors, tools for automated testing, validators, etc.

Likewise, a crucial activity is to describe the set of *plans* (i.e., the *pricing*) that depicts the functionality and performance being offered to clients. API providers usually define certain *limitations* in each instance of a plan (e.g., *quotas* and *rates*). For example, a free plan might be limited to having 100 monthly requests, and a professional plan to have 500 monthly requests.

With a similar approach as in other industries, several providers use the Service Level Agreement (SLA) concept to delimit the functionality and guarantee to which they commit to their customers. We identify both *computational SLAs* and *support SLAs*: they differ in that the former refers to a service provided by an automated system (such a web service), while the latter refers to a service provided by people. In this thesis, an SLA will always be a *computational SLA* and we will use interchangeably both *SLA* and *API*

[1] https://www.openapis.org

[2] https://apis.guru/awesome-openapi3

pricing terms to refer to that piece of information that holds the API limitations.

Besides, this information regarding the API pricing and API limitations is neither typically structured nor standardized. Wherefore answering questions regarding API limitations (e.g., determining whether or not a certain *pricing* is valid) is still a manual process with the consequent inconveniences (being tedious, time-consuming, error-prone, etc.).

Although formal specifications have been proposed for the definition of SLAs that arise in the context of the classical SOAs, web services or cloud environments (WSLA [11], SLAng [12], SLA* [13], WS-Agreement [14], SLAC [15], CSLA [16], rSLA [17], L-USDL Agreement [18], ySLA [19]), providers usually have an ad hoc approach with a low degree of automation. For example, some API marketplaces such as RapidAPI[3] do have simple models for applying simple rate or quota limitations, but they are just ad hoc solutions with no interoperability between different providers. This ad hoc approach suggests that the SaaS industry has not incorporated the idea of a model that can be integrated within the infrastructure as a first-class citizen.

Having a standard model for SLAs for APIs (i.e., API pricings) may boost an open and interoperable ecosystem of tools that would represent an improvement for the industry by automating certain tasks and assisting users (both providers and consumers). For example, code generators, visual editors, tools for automated testing, validators, etc. Moreover, this API pricing model also paves the way to a catalog of analysis operations. It may include aspects such as determining whether or not a certain *pricing* is valid, calculating the number of available requests in a certain period or deciding whether a plan is compliant with certain user needs. Those questions could be answered in an automated way if a sufficiently expressive model for API pricings arises.

However, not only single-purpose tools can be created, but a wider, eclectic approach is aimed in this thesis: we want to provide an automated *SLA-Driven governance* framework and to regulate the behavior of each component (i.e., API providers and clients) in the context of the agreed SLAs. For instance, an API provider limiting the service to those API clients who perform more than 10 requests secondly or an API client adapting their API consumption to a certain API plan to always hit the free tier.

We understand *governance* in a similar approach as ITIL does. Governance refers to the means by which an organization is directed and controlled. In service management, *governance* defines the common directions, policies and rules that the organization uses

[3] `https://rapidapi.com`

to deliver and maintain its services. Every organization, regardless of the size, takes direction from a *governance* body: an entity that is accountable at the highest level for the performance and compliance of the organization [20].

Assimilating the forenamed *governance* idea into our approach to SLAs (i.e., SLA-Driven governance), and RESTful APIs, the Service Level Agreement is responsible for directing the RESTful system in accordance with the API limitations, but also it is in charge of monitoring and evaluating the system. These tasks are not straightforward and should be assisted by open tools so that this complexity is hidden to users: that is our vision of an automated *SLA-Driven governance of RESTful Systems*, the object of this thesis.

In this respect, in our research group, *Governify*[4] has emerged as a framework to build SLA-Driven infrastructures. It can be considered as a set of components, exposed as RESTful APIs, utilities and techniques. Some of them are generic whereas others are extensions or adaptations for a domain-specific purpose. Broadly speaking, it is aimed at supporting the design, monitoring and implementation of SLAs-Driven infrastructures.

In this thesis, we seek to go forward and extend, closely aligned to industry standards such as the OpenAPI Specification, the concept of SLA to the RESTful API domain, thus supporting the *governance* of RESTful systems, that is, the design, monitoring and implementation of techniques and tools, aimed at improving the management and evolution of the microservices as part of RESTful system.

We believe that an expressive, fully-fledged specification for SLAs in APIs may boost an open ecosystem of tools that would represent an improvement for the industry. However, both the SLA model and its surrounding ecosystem should be carefully evaluated and validated in several contexts before we can affirm it is an actual improvement.

Hypothesis

There is no well-established ***model for describing API pricings (SLAs) in RESTful systems****, which is hindering the* ***automatic governance***

[4] https://www.isa.us.es/3.0/tool/governify

1.2 Thesis Context

This thesis has been developed in the context of the *Applied Software Engineering* Research Group[5] of the *Universidad de Sevilla* and it is intended to extend the scope of *BELI*[6] (TIN2015-70560-R), a Spanish national government project, whose objectives include the development of a *governance* system for hybrid services systems that integrate information systems (computer services) and teams of people. In this context, the initial stages have been covered with the Governify[7] ecosystem, a framework to build SLA-Driven infrastructures. It can be considered as a set of components, exposed as RESTful APIs, utilities and techniques. It is aimed at supporting the design, monitoring and implementation of SLAs-Driven infrastructures.

The relevance in the industry of the thesis topic is evidenced in numerous ICT projects. Specifically, we highlight the *SLA@OAI*[8] project in 2016, initially developed in conjunction with *Icinetic*, which aims at developing an operational extension to define SLAs within the framework of the OpenAPI Initiative (OAI). We named this extension *SLA for OpenAPI* (SLA4OAI). In 2018, the *OpenAPI Governance Board* approved the creation of a *Specific Interest Group*[9] (SIG) to coordinate this extension. It is integrated by a number of the companies that are part of the OpenAPI Initiative, including but not limited to: *API Evangelist*, *Apigee*, *Apimetrics*, *AsyncAPI*, *Google*, *Level 250*, *Metadev*, *Paypal* and *SpotLight*. In 2021, an initial proposal was presented to the *OpenAPI Goverance Board* and, with their feedback, the work on a next version started.

Other projects have had an impact on this thesis. On the one hand, in national research projects such as *TAPAS*[10], *THEOS*[11] and HORATIO[12] the concept of SLA has played a crucial role; during the execution of these projects we have refined and enhanced the concept of SLAs in a variety of contexts. On the other hand, ICT projects such as *PROSAS*[13] (2016) with *Accenture* and *GAUSS*[14] (2017) with *Everis Spain*, both focused on the SLA *governance* in business processes, had a tangential impact in this thesis as they help in the enhancing of the SLA model that later will be extended for APIs.

[5] https://isa.us.es
[6] https://investigacion.us.es/sisius/sis_proyecto.php?idproy=26807
[7] https://www.governify.io
[8] https://investigacion.us.es/sisius/sis_proyecto.php?idproy=26651
[9] https://github.com/isa-group/SLA4OAI-TC
[10] https://investigacion.us.es/sisius/sis_proyecto.php?idproy=21610
[11] https://investigacion.us.es/sisius/sis_proyecto.php?idproy=18670
[12] https://investigacion.us.es/sisius/sis_proyecto.php?idproy=29623
[13] https://investigacion.us.es/sisius/sis_proyecto.php?idproy=27485
[14] https://investigacion.us.es/sisius/sis_proyecto.php?idproy=28203

1.3 Thesis Goals

This thesis focuses on the study of the ways to support, improve and automate the SLA-Driven *governance* of RESTful APIs through the definition of methodologies, techniques and tools. Specifically, we have identified two main challenges:

- **CH1: Establish a sufficiently expressive specification for the description of API pricings and the analysis of their validity.**

  This challenge is focused on defining a specification having every necessary element to support the description of pricings in APIs with *limitations*. This specification should include aspects regarding the API itself as well as pricing aspects since the regulation of an API is closely related to its business model. This challenge involves several requirements, namely:

  – Analyze, systematically, a representative set of real-world APIs to evaluate the characteristics of the API limitations as part of the API pricing.

  – Identify both the importance and the role of SLAs during the API development lifecycle in different scenarios, considering both academic and industrial points of view.

  – Propose a comprehensive model for describing the API pricings.

  – Develop a catalog of operations for answering common questions regarding the validity in API limitations and API pricings.

  – Evaluate the proposal by modeling the API pricings of a representative set of real-world APIs.

- **CH2: Implement an ecosystem of tools and operations to support the SLA-Driven *governance* of RESTful APIs.**

  This challenge aims at addressing the evolution from the current SLAs management libraries, existing at the time of starting this thesis, to an open ecosystem of tools articulated in microservices as new components within the *Governify* framework. This ecosystem should allow integrating both models and analysis operations APIs so that it results in a useful and ready-to-use platform for researchers and practitioners. This challenge involves several requirements, namely:

  – Implementation of the tools aimed to support the *governance* of APIs.

  – Implementation of a catalog of operations regarding common questions in the context of API pricings validity.

– Validation of the ecosystem of tools in research and industrial contexts.

In order to succeed in our primary focus of supporting, improving and automating SLA-Driven *governance* of RESTful APIs, both challenges ought to be accomplished. We need an expressive specification of API pricings as well as a ready-to-use implementation of an ecosystem of tools and operations. We wherefore claim the main goal of this thesis to be:

Main goal

Creation of an ***expressive, fully-fledged specification of SLAs for RESTful APIs*** *endorsed with an* ***open ecosystem of tools and operations*** *aimed at the* ***SLA-Driven Governance of RESTful systems****.*

The next chapters will focus on refining how we have contributed to deal with the aforementioned challenges.

1.4 Outline

This thesis document is organized as follows:

Part I: PRELIMINARIES. First, Chapter §1 includes the research context, the justification of the relevance, the hypothesis and main goals. Next, Chapter §2 describes the main results of the research presented as part of this thesis.

Part II: PUBLICATIONS. It includes the three core publications that comprise this compendium as well as other supporting publications in both national and international forums. The overview of these publications can be found in Chapter §3. Specifically, Section §3.1 discusses the main contributions of the thesis with regard to the goals and Section §3.2 holds the list of relevant publications. Finally, the full-text copy of the three core publications can be found in Chapter §4, Chapter §5 and Chapter §6 respectively.

Part III: FINAL REMARKS. The Chapter §7 concludes the dissertation. Section §7.1 summarises the contributions and discusses their usefulness. Finally, Section §7.2 includes the limitations and future work.

Part IV: APPENDICES. It includes additional publications to serve as supplemental material. Specifically, Appendix §A and Appendix §B are papers published in a *demo* conference track while Appendix §C presents some ongoing work yet to be submitted to a journal.

Summary of Results

This chapter briefly describes the main results of the research presented in this thesis. Specifically, Section §2.2 summarises our work on the model, Section §2.3 highlight the analysis operations and, finally, Section §2.4 describes the ecosystem of tools.

2.1 Introduction: *Governify4APIs*

In this section, we summarize the main results of this thesis, which are part of the compendium presented in Section §3.2. The high-level contribution is the *Governify4APIs* Ecosystem, a set of new components or adaptations aimed at supporting the *SLA-Driven Governance of RESTful systems*.

This *Governify4APIs* ecosystem consists of two main differentiated parts; first, the *creation of a sufficiently expressive specification for the description of API pricings and the analysis of their validity* and secondly, the *implementation of an ecosystem of tools to support the SLA-Driven governance of RESTful APIs*.

To begin with, we briefly present this expressive specification, our *Governify4APIs* Model, by means of a real example in Section §2.2. Next, in Section §2.2.4, we present a serialization of the previous model as an extension of the OpenAPI Specification. Finally, in Section §2.3 we propose a validity analysis operation for API pricings.

Thereafter, Section §2.4 describes the ecosystem of tools that has been developed around the *Governify4APIs* Model, including the *SLA Editor*, *SLA Engine*, *SLA-Driven API Gateway*, *ELEcTRA* and the *SLA4OAI Analyzer*.

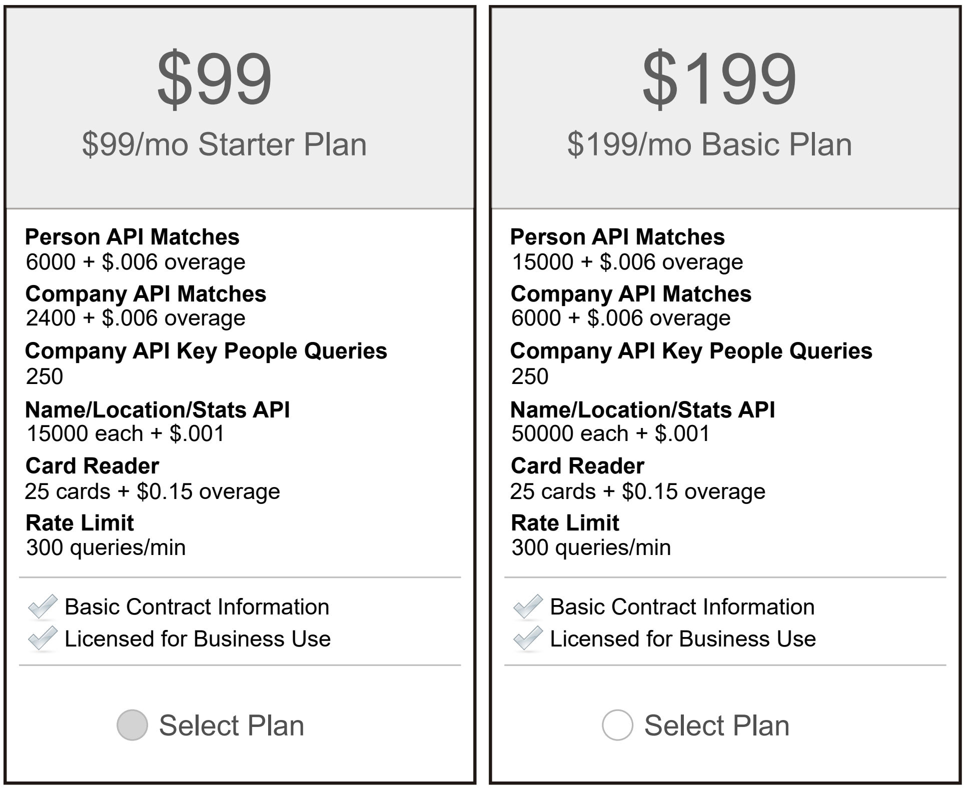
2.2 *Governify4APIs* Model

In the API Economy world, API providers have to make sufficient information available for the consumer to get informed about their products. This includes information regarding the API itself (endpoints and methods), the *plans* that a user can subscribe to, and the associated *cost*. A *plan* includes information regarding the API's limitations (*quota* and *rates*) for each of its resources.

All this information is typically found in a section called *pricing* (however, cloud infrastructure providers tend to refer to it as an *offering*). Consequently, we shall henceforth consider a *pricing* to be a set of *plans* having an associated *cost*.

In order to illustrate these concepts, we present a real example: the FullContact API, a tool for managing and combining contacts from different sources (Gmail, social media, etc.). The API allows users to programmatically look up information and match email addresses with publicly available information so as to enrich the contacts. Figure §2.1

depicts the pricing extracted from the *FullContact*[1] API.



$99	$199
$99/mo Starter Plan	$199/mo Basic Plan
Person API Matches 6000 + $.006 overage	**Person API Matches** 15000 + $.006 overage
Company API Matches 2400 + $.006 overage	**Company API Matches** 6000 + $.006 overage
Company API Key People Queries 250	**Company API Key People Queries** 250
Name/Location/Stats API 15000 each + $.001	**Name/Location/Stats API** 50000 each + $.001
Card Reader 25 cards + $0.15 overage	**Card Reader** 25 cards + $0.15 overage
Rate Limit 300 queries/min	**Rate Limit** 300 queries/min
✓ Basic Contract Information	✓ Basic Contract Information
✓ Licensed for Business Use	✓ Licensed for Business Use
● Select Plan	○ Select Plan

Figure 2.1: Plans of the FullContact API.

This *pricing* example consists of two paid *plans* having a fixed price *cost* billed monthly. With respect to the *limitations*, for each *operation*, a *quota* is applied. For example, in the starter *plan*, only 6000 matches on Person are available. Nevertheless, an *overage* is defined, i.e., it is possible to surpass the *limit* by paying a certain amount of money, in this case, $0.006 per request. Regardless of the plan, a common *rate* of 300 queries per minute is applied.

In this context, several analytical challenges can arise since the API providers need to understand the plans in depth before taking further action. In particular, they should verify the validity of their plans (i.e., that there is nothing inconsistent). Those challenges correspond to common questions on the API's pricing and plans that could be answered automatically with an appropriate model and analytical framework providing validity analysis operations.

In this thesis, we propose *Governify4APIs*, a model for API pricings (i.e., of each plan and the associated cost for a given API) which starts from the idea that each API resource

[1]https://www.fullcontact.com/developer. Accessed May 2019.

(HTTP path and method) has a related set of limitations (quotas and rates) for each API plan.

Figure §2.2 depicts the entire *Governify4APIs* model. For the sake of clarity, we have split it into three areas: (i) in dark grey, *pricing, plans and cost*; (ii) in light grey, *limitations and limits*; (iii) in medium grey, *capacity*. In the following subsections, we will detail each part of the model with examples extracted from the FullContact API in Figure §2.1, considering each part: the plan area (Subsection §2.2.1), the limitations area (Subsection §2.2.2), and the capacity area (Subsection §2.2.3).

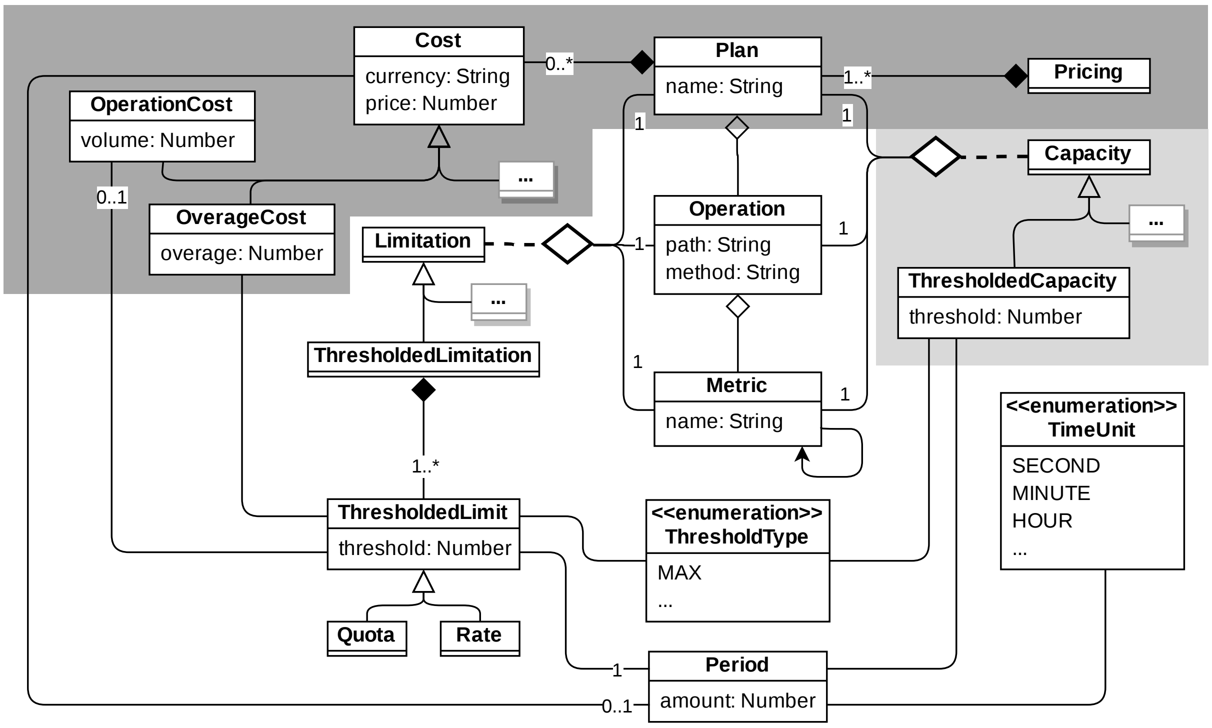


Figure 2.2: *Governify4APIs* model for API pricing.

2.2.1 Pricing, Plans, and Cost

As depicted in the model (dark grey in Figure §2.2), a `Pricing` consists of a set of `Plans`. A `Plan` has a name and a `Cost` that defines the price charged to users so that they can access the service. In our example, the FullContact API has two `plans`: a *starter* and a *basic* `Plan`.

The `Cost` may be very simple (e.g., assign a constant price to the `Plan`, e.g., *$99* or *$199* as in our example) or may depend on other properties. In this latter case, when the cost depends on a `Limitation`, we distinguish two costs: `OperationCost`, when an `Operation` is being charged for each time it is invoked; and `OverageCost`, when once a certain value of the `Limitation` has been reached (cf. Subsection §2.2.2), there start to be imposed charges per volume.

Either type of `Cost` can be periodic, defining a `Period` with an amount and a `TimeUnit`. In our example, the `Cost` of the *Starter* `Plan` is *99$* billed *monthly*, i.e., it has a *Period* with value 1 of the *TimeUnit* MONTH.

An `OperationCost` is frequent in pay-as-you-go payment models in which there is no monthly fixed `Cost` and the API consumer is only charged for, given a *requests* `metric`, the number of requests. In the model, this cost is associated with the operation by means of the Limitation. For example, a service might offer a `Plan` A in which each request can be charged at 0.10$ (volume: 1) and a `Plan` B where each pack of 1000 requests (volume: 1000) is charged at 75$. Depending on the client's needs, they might prefer `Plan` A or `Plan` B.

An `OverageCost` is usual when providers do not want to cut off the service once a `Limitation` has been reached, but want to continue providing it at a certain charge. Our example defines an overage when the `quota` values are reached: *each additional match after 6000 monthly matches is charged at $0.006*.

2.2.2 Limitations and Limits

As depicted in the model (unshaded section in Figure §2.2), in order to carry out this regulation of the consumption of an API, each `Operation` in a `Plan` can be subject to `Limitations` on a `Metric`. The most frequent type of `Limitation` is the `ThresholdedLimitation` which establishes one or more `ThresholdLimits` on the number of `Metric` units in a `Period`. The `ThresholdType` is usually MAX (i.e., the `ThresholdLimit` would therefore represent the *maximum* number of `Metric` units). In defining their `Pricing`, `Limitations` allow providers to adjust the API's consumption to the platform's total `Capacity` (cf. Subsection §2.2.3).

An `Operation` is defined by the pair formed by HTTP method and path. For example, *GET /contacts* would represent the query operation on a collection of user-type objects. A common example of a `Metric` is the *number of requests*. Nonetheless, other metrics can be defined such as storage, bandwidth or CPU consumption.

In accordance with the implementation, i.e., the algorithm used to enforce the `Limi-`

`tations`, we say that a `ThresholdedLimit` is a `Quota` if the computation of the number of metric units is done over a static window, i.e., in a *fixed* time window. For example, a *one-week static window* might be such that it always starts on Monday at 00:00 and ends on Sunday at 23:59, regardless of when the first metric unit is computed. On the contrary, if the time window is *sliding*, i.e., relative to the first metric unit computed, we say that the `ThresholdedLimit` is a `Rate`. For example, in a *one-week sliding window*, if the first metric unit were computed on Wednesday at 15:36:39, that window would close on the following Wednesday at 15:36:38.

Figure §2.3 illustrates graphically the differences between sliding and static windows. 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.

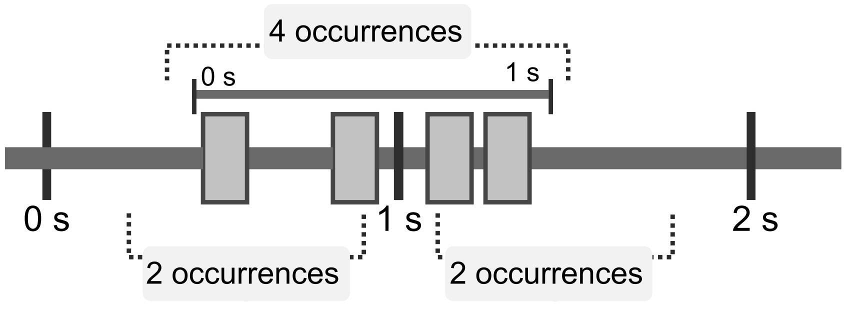


Figure 2.3: Sliding (rates) vs static (quotas) windows.

For example, if we use the *number of requests* as a `Metric`, and we want to prevent our users from making more than 4 requests per second, there are two different alternatives: a 1-second sliding time window with a limit of 4 requests, that opens after the first request and prevents more than 4 from being made during that second; or a 1-second static window with a limit of 2 requests, that could concentrate the first two requests at the end of the first second and the other two at the beginning of the next one.

In the industry, these limitations tend to follow definite patterns [21]. Specifically, `Quotas` tend to be defined over any metric and are measured in periods longer than an hour (e.g., daily, weekly, monthly or yearly), while `Rates` tend to be defined over the number of requests and are measured in shorter periods (e.g., secondly or minutely).

In our FullContact example, the *starter* plan has one `Rate` and four different `Quotas`. For example, the `Rate` is *300 requests in a 1-minute sliding window* and a `Quota` is *6000* *matches in a 1-month static window.*

The model distinguishes two concepts: `ThresholdedLimitation` and `ThresholdedLimit`. A `ThresholdedLimitation` over a certain metric and operation establishes a fraction of the overall `Capacity` of the service. A `ThresholdedLimitation`, however, can be expressed in various ways, one of which is by defining a set of `ThresholdedLimits` that, within a time period, restrict the percentage of `Capacity` that consumers are allowed to use. For example, a `ThresholdedLimitation` on a certain operation can be defined as a set of `ThresholdedLimits` as follows: *30 requests every 1 week* and *1 request every 1 second.*

A different way to express a `Limitation` (as represented by the ellipsis "..." in the model) would be to use frequency distributions [22], so that referring to percentiles would allow the form of the distribution and its different attributes to be considered. For example, a percentile, such as 99.0 or 99.9 would show a plausible value in the worst case, while the 50th percentile would emphasize the typical case. In the present communication, however, we will not address limitations specified as frequency distributions.

2.2.3 Capacity

Finally, a crucial aspect that is not explicitly depicted in a pricing or a plan is the `Capacity`. This is an internal aspect that providers do not put out publicly. The `Capacity` of the service represents a subset of the constraints of the platform or system on which the service is being deployed. It is the result of having to satisfy mainly technical and budget criteria (e.g., CPU or memory, number of nodes of the cluster, etc.).

Estimating the service's `Capacity` is fundamental to defining the `Pricing` and analysing the `Limitations`. In particular, all the `Limitations` ought to be satisfied by the service, i.e., they must not exceed the service's `Capacity`.

As depicted in the model (medium grey in Figure §2.2), once the `Capacity` has been identified, it is specified as if it were a `Limitation`, i.e., the number of certain `metric` units in a given `Period`. Therefore, analogously to the `Limitation`, the `ThresholdedCapacity` has a threshold value and a `ThresholdType` (usually MAX) in a given `Period` of a `TimeUnit`.

A possible way to express the `Capacity` on the metric *request* is the *number of requests per second (RPS)* for each operation and plan. For example, a capacity of *10 000 RPS* in *GET /pets* in the *free plan* would mean that the entire set of free-plan users will be able to make 10 000 RPS. The `Capacity` can be different for each plan since different infrastructures may be used to provide a better level of service to the clients.

For example, an organization might have calculated, based on performance and stress tests, that its production cluster is able to accept 10 000 RPS. Consequently, if a limitation had been set of 10 requests per second per client, the theoretical number of concurrent requests would be $10000/10 = 1000$ concurrent clients.

A useful instrument when analysing `Limitations` is the *percentage of capacity utilization* or simply the *percentage of utilization (PU)*. Intuitively, this percentage directly determines whether or not a `Limitation` can be set because this will be impossible if the PU is greater than 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*:

In a first approximation, an API consumer could assume that, since 1 day is 86 400 seconds, for every second, they will have $43200/86400 = 0.5$ requests. In this case, it is assumed a *uniform distribution* in which, little by little, the consumer will reach the 43 200 requests available in the day. This scenario corresponds to the *minimum PU*. But the `ThresholdedLimitation` states that for 1 day it is possible to make 43 200 requests, and in no case does it prevent the consumer from making all of them in a burst in the first instant of time. Indeed, in 1 second the consumer could make the whole set of 43 200 requests. This scenario implies a *burst distribution*, and corresponds to 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*.

Figure §2.4 illustrates different consumption scenarios for the same `ThresholdedLimitation` of *60 requests every 60 seconds*.

In a *uniform* consumption, 60 requests in 60 seconds would be equivalent to 1 request every 1 second. However, in a *burst* consumption, 2, 3, 6, or even a maximum of 60

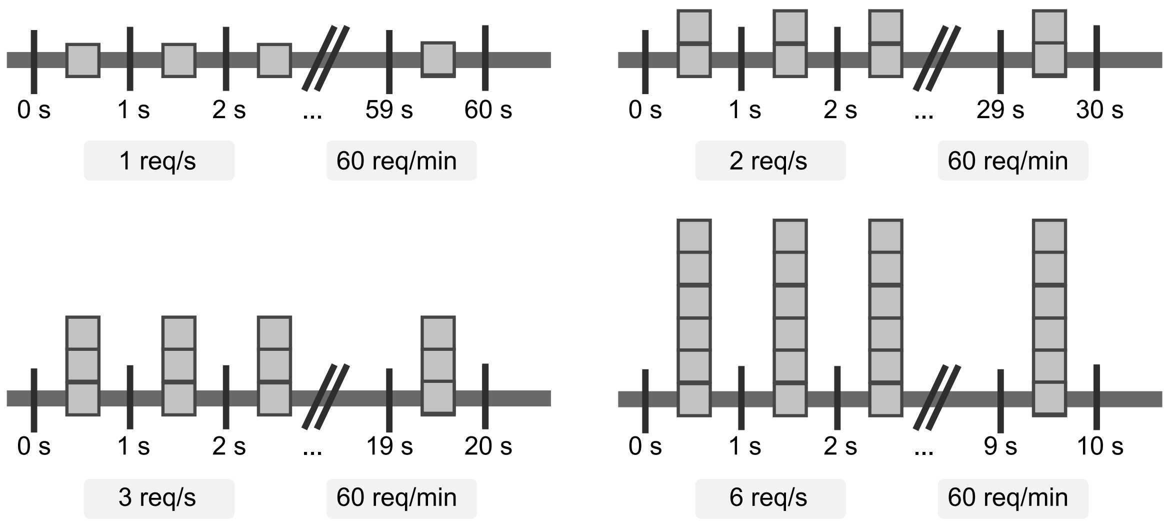


Figure 2.4: Examples of different consumption scenarios for the same `ThresholdedLimitation`.

requests could be made in 1 second. Therefore, to calculate the BPU in the limitation of *60 requests every 60 seconds*, we should take as a minimum value the *uniform* distribution of 1 request per second and as a maximum value the *burst* of 60 requests in 1 second in a 1 minute window.

2.2.4 SLA4OAI: A Serialization for our Model

The *Governify4APIs* model can be serialized to be aligned to a variety of API description specifications. Specifically, we propose SLA4OAI[2] [23, 24], 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).

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 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 a 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

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

the fields in a SLA4OAI file. Nevertheless, for a comprehensive description of the syntax, a JSON Schema document is available[25]. Further information is also available in the the specification's GitHub page [3].

As depicted in Listing 2.1, 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 [24] (i.e., a protocol as part of the SLA4OAI proposal, beyond the scope of the present communication), 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 2.1: Main elements in SLA4OAI

Specifically, as depicted in Listing 2.2, 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: FullContact
  sla: '1.0'
  type: plans
  api: ./fullcontact-oas.yaml
  provider: FullContact
infrastructure:
  supervisor: https://...
  monitor: https://...
availability: '2009-10-09T21:30:00.00Z'
```

Listing 2.2: Context, infrastructure and availability details in SLA4OAI

In the `Metrics` field, as depicted in Listing 2.3, 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

[3]https://github.com/isa-group/SLA4OAI-ResearchSpecification

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
matches:
  type: integer
  format: int64
  description: Number of matches
```

Listing 2.3: Metric details in SLA4OAI

The `Plans` section, as depicted in Listing 2.4, 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, a `period` with `amount` and a time `unit`, and the `scope` over which they should be enforced. As an extensible scope model, we propose two possible initial values (`tenant` or `account` as default).

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.

```
plans:
  Starter:
    pricing:
      cost: 99
      currency: USD
      period:
        amount: 1
        unit: month
    quotas:
      'v3/person.enrich':
        post:
          matches:
            - max: 6000
              cost:
                overage:
                  overage: 1
                  cost: 0.006
    rates:
      'v3/person.enrich':
        post:
          requests:
              - max: 10
                period:
                  amount: 1
                  unit: month
```

Listing 2.4: Plans details in SLA4OAI

2.3 Analysis

In this section, we propose an analysis framework to form a ground on which to reason about the pricing model presented. Consequently, this framework paves the way to exploiting the information contained in the model, and has been used to develop a validity analysis operation that could be useful in a real setting for both consumers and providers of APIs. As a foundation for the analysis operations, the first of the following subsections addresses the cornerstone of the analysis framework – the relationship between limitations and capacity.

2.3.1 Limits as Percentages of Capacity Utilization

Since the capacity of the platform on which the service is deployed is not unlimited, the pricings should be defined to be compatible and coherent with that capacity. As an example, ensuring that the total capacity is sufficient for the potential use of the service defined in a particular plan should be analyzed. Furthermore, we proposed (in Subsection §2.2.3) the notion that any given limitation corresponds to a Bounded set of Percentage of capacity Utilization (BPU) values derived from the potential usage scenarios a client could have for their consumption within the API while meeting its limitation.

In this context, the correspondence between limitations and BPU can be obtained by means of a normalization procedure that transforms the unit of the limitation to the capacity time unit, and then computing the minimum and maximum possible PUs. This procedure comprises just simple calculations, as is illustrated in the following example:

Consider a limitation with a limit of *43 200 rquests / 1 day* and assume a total capacity of 50 000 RPS. Since all limitations should be expressed using the time unit of the capacity (second), the limitation is *43 200 requests / 86 400 seconds*. First, assume a *uniform* consumption, i.e., if in 1 day (86 400 seconds) there are 43 200 requests, in there will be $43200/86400 = 0.5$ RPS. Given the value of the capacity, 50 000 RPS, the minimum PU is $0.5/50000 = 0.00001 = 0.001\%$. Now assume a single *burst* consumption, i.e., if a burst of 43 200 requests can occur during any 1 second window over 1 day. Given the value of the capacity, 50 000 RPS, the maximum PU is $43200/50000 = 0.864 = 86.4\%$. Therefore, the BPU of *43 200 requests / 1 day* subject to a capacity of 50 000 RPS is [0.001%-86.4%].

The calculation of the overall system capacity is a non-trivial procedure. It requires great technical effort in order to make a proper estimate. But, depending on the stage of development, even this will not always be feasible. In the present study, when the value of the system's capacity is unknown, we shall assign it the value of the highest capacity needed. To calculate this value, we shall assume uniform consumption after normalizing to the smallest time unit, and take the greatest value. The following is an example:

Consider the following two limitations: *1 RPS* and *100 RPW* (1 week, 604 800 s).In order to take the value of the highest capacity needed, we must first determine what the strongest limitation is. For this case, we normalize to the smallest unit, the second, $1\,RPS = 1$ and $100\,req/604800s = 0.000165\,RPS$, since $1 > 0.000165$ we have that the highest capacity needed is *1 RPS*. Therefore, we will take *1 RPS* as the value of the capacity. As a conclusion, it is worth noting that 1 RPS requires a higher capacity than 100 RPW, which only requires 0.000 165 RPS.

2.3.2 Pricing Validity

We define the **validity** of a pricing as checking whether it is valid depending on a set of validity criteria. These include the absence of different types of conflict, for example, two limits within a limitation that cannot be met at the same time. The validity of a *Governify4APIs* model is defined as certain validity criteria being met in each part of the model. In the model, a *pricing* has a set of *plans*, and these plans consist of *limitations*, each with its own *limits*. This hierarchy carries over to the validity operation. Hence, for example, a *pricing* will be valid, notwithstanding its satisfying other additional validity criteria, if all of its *plans* are also valid.

For solving validity conflicts, a **priority criterion** is required. For example, if two limits are defined with different values for a given metric and operation, which one should prevail over the other? In order to satisfy these requirements we assume henceforth the following default priority criteria: i) limitations with smaller periods over limitations with higher periods; ii) rates over quotas; iii) metric *number of requests* over any other metric. Notwithstanding, these criteria can be re-defined in other scenarios (e.g., metric *requests* is less important than the bandwidth in a certain business context).

We shall present the validity criteria in a hierarchy, starting from the fine-grained

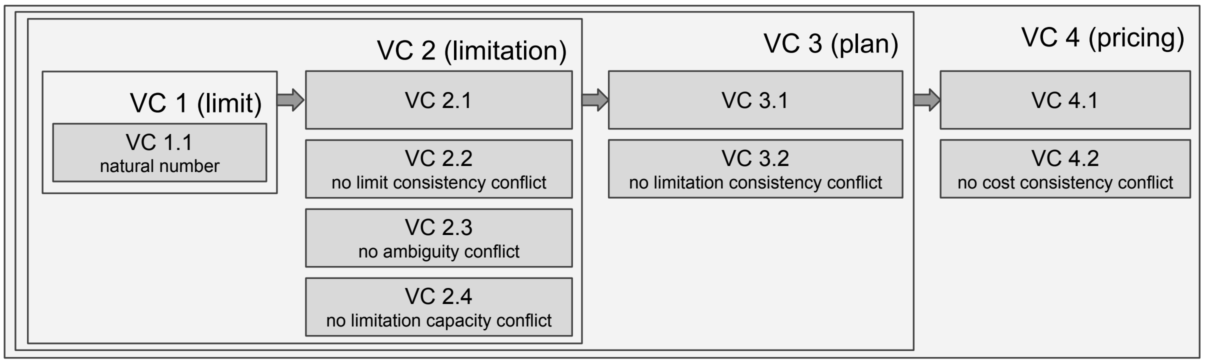


Figure 2.5: Validity criteria hierarchy.

(VC1 - limits, VC2 - limitation) to the coarse-grained (VC3 - plan, VC4 - pricing) validity criteria. Each validity criterion comprises multiple validity subcriteria. Figure §2.5 gives an overview of this hierarchy of validity criteria.

The details of each validity criterion are as follows:

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., there is no situation exceeding a *limit* with more priority while it is allowed by another *limit* with less priority (VC2.2); there are no *ambiguity conflicts* between any pair of its *limits*, i.e., two limits using the same period with different values (VC2.3) and there is no *capacity conflict*, i.e., the limitation does not surpass the associated *capacity* (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., two *limitations* on two related *metrics* (by a certain factor) cannot be met at the same time (VC3.2). If they happen to exist, the priority criteria will be used for determining which limit has to be prioritized.

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 one plan is less restrictive than the equivalent in another *plan* but the former *plan* is cheaper than the latter (VC4.2).

In order to understand these validity criteria, on the following subsections we will present examples of existence and absence of conflicts of each type.

Limit Consistency Conflict (VC2.2)

```
Capacity: 1000000 RPS
Limitations:
  Quota: 100 requests / 1 day
  Quota: 1000 requests / 1 week
```

Listing 2.5: Validity criterion VC2.2 (no limit consistency conflict)

An example of a situation where **there is no limit consistency conflict** can be observed in Listing 2.5. An inconsistency occurs when there is a possible situation exceeding a *limit* with more priority while it is allowed by another *limit* with less priority, according to the priority criteria hereinbefore mentioned. An example, using the size of the periods as the priority criterion, a conflict shall happen if the minimum PU of the limit with the longest period is less than the minimum PU of the limit with the shortest period.

The limit having the longest period is *1000 requests / 1 week* whose minimum PU is $1000/1\,000\,000 = 0.10\%$. The limit with the shortest period, *100 requests / 1 day*, has a minimum PU of $100/1\,000\,000 = 0.01\%$. Since $0.10\% \not< 0.01\%$, there is no conflict between these limits.

```
Capacity: 1000000 RPS
Limitations:
  Quota: 100 requests / 1 day
  Quota: 10 requests / 1 week
```

Listing 2.6: Validity criterion VC2.2 (limit consistency conflict)

On the other hand, in Listing 2.6 **there is a limit consistency conflict**. The limit with the longest period is *10 requests / 1 week* whose minimum PU is $10/1\,000\,000 = 0.001\%$. The limit with the shortest period, *100 requests / 1 day*, has a minimum PU of $100/1\,000\,000 = 0.01\%$. Since $0.001\% < 0.01\%$, there is a limit consistency conflict between these limits.

Ambiguity Conflict (VC2.3)

```
Limitation:
  Limit: 1 request / 1 second
  Limit: 100 requests / 1 day
```

Listing 2.7: Validity criterion VC2.3 (no ambiguity conflict)

An example where **there is no ambiguity conflict** is presented in Listing 2.7, because the limits of the limitation use different periods, i.e., *1 second* and *1 day*.

```
Limitation:
  Limit: 1 requests / 1 second
  Limit: 100 requests / 1 second
```

Listing 2.8: Validity criterion VC2.3 (ambiguity conflict)

Conversely, in Listing 2.8 **there is a consistency conflict** because the limits of the limitation use the same period, i.e., *1 second*.

Capacity Conflict (VC2.4)

```
Capacity: 100 requests / 1 second
  Limitations:
    Quota: 50 requests / 1 day
```

Listing 2.9: Validity criterion VC2.4 (no capacity conflict)

A possible situation where **there is no capacity conflict** is shown in Listing 2.9. First, we normalize using the unit of the capacity (i.e., seconds). Thus, there are *50 requests / 86 400s (1 day)*. Next, to calculate the BPU, we need both (i) the *minimum PU* (uniform distribution) and (ii) the *maximum PU* (burst distribution). For (i), if in 86 400 seconds there are 5 requests, in 1 second there will be $50/86\,400 = 0.000\,57$ requests. The *minimum PU* is $0.000\,57/100 = 0.0057\%$. For (ii), in 1 second there will be a burst of 50 requests. The *maximum PU* is $50/100 = 50\%$. Therefore, the BPU is [0.0057%,50%].

Since BPU is always less than 100%, there is no capacity conflict.

```
Capacity: 100 requests / 1 second (100 RPS)
Limitations:
  Quota: 200 requests / 1 day
```

Listing 2.10: Validity criterion VC2.4 (capacity conflict)

On the contrary, in Listing 2.10 **there is a capacity conflict**. First, we normalize using the unit of the capacity (i.e., seconds). Thus, there are *200 requests / 86 400s (1 day)*. Next, to calculate the BPU, we need both (i) the *minimum PU* (uniform distribution) and (ii) the *maximum PU* (burst distribution). For (i), if in 86 400 seconds there are 5 requests, in 1 second there will be $200/86\,400 = 0.0023$ requests. The *minimum PU* is $0.0023/100 = 0.000\,23\%$. For (ii), in 1 second there will be a burst of 200 requests. The *maximum PU* is $200/100 = 200\%$. Therefore, the BPU is [0.000 23%,200%].

Since BPU is greater than 100%, there is a capacity conflict because of the *maximum PU*.

```
Capacity: 100 requests / 1 second (100 RPS)
Limitations:
  Quota: 200 requests / 1 day
  Rate: 99 requests / 1 second
```

Listing 2.11: Validity criterion VC2.4 (no capacity conflict)

Additionally, Listing 2.11 presents another example where **there is no capacity conflict**. First, we normalize using the unit of the capacity (i.e., seconds). Thus, there are *200 requests / 86 400s (1 day)* and *99 requests / 1s* Next, we calculate the BPU in each limitation as in other examples. The first limitation's BPU is [0.000 23%,200%].

Next, to calculate the BPU, we need both (i) the *minimum PU* (uniform distribution) and (ii) the *maximum PU* (burst distribution). For (i), the *minimum PU* is $99/100 =$ 99%. For (ii), in 1 second there will be a burst of 99 requests. The *maximum PU* is $99/100 =$ 99%. Therefore, the BPU is [99%,99%].

Now, we aggregate both BPUs: first, we get the maximum value of the minimum PU: *max(0.000 23%, 99%)=99%*. Next, we obtain the minimum value of the maximum PU: *min(200%,99%)=99%*.

Therefore, as a result, we got [99%,99%]. Given that it does not surpass the capacity, we state that there is no capacity conflict.

2.4 *Governify4APIs* Ecosystem

Across the years, in our research group, *Governify*[4] has emerged as a framework to build SLA-Driven infrastructures. *Governify* can be considered as a set of components, exposed as RESTful APIs, utilities and techniques. Within this ecosystem, some components are generic whereas others are extensions or adaptations for a domain-specific purpose, ranging from the business process domain to ensuring best practices in software development.

In this thesis, we present the *Governify4APIs* Ecosystem, a set of new components or adaptations, on the top of *Governify*, for the *SLA-Driven Governance of RESTful systems*. Specifically, Figure §2.6 shows the big picture of the *Governify4APIs* architecture.

First, we have adapted two main top-level components: (i) an SLA Editor, for creating and editing SLA4OAI documents and (ii) an SLA Engine implementing a protocol for

[4]https://www.governify.io

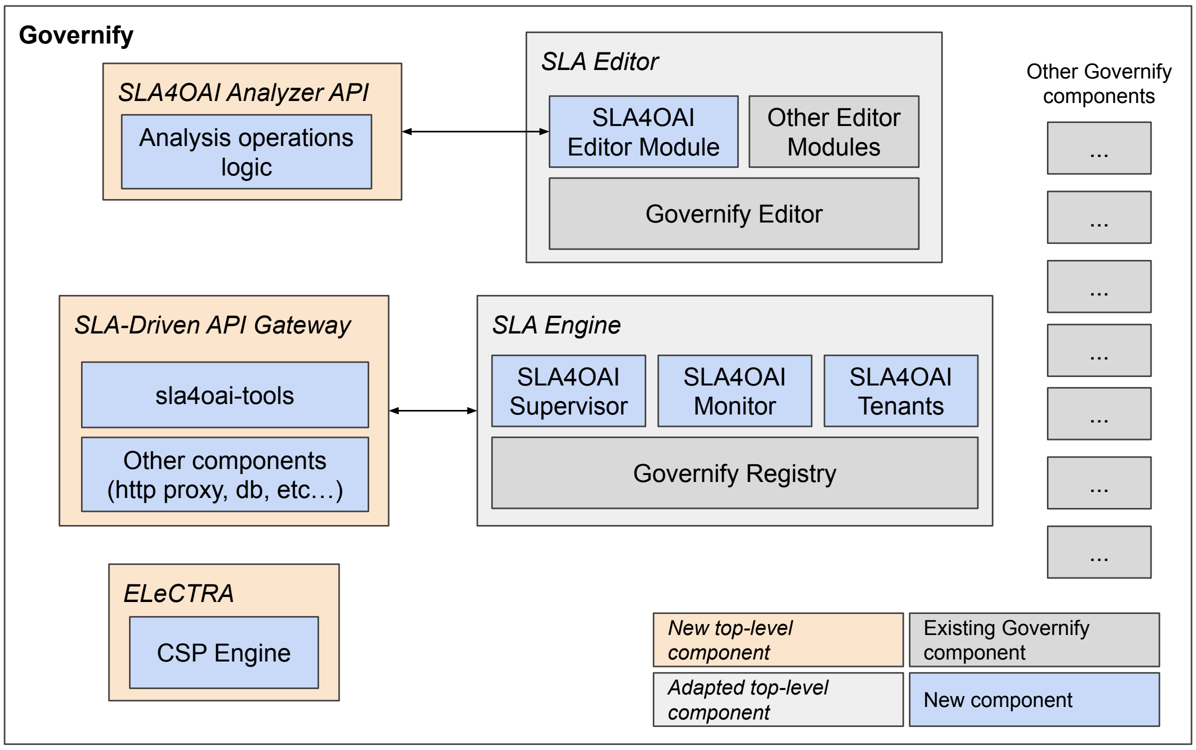


Figure 2.6: *Governify4APIs*' architecture

enforcing SLAs. Next, we have developed three main top-level components: (i) an SLA-Driven API Gateway which, using the SLA Engine, helps users to seamlessly govern their APIs; (ii) a SLA4OAI analyzer implementing the pricing validity analysis operation; (iii) ELeCTRA an initial prototype of an analysis tool based on CSP (Constraint Solving Programming) for performing complex analysis operations.

Our view of an *SLA-Driven Governance of RESTful systems* to success is that users should be assisted by a set of tools during each activity, from the development until the validation. Since we seek to offer a fully-fledged SLA4OAI language, we provide an initial working implementation of these tools.

The SLA Editor, the SLA Engine and the SLA-Driven API Gateway are extensively described at SLA [26]; next, ELeCTRA is described at [27]; finally, SLA4OAI Analyzer is not yet published as it is an ongoing work.

From a technical point of view, every *Governify* (and, consequently, each *Governify4APIs*) service is inspired in the microservice architectural style, they are mostly written in

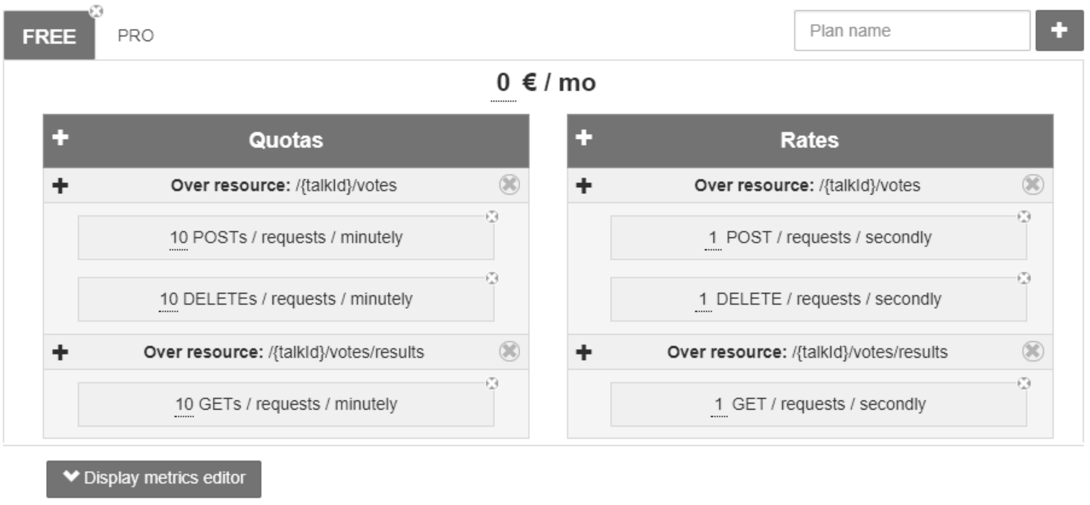
Node.js and they are have been packed as container images[5] and formerly deployed for public online access at the University's own infrastructure (under DNS domains **.governify.io*).

In the subsequent sections, we present each *Governify4APIs*' component:

2.4.1 SLA Editor

In modeling tasks, supporting tools are commonly provided to the users. In this scenario, we provide the *SLA editor*[6], aiming the modeling of pricings in APIs once it has been modeled with OAS. This editor is an extension of the *Governify Editor*: we have created a SLA4OAI module and it has been integrated into the editor.

Our editor is a user-friendly and web-based text editor specifically developed for assisting the user during modeling tasks, including auto-completion, syntax checking, and automatic binding. It is possible to create plans (e.g., free and pro) with quotas and rates. Clicking on the + sign, the user is able to select the path and method (previously defined in the OAS document) for entering the value of the limitation. Note that custom metrics can also be defined at the bottom, however, the calculation logic is left open for a specific implementation.



0 €/ mo

Quotas	Rates
Over resource: /{talkId}/votes	**Over resource:** /{talkId}/votes
10 POSTs / requests / minutely	1 POST / requests / secondly
10 DELETEs / requests / minutely	1 DELETE / requests / secondly
Over resource: /{talkId}/votes/results	**Over resource:** /{talkId}/votes/results
10 GETs / requests / minutely	1 GET / requests / secondly

Figure 2.7: Editing plans in the *SLA Editor* tool

[5] https://hub.docker.com/u/isagroup
[6] https://hub.docker.com/r/isagroup/designer-studio

2.4.2 SLA Engine

The SLA4OAI specification outlines the *Basic SLA Management Service*[7] (BSMS) defining the interaction flows and the endpoints */check* and */metrics*. Our engine provides a concrete implementation which also includes a particular way to handle SLA saving/retrieving tasks. Specifically, *Monitor*[8] is an implementation of the *Metrics* BSMS service and *Supervisor*[9], of the *Check* service.

The *Monitor* service exposes a POST operation in the route */metrics* for gathering the metrics collected from different services. It can collect a set of basic metrics and send them to a data store for aggregation and later consumption. The metrics can be grouped in batches or sent one by one to fine-tune performance versus real-time SLA tracking.

The *Supervisor* service has a POST */check* endpoint for the verification of the current state of the SLA for a given operation in a certain scope. For each request, this service will evaluate the state of the SLA and will respond with a positive or negative response depending on whether a limitation has been overcome.

In addition, this service also implements (outside the scope of the BSMS) these additional endpoints: GET/POST */tenants*, GET/POST */slas* and PUT/DELETE *slas/<id>* for managing both users (tenants and accounts) and SLA4OAI documents themselves.

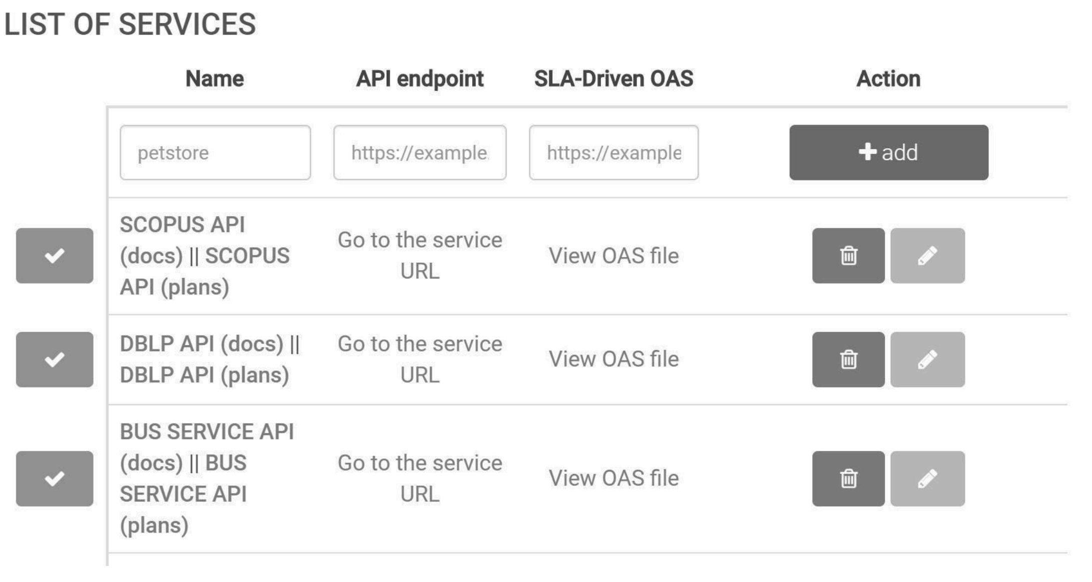
2.4.3 SLA-Driven API Gateway

We provide a *SLA-Driven API Gateway*[10], an implementation for registering APIs that will be automatically governed regarding the specified pricing.

Particularly, we provide an online preconfigured instance of an SLA-Driven API Gateway. As depicted in Figure §2.8, API providers are only required to enter: (i) The real endpoint of their API; (ii) A URL pointing to the SLA4OAI document. Once an API is registered, the SLA-Driven API Gateway exposes a public and SLA-regulated endpoint, as well as the */plans* endpoint for the provisioning portal. Clients who have selected a plan will get an API-key from the portal that will be a bearer token to consume the SLA-regulated API.

Figure §2.9 represents how the gateway works. First, requests will pass through the

[7]`https://sla4oai.specs.governify.io/operationalServices.html`
[8]`https://hub.docker.com/r/isagroup/governify-project-oai-monitor`
[9]`https://hub.docker.com/r/isagroup/governify-project-oai-supervisor`
[10]`https://hub.docker.com/r/isagroup/governify-project-oai-gateway`



LIST OF SERVICES

	Name	API endpoint	SLA-Driven OAS	Action
	petstore	https://example	https://example	+ add
✓	SCOPUS API (docs) || SCOPUS API (plans)	Go to the service URL	View OAS file	
✓	DBLP API (docs) || DBLP API (plans)	Go to the service URL	View OAS file	
✓	BUS SERVICE API (docs) || BUS SERVICE API (plans)	Go to the service URL	View OAS file	

Figure 2.8: Configuration UI of the SLA-Driven API Gateway

API Gateway until they are directed to the node that will serve it (step 1). Next, the API Gateway query the *SLA Check* API to determine if the request is authorized to develop the actual operation based on the appropriate SLA (step 2). Afterward, if it is authorized, the actual API is invoked and the response is returned (step 3). If it is not, a status code and a summary of the reason (as generated by the SLA check API) are returned (step 3). After the consumption ends (step 4), the metrics are sent to the *SLA Metrics* API (step 5), which is in charge of updating the status of the agreement with the new metrics introduced (step 6).

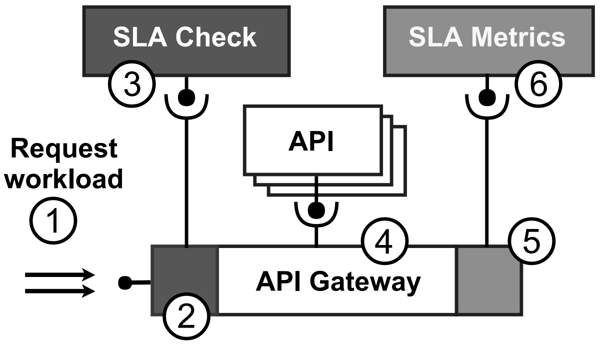


Figure 2.9: *Gateway* enforce defined in the SLA4OAI BSMS

Internally, this gateway uses an SLA Instrumentation Library in Node.js[11], which is a middleware (i.e., a filter that intercepts the HTTP requests and performs transformation

[11]https://www.npmjs.com/package/sla4oai-tools

if necessary) written for Express, the most used Node.js web application framework. This middleware intercepts all inbound/outbound traffic to perform the BSMS flow.

2.4.4 ELEcTRA

Providers, when building complex services, they often depend on third-party APIs. Before defining their own API pricing, it is crucial to know each pricing from the APIs you depend on (i.e., that is what we mean by *induced limitations*) Each API provider typically defines different limitations, therefore, performing a manual analysis of external APIs and their impact in a microservice architecture is a complex and tedious task.

ELeCTRA[12] is a tool to automate the analysis of *induced limitations* in an API, derived from its usage of external APIs. This tool takes the structural, conversational and SLA specifications of the API, generates a visual dependency graph and translates the problem into a constraint satisfaction problem (CSP) to obtain the effective limitations.

Figure §2.10 depicts a screenshot of ELeCTRA tool. The details of the implementation and a demonstration[13] are in [27].

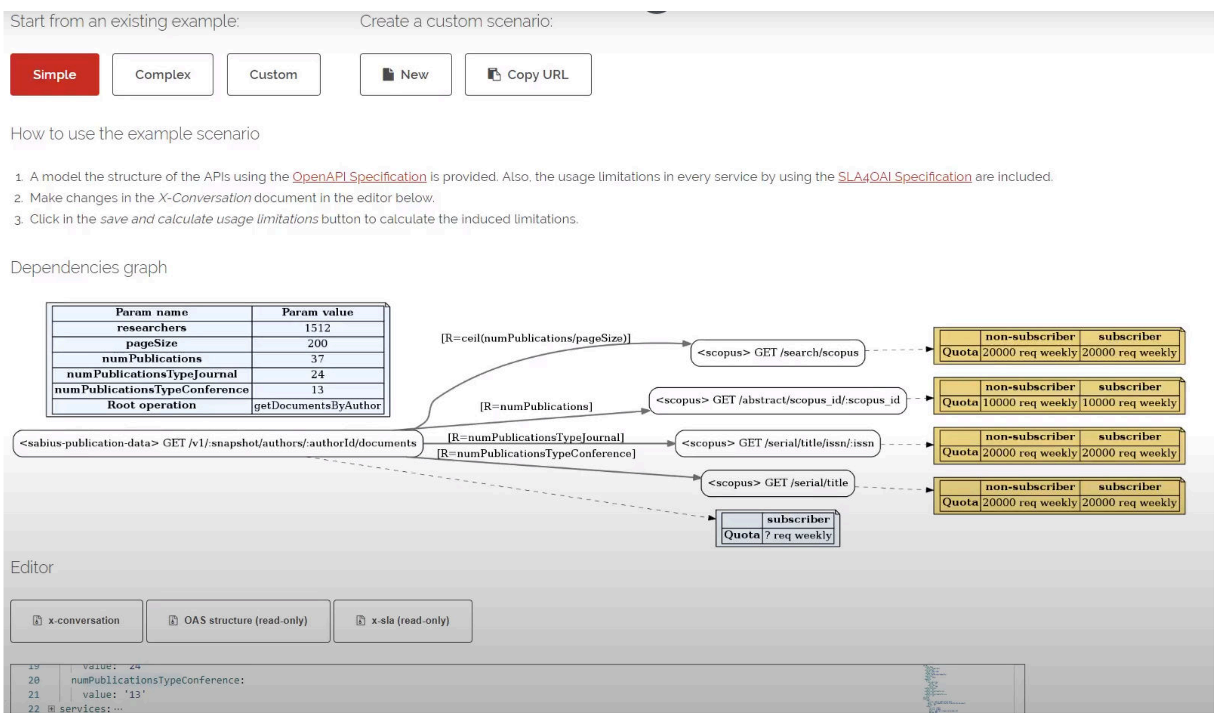
2.4.5 SLA4OAI Analyzer

The main operation regarding API limitations is *Validity* (cf. Section §2.3.2). 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*, an initial version of a publicly available command-line tool [28]. Once installed, given a SLA4OAI file, the command *sla4oai-analyzer -o <operation> -f <myFile.yaml>* will initiate the validity analysis for this file. In order to be integrated into the Governify framework, this tool is also available as an API [29].

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). Figure §2.12 depicts a consistency conflict detected by this tool, caused by a modeling mistake.

As illustrations of some outputs of the tool, Figure §2.11 shows a pricing with *syntax errors* and Figure §2.12 a *consistency conflict*.

[12] https://hub.docker.com/r/isagroup/governify-electra
[13] http://youtu.be/axbkDax1N9g



How to use the example scenario

1. A model the structure of the APIs using the OpenAPI Specification is provided. Also, the usage limitations in every service by using the SLA4OAI Specification are included.
2. Make changes in the *X-Conversation* document in the editor below.
3. Click in the *save and calculate usage limitations* button to calculate the induced limitations.

Dependencies graph

Param name	Param value
researchers	1512
pageSize	200
numPublications	37
numPublicationsTypeJournal	24
numPublicationsTypeConference	13
Root operation	getDocumentsByAuthor

<sabius-publication-data> GET /v1/:snapshot/authors/:authorId/documents
[R=ceil(numPublications/pageSize)]
<scopus> GET /search/scopus
[R=numPublications]
<scopus> GET /abstract/scopus_id/:scopus_id
[R=numPublicationsTypeJournal]
<scopus> GET /serial/title/issn/:issn
[R=numPublicationsTypeConference]
<scopus> GET /serial/title

	non-subscriber	subscriber
Quota	20000 req weekly	20000 req weekly

	non-subscriber	subscriber
Quota	10000 req weekly	10000 req weekly

	non-subscriber	subscriber
Quota	20000 req weekly	20000 req weekly

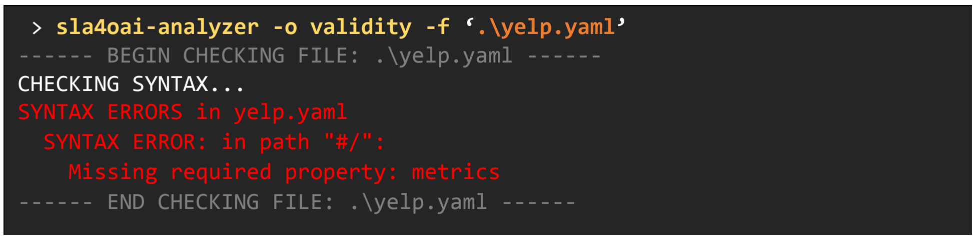
	non-subscriber	subscriber
Quota	20000 req weekly	20000 req weekly

	subscriber
Quota	? req weekly

Editor

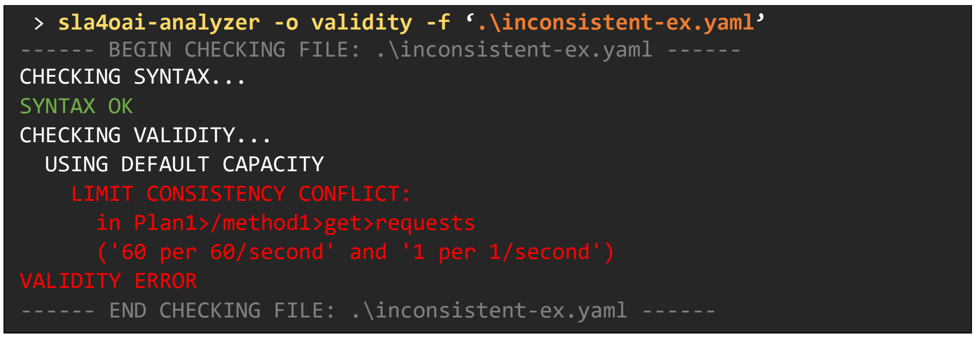
```
      value: '24'
    numPublicationsTypeConference:
      value: '13'
  services: ⋯
```

Figure 2.10: Sample calculation of induced limitations using ELEcTRA.



```
 > 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 2.11: 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 2.12: Tool running the validity operation with errors.

Part II

PUBLICATIONS

List of Publications

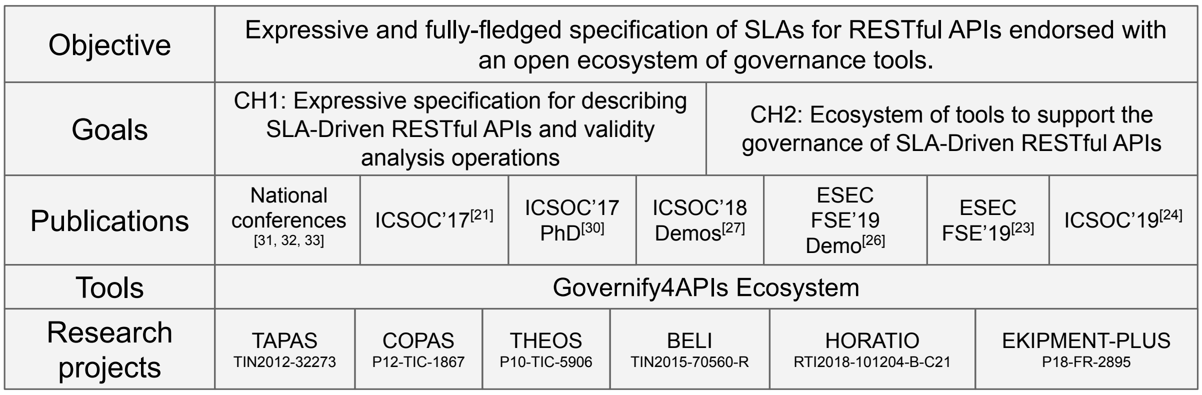
This chapter outlines the main contributions of this thesis and presents the list of publications. Precisely, Section §3.1 discusses the main contributions of the thesis with regard to the goals mentioned hereinbefore. Next, Section §3.2 holds the list of relevant publications that comprise this compendium. Finally, the full-text copy of the three core publications can be found in Chapter §4, Chapter §5 and Chapter §6 respectively.

3.1 Discussion of Results

This section discusses the main contributions of the thesis aimed at addressing the research goals described in Section §1.1.

3.1.1 General Overview

Figure §3.1 depicts an overview of the objectives, goals, related publications, developed tools and associated research projects.



Objective	Expressive and fully-fledged specification of SLAs for RESTful APIs endorsed with an open ecosystem of governance tools.						
Goals	CH1: Expressive specification for describing SLA-Driven RESTful APIs and validity analysis operations				CH2: Ecosystem of tools to support the governance of SLA-Driven RESTful APIs		
Publications	National conferences [31, 32, 33]	ICSOC'17[21]	ICSOC'17 PhD[30]	ICSOC'18 Demos[27]	ESEC FSE'19 Demo[26]	ESEC FSE'19[23]	ICSOC'19[24]
Tools	Governify4APIs Ecosystem						
Research projects	TAPAS TIN2012-32273	COPAS P12-TIC-1867	THEOS P10-TIC-5906	BELI TIN2015-70560-R	HORATIO RTI2018-101204-B-C21	EKIPMENT-PLUS P18-FR-2895	

Figure 3.1: Overview of this thesis.

The context and goals are extensively described at Section §1.1 and Section §1.3 respectively. Next, the summary of publications is presented in Section §3.1.2. Then, the tools are described in Section §2.4 and, finally, the associated research projects are outlined in Section §1.2.

3.1.2 Summary of the Contributions

Figure §3.2 depicts the general outline of the contributions of this dissertation. It highlights the main two contributions and their relationship with the publications as part of this compendium.

The high-level contribution is *Governify4APIs*, a set of new components or adaptations aimed at supporting the *SLA-Driven Governance of RESTful systems*. It has been built on the top of *Governify*, a framework to build SLA-Driven infrastructures. For building *Governify4APIs* two main contributions have been done: C1, *creation of a sufficiently expressive specification for the description of API pricings and the analysis of their validity* and C2, *implementation of an ecosystem of tools to support the SLA-Driven governance*

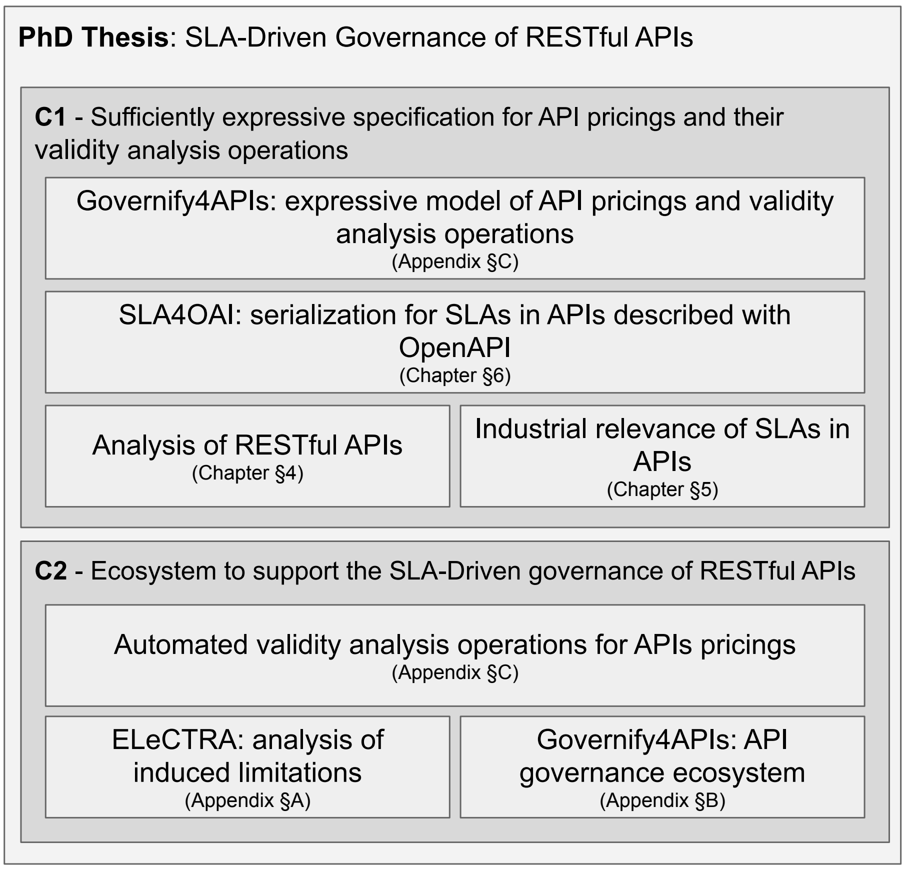

PhD Thesis: SLA-Driven Governance of RESTful APIs
C1 - Sufficiently expressive specification for API pricings and their validity analysis operations
Governify4APIs: expressive model of API pricings and validity analysis operations (Appendix §C)
SLA4OAI: serialization for SLAs in APIs described with OpenAPI (Chapter §6)
Analysis of RESTful APIs (Chapter §4)
Industrial relevance of SLAs in APIs (Chapter §5)
C2 - Ecosystem to support the SLA-Driven governance of RESTful APIs
Automated validity analysis operations for APIs pricings (Appendix §C)
ELeCTRA: analysis of induced limitations (Appendix §A)
Governify4APIs: API governance ecosystem (Appendix §B)

Figure 3.2: Overview of the contributions of this thesis.

of RESTful APIs.

C1: Creation of a sufficiently expressive specification for the description of API pricings and the analysis of their validity: the *Governify4APIs* Model

To this end, we first performed a systematic analysis of 69 RESTful APIs to thoroughly evaluate the characteristics of the API pricings and API limitations. This analysis paved the way to identify the requirements for the creation of an expressive model to describe SLAs in APIs. This contribution is presented in [21] (Chapter §4).

Second, we analyzed the landscape of SLAs in RESTful APIs in two different directions: i) Clarifying the SLA-driven API development lifecycle: its activities and participants; 2) Developing a catalog of relevant concepts and ulterior prioritization based on different perspectives from both Industry and Academia. As a main result, we presented a scored list of concepts that paves the way to establish a concrete roadmap for a standard industry-aligned specification to describe SLAs in APIs. This contribution is presented in [23] (Chapter §5).

Next, we presented SLA4OAI, pioneering in extending the OpenAPI Specification not only allowing the specification of SLAs but also supporting some stages of the SLA-Driven API lifecycle with an open-source ecosystem. Finally, we validated our proposal having modeled 5488 limitations in 148 plans of 35 real-world APIs. This contribution is presented in [24] (Chapter §6).

Finally, we continue evolving the SLA4OAI model, which is just a simplified serialization intended to be compatible with the OpenAPI Specification, into the *Governify4APIs* model. We have incorporated some missing features after having analyzed with rigor the concept of an API pricing and its properties. We also paved the way for a catalog of analysis operations, starting from defining what is the definition of a API pricing validity. This contribution is, at the time of writing, under active elaboration. However, an initial version is available in this thesis as an appendix in Appendix §C.

C2: Implementation of an ecosystem of tools to support the SLA-Driven governance of RESTful APIs: the *Governify4APIs* Ecosystem

First, we developed ELeCTRA, a simple tool to automate the analysis of induced limitations in an API, derived from its usage of external APIs. This tool takes the structural, conversational and SLA specifications of the API, generates a visual dependency graph and translates the problem into a constraint satisfaction optimization problem (CSOP) to obtain the optimal usage limitations. This contribution is presented in [27] (Appendix §A).

Next, we introduced *Governify4APIs*, an ecosystem of tools aimed at the SLA-Driven *governance* of RESTful APIs. Namely, an SLA Editor, an SLA Engine and an SLA Instrumentation Library. We also present a fully operational SLA-Driven API Gateway built on top of our ecosystem of tools. To evaluate our proposal, we used three sources for gathering validation feedback: industry, teaching and research. This is presented in [26] (Appendix §B).

Finally, we evolved *Governify4APIs* by enhancing the tools with new analysis operations regarding the validity of an API pricing, the calculation of the effective limitation in a period and the compliance of a certain API pricing with the given user needs. As stated above, this contribution is under active elaboration, but an initial version is available in this thesis (Appendix §C).

Regarding the evaluation of this contribution, we seek three main contexts: industrial, educational and research.

Concerning the industrial evaluation, some OpenAPI Initiative members have expressed their interest in SLA4OAI, the SLA modeling proposal, and in promoting a working group for evolving and extending it. We collaborated with people from Google, Paypal, AsyncAPI Initiative and Metadev for analyzing, starting from SLA4OAI, the status of SLAs and limitations in the industry. Furthermore, in spite of the fact the SLA4OAI extension and tools have not been widely announced nor promoted, we have disclosed the tooling ecosystem into the main public Node.js artifact repository (i.e., NPM) and this platform provides a set of analytics of usage since their publishing. Specifically, based on its data we observe that *SLA Instrumentation Library* has been downloaded and installed more than 600 times while the *SLA Engine* was downloaded more installed than 2800 times[1].

Regarding the use of *Governify4APIs* in teaching, it has been extensively used in, at least, two undergraduate service-oriented related subjects. As students were required to create their own APIs[2], they also had to set the rate and quota limitations using *Governify4APIs*. Whereas we do not have any specific usage report, we collected useful information, issues and bugs derived from running in production.

As for the research context, we are validating our proposal (language and tools) in a national research network. Several members are exposing their research results by creating an API and applying limitations using *Governify4APIs*. Then, all these artifacts are being deployed in a central publicly available catalog[3].

3.2 Main Papers

- A. Gámez-Díaz, P. Fernández, A. Ruiz-Cortés. **An Analysis of RESTful APIs Offerings in the Industry**. *In proceedings of the 15th International Conference on Service-Oriented Computing (ICSOC 2017)* [21]. GGS class 2 (A-).

- A. Gámez-Díaz, P. Fernández, A. Ruiz-Cortés, P. Molina, N. Kolekar, P. Bhogill, M. Mohaan, F. Méndez. **The Role of Limitations and SLAs in the API Industry**. *In 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+).

[1]https://npm-stat.com/charts.html?package=sla4oai-tools
[2]https://github.com/gti-sos
[3]https://www.isa.us.es/rcis. Accessed Oct 2020.

- A. Gámez-Díaz, P. Fernández, A. Ruiz-Cortés. **Automating SLA-Driven API development with SLA4OAI**. *In proceedings of the 17th International Conference on Service-Oriented Computing (ICSOC 2019)* [24]. GGS class 2 (A-).

3.3 Supporting Papers

International Conferences

- A. Gámez-Díaz, P. Fernández, A. Ruiz-Cortés. **Governify for APIs: SLA-Driven ecosystem for API governance**. *In proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)* [26]. GGS class 1 (A+).

- A. Gámez-Díaz, P. Fernández, C. Pautasso, A. Ivanchikj, A. Ruiz-Cortés. **ELeCTRA: Induced Usage Limitations Calculation in RESTful APIs**. *In proceedings of the 16th International Conference on Service-Oriented Computing - Workshops (ICSOC 2018)* [27].

- A. Gámez-Díaz, P. Fernández, A. Ruiz-Cortés. **SLA-Driven Governance for RESTful Systems**. *In proceedings of the 15th International Conference on Service-Oriented Computing - Workshops (ICSOC 2017)* [30].

3.3.1 National Conferences

- A. Gámez-Díaz, P. Fernández, A. Ruiz-Cortés. **Fostering SLA-Driven API Specifications**. *Actas de las XIV Jornadas de Ingeniería de Ciencia e Ingeniería de Servicios (JCIS 2018)*[31].

- A. Gámez-Díaz, P. Fernández, A. Ruiz-Cortés. **Towards SLA modeling for RESTful APIs**. *Actas de las XIII Jornadas de Ingeniería de Ciencia e Ingeniería de Servicios (JCIS 2017)* [32].

- A. Gámez-Díaz, P. Fernández, A. Ruiz-Cortés. **Towards SLA-Driven API Gateways**. *Actas de las XI Jornadas de Ingeniería de Ciencia e Ingeniería de Servicios (JCIS 2015)* [33].

3.3.2 Journal Papers Awaiting Response

- A. Gámez-Díaz, R. Fresno, P. Fernández, A. Ruiz-Cortés. **A Pricing Modeling Framework for RESTful APIs Regulated with Limitations**. *IEEE Transactions on Services Computing.* (Appendix §C).

3.3.3 Other Publications

- J. Troya, J. A. Parejo, S. Segura, A. Gámez-Díaz, A. E. Márquez-Chamorro and A. del-Río-Ortega, **Flipping Laboratory Sessions in a Computer Science Course: An Experience Report**. *IEEE Transactions on Education* (Sept. 2020) [34].
- J. A. Parejo, J. Troya, S. Segura, A. del-Río-Ortega, A. Gámez-Díaz and A. E. Márquez-Chamorro, **Flipping Laboratory Sessions: An Experience in Computer Science**. *IEEE Revista Iberoamericana de Tecnologías del Aprendizaje* (Aug. 2020) [35].
- J. Troya, S. Segura, J. A. Parejo, A. del-Río Ortega, A. Gámez-Díaz and A. E. Márquez-Chamorro. **Invirtiendo las clases de laboratorio en Ingeniería Informática: Un enfoque ágil**. *Actas de la XXV Edición de las Jornadas sobre la Enseñanza Universitaria de la Informática (JENUI 2019)* [36].

An Analysis of RESTful APIs Offerings in the Industry

Published in the 15th International Conference on Service-Oriented Computing (ICSOC), Málaga, Spain. November 2017.

***Authors**: Antonio Gámez Díaz, Pablo Fernández Montes, Antonio Ruiz Cortés.*
***DOI**: 10.1007/978-3-319-69035-3_43.*
***Rating**: GGS class 2 (A-).*

An Analysis of RESTful APIs Offerings in the Industry

Antonio Gamez-Diaz(✉), Pablo Fernandez, and Antonio Ruiz-Cortes

Universidad de Sevilla, Seville, Spain
{agamez2,pablofm,aruiz}@us.es

Abstract. As distribution models of information systems are moving to *XaaS* paradigms, microservices architectures are rapidly emerging, having the RESTful principles as the API model of choice. In this context, the term of *API Economy* is being used to describe the increasing movement of the industries in order to take advantage of exposing their APIs as part of their service offering and expand its business model.

Currently, the industry is adopting standard specifications such as OpenAPI to model the APIs in a standard way following the RESTful principles; this shift has supported the proliferation of API execution platforms (*API Gateways*) that allow the XaaS to optimize their costs. However, from a business point of view, modeling offering plans of those APIs is mainly done ad-hoc (or in a platform-dependent way) since no standard model has been proposed. This lack of standardization hinders the creation of API governance tools in order to provide and automate the management of business models in the XaaS industry.

This work presents a systematic analysis of 69 XaaS in the industry that offer RESTful APIs as part of their business model. Specifically, we review in detail the plans that are part of the XaaS offerings that could be used as a first step to identify the requirements for the creation of an expressive governance model of realistic RESTful APIs. Additionally, we provide an open dataset in order to enable further analysis in this research line.

1 Introduction

In the last decade, distribution models of information systems are evolving into *XaaS* [10] paradigms where customers no longer need to buy a perpetual license, host the software or maintain the infrastructure [5]. As part of this trend, the microservices architectures are rapidly emerging as they provide a flexible evolution model [7]. In particular, this architectural model proposes a division of the information system into a set of small services deployed independently which communicate each other using Web APIs that adhere typically to REST principles [6].

This work has been partially supported by the European Commission (FEDER), the Spanish and the Andalusian R&D&I programs (grants TIN2015-70560-R (BELI) and P12–TIC-1867 (COPAS)) and the FPU scholarship program, granted by the Spanish Ministry of Education, Culture and Sports (FPU15/02980).

© Springer International Publishing AG 2017
M. Maximilien et al. (Eds.): ICSOC 2017, LNCS 10601, pp. 589–604, 2017.
https://doi.org/10.1007/978-3-319-69035-3_43

In this context, the term of *API Economy* is being increasingly used to describe the movement of the industries to share their internal business assets as APIs [21] not only across internal organizational units but also to external third parties; in doing so, this trend has the potential of unlocking additional business value through the creation of new assets [3]. In fact, we can find a number of XaaS examples in the industry that are deployed solely as APIs (such as Meaningcloud[1], Flightstats[2] or Twilio[3]).

In order to be competitive in this such a growing market of APIs, at least two key aspects can be identified: (i) *ease of use* for its potential developers; (ii) a flexible usage *plan* that fits their customer's demands.

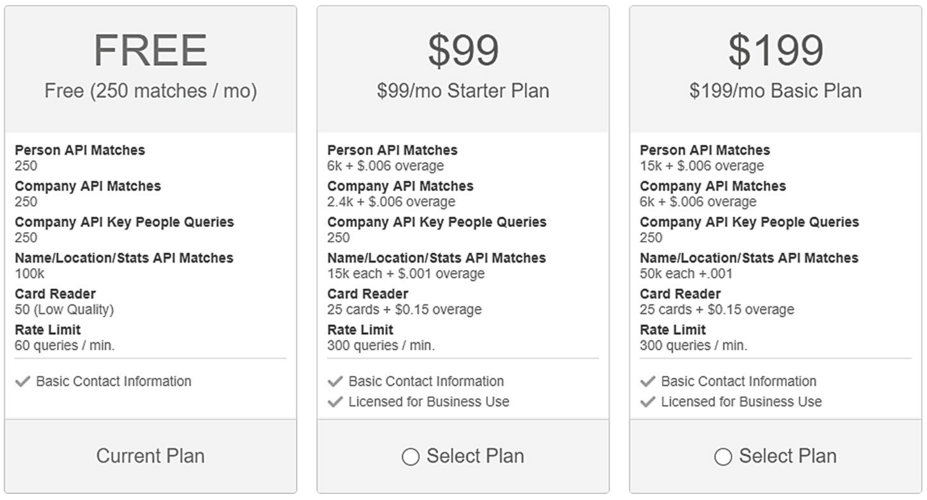
Regarding the *ease of use* perspective, third party developers need to understand how to use the exposed APIs so it becomes necessary to provide a good training material but, unfortunately, several API providers do not often write a good documentation of their products [8]. Alternatively, in the last year, we found the promising proposal of the *Open API Initiative*[4] (OAI) whose aim is to support the creation, evolution and promotion of a vendor neutral description format for RESTful APIs and that is currently being backed by a growing number of leading industrial stakeholders.

Conversely, from the usage *plans* perspective, to the best of our knowledge, do not exists a widely accepted model to describe usage plans including elements such as cost, functionality restrictions or limits. In this context, we can find some example of API management platforms in the industry (commonly known as *API Gateways*), which have tried to address the problem of usage plans modeling but they are typically constrained by their platform architecture and no interoperable usage plan specification is provided. For instance, Mashape presents a limited governance ecosystem, since it only allows users to define quotas and not rates.

Figure 1 illustrates a real plan extracted from *FullContact*[5], a real-world SaaS offering which includes an API that manages and organizes contacts in a collaborative way, it also matches emails addresses and tries to find as much information as available on the Internet to complete the profiles. Note that in this work, we focus on XaaS offering a RESTful API in order to access either fully or partially to the functionality they offer. In traditional XaaS, these actions are accessed using the graphic user interface.

This example is composed of three plans, one of them is free whereas the remaining are paid. Focusing on paid ones, they have a fixed *price* that is monthly *billed*. Regarding the *limits*, for each resource, a *quota* is being applied; for instance, in the starter plan, only 6000 matches over *Person* are available. Nevertheless, an *overage* is defined, that is, it is possible to overcome the limit by paying a certain amount of money; in this case, \$0.006 per each request. Regardless of the accessed resources, a common *rate* of 300 queries per minute is

[1] https://www.meaningcloud.com/products/pricing.
[2] https://developer.flightstats.com/getting-started/pricing.
[3] https://www.twilio.com/sms/pricing.
[4] https://www.openapis.org/.
[5] https://www.fullcontact.com/developer/.



FREE	$99	$199
Free (250 matches / mo)	$99/mo Starter Plan	$199/mo Basic Plan
Person API Matches 250	**Person API Matches** 6k + $.006 overage	**Person API Matches** 15k + $.006 overage
Company API Matches 250	**Company API Matches** 2.4k + $.006 overage	**Company API Matches** 6k + $.006 overage
Company API Key People Queries 250	**Company API Key People Queries** 250	**Company API Key People Queries** 250
Name/Location/Stats API Matches 100k	**Name/Location/Stats API Matches** 15k each + $.001 overage	**Name/Location/Stats API Matches** 50k each +.001
Card Reader 50 (Low Quality)	**Card Reader** 25 cards + $0.15 overage	**Card Reader** 25 cards + $0.15 overage
Rate Limit 60 queries / min.	**Rate Limit** 300 queries / min.	**Rate Limit** 300 queries / min.
✓ Basic Contact Information	✓ Basic Contact Information ✓ Licensed for Business Use	✓ Basic Contact Information ✓ Licensed for Business Use
Current Plan	○ Select Plan	○ Select Plan

Fig. 1. Example of an API plan.

being applied. In this plan, there are not any *functionality limitation*, even the free plan has the same functionality that paid ones have. In this case, the *free tier* is regulated by *limits* such as *quotas* and *rates*.

The main aim of this paper is to develop the first step towards an expressive, platform neutral, usage plan model that could be used to create open API governance tools. Specifically, this work presents a systematic analysis of the usage plans identified in a wide spectrum of real-world APIs; in doing so, the main contributions of this paper are: (i) present a systematic method to analyze XaaS offerings in the industry including RESTful APIs; (ii) undertake a comparative analysis of 69 industrial APIs selected from two widely used API directories, identifying the common trends related to the modeling of usage plans; (iii) provide an open dataset that can be used to replicate our analysis and to be extended in further researches.

This paper is organized as follows: Sect. 2 shows the methodology that we use in our study as well as the characteristics we analyze. Next, in Sect. 3, we discuss the results of the analysis. In addition, Sect. 4 shows the existing work related to this paper. Finally, Sect. 5 shows some remarks and conclusions.

2 Research Method and Conduct

The study[6] presented herein was entirely conducted during the 2017 first quarter and it is a primary study in which we analyze real-world APIs. Whereas primary research data are collected from, for instance, research subjects or experiments, secondary studies involve the synthesis of existing research. Specifically, our work

[6] Data used in this study is publicly available at https://goo.gl/gQPDxz.

is based on the guidelines provided by Kitchenham and Charters in [12], adapting these guides about how to carry out secondary studies to a primary study. We consider that using these guidelines helps to systematize the research we are doing since they define a workflow directly applicable to primary research and give recommendations with the aim of avoiding undesired bias.

In our work, we systematically analyze a set of characteristics in real-world APIs following the steps depicted in Fig. 2.

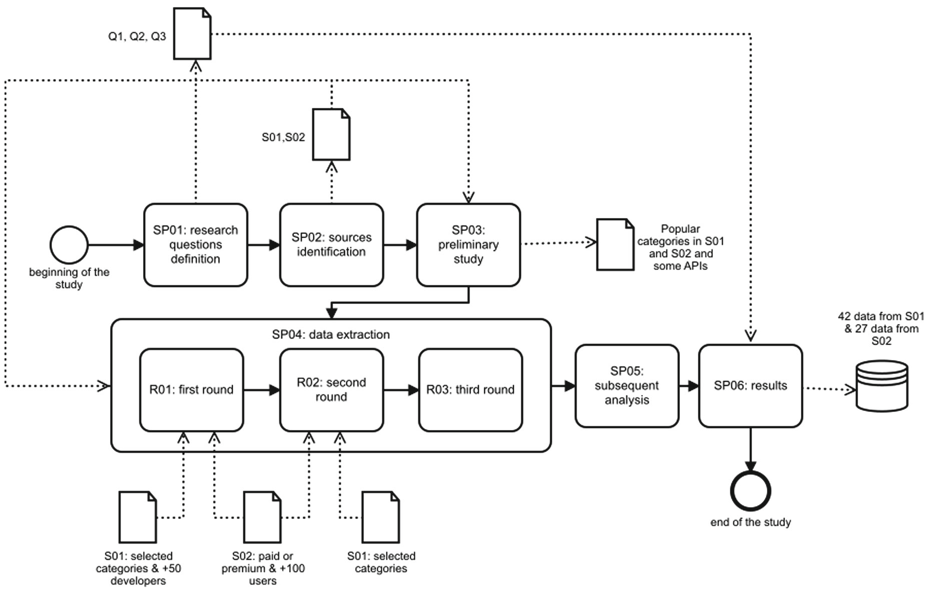

Q1, Q2, Q3
S01,S02
beginning of the study
SP01: research questions definition
SP02: sources identification
SP03: preliminary study
Popular categories in S01 and S02 and some APIs
SP04: data extraction
R01: first round
R02: second round
R03: third round
SP05: subsequent analysis
SP06: results
42 data from S01 & 27 data from S02
end of the study
S01: selected categories & +50 developers
S02: paid or premium & +100 users
S01: selected categories

Fig. 2. BPMN representation of the research process.

- **SP01-Research questions definition.** We start our systematic analysis with a series of motivating questions which will drive the investigation. We consider that these questions can pave the way for future research activities. Specifically, we define the following questions:
  - **RQ01.** What are the most common business models in the context of XaaS that offer a RESTful API?
  - **RQ02.** How are the plans, in terms of the characteristics that they have, used in XaaS that provide a RESTful API?
  - **RQ03.** Which regulations do XaaS offerings state over the RESTful APIs?
- **SP02-Sources identification.** Based on the literature and the analysis of the industry that we have conducted, 10 API repositories were collected. Nevertheless, we have considered those ones which *included more than 5000 APIs* and whose *last update date was in the year 2017*, remaining 2 valid sources: **S01-ProgrammableWeb**[7]: with 17511 APIs distributed in 478 categories

[7] https://www.programmableweb.com.

and **S02-Mashape**[8] with 7500 APIs distributed in 28 categories. Note that Mashape directory has been recently moved to RapidAPI[9] catalog, so subsequent analysis should be made over RapidAPI rather than Mashape.

– **SP03-Preliminary study.** After a preliminary examination of API directories S01 and S02, the more popular categories in each one were identified. We did a percentile study over the categories and the number of users in each one. Particularly, we fixed P_{97} for S01 and P_{50} for S02. Additionally, we included some handpicked APIs, looking for these ones with complex plans.
– **SP04-Data extraction.** We designed two different forms since the quality criteria have to be used to identify inclusion/exclusion criteria, according to the Kitchenham guidelines [12]. The first one[10] tried to identify basic information about the analyzed API as well as information regarding the plans. The second one[11] went in depth into the overage and both functionality and quota/rate limitations, including the API characteristics showed in Sect. 2.1. 30 students were given S01 and S02 API directories so that they chose two XaaS offerings following the eligibility criteria. They collected manually the required information in a session guided by the authors and they filled out the forms. In order to have a broader vision of the APIs offered in the industry, we defined an incremental process composed of three rounds. We started from defining strict eligibility criteria and the number of developers that the API has. Then we relaxed some criterion so that a new set of APIs was included.

  In the first round (R01) we limited the APIs selected from S01, considering only a certain set of categories[12], according to its popularity (see SP03). In addition, we set a threshold of 50 registered developers in S01 and limited the APIs selected from S02 having, at least, 100 users and being in categories either *paid* or *premium*.

  In the second round (R02) we were informed by some students about they did not found any API according to the established search restrictions. At this moment, we determined to relax the criteria in S01, removing the 50 developers' threshold. After finishing this round, we have collected 62 APIs.

  In the third round (R03) we started the guided session in class to fill out the form. Nevertheless, we noticed that there was a number of APIs without a clear plan, and students found quite difficult to find all the information that we asked for. At this point, we decided to start a new API gathering session with the help of the instructors. After finishing this round, we harvested extra 28 APIs.

[8] https://www.mashape.com.
[9] https://rapidapi.com.
[10] Available at https://goo.gl/rqwvH7.
[11] Available at https://goo.gl/sbzXEh.
[12] Mapping, social, e-commerce, mobile, search, tools, messaging, API, video, financial, cloud, payments, enterprise, analytics, data.

– **SP05-Subsequent analysis.** We did a subsequent analysis in two different steps: (i) manual data validation and classification: giving a result a set of 69 analyzed XaaS offerings with more than one plan. We detected some inconsistencies in some points that were manually reviewed and corrected; (ii) ulterior results classification: in which we separated the data gathered regarding the source, obtaining 42 APIs from S01 and 27 from S02.

2.1 Analyzed Attributes

We developed a comparative framework based on 60 attributes grouped in 7 areas illustrating the traceability between the research questions and the gathered characteristics. Following, we describe each group of attributes.

General information (see Table 1). We collected information about the API itself, including the *name* (**GI01**) and the *source* (**GI02**) where these APIs was selected from (i.e. *Mashape* or *ProgrammableWeb*); and the *plans URL* (**GI03**).

API characterization (see Table 1). We distinguished two attributes, *API type* (**AC01**) and *API maturity level* (**AC02**), in terms of giving a more precise classification of APIs. Specifically, we propose a classification of four types for the *API type*: **T01** if the XaaS offering does not provide any API at all; **T02** when the XaaS offering does provide a non-RESTful API; **T03** if the XaaS offering does provide as part of its offer a RESTful API, (e.g., a SaaS which allows customers to access their data in a RESTful way, but the primary access way is a GUI); and **T04** if the XaaS offering is, actually, a RESTful API (e.g., an API to send emails or SMS). For *API type* T03 or T04, we identify a set of three *API maturity levels*: **ML01** if the API does not define any limitations nor explicit Service Level Agreement (SLA); **ML02** when the API defines limitations and/or explicit SLAs but they are not in the plans (i.e., the limitations are applied regardless of the selected plan); and **ML03** if the API defines limitations and/or explicit SLAs depending on the selected plan.

Pricing (see Table 1). We identify economic information of the API pricing including the *currency* (**P01**) in which clients are billed, the *billing cycle* (**P02**) and a set of statistics of the *plan cost* (**P03**, **P04**, **P05**).

Business model (see Table 1). We consider the main *primary business model* (**BM05**) in the API, inspired by a number of works in the literature, as shown in Sect. 4. Namely: *free* (**FR**), when no payment is needed; *pay-as-you-go simple* (**PG-S**), when you pay just for the usage you do (e.g., you pay per each request made); *pay-as-you-go with intervals* (**PG-I**), when the payment for each unit depends on the usage volume (e.g., the first 1 K request cost $0.1 each, but the subsequent $0.05 each); *tiered with fixed prices* (**TO1**), when each plan has a non-variable price; *tiered with overage* (**TO2**), when existing plans with a certain price and limitations you can overcome the limits by paying an extra amount. We also gathered the *number of plans* (**BM06**) and discover the *existence of discounts per annual upfronts* (**BM01**), the *existence of customs plans* (**BM03**), the *main limitation of the free plan* (**BM04**); or the *existence of a free plan* (**BM02**).

Table 1. First set of API analyzed attributes.

General information	RQ01	RQ02	RQ03
GI01-Name of the API			
GI02-Source	✓	✓	✓
GI03-Plans URL			
API characterization			
AC01-API type	✓	✓	
AC02-API maturity	✓	✓	
Pricing			
P01-Currency used	✓	✓	
P02-Billing cycle	✓	✓	
P03/P04/P05-Plan cost(max/min/avg)	✓	✓	
Business model			
BM01-Existence of discounts per annual upfront	✓	✓	
BM02-Existence of a free plan	✓	✓	
BM03-Existence of custom plans	✓	✓	
BM04-Main limitation of the free plan	✓	✓	
BM05-Main business model	✓	✓	
BM06-Number of plans	✓	✓	

Overage (see Table 2). We define *overage* as the extra cost in which a customer incurs when a certain limitation or set of limitations is exceeded (**O01**). The *overage scope* (**O02**) depends over what item the limitation is made (e.g., requests, the number of resources, etc.). Moreover, we collected data about the *overage cost* (maximum **-O09-**, minimum **-O10-** and average **-O11-** across the different plans) and the *overage limit* (maximum **-O03-**, minimum **-O07-** and average **-O08-** across the different plans), i.e., the amount of scoped data allowed per each overage payment. Furthermore, we consider the *existence of an overage in every paid plan* (**O04**) and we analyze whether in the same paid plan *all the resources have an overage* (**O05**) and *all the resources have the same overage value* (**O06**).

Functionality limitations (see Table 2). We identify the limitations over the API functionality (**FL01**) and study the granularity: *resource access granularity* (**FL02**), if the limitation is applied to the resource endpoint (e.g. it is not possible to access some parts of the resource in some plans); *HTTP method granularity* (**FL03**), if the limitation is applied to a certain HTTP verb (e.g., it is not possible to make a POST in some plans) *request body granularity* (**FL04**), when the limitation is based on the specific payload sent to an endpoint. Furthermore, we identify the *existence of a functionality limitation in every paid plan* (**FL05**) and we analyze whether in the same paid plan *all the resources have a*

Table 2. Second set of API analyzed attributes.

Overage	RQ01	RQ02	RQ03
O01-Existence of an overage	✓	✓	✓
O02-Overage scope		✓	✓
O04-Existence of an overage in every paid plan		✓	✓
O05-In the same paid plan all the res. have an overage		✓	✓
O06-In the same paid plan all the res. have the same overage value		✓	✓
O03/O07/O08-Overage limit value(max/min/avg)		✓	✓
O09/O10/O11-Overage cost (max/min/avg)		✓	✓
Functionality limitations			
FL01-Existence of functionality limitations	✓	✓	✓
FL02-Limitation granularity: resource access			✓
FL03-Limitation granularity: HTTP methods			✓
FL04-Limitation granularity: request body			✓
FL05-Existence of functionality limitations in every paid plan			✓
FL06-In different paid plans each one has the same func. lim.			✓
FL07-In the same paid plan all the resources have a func. lim.			✓

functionality limitation (**FL06**) and *all the resources have the same functionality limitations* (**FL07**).

Quotas/Rates (see Table 3). We analyze two time-based limitations in the API, commonly known as quotas and rates. The main difference is the sliding window that rates have: whereas with quotas it is possible to define limits such as *up to 1000 requests per day*, with rates it is possible to express limits with a relative period of time, such as *up to 100 requests in the last minute*. Specifically, we identify the scope of these limitations: (i) *requests scope* (**Q02/R02**), (ii) *storage scope* (**Q03/R03**); (iii) *resource scope* (**Q04/R04**); (iv) *transaction size scope* (**Q05/R05**) and *other scopes* not explicitly mentioned (**Q06/R06**). Moreover, we collected the value of the limitation (maximum **-Q12/R12-**, minimum **-Q13/R13-** and average **-Q14/R14-** across the plans) and periodicity. Furthermore, we consider the *existence of a functionality limitation in every paid plan* (**Q07/R07**), we analyze if *in different plans each one has the same quotas/rates.* (**Q08/R08**), whether in the same paid plan *all the resources have a quotas/rates* (**Q09/R09**) and, finally, if *all the resources have the same quota/rate value.* (**Q10/R10**).

Table 3. Third set of API analyzed attributes.

Quotas and rates	RQ01	RQ02	RQ03
Q01/R01-Existence of quotas/rates	✓	✓	✓
Q02/R02-Quotas/Rates over requests			✓
Q03/R03-Quotas/Rates over storage			✓
Q04/R04-Quotas/Rates over resources			✓
Q05/R05-Quotas/Rates over transaction size			✓
Q06/R06-Quotas/Rates over another scope			✓
Q07/R07-Quotas/Rates in every paid plan			✓
Q08/R08-Quota/Rates in all resources of different plans			✓
Q09/R09-Quota/Rates in all resources of the same plan			✓
Q10/R10-Same quota/rate value for a given plan & resource			✓
Q11/R11-Quota/rate periodicity			✓
Q12/R12/Q13/R13/Q14/R14-Quota/Rate value (max/min/avg)			✓

3 SP06-Results

In this section, we present the results of the study grouped in three different blocks: (i) attributes regarding the business model and pricing; (ii) aspects related to limitations and overage application; (iii) quotas and rates limitations. Due to the fact that there exist notable differences between the APIs and their governance models, we decided to perform a separate analysis regarding the source of the API: Mashape and ProgrammableWeb.

In Fig. 3 we observe that most of the APIs analyzed are, indeed, the XaaS offering (AC01). In the case of Mashape, all the APIs are T04. Regarding the maturity (AC02), in both cases, we observe that the defined limitations depend on the plan that the client selects. Note we have established a search protocol that picked primarily popular APIs from popular categories, a fact that explains this polarization in AC01 and AC02. A small number of APIs offer a discount per an anticipated payment or upfront (BM01), but the vast majority define a free tier with some specific limitations (B02). In addition, it is frequent to have a way to define custom plans by talking directly to the company (BM03). Regarding the business models (BM05), it is very likely for APIs from Mashape to have a tiered plan with an overage, in contrast to the ones from ProgrammableWeb, in which is common to have a tiered plan with fixed prices. It is remarkable that the more common billing cycle (P02) is *monthly* and the number of plans (BM06) oscillates between two and four.

Figure 4 depicts the most interesting attribute analysis about how limitations are being applied in APIs. First, we observe that a high number limits the operations, rather than functionality or time (BM04). Secondly, from the providers

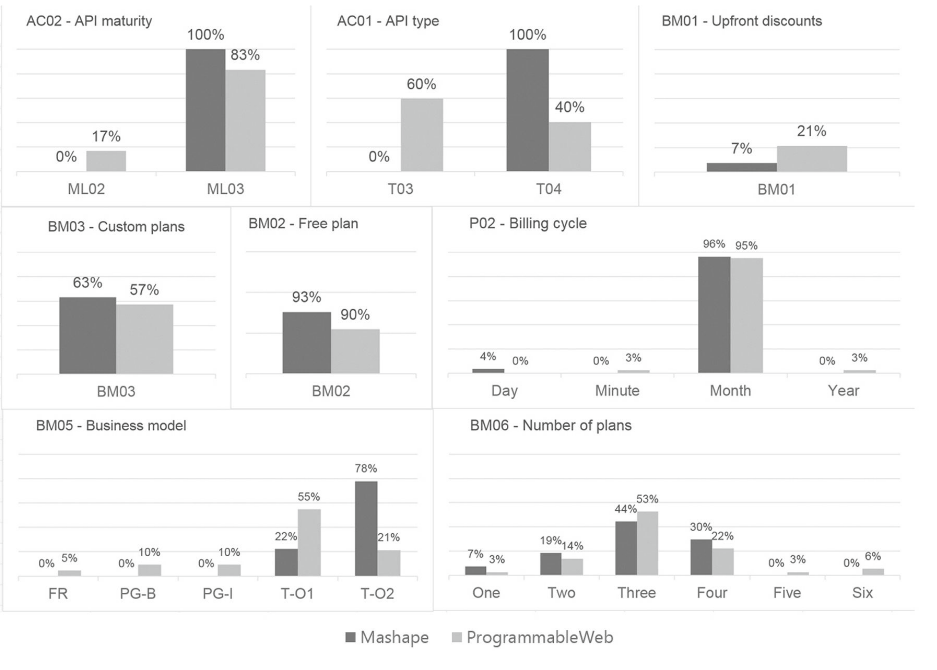


Fig. 3. Business model and pricing analysis.

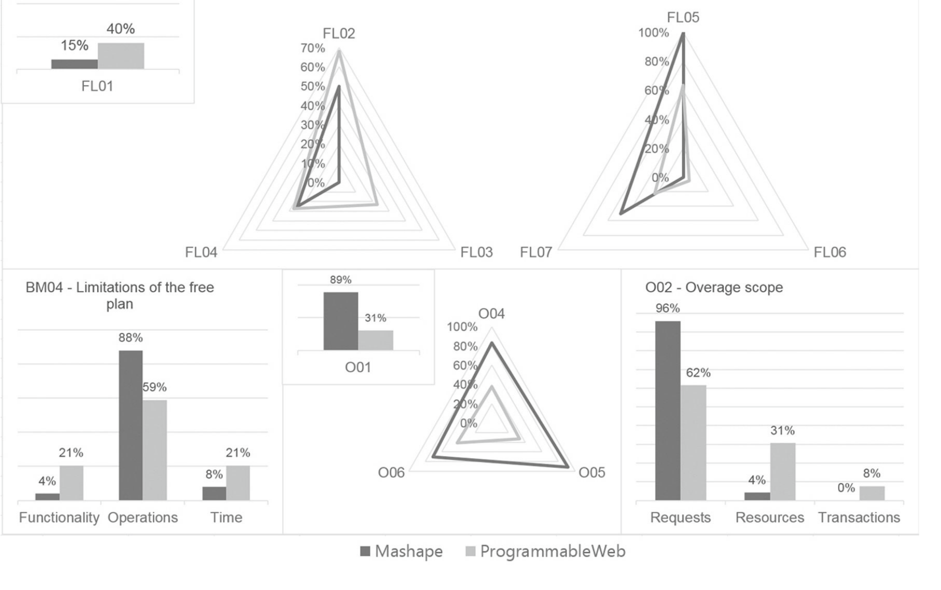


Fig. 4. Limitations and overage analysis.

that apply an overage if a certain limit is reached (O01), it is frequent that all the resources have an overage (O05), but it has not to be the same (O06). The most common scope (O02) is *requests*. On the other hand, some APIs apply limitations over the functionality (FL01), being more frequent in the APIs chosen from ProgrammableWeb. Most of the limitations are applied to the resource itself (FL2). Furthermore, functionality limitations use to be present in every plan (FL05), but they neither are the same across the plans (FL06) nor have the same values (FL07).

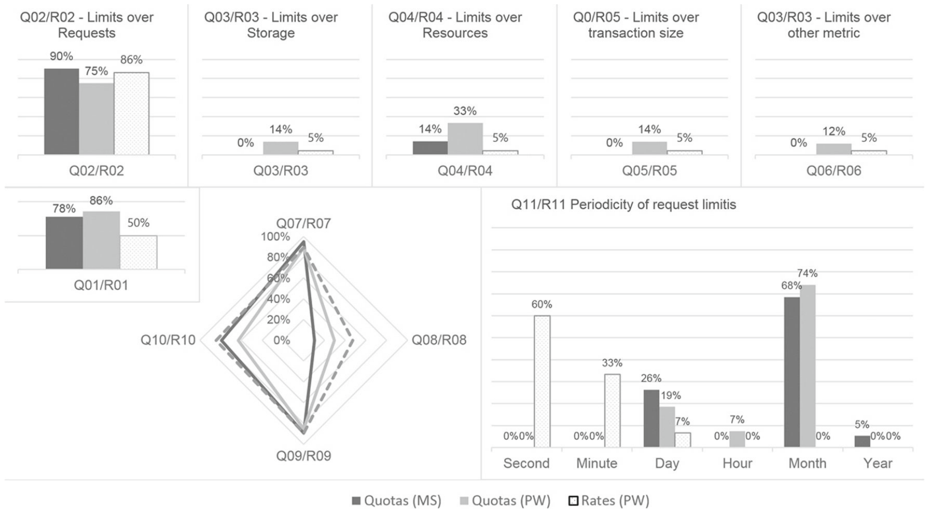


Fig. 5. Quotas and Rates analysis.

In Fig. 5 we observe some charts regarding the limitations using quotas and rates. Whereas both quotas and rates are very frequent (Q01/R01), we have noticed that Mashape does not allow users to define rates. Quotas are usually defined using *monthly* periods, whereas rates are more common to be *secondly* or *minutely* (Q11/R11). Furthermore, most of quotas and rates are defined over requests (Q02/R02), rather than over resources (Q04/R04) or storage (Q03/R03). It is also remarkable that most of quotas and rates have the same values within a plan (Q10/R10), but in different plans they usually have different values (Q08/R08).

Each of these attributes paves the way to give an answer to the stated research questions. Specifically, (i) regarding the most common business models (**RQ01**), as depicted in Fig. 3, BM05 attribute points out that the more common business models are the tiered ones with or without overage; (ii) regarding the plans (**RQ02**), as shown in Fig. 3, most APIs define between two or four plans, with a monthly billing cycle; (iii) regarding the regulations (**RQ03**), as illustrated in Figs. 4 and 5 most XaaS providers apply limitations in somehow. They limit the free tier by restricting the operations allowed and, for paid plans, they define both quotas and rates. These limitations unusually are scoped over the number of requests, and the periodicity intervals range from minutely for quotas, to secondly for rates. This situation may be caused by the lack of versatility and expressivity existent in current modeling tools.

In our analysis, we identify two different threats to the validity of the results herein presented: (i) the size of the sample may not be statistically representative regarding the total population of APIs in the real world. Nevertheless, we have tried to prioritize the more popular categories in each repository so that we can maximize the API usage; (ii) despite the fact that we have tried to do our best when validating data, there may be some errors since the process is manual. Apart from offering the open dataset we plan, as future work, to revisit it and undertake a comprehensive examination.

4 Related Work

A number of analyses of web services in the industry and, especially, of RESTful APIs, have been presented. They usually focus on characteristics inherent to the API design. This work presents a new research direction by developing a systematic study of RESTful APIs focusing on how providers deal with non-functional properties in plans by establishing limitations, such as rates and quotas. We emphasize our work in providing an open and machine-readable dataset to other researchers.

The more relevant literature we have revised is summarized in the following:

A first set of studies is focused on traditional web services (WSDL/XML/SOAP). On the one hand, Li et al. show a study on Web services [13] in order to get the diversity of the specification of key elements in the industry. Specifically, they focus on statistics based on the number of defined operations, WSDL document size, average words used in the description fields and function diversity. They crawled some web services catalogs and collected information about 570 WSDL documents from active services, nevertheless, they focus only on a single search engine. On the other hand, Al-Masri et al. present a broader study [1] in which the authors have developed a crawler for collecting information about 5077 WSDL references available in different sources, such as Google, Yahoo, Alltheweb and Baidu. They determine statistics about object sizes, technology and function among others. They also point out the disconnection between UDDI registries and the current web, since these registries are incapable of providing Quality of Service (QoS) measurements for registered Web services and they do not clearly define how service providers can advertise business models.

Coinciding with the progressive increase of RESTful APIs, a second set of works are focused on these services. In [14], Maleshkova et al. analyze a set of randomly chosen 222 APIs of ProgrammableWeb, not just RESTful APIs but RPC and hybrid style also. They analyze six API characteristics: general information, types, input parameters, output formats, invocation details and complementary documentation. They found that a lack of a standard format to document APIs. In particular, it shows that APIs suffer from under-specification because some important information (e.g., data type and HTTP methods) are missing. Furthermore, in [18], Renzel et al. show a study over the 20 most popular RESTful Web Services from ProgrammableWeb against 17 RESTful design criteria found in the literature. The point out that hardly any of the services claiming to be RESTful is truly RESTful. This study also offers the full dataset showing the values for each analyzed characteristic. Finally, in [4], Bülthoff et al. analyze a dataset which comprises 45 Web APIs in total, primarily chosen from ProgrammableWeb directory, and provide conclusions about common description forms, output types, usage of API parameters, invocation support, the level of reusability, API granularity and authentication details. In this study, the authors show that an 89% of APIs state and implement rate limitations, either written down as part of the documentation or included with the general terms and conditions.

In a third set of studies in the last years, authors are moving to conducting other analysis to determine how the APIs are evolving and whether best practices are being followed. For instance, in [20], Sohan et al. conduct a case study of 9 evolving APIs to investigate what changes are made between versions and how the changes are documented and communicated to the API users. Furthermore, they extract some recommendations, such as the use of semantic versioning, separate releases for bug fixes and new features, auto-generated API documentation cross-linked with changelogs and providing live API explorers. Next, Palma et al. in [15,16], present a framework to undertake API analysis, specifically, in the first work, they analyze 12 APIs in order to recognize some patterns and anti-patterns for RESTful APIs; in the second work, analogously, they study 15 APIs to detect some linguistic patterns and anti-patterns in URL paths. Furthermore, in [17], Petrillo et al. present a study evaluating and comparing the design of the RESTful APIs of 3 cloud providers in terms of the fulfillment of a catalog of 73 best practices. They show that APIs reach an acceptable level of maturity when they consider best practices related to understandability and reusability. Moreover, in [19], Rodriguez et al. evaluate some good and bad practices in RESTful APIs. In particular, they analyze data logs of HTTP calls collected from the Internet traffic, identify usage patterns from logs and compare these patterns with design best practices.

Furthermore, from an industrial perspective some studies have been carried out; Musser, VP of ProgrammableWeb, highlights in a conference[13] what are the more common business models nowadays. In this sense, Yu et al. carried out a study [25] that analyzes structure and dynamics of ProgrammableWeb, determining that cumulative API use follows a power law distribution: a large number of APIs is used in a few mashups and a small number of APIs is used by many mashups. Furthermore, Haupt et al. present a study [11] of some API properties over 286 Swagger descriptions using a custom framework to analyze these Swagger documents.

[13] Available at https://goo.gl/8eZwwv.

In a pricing model perspective, we found initial works such as [2] in which Andrikopoulos et al. present a cost calculator for cloud ecosystems. More specifically, Vukovic et al. have presented some relevant works in the sense API ecosystems analysis and formal representations of service licenses. In [24] they presented a graph-based data model for API ecosystem built on an RDF data store. It stores temporal information about when entities and relationships were created and possibly deleted, allowing insights into the evolution of API ecosystems. On the other hand, in [22] they present a data model for API terms of service that captures a set of non-functional properties of APIs and allows for terms and conditions to be automatically assessed and composed. Later, in [23] they define a formal representation of service license description that facilitates automated license generation and composition. They also care about some QoS parameters and its relationship between the agreed SLA. Nevertheless, they do not identify any limitation that actually exists in real API plans, such as quotas and rates. Moreover, they restrict the concept of Service Level Agreements (SLAs) to two components: condition and action, whereas our approach pretends to go further.

To the best of our knowledge, our work differs from the one presented herein in three specific points: (i) Any of the analyzed works present a study over a number of RESTful APIs in terms of non-functional aspects and limitations (e.g., quotas and rates), plans and business models. (ii) We have carried out our analysis systematically, defining a specific set of objectives and research questions, rules to select the APIs and a specific methodology to analyze the gathered data. (iii) None of the works provides an open dataset in a machine-readable format so that researchers could improve and use the data gathered by authors in further studies. The only one that presents a dataset is [18], nevertheless, they do not offer it in a machine-readable way.

5 Conclusions and Future Work

In this paper, we have systematically studied 69 RESTful APIs of XaaS offerings; after identifying the research questions, we selected two valid sources to extract APIs from: Mashape and ProgrammableWeb. Next, we analyzed a set of characteristics regarding the type of the API, pricing, business models used in the XaaS offering, functionality limitations, overage and quotas and rates. We found that there exists a wider expressibility in terms of API limitations when the API is not explicitly regulated by an API Gateway, such as Mashape.

As an additional value, we believe the results of this study can also be useful for practitioners who plan to design a new plan for an API. Finally, as a future work, we plan to identify: (i) a correlation between the price plan offered and the types of limits; (ii) a specific set of requirements to define a formal governance model that supports a realistic usage plan specification for RESTful APIs, including temporality elements such as scheduling restrictions as defined in [9].

References

1. Al-Masri, E., Mahmoud, Q.H.: Investigating web services on the world wide web. In: WWW 2008, vol. 32(3), pp. 795–804 (2008)
2. Andrikopoulos, V., Song, Z., Leymann, F.: Supporting the migration of applications to the cloud through a decision support system. In: ICSOC 2013, pp. 565–572. IEEE, June 2013
3. Bonardi, M., Brioschi, M., Fuggetta, A.: Fostering collaboration through API economy. In: SER&IP 2016, pp. 32–38 (2016)
4. Bülthoff, F., Maleshkova, M.: RESTful or RESTless – current state of today's top web APIs. In: Presutti, V., Blomqvist, E., Troncy, R., Sack, H., Papadakis, I., Tordai, A. (eds.) ESWC 2014. LNCS, vol. 8798, pp. 64–74. Springer, Cham (2014). doi:10.1007/978-3-319-11955-7_6
5. Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P.: Cloud Computing Patterns. Springer, Heidelberg (2014)
6. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Building **54**, 162 (2000)
7. Fowler, M.: Microservices, pp. 1–14 (2014)
8. Forrester. API Management Solutions, Q3 2014. Technical report (2015)
9. García, J.M., Martín-Díaz, O., Fernandez, P., Ruiz-Cortés, A., Toro, M.: Automated analysis of cloud offerings for optimal service provisioning. In: Maximilien, M., Vallecillo, A., Wang, J., Oriol, M. (eds.) ICSOC 2017. LNCS, vol. 10601, pp. 331–339. Springer, Cham (2017)
10. Geelan, J.: Twenty-one experts define cloud computing. Cloud Comput. J. **4**, 5 (2009)
11. Haupt, F., Leymann, F., Scherer, A., Vukojevic-Haupt, K.: A framework for the structural analysis of REST APIs. In: ICSA 2017, p. 4 (2017)
12. Kitchenham, B., Charters, S.: Guidelines for performing systematic literature reviews in software engineering Version 2.3. Engineering **45**(4ve), 1051 (2007)
13. Li, Y., Liu, Y., Zhang, L., Li, G., Xie, B., Sun, J.: An exploratory study of web services on the internet. In: ICWS 2007, pp. 380–387. IEEE (2007)
14. Maleshkova, M., Pedrinaci, C., Domingue, J.: Investigating web APIs on the World Wide Web. In: ECOWS 2010, pp. 107–114. IEEE, December 2010
15. Palma, F., Dubois, J., Moha, N., Guéhéneuc, Y.-G.: Detection of REST patterns and antipatterns: a heuristics-based approach. In: Franch, X., Ghose, A.K., Lewis, G.A., Bhiri, S. (eds.) ICSOC 2014. LNCS, vol. 8831, pp. 230–244. Springer, Heidelberg (2014). doi:10.1007/978-3-662-45391-9_16
16. Palma, F., Gonzalez-Huerta, J., Moha, N., Guéhéneuc, Y.-G., Tremblay, G.: Are RESTful APIs well-designed? Detection of their linguistic (anti)patterns. In: Barros, A., Grigori, D., Narendra, N.C., Dam, H.K. (eds.) ICSOC 2015. LNCS, vol. 9435, pp. 171–187. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48616-0_11
17. Petrillo, F., Merle, P., Moha, N., Guéhéneuc, Y.-G.: Are REST APIs for cloud computing well-designed? An exploratory study. In: Sheng, Q.Z., Stroulia, E., Tata, S., Bhiri, S. (eds.) ICSOC 2016. LNCS, vol. 9936, pp. 157–170. Springer, Cham (2016). doi:10.1007/978-3-319-46295-0_10
18. Renzel, D., Schlebusch, P., Klamma, R.: Today's top "RESTful" services and why they are not RESTful. In: Wang, X.S., Cruz, I., Delis, A., Huang, G. (eds.) WISE 2012. LNCS, vol. 7651, pp. 354–367. Springer, Heidelberg (2012). doi:10.1007/978-3-642-35063-4_26

19. Rodríguez, C., Baez, M., Daniel, F., Casati, F., Trabucco, J.C., Canali, L., Percannella, G.: REST APIs: a large-scale analysis of compliance with principles and best practices. In: Bozzon, A., Cudre-Maroux, P., Pautasso, C. (eds.) ICWE 2016. LNCS, vol. 9671, pp. 21–39. Springer, Cham (2016). doi:10.1007/978-3-319-38791-8_2
20. Sohan, S.M., Anslow, C., Maurer, F.: A case study of web API evolution. In: SERVICES 2015, pp. 245–252. IEEE, June 2015
21. Tan, W., Fan, Y., Ghoneim, A., Hossain, M.A., Dustdar, S.: From the service-oriented architecture to the web API economy. IEEE Internet Comput. **20**(4), 64–68 (2016)
22. Vukovic, M., Laredo, J., Rajagopal, S.: API terms and conditions as a service. In: ISCC 2014, pp. 386–393. IEEE, June 2014
23. Vukovic, M., Zeng, L.Z., Rajagopal, S.: Model for service license in API ecosystems. In: Franch, X., Ghose, A.K., Lewis, G.A., Bhiri, S. (eds.) ICSOC 2014. LNCS, vol. 8831, pp. 590–597. Springer, Heidelberg (2014). doi:10.1007/978-3-662-45391-9_51
24. Wittern, E., Laredo, J., Vukovic, M., Muthusamy, V., Slominski, A.: A graph-based data model for API ecosystem insights. In: ICWS 2014, pp. 41–48. IEEE, June 2014
25. Yu, S., Woodard, C.J.: Innovation in the programmable web: characterizing the mashup ecosystem. In: Feuerlicht, G., Lamersdorf, W. (eds.) ICSOC 2008. LNCS, vol. 5472, pp. 136–147. Springer, Heidelberg (2009). doi:10.1007/978-3-642-01247-1_13

The Role of Limitations and SLAs in the API Industry

Published in the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Tallinn, Estonia. August 2019.

Authors: *Antonio Gámez Díaz, Pablo Fernández Montes, Antonio Ruiz Cortés, Pedro J. Molina, Nikhil Kolekar, Prithpal Bhogill, Madhurranjan Mohaan, Francisco Méndez.*
DOI: *10.1145/3338906.3340445.*
Rating: *GGS class 1 (A+).*

The Role of Limitations and SLAs in the API Industry

Antonio Gamez-Diaz
Universidad de Sevilla
Seville, Spain
antoniogamez@us.es

Pablo Fernandez
Universidad de Sevilla
Seville, Spain
pablofm@us.es

Antonio Ruiz-Cortés
Universidad de Sevilla
Seville, Spain
aruiz@us.es

Pedro J. Molina
Metadev
Seville, Spain
pjmolina@metadev.pro

Nikhil Kolekar
PayPal
San Jose, California, USA
nikhil@openweave.ai

Prithpal Bhogill
Google
Mountain View, California, USA
prithpal@google.com

Madhurranjan Mohaan
Google
Mountain View, California, USA
madhurranjanm@google.com

Francisco Méndez
AsyncAPI Initiative
Barcelona, Spain
fmvilas@gmail.com

ABSTRACT

As software architecture design is evolving to a microservice paradigm, RESTful APIs are being established as the preferred choice to build applications. In such a scenario, there is a shift towards a growing market of APIs where providers offer different service levels with tailored limitations typically based on the cost.

In this context, while there are well established standards to describe the functional elements of APIs (such as the OpenAPI Specification), having a standard model for Service Level Agreements (SLAs) for APIs may boost an open ecosystem of tools that would represent an improvement for the industry by automating certain tasks during the development such as: SLA-aware scaffolding, SLA-aware testing, or SLA-aware requesters.

Unfortunately, despite there have been several proposals to describe SLAs for software in general and web services in particular during the past decades, there is an actual lack of a widely used standard due to the complex landscape of concepts surrounding the notion of SLAs and the multiple perspectives that can be addressed.

In this paper, we aim to analyze the landscape for SLAs for APIs in two different directions: i) Clarifying the SLA-driven API development lifecycle: its activities and participants; 2) Developing a catalog of relevant concepts and an ulterior prioritization based on different perspectives from both Industry and Academia. As a main result, we present a scored list of concepts that paves the way to establish a concrete road-map for a standard industry-aligned specification to describe SLAs in APIs.

CCS CONCEPTS

• **Information systems** → **RESTful web services**; • **Software and its engineering** → **Extra-functional properties**; **System description languages**.

KEYWORDS

RESTful APIs, SLA, OpenAPI Specification, SLA-driven APIs, API Gateways

ACM Reference Format:
Antonio Gamez-Diaz, Pablo Fernandez, Antonio Ruiz-Cortés, Pedro J. Molina, Nikhil Kolekar, Prithpal Bhogill, Madhurranjan Mohaan, and Francisco Méndez. 2019. The Role of Limitations and SLAs in the API Industry. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3338906.3340445

1 INTRODUCTION

In the last decade, RESTful APIs are becoming a clear trend as composable elements that can be used to build and integrate software [6, 12]. One of the key benefits this paradigm offers is a systematic approach to information modeling leveraged by a growing set of standardized tooling stack. In this context, the term of *API Economy* is being increasingly used to describe the movement of the industries to share their internal business assets as APIs [22] not only across internal organizational units but also to external third parties; in doing so, this trend has the potential of unlocking additional business value through the creation of new assets [4]. In fact, we can find a number of examples in the industry that are deployed solely as APIs (such as Meaningcloud[1], Flightstats[2] or Twilio[3]).

In order to be competitive in this such a growing market of APIs, at least two key aspects can be identified: i) *ease of use* for its potential developers; ii) a flexible usage *plan* that fits their customer's demands.

Regarding the *ease of use* perspective, third-party developers need to understand how to use the exposed APIs so it becomes necessary to provide good training material but, unfortunately, several API providers do not often write good documentation of their products [7]. Notwithstanding, during the last years, the *OpenAPI*

[1] https://www.meaningcloud.com/products/pricing
[2] https://developer.flightstats.com/getting-started/pricing
[3] https://www.twilio.com/sms/pricing

Specification[4] (OAS), formerly known as *Swagger* specification, has become the *de facto* standard to describe RESTful APIs from a functional perspective providing an ecosystem that helps the developer in several aspects of the API development lifecycle[5].

The benefits are twofold: from the API provider's perspective, there are tools aimed to automate the server scaffolding, an interactive documentation portal creation or the generation of unit test cases; from API consumer's perspective, there are tools to automate the creation of API clients, the security configuration or the endpoints discovery and usage [1, 19, 21].

Concerning the usage *plans* perspective, as APIs are deployed and used in real settings, the need for non-functional aspects is becoming crucial. In particular, the adoption of Service Level Agreements (SLAs) [17] could be highly valuable to address significant challenges that industry is facing, as they provide an explicit placeholder to state the guarantees and limitations that a provider offers to its consumers. Exemplary, these limitations (such as *quotas* or *rates*) are present in most common industrial APIs [8] and both API providers and consumers need to handle how they monitor, enforce or respect them with the consequent impact in the API deployment/consumption.

However, to the best of our knowledge, there is no widely accepted model to describe usage plans including elements such as cost, functionality restrictions or limits. In this context, a new type of infrastructure, coined as API Gateway [10], has emerged to support API developers in the management of multiple non-functional aspects such as consumer authentication, request throttling or billing. From a deployment perspective, API Gateways are usually implemented as virtual appliances, virtual machine images or reverse proxies that promote a decoupling from the main API artifact. In contrast, the vendor-specific approach to non-functional concerns typically represents a strong dependence with the API Gateway provider.

In this paper, we aim to analyze the landscape in the SLA and limitations for APIs directly from those participants who have shown interest on participating in the definition of an industrial standard for SLAs in APIs. Specifically, we have started up conversations with members of the OpenAPI Initiative who belong to the SLA interest group aiming to gather information about their industrial perspective of the role of SLAs and limitations in the APIs.

The rest of the paper is structured as follows: in Section 2 we introduce, briefly, the idea of Service Level Agreements (SLA) and its importance in the API ecosystem. Next, in Section 3, we describe the related work. Continuing, in Section 4 we describe the SLA-driven API lifecycle. Further, in Section 5 we present the industrial insights from different participants. Finally, in Section 6, we show some final remarks and conclusions.

2 SLAS IN A NUTSHELL

Service Level Agreements (SLAs) consist of a set of terms that include information about functional features, non-functional guarantees, compensation, termination terms and any other terms with relevant information to the agreement. An agreement signed by all interested parties should be redacted carefully because a failure to specify their terms could carry penalties to the initiating or responding party. Therefore, agreement terms should be specified in a consistent way, avoiding contradictions between them. However, depending on the complexity of the agreement, this may become a challenging task. SLAs can, therefore, be used to describe the rights and obligations of parties involved in the transactions (typically the service consumer and the service provider); among other information, SLA could define guarantees associated with the idea of Service Level Objectives (SLOs) that normally represent key performance indicators of either the consumer or the provider. In case the guarantee is under-fulfilled or over-fulfilled SLAs could also define some compensations (i.e. penalties or rewards). In such a context, during the last years, there have been important steps towards the automation of the management of SLAs, however, the formalization in SLAs still remains an important challenge.

A *SLA* typically contains these concepts:

Name identifies the agreement and can be used for reference.

Context includes information such as the name of the parties and their roles as initiator or responder in the agreement. Additionally, it can include other important information for the agreement.

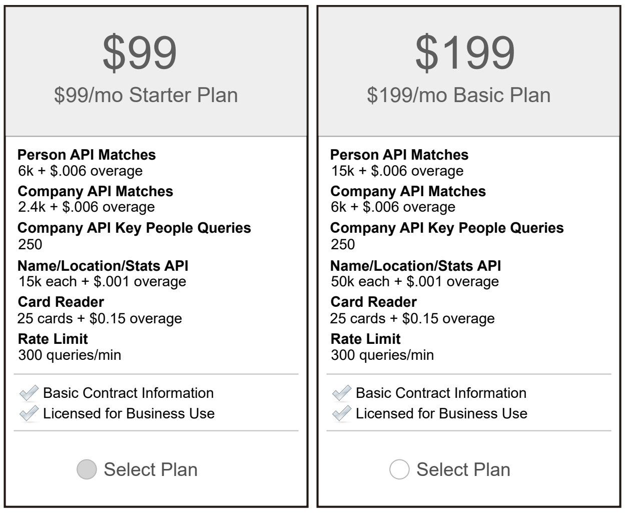
Terms the two main types of terms are:

- **Service terms** they provide service information by means of:
  - **Service description terms** which includes information to instantiate or identify the services and operations involved in the agreement.
  - **Service properties** which includes the measurable properties that are used in expressing guarantee terms. They consist of a set of variables whose values can be established inside the service description term. These terms play an key role in the definition of the service level which is actually offered to clients and the price they pay for. For instance, in APIs, it is common to see quota (e.g., 30K request/month) and rate (e.g., 1 request/second) limitations that define the service.
- **Guarantee terms** they describe the service level objectives (*SLOs*) agreed by a specific obligated party, using Service Level Indicators (*SLIs*), a set of carefully defined quantitative measures of some aspect of the level of service that is provided. It also includes the scope of the term (e.g. if it applies to a certain operation of a service or the whole service itself) and a qualifying condition that specifies the validity condition under which the term is applied. Guarantee terms often include compensations [17], that is, penalties (or rewards) applied when the SLO is unfulfilled or overfulfilled.

The concept of SLA is, very frequently, misunderstood: some services claim to have an SLA when they are only defining the service description terms (e.g., limitations). SLAs are agreements, that is, an explicit or implicit contract with your users that includes consequences of the meeting (or missing) the SLOs they contain [3, 20]. In many services, including APIs, there is no SLA: if nothing happens if the SLOs are not being met, it is not an SLA, but a mere description of SLOs and service properties.

[4]The latest version of the OpenAPI Specification is available at https://github.com/OAI/OpenAPI-Specification
[5]https://openapi.tools

In the industry, the way in which a customer can select and purchase a certain service level is by using pricing plans. In Figure 1 it is depicted a real plan extracted from *FullContact*[6], a product which includes an API for managing and organizing contacts in a collaborative way and it also matches emails addresses looking for publicly available information on the Internet to enrich the profiles.



$99
$99/mo Starter Plan

Person API Matches
6k + $.006 overage
Company API Matches
2.4k + $.006 overage
Company API Key People Queries
250
Name/Location/Stats API
15k each + $.001 overage
Card Reader
25 cards + $0.15 overage
Rate Limit
300 queries/min

✓ Basic Contract Information
✓ Licensed for Business Use

● Select Plan

$199
$199/mo Basic Plan

Person API Matches
15k + $.006 overage
Company API Matches
6k + $.006 overage
Company API Key People Queries
250
Name/Location/Stats API
50k each + $.001 overage
Card Reader
25 cards + $0.15 overage
Rate Limit
300 queries/min

✓ Basic Contract Information
✓ Licensed for Business Use

○ Select Plan

Figure 1: Example of an API plan

This example is composed of two paid plans having a fixed price that is monthly billed. Regarding the *limitations*, for each resource, a *quota* is being applied; for instance, in the *starter* plan, only 6000 matches over Person are available. Nevertheless, an overage is defined, that is, it is possible to overcome the limit by paying a certain amount of money; in this case, $0.006 per each request. Regardless of the accessed resources, a common *rate* of 300 queries per minute is being applied.

In this example, there is neither guarantee term nor SLOs. All these elements belong to the set of service properties, particularly, the limitations, which are, actually, defining the service level (e.g., *free*, *starter* or *basic*)

3 RELATED WORK

The software industry has embraced integration as a key challenge that should be addressed in multiple scenarios. In such a context, the proliferation of APIs is a reality that has been formally analyzed: in [18], authors performed an analysis of more than 500 publicly-available APIs to identify the different trends in current industrial landscape with the following key results: in terms of paradigm they conclude that 500 out of 522 analyzed APIs provide an API based on REST; regarding the *format*, the authors identified that nearly two thirds of the APIs support JSON without supporting XML. Concerning the *access control*, authors showed that most APIs require some form of service registration for developers to start using the API. Regarding the *documentation*, they showed that generated documentation is being used in about a half of the APIs, with documents generated by SwaggerUI (from an OpenAPI Specification) taking the lead, suggesting some tendency to make the API documentation machine-readable and understandable as well. Specifically, from a functional point of view, there is a clear trend with respect to the functional description of the service: during the last years, the OpenAPI Specification has consolidated as a *de-facto* standard to define the different functional properties an API provides. One of the reasons behind this success has been a growing ecosystem of tools that leverages from the API development life-cycle based on the information included in OAS: from automated code generators that create an initial scaffolding of the API to dynamic documentation portals that allow developers to understand and test the API usage.

[6]https://www.fullcontact.com/developer

Table 1: Analysis of SLA Models

Name	F1	F2	F3	F4	F5	F6	F7
SLAC [24]	DSL					✓	✓
CSLA [14]	XML		✓			✓	
L-USDL Ag. [11]	RDF	✓	✓		†	✓	
rSLA [23]	Ruby	✓		✓	✓		✓
SLAng [15]	XML	✓					
WSLA [16]	XML	✓	✓		✓		
SLA* [13]	XML	✓	✓		✓		
WS-Ag. [2]	XML	✓	✓	✓	†		

† Supported with minor enhancements or modifications.

In such a consolidated market of APIs, non-functional aspects are also becoming a key element in the current landscape. In [8], authors analyze a set of the 69 real APIs in the industry to characterize the variability in its offerings, obtaining a number of valuable conclusions about real-world APIs, such as: (i) Most APIs provide different capabilities depending on the tier or plan of the API consumer is willing to pay. (ii) Usage limitations are a common aspect all APIs describe in their offerings. (iii) Limitations over API requests are the most common including quotas over static periods of times (e.g., *1.000 request each natural day*) and rates for dynamic periods of times (*3 request per second*). (iv) Offerings can include a wide number of metrics over other aspects of the API that can be domain-independent (such as the number of returned results or the size in bytes of the request) or domain-dependent (such as the CPU/RAM consumption during the request processing or the number of different resource types). Based on these conclusions, we identify the need for non-functional support in the API development life-cycle and the high level of expressiveness present in the API offerings.

Furthermore, as monitoring is a key aspect, a number of works have been presented aiming to analyze different approaches for runtime monitoring. In [20], authors developed a comparison framework for runtime monitoring approaches and validate it by applying it to 32 existing approaches and by comparing 3 selected approaches in the light of different monitoring scenarios.

Furthermore, during the last decade, a number of SLA models have been presented. We have analyzed the most prominent academic and industrial proposals aimed to the definition of SLAs in both traditional web services and cloud scenarios.

Specifically, in Table 1, we have considered 7 aspects to analyze in each SLA proposal, namely: **F1** determines the format in which the document is written syntax; **F2** shows whether the target domain is web services; **F3** indicates if it can model more than one offering (i.e., different operations of a web service); **F4** determines if it allows modeling hierarchical models or overriding properties and metrics; **F5** shows whether temporal concerns can be model (e.g., in metrics); **F6** indicates if there exists a tool for assisting users to model using this proposal; **F7** determines if there exists a tool/framework for enacting the SLA.

Based on the comparison of the different SLA models (summarized in Table 1), we highlight the following conclusions: (i) None of the specifications provides any support or alignment with the OpenAPI Specification; (ii) Most of the approaches provide a concrete syntax on XML, RDF (some of them they even lack concrete syntax) and there is no explicit support to YAML or JSON serializations. (iii) An important number of proposals are complete, but others leave some parts open to being implemented by practitioners. (iv) Besides the fact that a number of proposals are aimed to model web services, they are focused on traditional SOAP web services rather than RESTful APIs. In this context, they do not address the modeling standardization of the RESTful approach: i.e., the concept of a resource is well unified (a URL), and the amount of operations is limited (to the HTTP methods, such as GET, POST, PUT and DELETE). This lack of support of the RESTful modeling prevents the approaches to have a concise and compact binding between functional and non-functional aspects. (v) They do not have enough expressiveness to model limitations such as quotas and rates, for each resource and method and with complete management of temporally (static/sliding time windows and periodicity) present in the typical industrial API SLAs. (vi) Most proposals are designed to model a single offering and they mostly lack support to modeling hierarchical models or overriding properties and metrics (F4); in such a context, they cannot model a set of tiers or plans that yield a complex offering that maintains the coherence by model and instead they rely on a manual process that is typically error-prone. (vii) finally, the ecosystem of tools proposed in each approach (in the case of its existence) is extremely limited and aimed to be solely as a prototype; moreover, they apparently are not integrated into a developer community nor there is evidence of this usage by practitioners in the industry.

4 INTRODUCING SLAS IN THE API LIFECYCLE

In spite of the fact that each organization could address the API lifecycle with slightly different approaches, we identify a minimal set of general stages and activities. The first activity corresponds with the actual *Functional Development* of the API implementing and testing the logic; next a *Deployment* activity where the developed artifact is configured to be executed in a given infrastructure; finally, once the API is up and running, an *Operation* activity starts where the requests from consumers can be accepted. This process is a simplification that can be evolved to add intermediate steps (such as testing) or to include an evolutionary cycle where different versions are deployed progressively. In order to incorporate SLAs in this process, we expand to this basic lifecycle where both API Provider and API Consumer interact (as depicted in Figure 2).

Specifically, from the provider's perspective, the *Functional Development* can be developed in parallel with a *SLA modeling* where the actual SLA offering is written and stored in a given *SLA Registry*. Once both the functional development and the SLA modeling has concluded, the *SLA instrumentation* must be carried out, where the tools and/or developed artifacts are parameterized, so they can adjust their behavior depending on a concrete SLA and provide the appropriate metrics to analyze the SLA status. Next, while the *deployment* of the API takes place, a parallel activity of *SLA enactment* is developed where the deployment infrastructure should be configured in order to be able to enforce the SLA before the API reaches the *operation* activity.

Complementary, from consumer's perspective, once the provider has published the SLA offering (i.e., *Plans*) in the *SLA Registry*, it starts the *offer analysis* to select the most appropriate option (*offer selection* activity) and to create and register its actual SLA; finally, the API *Consumption* is carried out as long as the API is the *Operation* activity and its regulated based on the terms (such as quotas or rates) defined in the SLA.

In order to implement this lifecycle, it is important to highlight that the *SLA instrumentation*, *SLA enactment* and *Operation* activities should be supported by an SLA enforcement protocol aimed to define the interactions for *checking* if the consumption of the API for a given consumer is allowed (e.g., it meets the limitations specified in its SLA) and to gather the actual values of the *metrics* from the different deployed artifacts that implement the API.

From an industrial perspective and regarding the implication across the entire development lifecycle of APIs, different roles or stakeholders appear, as discussed below. The mapping role-activity is also depicted in Figure 2 by using the RALPH notation [5].

Developer This role is composed by the team responsible for the development of a certain API and making it available for other teams. Their use cases are related to the definition of Service Level Objectives (SLOs) since they are the role most aware of the internal functioning of the API. Namely:
- a better understanding of what SLOs can they reasonably target so that they can offer an SLO for the API.
- a better understanding of the performance of their downstream dependencies (e.g., back-ends) so that they can determine their effect on the SLOs.
- a better understanding of the performance of policies in the proxy so that they can determine their effect on the SLOs.

Product manager This role is composed of business people, aligned with the company's objectives. Their use cases aim to satisfy customer's needs and be aware of the overall picture of the dependencies between services. Namely, knowing the SLOs of the downstream dependencies so that they can create products which meet the customers' needs.

Product operator This role is composed of system administration people, who are responsible for monitoring and reporting the service performance in SLOs. Their use cases aim to be notified of any alert or incident and take remedial actions. Namely:

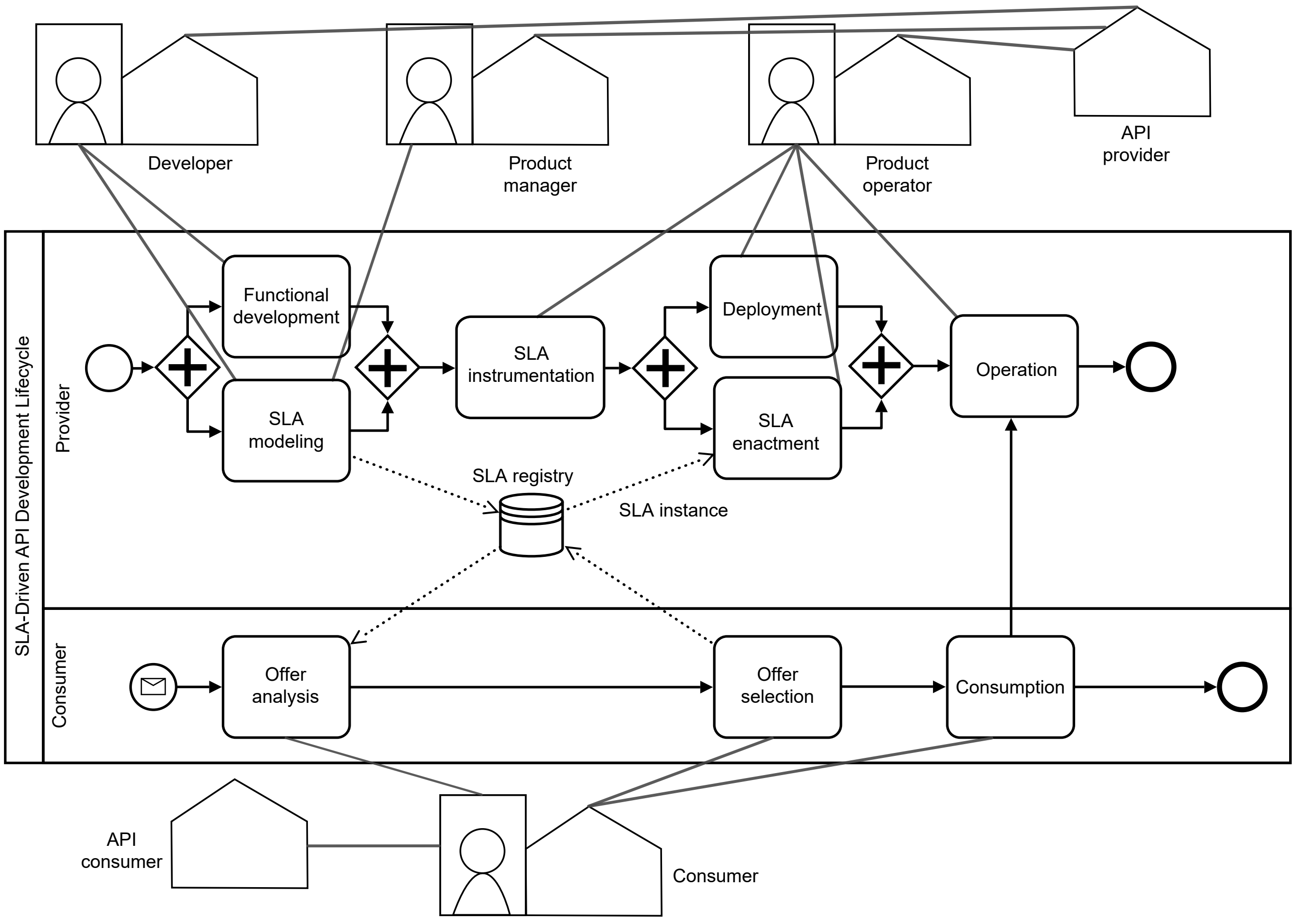


Figure 2: SLA-Driven API development lifecycle

- having alerts automatically set based on SLOs to alert them of the risk of missing the objective so that they can take remedial action.
- receiving regular reports detailing API performance against SLOs, so that they can report to the business owners.
- watching both the internal and external SLO commitments for various APIs or Products so that they can quickly categorize and prioritize the operational efforts.

Consumer This role is composed of the set of API clients. Their use cases aim to be informed of the different service levels and claim if the SLOs are not being met. Namely:

- knowing what service level is offered so that they can make an informed decision about adopting the API.
- understanding the historical actual performance of an API so that they can know how reliable they might expect them to be.
- assuring that they are getting the service level that they are paying for so that they can claim remedies if SLOs are not met.

5 INDUSTRIAL DISCUSSION

5.1 The Discussion Process

We opened a call for interest on participating in a research paper open to the OAI members belonging to the SLA4OAI group[7], as part of the OpenAPI Initiative. Our main goal is to gather information about their industrial perspective of the role of SLAs in the APIs.

In order to present general vision, we have classified the participants in different groups regarding their role in the API industry, namely: i) *API infrastructure manager*: are the creator of middleware solutions such as API Gateways or proxies, they do not develop any particular API, but they enhance and enrich third-party ones with other features; ii) *API providers*: are the developers of one or many APIs and also responsible for setting the proper service level and limitations values; iii) *Others*: represent a different set of participant not included before, for instance, API enthusiasts and people who have been involved in the creation of other specifications.

[7]More information at sla@openapi.groups.io

As for an *API infrastructure manager*, we have *Google Apigee.* As for an *API provider*, we have *PayPal.* Finally, *other* participants include *Async API* and *Metadev.*

5.2 Describing some API Concepts

In order to have a common vocabulary prior discussion, some considerations about the concepts and terminology took place:

5.2.1 *SLA General Concepts.*

Context describes aspects such as the version, stakeholders or the validity period.

Metrics are the elements that are being gathered and computed.

Service Level Indicator (SLI) is a particular case of metric which is used to assess one key aspect of the system. They are typically implemented as a time series and may involve some level of sophistication (e.g., sliding windows) in its calculation.

Service Level Objective (SLO) is a precise numerical target (often a ratio) for one or more SLIs, describing the minimum acceptable reliability or performance of a system. A given system may have different SLOs for different users, e.g., an internal objective and an external one.

Guarantee terms describe the commitments over certain SLI. They also should describe the consequences of not meeting this commitment in terms of compensations.

Service Level Agreement (SLA) is, therefore, a contract signed with a user. Notably, SLIs and SLOs are technical constructs whereas SLAs are business constructs.

Service properties (or configuration) are the attributes constraints that are being used to drive the API behavior.

5.2.2 *API Constraints.*

Quotas describe the limitations of use for a fixed/static period of time. It is an entitlement to API usage over a (usually relatively long) time period, e.g., 100000 calls per month.

Rates describe the limitations of use for a dynamic period of time. It is an entitlement to API usage over a (usually short) time period, e.g., 10 calls per second per consumer.

Time constraint some APIs can offer a set of limitations regarding the time in which it is being requested. For instance, some calls could be thought to be cheaper during off-peak hours.

Authentication is the verification of the credentials of the request. This process is based on sending the credentials from the remote client to the server by using an authentication protocol. Likewise, the authorization is the process of verification that the connection attempt is allowed. These mechanisms are required for the API monetization.

5.2.3 *API Monetization.*

Pricing is the way in which APIs are monetized. Typically, some pricing models are: fixed (with or without overage) and pay-as-you-go. The first allows a developer to purchase fixed values for a set of metrics (e.g., number of calls) within a period (e.g., per month), but they cannot exceed the established limitations; when overage is allowed, a small fee is charged if the developer exceeds the values of the metrics (e.g., number of calls).

Plans is an approach to fit a wide range of business needs by organizing the pricing in a set of tiers of plans.

Metering is the recording of the API usage in sufficient detail to perform rating.

Rating is the conversion of records of API usage into an owed amount of money. This conversion may involve simply a fixed charge per API call, or considerably more complex schemes.

Billing is the presentation to an API user of a report of amounts owed, taking into account any discounts, service credits, taxes, and revenue sharing.

Collection is the way of receiving and recording payments of amounts owed by users of APIs.

Enforcement is preventing a user from using an API once they have exhausted their pre-paid service credit, or reached a credit limit.

5.3 API Provider's Vision

For some API providers, the inclusion of SLAs is something relatively new (less than five years ago), but the main issue is the SLA field is the set of activities surrounding the SLOs to improve the customer experience; for instance, the definitions of metrics and SLIs and the monitoring process.

They believe that, in general, SLOs are drivers for customer experience and digital businesses. As applications and experiences are composed of business capabilities and they are realized as APIs which may use other APIs to achieve their business function, the customer experience is fueled by complex tiered orchestration of APIs and, therefore, performance and availability of experiences is a function of those underlying services.

SLOs for APIs dictate suitability and choice of utilization and, hence, having the ability to accurately measure and monitor SLOs is a fundamental requirement. SLOs, also, dictate performance and availability profiles for the application and provide individual accountability for performance and availability across enabling services. The common thread is the correlation and tracking of the call-chain for service invocation, the identification of the API subscription for applications, monitoring aggregated and apportioned performance profiles for applications and, finally, a common set of performance metrics need to be defined, logged, monitored, analyzed and reported.

As API providers, they use to consider the following set of metrics/SLIs in their APIs:

- **Call volume**: number of API operations invocations irrespective of response.
- **Response time**: the total amount of time, in milliseconds, it takes the service to respond to an API operation request aggregated as the 95th percentile, 90th, and 50th.
- **Availability**: percentage of API calls completed without causing a *Failed Customer Interaction.*
- **Business Error Rate**: percentage of API calls with business error responses. A business error is an error that is not a system error and could be caused by invalid input, user error, business rules, policy constraints, or lack of authorization.
- **System Error Rate**: percentage of API calls with system error responses. A system error is an error that is caused by

a code defect, timeouts for underlying services, or a framework failure, including a hardware network or environment failure.

Regarding the SLI, these metrics need to be measured at the individual API operation level. For REST APIs, the URI for the resource and the HTTP method need to be used as identifiers for API operation. The method identifier from the API specification must be used for correlation. The API operation metrics need to be correlated to the API product and its major and minor version. This correlation will provide insights into capability and ownership attribution form the observed quality of service with respect to published SLAs. The application identity of the originating application, along with that of the immediate application invoking the API operations must be tracked. The identity of Remote Availability Zone (RAZ) for the service application must be tracked to help understand the quality of service across RAZs.

Concerning the monitoring, the published SLOs for API operations must be monitored for compliance. Since there could be variance in API metrics for diverse application use-cases, compliance must be computed using 95th, 90th percentiles, and average aggregations initially, before being base-lined for a longer term. As a daily basis, developer and operation teams are responsible for checking the service status and monitoring the key metrics. Specifically, the SLIs are expected to be in an acceptable range, as defined in the SLOs. For instance, the SLIs availability and latency are measured to meet the target metrics in the SLOs.

Regarding the SLAs, they see SLAs as part of a wider contract, which includes other legal aspects. In such a context, the SLA is just a part of the service contract. At some organizational levels, the value of the SLAs is concentrated in the fulfillment of the guarantee terms when negotiating contractual agreements and invoicing, that is, the SLA reporting. At this point, the SLA of the API services should be considered to be reportable, that is, showing, at a glance, the overall picture of the SLA state in each moment.

In service-based applications (SBAs) the fruitful composition of different services and APIs play a crucial role. There is a strong dependency between different components and, therefore, they are expected to be as reliable as possible (and agreed in the SLA). As an SBA provider, it is strictly necessary to know in advance all the values of the limitations and the agreed SLA terms. Otherwise, the provider is not able to set its own SLOs

5.4 API Infrastructure Manager's Vision

As API infrastructure manager, such as an API Gateway, their platforms aim to define API concerns such as different service levels, API limitations (or entitlement) and pricing. They also lay out their position on extending the OpenAPI specification in this area.

Regarding the pricing, their platform provides support for: *fixed fee per API call*, *fixed fee per time period*, *volume-based tiers of fees per API call*, *volume-based bundles of API calls*, *revenue sharing schemes*, *charging variable amounts* based on arbitrary runtime attributes (parameters in the request, elements of the response, time, geography, current load on the API, etc).

They consider two different types of API limitations: quotas and rate limits: i) Quotas are the business level construct of enforcing how much access does one client have to an API based on their tier. For instance: a *gold tier* customer may have access to invoking a set of APIs 1000 per day, whereas a *bronze tier* customer may only able to invoke 100 per day. 2) Rate limiting, on the other hand, has a system-centric connotation. For instance: if the infrastructure is only expected to work for loads under 100 transactions per second, the proper level of rate limiting policy would be irrespective of the kind of customer invoking it.

Regarding the roles, they consider API producers as a team responsible for API development and making the APIs available for every other team. Additionally, they identify the role of an API Product Manager as the one that has business ownership of a portfolio of APIs also known as an API Product. Their main focus is to manage these products and look into ways of monetizing them via partners and external developers. As API infrastructure managers, they use to consider the following set of metrics/SLIs in their APIs:

- **Availability**: percentage of API calls completed without errors.
- **Error rates**: percentage of API calls with error responses
- **Latency**: the total amount of time that takes the service to respond to an API operation request aggregated as a percentile.

Concerning the modeling issues, their current priority would be to codify SLIs and SLOs for APIs in a formal description language by extending the OpenAPI Specification. Based on such a codification their tooling could then offer richer native support for the user stories. Nevertheless, they recommend focusing first on defining an extension to describe technical concerns (e.g., SLIs and SLOs) and keep SLAs (as a business contract) out of the scope for a later extension. They believe that SLAs, as well as not being readily amenable to such a codification, probably don't belong in OpenAPI Specification in any case.

They also suggest that monetization and pricing definition should be part of a separate initiative. In the real world, there is significant complexity in rating API usage, likely deserving of its own OpenAPI extension.

5.5 Discussion's Results

In this section, we show some final remarks aiming to be able to define a roadmap in the standardization of the SLA and limitations in an API context.

The relevance of each concept described in Section 5 is different for each provider. After asking them for scoring each one, we gathered and aggregated the responses, as stated in Table 2.

The most important concepts are *metrics/SLIs*, *quotas* and *rates*. The importance of the definition of SLOs for both API producers and infrastructure manager is notorious. As also stated by other participants, it is important to keep separate concerns and different aspects (i.e., SLOs, plans, metrics); they can be always be referenced externally if needed. The granularity of definitions when defining an SLA model is a problem: there exists the dichotomy between a fine-grained approach (i.e., a fully comprehensive model description) and a coarse-grained one (i.e., a description the most common elements and paving the way for custom extensions).

Table 2: Relevance of concepts for industrial participants

Items		Score
General concepts	Context	●●○
	Metrics	●●●
	SLIs	●●●
	SLOs	●●○
	Guarantees	●●○
	SLAs	●○○
	Configuration	○○○
API constraints	Quotas	●●●
	Rates	●●●
	Time constraints	●○○
	Authorization	●○○
API monetization	Pricing	●○○
	Plans	●○○
	Metering	●●○
	Rating	●●○
	Billing	●○○
	Collection	●○○
	Enforcement	●●○

Symbol denotes the relevance for the industrial participants.

In general terms, the participants belonging to the SLA4OAI group, as part of the OpenAPI Initiative, tend to agree in a manifesto during the standardization tasks:

Motivation fostering the importance of the SLA inside the API development lifecycle is that SLAs are already present in most commercial APIs. Since OAI is becoming the *de facto* standard for the definition of APIs, natural evolution to describe SLAs into OpenAPI Specification would expand the OAI benefits.

Goals Three are identified:
- Be as aligned as possible with the OpenAPI principles.
- Describe the most common elements in SLAs (e.g., plans, metrics, quotas, rates).
- Be integrated with the main OpenAPI Specification.

Non-goals There are two:
- Define a particular way to enforce SLAs.
- Be fully comprehensive including a wide set of elements found in different industrial APIs.

Design principles They are two:
- Pragmatism to spot the most common elements;
- Promote tooling to take advantage of the SLA4OAI Specification.

6 CONCLUSIONS

From the Academia's point of view, the fact of having a standard model for the definition of SLAs in APIs could foster the development of novel techniques aiming to deal with the information contained in the SLAs. There is already a number of works in the SLA field, as pointed out in Section 3, so aligning that with the API ecosystem would pave the way for new challenges.

As an example, this SLA model could enable *SLA-aware monitoring and testing* techniques: including non-functional and QoS requirements into the test cases. Moreover, a *formal analysis on the SLA model* could unveil inconsistencies in the set of API limitations. Furthermore, *SLA-aware model-driven development* would experience an improvement, since taking into account the SLA could be helpful when deciding among different architectures. A first step in this direction, in [9], we presented *Governify for APIs*, an initial set of tools aimed to settle down our idea of SLA-driven APIs.

Finally, this work is intended to collect the industrial perspective on the challenge of standardizing the modeling of SLAs and limitations in the API context, under the umbrella of a well-assented specification for APIs as it is the OpenAPI Specification. The contribution presented herein just lay the first stone on the roadmap that is the modeling effort in conjunction with relevant industrial players.

ACKNOWLEDGMENTS

This work is partially supported by the European Commission (FEDER), the Spanish Government under projects BELI (TIN2015-70560-R) and HORATIO (RTI2018-101204-B-C21), and the FPU scholarship program, granted by the Spanish Ministry of Education, Culture and Sports (FPU15/02980).

The authors would also like to thank for their time and their valuable contributions to all the members of the OpenAPI Technical Steering Committee and, specially, to the rest of the Technical Committee behind the SLA4OAI Specification: Isaac Hepworth (Google), Jeffrey ErnstFriedman (The Linux Foundation), Kin Lane (API Evangelist), Mike Ralphson (The Linux Foundation) and Scott Ganyo (Google).

REFERENCES

[1] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. 2007. Mining API patterns as partial orders from source code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE '07*. ACM Press, New York, New York, USA, 25. https://doi.org/10.1145/1287624.1287630

[2] Alain Andrieux, Karl Czajkowski, Kate Keahey, A. Dan, Kate Keahey, H. Ludwig, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. 2004. Web Services Agreement Specification (WS-Agreement). (2004), 80 pages. http://forgc.gridforum.org/Public_Comment_Docs/Documents/Oct-2006/WS-AgreementSpecificationDraftFinal_sp_tn_jpver_v2.pdf

[3] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. 2016. *Site Reliability Engineering: How Google Runs Production Systems* (1st ed.). O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, USA.

[4] Michele Bonardi, Maurizio Brioschi, Alfonso Fuggetta, Emiliano Sergio Verga, and Maurilio Zuccalà. 2016. Fostering Collaboration Through API Economy: The E015 Digital Ecosystem. In *Proceedings of the 3rd International Workshop on Software Engineering Research and Industrial Practice (SER&#38;IP '16)*. ACM, New York, NY, USA, 32–38. https://doi.org/10.1145/2897022.2897026

[5] Cristina Cabanillas, David Knuplesch, Manuel Resinas, Manfred Reichert, Jan Mendling, and Antonio Ruiz-Cortés. 2015. RALph: A Graphical Notation for Resource Assignments in Business Processes. In *Advanced Information Systems Engineering*, Jelena Zdravkovic, Marite Kirikova, and Paul Johannesson (Eds.). Springer International Publishing, Cham, 53–68.

[6] Roy Thomas Fielding. 2000. Architectural Styles and the Design of Network-based Software Architectures. *Building* 54 (2000), 162. https://doi.org/10.1.1.91.2433

[7] Forrester. 2015. *API Management Solutions , Q3 2014.* Technical Report. Forrester.

[8] Antonio Gamez-Diaz, Pablo Fernandez, and Antonio Ruiz-Cortes. 2017. An Analysis of RESTful APIs Offerings in the Industry. In *Service-Oriented Computing*, Michael Maximilien, Antonio Vallecillo, Jianmin Wang, and Marc Oriol (Eds.). Springer International Publishing, Cham, 589–604.

[9] Antonio Gamez-Diaz, Pablo Fernandez, and Antonio Ruiz-Cortes. 2019. Governify for APIs: SLA-Driven ecosystem for API governance. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the*

Foundations of Software Engineering (ESEC/FSE 2019). ACM, Tallin, Estonia. https://doi.org/10.1145/3338906.3341176

[10] Antonio Gámez-Díaz, Pablo Fernández-Montes, and Antonio Ruiz-Cortés. 2015. Towards SLA-Driven API Gateways. In *Actas de las XI Jornadas de Ingeniería de Ciencia e Ingeniería de Servicios*, Juan Manuel Murillo (Ed.), Vol. 201232273. Sistedes, Santander, 9. https://doi.org/10.13140/RG.2.1.4111.5609

[11] José María García, Pablo Fernández, Carlos Pedrinaci, Manuel Resinas, Jorge Cardoso, and Antonio Ruiz-Cortés. 2017. Modeling Service Level Agreements with Linked USDL Agreement. *IEEE Transactions on Services Computing* 10, 1 (1 2017), 52–65. https://doi.org/10.1109/TSC.2016.2593925

[12] Holger Harms, Collin Rogowski, and Luigi Lo Iacono. 2017. Guidelines for Adopting Frontend Architectures and Patterns in Microservices-based Systems. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 902–907. https://doi.org/10.1145/3106237.3117775

[13] Keven T. Kearney, Francesco Torelli, and Constantinos Kotsokalis. 2010. SLA * An abstract syntax for Service Level Agreements. In *2010 11th IEEE/ACM International Conference on Grid Computing*. IEEE, Brussels, Belgium, 217–224. https://doi.org/10.1109/GRID.2010.5697973

[14] Yousri Kouki, Frederico Alvares de Oliveira, Simon Dupont, and Thomas Ledoux. 2014. A language support for cloud elasticity management. In *Proceedings - 14th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2014*. IEEE, Chicago, IL, USA, 206–215. https://doi.org/10.1109/CCGrid.2014.17

[15] D. D. Lamanna, J. Skene, and W. Emmerich. 2003. SLAng: A language for defining service level agreements. In *FTDCS*, Vol. 2003-Janua. IEEE, San Juan, Puerto Rico, USA, USA, 100–106. https://doi.org/10.1109/FTDCS.2003.1204317

[16] H. Ludwig, A. Keller, A. Dan, and R. King. 2002. A service level agreement language for dynamic electronic services. In *WECWIS 2002*. IEEE Comput. Soc, Newport Beach, CA, USA, USA, 25–32. https://doi.org/10.1109/WECWIS.2002.1021238

[17] C. Muller, A. Gutierrez Fernandez, P. Fernandez, O. Martin-Diaz, M. Resinas, and A. Ruiz-Cortes. 2018. Automated Validation of Compensable SLAs. *IEEE Transactions on Services Computing* (jan 2018), 1–1. https://doi.org/10.1109/TSC.2018.2885766

[18] Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino. 2018. An Analysis of Public REST Web Service APIs. *IEEE Transactions on Services Computing* (2018). https://doi.org/10.1109/TSC.2018.2847344

[19] Tien N. Nguyen, Anh Tuan Nguyen, Trong Nguyen, Thanh Nguyen, Hoan Anh Nguyen, Ngoc Tran, Hung Phan, and Linh Truong. 2018. Complementing global and local contexts in representing API descriptions to improve API retrieval tasks. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. ACM Press, New York, New York, USA, 551–562. https://doi.org/10.1145/3236024.3236036

[20] Rick Rabiser, Sam Guinea, Michael Vierhauser, Luciano Baresi, and Paul Grünbacher. 2017. A comparison framework for runtime monitoring approaches. *Journal of Systems and Software* 125 (3 2017), 309–321. https://doi.org/10.1016/j.jss.2016.12.034

[21] Anastasia Reinhardt, Tianyi Zhang, Mihir Mathur, and Miryung Kim. 2018. Augmenting Stack Overflow with API Usage Patterns Mined from GitHub. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 880–883. https://doi.org/10.1145/3236024.3264585

[22] W. Tan, Y. Fan, A. Ghoneim, M. A. Hossain, and S. Dustdar. 2016. From the Service-Oriented Architecture to the Web API Economy. *IEEE Internet Computing* 20, 4 (July 2016), 64–68. https://doi.org/10.1109/MIC.2016.74

[23] S. Tata, M. Mohamed, T. Sakairi, N. Mandagere, O. Anya, and H. Ludwig. 2016. rSLA: A Service Level Agreement Language for Cloud Services. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE, San Francisco, CA, USA, 415–422. https://doi.org/10.1109/CLOUD.2016.0062

[24] Rafael Brundo Uriarte, Francesco Tiezzi, and Rocco De Nicola. 2014. SLAC: A Formal Service-Level-Agreement Language for Cloud Computing. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC '14)*. IEEE Computer Society, Washington, DC, USA, 419–426. https://doi.org/10.1109/UCC.2014.53

Automating SLA-Driven API development with SLA4OAI

Published in the 17th International Conference on Service-Oriented Computing (ICSOC), Toulouse, France. October 2019.

Authors: *Antonio Gámez Díaz, Pablo Fernández Montes, Antonio Ruiz Cortés.*
DOI: *10.1007/978-3-030-33702-5_2.*
Rating: *GGS class 2 (A-).*

Automating SLA-Driven API Development with SLA4OAI

Antonio Gamez-Diaz(✉), Pablo Fernandez, and Antonio Ruiz-Cortes

Universidad de Sevilla, Seville, Spain
{antoniogamez,pablofm,aruiz}@us.es

Abstract. The OpenAPI Specification (OAS) is the *de facto* standard to describe RESTful APIs from a functional perspective. OAS has been a success due to its simple model and the wide ecosystem of tools supporting the SLA-Driven API development lifecycle. Unfortunately, the current OAS scope ignores crucial information for an API such as its Service Level Agreement (SLA). Therefore, in terms of description and management of non-functional information, the disadvantages of not having a standard include the vendor lock-in and prevent the ecosystem to grow and handle extra functional aspects.

In this paper, we present SLA4OAI, pioneering in extending OAS not only allowing the specification of SLAs, but also supporting some stages of the SLA-Driven API lifecycle with an open-source ecosystem. Finally, we validate our proposal having modeled 5488 limitations in 148 plans of 35 real-world APIs and show an initial interest from the industry with 600 and 1900 downloads and installs of the SLA Instrumentation Library and the SLA Engine.

1 Introduction

In the last decade, RESTful APIs are becoming a clear trend as composable elements that can be used to build and integrate software [7,18]. One of the key benefits this paradigm offers is a systematic approach to information modeling leveraged by a growing set of standardized tooling stack from both the perspective of the API consumer and the API provider.

Specifically, during the last years, the *OpenAPI Specification*[1] (OAS), formerly known as *Swagger* specification, has become the *de facto* standard to describe RESTful APIs from a functional perspective providing an ecosystem

[1] https://github.com/OAI/OpenAPI-Specification.

This work is partially supported by the European Commission (FEDER), the Spanish Government under projects BELI (TIN2015-70560-R) and HORATIO (RTI2018-101204-B-C21), and the FPU scholarship program, granted by the Spanish Ministry of Education, Culture and Sports (FPU15/02980).

S. Yangui et al. (Eds.): ICSOC 2019, LNCS 11895, pp. 20–35, 2019.
https://doi.org/10.1007/978-3-030-33702-5_2

that helps the developer in several aspects of the API development lifecycle[2]. As an example, from the API provider perspective, there are tools that aim to automate the server scaffolding, an interactive documentation portal creation or the generation of unit test cases; from the perspective of the consumer, there are tools to automate the creation of API clients, the security configuration or the endpoints discovery and usage [1,15,16].

However, as APIs are deployed and used in real settings, the need for non-functional aspects is becoming crucial. In particular, the adoption of Service Level Agreements (SLAs) [13] could be highly valuable to address significant challenges that the industry is facing, as they provide an explicit placeholder to state the guarantees and limitations that a provider offers to its consumers. For example, these limitations (such as *quotas* or *rates*) are present in most common industrial APIs [3] and both API providers and consumers need to handle how they monitor, enforce or respect them with the consequent impact in the API deployment/consumption.

In this paper, we address the challenge of SLA modeling and management in APIs by providing the following contributions:

– SLA4OAI, an open SLA specification that is integrated with the OpenAPI Specification joint with a Basic SLA Management Service (i.e., a minimum definition of endpoints required for the SLA enforcing in the APIs) that can be used to promote the vendor independence.
– A set of tools to support the different activities of the API development lifecycle when it becomes aware of the existence of an SLA.
– An initial validation over 5488 limitations in 35 of real-world APIs showing the expressiveness coverage and the potential evolution roadmap for the specification.

The rest of the paper is structured as follows: in Sect. 2, we describe the related work and motivate the need for our proposal. In Sect. 3 we describe in brief words the OpenAPI Specification focusing on its extension's capabilities. In Sect. 4 we describe our SLA4OAI model proposal. In Sect. 5 we show the ecosystem of tools that have been built around our proposal. In Sect. 6 we validate our proposal by modeling 5488 limitations in 35 of real-world APIs. Finally, in Sect. 7 we show some remarks and conclusions.

2 Motivation and Related Work

The software industry has embraced integration as a key challenge that should be addressed in multiple scenarios. In such a context, the proliferation of APIs is a reality that has been formally analyzed: in [14], authors performed an analysis of more than 500 publicly-available APIs to identify the different trends in the current industrial landscape. Specifically, regarding the *documentation*, there is a clear trend with respect to the functional description of the service: during

[2] https://openapi.tools.

the last years, the OpenAPI Specification has consolidated as a *de-facto* standard to define the different functional properties an API provides. For instance, in [12], authors study on the presence of dependency constraints among input parameters in web APIs in industry.

With such a consolidated market of APIs, non-functional aspects are also becoming a key element in the current landscape. In [3], authors analyze a set of the 69 real APIs in the industry to characterize the variability in its offerings, obtaining a number of valuable conclusions about real-world APIs, such as: (i) Most APIs provide different capabilities depending on the tier or plan of the API consumer is willing to pay. (ii) Usage limitations are a common aspect all APIs describe in their offerings. (iii) Limitations over API requests are the most common including quotas over static periods of times (e.g., *1.000 request each natural day*) and rates for dynamic periods of times (*3 request per second*). (iv) Offerings can include a wide number of metrics over other aspects of the API that can be domain-independent (such as the number of returned results or the size in bytes of the request) or domain-dependent (such as the CPU/RAM consumption during the request processing or the number of different resource types). Based on these conclusions, we identify the need for non-functional support in the API development life-cycle and the high level of expressiveness present in the API offerings.

From the perspective of the API development life-cycle, the lack of a standard spec for non-functional aspects integrated with existing standards OpenAPI, prevents the tooling ecosystem to grow and provide support advanced issues: as an example, to support the API consumer, it could be possible to develop tools to automate the generation of SLA-aware API clients able to self-adapt the request rate to the API limitations; to support the API provider, it could be possible to create of SLA-aware API testers enriching the habitual tests with information about limitations in order to analyze the actual performance capabilities to decide the maximum number of API consumers to be allowed with a certain SLA that explicitly states the limitations in their usage. We have analyzed the most prominent academic and industrial proposals that aim to the definition of SLAs in both traditional web services and cloud scenarios in order to outline their scope and limitations. Specifically, in Table 1, we have considered 7 aspects to analyze in each SLA proposal, namely: **F1** determines the format in which the document is written; **F2** shows whether the target domain is web services; **F3** indicates if it can model more than one offering (i.e., different operations of a web service); **F4** determines if it allows modeling hierarchical models or overriding properties and metrics; **F5** shows whether temporal concerns can be model (e.g., in metrics); **F6** indicates if there exists a tool for assisting users to model using this proposal; **F7** determines if there exists a tool/framework for enacting the SLA.

Based on this comparison of the different SLA models, we highlight the following conclusions: (i) None of the specifications provides any support or alignment with the OpenAPI Specification; (ii) Most of the approaches provide a concrete syntax on XML, RDF (some of them they even lack concrete syntax) and there is no explicit support to YAML or JSON serializations. (iii) An important number of proposals are complete, but others leave some parts open to being implemented by practitioners. (iv) Besides the fact that a number of proposals are that aims to model web services, they are focused on traditional SOAP web services rather than

Table 1. Analysis of SLA models

Name	F1	F2	F3	F4	F5	F6	F7
SLAC [19]	DSL					✓	✓
CSLA [9]	XML		✓			✓	
L-USDL Ag. [6]	RDF	✓	✓		†	✓	
rSLA [17]	Ruby	✓		✓	✓		✓
SLAng [10]	XML	✓					
WSLA [11]	XML	✓	✓		✓		
SLA* [8]	XML	✓	✓		✓		
WS-Ag. [2]	XML	✓	✓	✓	†		

† Supported with minor enhancements or modifications.

RESTful APIs. In this context, they do not address the modeling standardization of the RESTful approach: i.e., the concept of a resource is well unified (a URL), and the amount of operations is limited (to the HTTP methods, such as GET, POST, PUT and DELETE). This lack of support of the RESTful modeling prevents the approaches to have a concise and compact binding between functional and non-functional aspects. (v) They do not have enough expressiveness to model limitations such as quotas and rates, for each resource and method and with complete management of temporally (static/sliding time windows and periodicity) present in the typical industrial API SLAs. (vi) Most proposals are designed to model a single offering and they mostly lack support to modeling hierarchical models or overriding properties and metrics (F4); in such a context, they cannot model a set of tiers or plans that yield a complex offering that maintains the coherence by model and instead they rely on a manual process that is typically error-prone. (vii) finally, the ecosystem of tools proposed in each approach (in the case of its existence) is extremely limited and that aims to be solely as a prototype; moreover, they apparently are not integrated into a developer community nor there is evidence of this usage by practitioners in the industry.

In order to overcome the limitations of existing approaches, the main goals of this paper can be summarized as follows: (i) An interoperable model fully-integrated with leading API description language (OAS) to express the API limitations. (ii) an initial ecosystem of tools to provide support to different parts of the SLA-Driven API development lifecycle. (iii) validation of this model in real-world scenarios to assess its expressiveness.

3 OAS in a Nutshell

In this section, we briefly present the OpenAPI Specification (OAS), considering its goals, structure and extension capabilities. OAS, formerly known as Swagger, is a vendor-neutral, portable and open specification for the functional deception of APIs. It is promoted by the OpenAPI Initiative (OAI), an open source consortium hosted by The Linux Foundation and supported by a growing number of leading industry stakeholders, such as Google, IBM, Microsoft or

Oracle, amongst others. Both API clients and vendors are able to benefit from the formal definition using the OAS: from the clients' point of view, they can use any tool from the extensive ecosystem created around the OAI; conversely, from the vendors' point of view, they can generate interactive documentation portals, create auto-generated prototypes and perform automatic API monitoring and testing. Specifically, as a minimum content, an OAS document should describe a set of aspects including *API general information* (such as title, description and version), a list of *Resources*, *Paths* and *Methods* allowed, and set of *Schemas* (following the JSON-schema specification) to identify the structure of the data to be exchanged with the API (e.g., a resource structure). In order to have a more concise description, it is possible to reuse definitions of schemes by means of the *$ref* constructor as proposed in the JSON-schema standard. Complementary, API provider can include optional elements such as the different *API endpoints*, where the API can be accessed. This is especially useful in scenarios with different endpoints for development and production stages.

```
1  openapi: 3.0.0
2  info:
3   title: Simple petstore API
4   description: ...
5   version: ...
6   x-sla: ./pets-plans.yaml
7  servers:
8  - url: ....
9  paths:
10  /pets:
11   get:
12    description: ...
13    parameters: ..
14    responses:
15     200:
16      description: pet response
17      content:
18       application/json:
19        schema:
20         $ref: "#/components/schemas/pet"
21   post:
22    ...
23 components:
24  schemas:
25   pet:
26    title: pet model
27    ...
```

Listing 1.1. RESTful API in OAS

```
1  context:
2   id: plans
3   sla: '1.0'
4   type: plans
5   ...
6  infrastructure: ...
7  metrics:
8   requests:
9    type: integer
10   format: int64
11   description: #requests
12   resolution: consumption
13 ...
14 plans:
15  free:
16   pricing:
17    cost: 0
18    currency: USD
19    billing: monthly
20   quotas:
21    /pets:
22     post:
23      requests:
24       - max: 100
25        period: daily
26   rates:
27    /pets:
28     get:
29      requests:
30       - max: 2
31        period: secondly
32        scope: tenant
33  pro:
34   ...
```

Listing 1.2. SLA written in SLA4OAI

As an example, Listing 1.1 shows an OAS fragment from a basic RESTful API that corresponds with a single endpoint (*/pets*) and two methods. Lines 9–22 describe the definition of the *pet* resource including the *GET* and *POST* methods for retrieving and creating resources; specifically, line 11 starts modeling the *GET* method with a *description* and the *parameters* that the request might be able to handle and *responses* section (lines 14–20) describe the model of a successful HTTP response (i.e., status code *200*) returning a *pet* resource conforming with the appropriate schema reference (line 20). Finally, in lines 24–27, the data model (schema) of the pet object is being defined. A key feature of the OAS is the capability of being extended with the definition of custom properties starting with *x-*, paving the way for customizing or adding additional features according to specific business needs. As an example, line 6 shows the use of the *x-* extension point to include a reference to the SLA description of the API following our proposal (c.f., Sect. 4).

4 Our Proposal

4.1 SLA4OAI Language

SLA4OAI[3] is a language which provides a model for describing SLA in APIs in a vendor-neutral way by means of extending the main specification. This proposal is open for evolution based on the discussion with the community and other partners of the OpenAPI Initiative, hosted by the Linux Foundation. For the sake of completeness, always refer to the online version so as to have a complete reference of the language.

The figure available online[4] depicts an abstract syntax of an SLA4OAI description. Starting with the top-level placeholder (denoted as *SLA4OAI Document* in the figure) we can describe basic information about the *context*, the *infrastructure* endpoints that implement the Basic SLA Management Service, the *metrics* and a default value for *quotas*, *rates*, *guarantees* and *pricing*.

Context contains general information, such as the *id*, the *version*, the URL pointing to the *api* OAS document, the *type* and the *validity* of the document; in this context, the *type* field can be either *plans* or *instance* and it indicates whether the document corresponds with the general plan offering or it correspond with a specific SLA agreed with a given customer. The *Metrics* enables the definition of custom metrics which will be used to define the limitations, such as the number of requests, or the bandwidth used per request. For each metric, the *type*, *format*, *unit*, *description*, and *resolution* should be defined. The *Plan configuration* (configuration parameters for the service tailored for the plan), availability (availability of the service for this plan expressed via time slots using the ISO 8601 time intervals format), and the rest of the elements that will override the default with plan-specific values: quotas, rates and guarantees, pricing. In this context, it is important to highlight that the *Plan* section maps the structure in the OAS document to attach the specific limitations (quotas or rates) for each path and method. Specifically, after defining the *configuration*, the *availability*, *pricing*, *guarantees*, the limitations *quotas* and *rates* can be modeled; particularly, the limitations are described in the *Limit* with a *max* value

[3] https://sla4oai.specs.governify.io.
[4] https://isa-group.github.io/2019-05-sla4oai/files/sla4oai_diagram.png.

that can be accepted, a *period* (i.e., secondly, minutely, hourly, daily, monthly or yearly) and the *scope* where they should be enforced; as an extensible scope model, we propose two possible initial values (*tenant* or *account* as default) corresponding with a two-level structure: a limitation or guarantee with a tenant scope will be applicable to the whole organization while an account scope would be applicable to each specific user or account (typically with a different API key) in the organization.

Considering the features of the existing SLA proposals previously analyzed and available in the online appendix, SLA4OAI is a proposal serialized using the YAML/JSON syntax (F1) specifically designed for web services (F2), concretely, RESTful APIs. It is able to model one or more offerings (F3) in a hierarchical model (F4) since *plans* can override the default values for the limitations. Furthermore, our proposal takes into account the temporality (F5), since each limitation is scoped to a precise period of time and each plan has its own *availability* information. Finally, as stated in following sections, SLA4OAI has a set of tools for assisting users to write the model (F6) and an initial ecosystem of tools to support parts of the development lifecycle (F7).

Let us consider the aforementioned example (as modeled in Listing 1.1) to be extended with a basic SLA: as a provider, it would be useful to limit, on the one hand, the number of requests a consumer is allowed to make in a static window (quota) of 1 day depending on the plan purchased and, on the other hand, the requests allowed to be made in a sliding window (rate), differing from GET and POST methods to avoid the API saturation derived from abusive customers. Specifically, Listing 1.2 illustrates the model in SLA4OAI of the limitations of this example API: in lines 14–34 the *free* and *pro* plans are being modeled. Focusing on the first, line 15 define a specific *plan* by its limitations *quotas* (lines 20–25) and *rates* (lines 26–32). For instance, a quota of 100 POST requests over the resource */pets* in a static window of 1 day is defined in lines 23–25. Conversely, a rate of 2 requests per second is defined for */pets* GET requests (lines 29–32). Finally, note that line 4 indicates that this document is for describing *plans*. Whenever a client accepts a specific plan, *type* field would become an *instance* one. It is interesting to highlight the *scope: tenant* (line 32) in the rates for the GET request represents a limitation for the whole consumer organization affecting all the accounts of the organization, while the rest of the quotas and rates are enforced on a default per-account basis.

4.2 SLA-Driven API Development Lifecycle

In spite of the fact that each organization could address the API development lifecycle with slightly different approaches, a minimal set of activities can be identified: a first activity corresponds with the actual *Functional Development* of the API implementing and testing the logic; next a *Deployment* activity where the developed artifact is configured to be executed in a given infrastructure; finally, once the API is up and running, an *Operation* activity starts where the requests from consumers can be accepted. This process is a simplification that can be evolved to add intermediate steps (such as testing) or to include an evolutive cycle where different versions are deployed progressively. In order to incorporate SLAs in this process, we expand this basic lifecycle where both API Provider and API Consumer can interact (as depicted in Fig. 1).

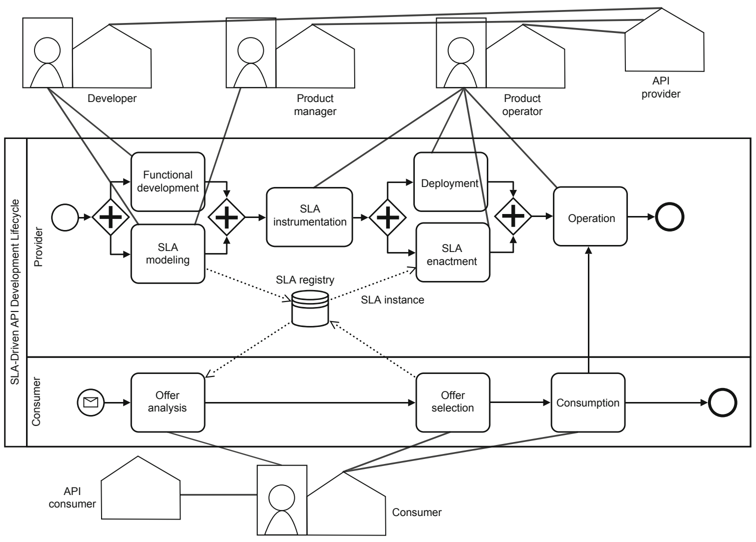


Fig. 1. SLA-Driven API development lifecycle

Specifically, from the provider's perspective, the *Functional Development* can be developed in parallel with a *SLA modelling* where the actual SLA offering (type *plans*) is written and stored in a given *SLA Registry*. Once both the functional development and the SLA modeling has concluded, the *SLA instrumentation* must be carried out, where the tools and/or developed artifacts are parameterized so they can adjust their behavior depending on a concrete SLA and provide the appropriate metrics to analyze the SLA status. Next, while the *deployment* of the API takes place, a parallel activity of *SLA enactment* is developed where the deployment infrastructure should be configured in order to be able to enforce the SLA before the API reaches the *operation* activity.

Complementary, from consumer's perspective, once the provider has published the SLA offering (i.e., *Plans*) in the *SLA Registry*, it starts the *offer analysis* to select the most appropriate option (*offer selection* activity) and to create and register its actual SLA (type *instance*); finally, the API *Consumption* is carried out as long as the API is the *Operation* activity and its regulated based on the terms (such as quotas or rates) defined in the SLA.

In order to implement this lifecycle, it is important to highlight that the *SLA instrumentation*, *SLA enactment* and *Operation* activities should be supported by an SLA enforcement protocol that aims to define the interactions for *checking* if the consumption of the API for a given consumer is allowed (e.g., it meets the limitations specified in its SLA) and to gather the actual values of the *metrics* from the different deployed artifacts that implement the API.

4.3 Basic SLA Management Service

The *Basic SLA Management Service* (BSMS) is a basic non-normative API description to provide basic support for the SLA enforcing protocol as motivated in the SLA-Driven API development lifecycle (c.f., Sect. 4.2) and addresses the following features: (i) Checking the current state of a given SLA (SLA Check). (ii) Reporting metrics to calculate the current state of a given SLA (SLA Metrics). To this end, this BSMS proposal represents a descriptive interface that could be implemented in different technologies and acts as a decoupling mechanism to the underlying infrastructure that actually provides support to the development lifecycle.

Moreover, the definition of a BSMS paves the way to define multiple SLA enforcing architectures that could be selected depending on the performance or technological constraints of a given scenario. Specifically, Figs. 2 and 3 represent an overview of two different SLA enforcing architectures: on the one hand, the *Standalone* enforcing define an SLA instrumentation as part of the API with a direct communication with the SLA management infrastructure; on the other hand, a *Gateway* enforcing relays on the front load balancer to connect with the SLA management infrastructure so a potential set of API instances do only provide the functional logic.

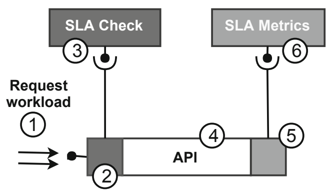


Fig. 2. Standalone SLA enforcing arch.

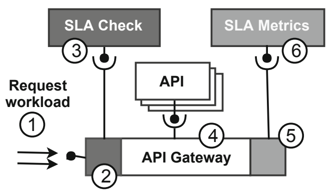


Fig. 3. Gateway SLA enforcing arch.

In order to illustrate the interactions and behavior of each component implementing (or interacting with) the BSMS, we will focus on the *Gateway* enforcing architecture (See Fig. 3) as it is a more complete scenario:

1. Requests will pass through the API Gateway until they are directed to the node that will serve it (step 1).
2. The API Gateway query the SLA Check component to determine if the request is authorized to develop the actual operation based on the appropriate SLA (step 2).
   (a) If it is authorized, the actual API is invoked and the response is returned (step 3).
   (b) If it is not authorized, a status code and a summary of the reason (as generated by the SLA check component) is returned (step 3).
3. After the consumption ends (step 4), the metrics are sent to the SLA Metrics component (step 5). This component is in charge of updating the status of the agreement with the new metrics introduced (step 6). This new information

could be processed to determine the SLA state that should be taken into account in further requests.

In the following subsections, we overview the interface and the expected behavior of the SLA Check and SLA Monitor components; a complete description of the proposed API is available online[5].

SLA Check. This component should support the verification process to decide whether an API request can be satisfied based on the current state of its SLA. In particular, it should provide two different endpoints:

- A query (*GET*) operation over the */tenants* path in order to locate the SLA scope and the SLA id that should regulate the consumption based on a given token (typically an API key sent by the consumer as a query or header parameter). The SLA scope should determine the actual tenant (the consumer organization that has signed the SLA) and the account (that belongs to the consumer organization).
- A verification (*POST*) operation over the */check* path in order verify whether a specific request can be done; specifically, it will respond true or false to notify the provider if it is: (i) Acceptable to fulfill the request (positive case), or on the contrary; (ii) Not acceptable and then, the request should be denied (negative case); in such a case, it could include optional information describing the reason for the SLA violation. Concerning the HTTP status code, in a general case, a negative response should correspond with standard *403 Forbidden*; if the denial reason is rate/quota limit enforcement, then the recommendation is to use *429 Too Many Requests* and include rate limit information as metadata into the consumer response to explain the denial of service: as an example it could include the actual metric computation, the limit or a future timestamp when the rate/quota will be reset for the given consumer.

It is important to note that, while a complete interaction with the SLA Check component involves the invocation to both endpoints, in demanding scenarios, a local API key cache can be introduced in order to avoid the first query over the*/tenants* path.

SLA Metrics. This component should implement a mechanism for metric gathering in order to support the analysis of SLA fulfillment. In particular, it should provide a storage (*POST*) operation over the */metric* path in order to register a certain metric. In addition to the actual metric value, as mandatory elements, it should also include information about the metric context including the *SLA Scope*, the *SLA Id* and the *sender* (i.e., the specific API instance or API Gateway generating the metric).

The metrics can correspond with a standard set of well-defined domain-independent metrics such as *request count* or *response time*, or domain-dependent metrics such as a certain payload attribute (e.g., the size of a specific parameter).

[5] https://sla4oai.specs.governify.io/operationalServices.html.

Since metrics flow could be dense in the same scenarios a buffering can be introduced; to this respect, the SLA Metric component should allow reception of multiple metrics values in a single operation. Consequently, metrics can be grouped in batches or sent one by one to fine-tune performance versus real-time SLA tracking in each scenario.

5 Tool Support

The SLA-Driven API development lifecycle, depicted in Fig. 1 and explained in Sect. 4.2, should be assisted by a set of tools during certain activities. Since we seek to provide a fully-fledged language, we provide an initial working implementation of these tools [4]. Specifically, for the *SLA modeling* activity we present the *SLA Editor* for hiding the complexity of the language to the end user. The concrete implementation of the *SLA instrumentation* activity is provided in the *SLA Engine*, an implementation of the *Basic SLA Management Service*, defining the */metrics* and */check* endpoints. On the one hand, for the *Standalone* SLA enforcing architecture, we support the *SLA instrumentation* and *SLA enactment* activities with the *SLA Instrumentation Library* in a Node.js module; on the other hand, for the *Gateway* SLA enforcing architecture, a complete *SLA-Driven API Gateway* is provided as a service.

SLA Editor. In modeling tasks, supporting tools are commonly provided to the users. In this scenario, we provide the *SLA editor*[6], for the *SLA modeling* activity in the SLA-Driven API development lifecycle. *SLA editor* is a user-friendly and web-based text editor specifically developed for assisting the user during the modeling tasks, including auto-completion, syntax checking, and automatic binding. It is possible to create plans (e.g., free and pro) with quotas and rates. Clicking on the + sign, the user is able to select the path and method (previously defined in the OAS document) for entering the value of the limitation. Note that custom metrics can also be defined at the bottom, however, the calculation logic is left open for a specific implementation.

SLA Engine. Whereas the BSMS (c.f., Sect. 4.3 defines the interaction flows and the endpoints */check* and */metric*, a reference implementation should be provided in order to properly carry out the *SLA instrumentation* activity in the SLA-Driven API development lifecycle. The *SLA Engine*, thus, provides a concrete implementation which also includes a particular way to handle SLA saving/retrieving tasks. Specifically, *Monitor*[7] is an implementation of the *Metrics* BSMS service and *Supervisor*[8], of the *Check* service.

The *Monitor* service exposes a POST operation in the route */metrics* for gathering the metrics collected from other different services. It can collect a

[6] https://designer.governify.io.
[7] http://monitor.oai.governify.io/api/v1/docs.
[8] http://supervisor.oai.governify.io/api/v1/docs.

set of basic metrics and send them to a data store for aggregation and later consumption. The metrics can be grouped in batches or sent one by one to fine-tune performance versus real-time SLA tracking.

The *Supervisor* service has a POST */check* endpoint for the verification of the current state of the SLA for a given operation in a certain scope. For each request, this service will evaluate the state of the SLA and will respond with a positive or negative response depending on whether a limitation has been overcome. In addition, this service also implements (outside the scope of the BSMS) these additional endpoints: GET/POST */tenants*, GET/POST */slas* and PUT/DELETE *slas/<id>* for managing both users (tenants and accounts) and SLA4OAI documents themselves.

SLA Instrumentation Library. Despite the fact that the BSMS defines the interaction flows between the endpoints, the concrete implementation of these interactions is left open for the activities of *SLA instrumentation* and *SLA enactment* of the SLA-Driven API development lifecycle. The tool that we present aims to cover this lack in the *Standalone* SLA enforcing architectures. Specifically, we present an SLA Instrumentation Library for Node.js[9], which is a middleware (i.e., a filter that intercepts the HTTP requests and perform transformation if necessary) written for Express, the most used Node.js web application framework. This middleware intercepts all the inbound/outbound traffic to perform the BSMS flow.

Specifically, *Monitor* is an implementation of the *Metrics* BSMS service and *Supervisor*, of the *Check* service, as explained in the SLA Engine section.

Once the API uses the SLA Instrumentation Library, a new endpoint */plans* is added. It creates a provisioning portal for clients to purchase a plan. Once the customer purchases (or simply selects, in case of the free ones) a plan, this customer will get an API-key, acting as a bearer token for HTTP authentication.

SLA-Driven API Gateway. A more transparent way to implement the interaction flows defined is the BSMS is achieved by using an *SLA-Driven API Gateway*[10]. We provide an open-source implementation for deploying SLA-Driven API Gateways using any *SLA Engine* and supporting the *SLA instrumentation* and *SLA enactment* activities of the SLA-Driven API development lifecycle in a *Gateway* SLA enforcing architecture.

Particularly, we provide as a service, an online preconfigured instance (using the aforementioned SLA Instrumentation Library) of an SLA-Driven API Gateway. API providers are only required to enter: (i) The real endpoint of their API; (ii) A URL pointing to the SLA4OAI document. Once an API is registered, the SLA-Driven API Gateway exposes a public and SLA-regulated endpoint, as well as the */plans* endpoint for the provisioning portal. Clients who have selected a plan will get an API-key from the portal that will be as a bearer token to consume the SLA-regulated API.

[9] https://www.npmjs.com/package/sla4oai-tools.
[10] https://gateway.oai.governify.io.

6 Validation

In this section, we describe how we have evaluated our proposal. In particular, the goal of the evaluation was to answer the following research questions:

RQ1: How expressive is our SLA4OAI model in comparison to real-world APIs' SLAs We want to know whether the SLA4OAI model that we use is expressive enough to model a wide variety of real-world SLAs and which are the characteristics of the SLAs that we are not able to express.

RQ2: Which difficulties appear when modeling SLAs defined are expressed in natural language? All real-world APIs' SLAs are expressed in natural language. Therefore, before checking their limitations, it is necessary to formalize them. With this question, we examine the problems that may appear in this step.

RQ3: What is the reception of our SLA4OAI model and tools in the community? Besides this proposal has not been officially published, it is publicly available in our code and artifact repositories (such as NPM). We wonder whether our proposal is being used by a set of external users and how large this set is.

RQ1: Expressiveness of SLA4OAI. To evaluate the expressiveness of the SLA4OAI proposal, we have modeled the limitations of a set of APIs. For selecting this set we considered the work of [3], where the authors analyzed a set of 69 APIs from two of the largest API directories, Mashape (now integrated into RapidAPI) and ProgrammableWeb, studying 27 and 41 respectively.

For our evaluation, we have manually selected a subset of these APIs, giving, as a result, a number of 35 APIs whose modeling using SLA4OAI is challenging (i.e., the 27 ones from RapidAPI have the same expressiveness, as the authors noted). Specifically, have modeled 5488 limitations (quotas/rates) over 7055 combinations of metrics (e.g., number of requests) and periods (e.g., secondly, monthly) in 148 plans of 35 real-world APIs. We provide a workspace[11] with the 35 modeled APIs and the statistical analysis that we have performed. Focusing on these limitations, the quotas use to be defined over custom metrics based on their business logic (e.g., credits spent by request, the number of returned results or the storage consumed). On the other hand, rates are mostly defined over the number of requests. In both cases, APIs usually define their limitations over one or two different metrics. Finally, regarding the periods, both limitations are usually over just one period: *monthly* for quotas, and *secondly* for rates.

RQ2: Modeling Issues. During the modeling process we have noticed a few issues, namely: (i) When an overage exists (i.e., one can overcome the limitation value by paying an extra amount of money per request), the quotas are *soft*, that is, the service is still accessible, but this situation should be taken into account. (ii) Sometimes plans in real APIs are the result of an aggregation of other plans. For instance, one can buy a *base plan* with N requests/s, but, purchasing an upgrade, it is possible to reach the N+1 requests/s. (iii) Using

[11] https://isa-group.github.io/2019-05-sla4oai.

more than one period for limitations. For instance, (1000 requests/month and 100 requests/week). Despite the fact that it is supported in SLA4OAI, it is not present in the current reference implementation. (iv) Some limitations use a custom period by means of defining the amount and unit, for example, every 5 min, every 2.5 months, etc. (v) In a few APIs, especially for trial plans, *forever* periods are often used.

RQ3: SLA4OAI Interest in the Community. Despite the SLA4OAI extension and tools have not been widely announced nor promoted, we have disclosed the tooling ecosystem into the main public NodeJS artifact repository (i.e., NPM) and this platform provides a set of analytics, refering to individual installations[12], of the usage since it was published. Specifically, based on its data, it is observed that the SLA Instrumentation Library has been downloaded and installed more than 600 times[13] while the SLA Engine was downloaded more installed than 1900 times. Furthermore, several industry members of the Open API Initiative (including Google or PayPal) have expressed their interest in this proposal and to promote a working group for evolving and extending the SLA4OAI proposal [5].

7 Conclusions

The current *de facto* standard for modeling functional aspects of RESTful APIs, the OpenAPI Specification, ignore crucial non-functional information for an API such as its Service Level Agreement (SLA). This lack of a standard to define the non-functional aspects leads to vendor lock-in and it prevents the open tool ecosystem to grow and handle extra functional aspects. In this paper, we pioneer in extending OAS to define a specific model for SLAs description and we provide an initial set of open-source tools that leverage the pre-existing OAI ecosystem in order to automate some stages of the SLA-Driven API lifecycle. Our proposal has been validated in terms of expressivity in 35 real-world APIs and, in spite of the lack of promotion, the initial metrics of usage of the tools proof an interest from the industry.

As future work, the modeling issues identified in Sect. 6 spot the potential improvements of SLA4OAI specification and the ecosystem of tools, namely: (i) Incorporate the concept of *hard*/*soft* limitation types. (ii) Add the definition of custom periods, rather than limiting them to a fixed set of values. (iii) Design a process for creating composite plans on the top of simpler ones. (iv) Improve the reference implementation of the tools to support more than one period in each limitation. From a community perspective, based on the interest received in the industry, we are in the process of creating an official working group for the industrial members in OAI to incorporate more feedback from the industry and

[12] Details about how this calculation is being made is available at http://bit.ly/npm-calculation.

[13] https://npm-stat.com/charts.html?package=sla4oai-tools.

define a coordinated mechanism of evolution for future versions of the current SLA4OAI proposal.

References

1. Acharya, M., Xie, T., Pei, J., Xu, J.: Mining API patterns as partial orders from source code. In: ESEC-FSE 2007, p. 25. ACM Press, New York (2007)
2. Andrieux, A., et al.: Web Services Agreement Specification (WS-Agreement) (2004)
3. Gamez-Diaz, A., Fernandez, P., Ruiz-Cortes, A.: An analysis of RESTful APIs offerings in the industry. In: Maximilien, M., Vallecillo, A., Wang, J., Oriol, M. (eds.) ICSOC 2017. LNCS, vol. 10601, pp. 589–604. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69035-3_43
4. Gamez-Diaz, A., Fernandez, P., Ruiz-Cortes, A.: Governify for APIs: SLA-Driven ecosystem for API governance. In: ESEC-FSE 2019. ESEC/FSE 2019, Tallin, Estonia. ACM (2019)
5. Gamez-Diaz, A., et al.: The role of limitations and SLAs in the API industry. In: ESEC-FSE 2019. ESEC/FSE 2019, Tallin, Estonia. ACM (2019)
6. Garcia, J.M., Fernandez, P., Pedrinaci, C., Resinas, M., Cardoso, J., Ruiz-Cortes, A.: Modeling service level agreements with linked USDL agreement. IEEE TSC **10**(1), 52–65 (2017)
7. Harms, H., Rogowski, C., Lo Iacono, L.: Guidelines for adopting frontend architectures and patterns in microservices-based systems. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 902–907 (2017)
8. Kearney, K.T., Torelli, F., Kotsokalis, C.: SLA*: an abstract syntax for service level agreements. In: GRID, pp. 217–224. IEEE, October 2010
9. Kouki, Y., Alvares de Oliveira, F., Dupont, S., Ledoux, T.: A language support for cloud elasticity management. In: CCGrid 2014, pp. 206–215. IEEE, May 2014
10. Lamanna, D.D., Skene, J., Emmerich, W.: SLAng: a language for defining service level agreements. In: FTDCS, pp. 100–106, January 2003
11. Ludwig, H., Keller, A., Dan, A., King, R.: A service level agreement language for dynamic electronic services. In: WECWIS 2002, pp. 25–32. IEEE Computer Society (2002)
12. Martin-Lopez, A., Segura, S., Ruiz-Cortes, A.: A catalogue of inter-parameter dependencies in restful web APIs. In: Yangui, S., et al. (eds.) ICSOC 2019. LNCS, vol. 11895, pp. 399–414. Springer, Cham (2019)
13. Muller, C., Gutierrez Fernandez, A.M., Fernandez, P., Martin-Diaz, O., Resinas, M., Ruiz-Cortes, A.: Automated validation of compensable SLAs. IEEE TSC, 1 (2018)
14. Neumann, A., Laranjeiro, N., Bernardino, J.: An analysis of public REST web service APIs. IEEE TSC, 1 (2018)
15. Nguyen, T.N., et al.: Complementing global and local contexts in representing API descriptions to improve API retrieval tasks. In: ESEC/FSE 2018, pp. 551–562. ACM Press, New York (2018)
16. Reinhardt, A., Zhang, T., Mathur, M., Kim, M.: Augmenting stack overflow with API usage patterns mined from GitHub. In: ESEC/FSE 2018, pp. 880–883 (2018)
17. Tata, S., Mohamed, M., Sakairi, T., Mandagere, N., Anya, O., Ludwiga, H.: RSLA: a service level agreement language for cloud services. In: CLOUD, pp. 415–422, June 2017

18. Thomas Fielding, R.: Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California, Irvine (2000)
19. Uriarte, R.B., Tiezzi, F., De Nicola, R.: SLAC: a formal service-level-agreement language for cloud computing. In: UCC, pp. 419–426. IEEE, December 2014

Part III

FINAL REMARKS

Final Remarks

This chapter concludes this dissertation by highlighting the main conclusions and future work. Specifically, Section §7.1 details the conclusions as well as the limitations of our proposal. Next, Section §7.2 discusses the possible extensions to this work, new research interests and collaborations to be explored in the future.

7.1 Conclusions

This thesis dissertation focuses on the problem of defining an *expressive, fully-fledged specification of SLAs for RESTful APIs* with two main challenges: (CH1) Establish a sufficiently expressive specification for the description of API pricings and the analysis of their validity; (CH2) Implement an ecosystem of tools and operations to support the SLA-Driven governance of RESTful APIs. As a result of the herein accomplished work, the main thesis' output is:

Main conclusion

An ***expressive, fully-fledged specification of SLAs for RESTful APIs*** *endorsed with an* ***open ecosystem of tools*** *can be created to support the* ***SLA-Driven Governance for RESTful systems***

7.1.1 Discussion of Results

To this end, we, first of all, had to analyze the state of the art on API: plans, SLAs, pricing, etc. To this end, we performed a systematic analysis of 69 RESTful APIs as part of their business model (Chapter §4). It paved the way to identify the requirements for the creation of an expressive *governance* model of realistic RESTful APIs.

Later, we continued analyzing the landscape aimed at finding a catalog of relevant concepts based on different perspectives from both industry and academia by joining forces with representatives from top-level organizations belonging to the OpenAPI Consortium (Chapter §5).

Then, we presented SLA4OAI, pioneering in extending the OpenAPI Specification not only allowing the specification of SLAs (Chapter §6), but also supporting some stages of the SLA-Driven API lifecycle with an open-source ecosystem of tools, Governify4APIs (Appendix §B).

Finally, we also started leveraging from this SLA4OAI specification by the development of ELeCTRA, a tool to automate the analysis of induced usage limitations in an API, derived from its usage of external APIs (Appendix §A).

7.1.2 Limitations

The work herein presented is subject to numerous improvements and extensions in either the proposed model and the implemented tools. We will address them from a threefold perspective: descriptive, formal and methodology, as Broy identifies in [37].

In the descriptive perspective, SLA4OAI provides an expressive, fully-fledged specification of SLAs for RESTful APIs. To this end, we have tried to cover as many realistic use cases as possible, by exploring industrial API offerings and validating the proposal in certain scenarios. Notwithstanding, the methods used to analyze and capture the expressiveness in real APIs are error-prone and we may have missed some interesting variability. Besides, as RESTful APIs are becoming so popular nowadays, newer APIs could have defined more richer pricings that our model is not able to represent. Additionally, a well-known limitation of our model is the multitenancy issue: currently, the analysis operations do not consider more than one user, since they are aimed at answering questions with regards to the pricing plan as a whole (i.e., *is the plan valid?* but no *is the plan valid for 30K users?*.

In the formal perspective, our model has been defined with rigor aimed at capturing the plurality of business and pricing models for defining the nature of an API limitation. However, we have not closely dug into any formal semantics or formal specification. We came to the decision of putting aside the strictly formal perspective in favor of sticking close to industrial standards. This perspective, therefore, is the one that should be more extensively improved.

In the methodology perspective, it also extends to the ecosystem of tools and the ulterior validation, there is a long way to go. Even if we have worked in providing a wide set of tools, some of them being actively used in the community, the vast majority still are just proofs of concept. We are currently working on evolving SLA4OAI so that it becomes part of the OpenAPI Initiative; this will be the main scenario for providing and validating more methodology aspects.

7.2 Future Work

This thesis dissertation has opened new research challenges that can be explored in the future, including:

- Considering temporality in our SLA4OAI model: for instance, in [38], the authors distinguish five types of incoming workloads: *static*, *periodic*, *once-in-a-lifetime*, *unpredictable*, and *continuously changing*. If we introduce the concept of temporality in the pricing, i.e., to consider that certain plans have a determined temporal validity (e.g., day/night plan), the operations have to be adapted to consider this temporality. Joining temporality with workload models, one could automate the management of this type of advanced scenarios which require infrastructures that are dynamic (e.g., instances that start or stop and have a variable cost).

- Taking multitenancy into account: a limitation of our model is that our analysis operators are scoped for a single user. Once we add the concept of consumption scenarios we will able to deal with new operations, such *is this plan valid for periodic workload with peaks of 30K users?*, *can we guarantee that our plan is valid 99% of times?*. This research path will focus on the paramount importance of the SLA classical concept.

- Adding more dimensions in the model: we have considered continuing working in giving a unified and comprehensive description model for RESTful APIs, combining structural, conversational, and SLAs. This path was only explored in [27].

- Automated discovery/validation of RESTful APIs usage plans: once we have the SLA4AOI model, we can implement a tool that monitors a given set of API resources to determine which number of metric units (e.g., incoming requests) the service is able to provide. It would be useful to verify if the pricing plans being offered by the providers can, indeed, be fulfilled.

- Continue validating the proposal in industrial environments: we will continue fostering the adoption of the SLA4OAI model as part of the OpenAPI Initiative and other similar forums in the open-source community.

Part IV

APPENDICES

ELeCTRA: Induced Usage Limitations Calculation in RESTful APIs

Published in the 16th International Conference on Service-Oriented Computing - Workshops (ICSOC), Hangzhou, China. October 2018.

***Authors**: Antonio Gámez Díaz, Pablo Fernández Montes, Antonio Ruiz Cortés.*

***DOI**: 10.1007/978-3-030-17642-6_39.*

ELeCTRA: Induced Usage Limitations Calculation in RESTful APIs

Antonio Gamez-Diaz[1](✉), Pablo Fernandez[1], Cesare Pautasso[2], Ana Ivanchikj[2], and Antonio Ruiz-Cortes[1]

[1] Universidad de Sevilla, Seville, Spain
{agamez2,pablofm,aruiz}@us.es

[2] Software Institute, Faculty of Informatics, USI Lugano, Lugano, Switzerland
{cesare.pautasso,ana.ivanchikj}@usi.ch

Abstract. As software architecture design is evolving to microservice paradigms, RESTful APIs become the building blocks of applications. In such a scenario, a growing market of APIs is proliferating and developers face the challenges to take advantage of this reality. For example, third-party APIs typically define different usage limitations depending on the purchased Service Level Agreement (SLA) and, consequently, performing a manual analysis of external APIs and their impact in a microservice architecture is a complex and tedious task. In this demonstration paper, we present ELeCTRA, a tool to automate the analysis of induced usage limitations in an API, derived from its usage of external APIs. This tool takes the structural, conversational and SLA specifications of the API, generates a visual dependency graph and translates the problem into a *constraint satisfaction optimization problem* (CSOP) to obtain the optimal usage limitations.

1 Motivation

In recent years, there has been a clear trend towards the micro-service architectural style where each component (i.e. micro-service) can evolve, scale and get deployed independently. This style increases the flexibility of the system and has been applied in demanding web applications such as eBay, Amazon or Netflix.

From an engineering perspective, a key element of these architectures, with respect to the modeling and implementation of microservices, is the use of the RESTful paradigm. From a business perspective, this microservice architecture tendency represents a recent shift in software engineering towards API-driven building and consumption, fostering a breeding landscape for RESTful API market in which composite service providers face new issues, such as, how to set their usage limitations accordingly to 3rd party providers' usage limitations [6].

This work has been partially supported by the European Commission (FEDER), the Spanish and the Andalusian R&D&I programs (grants TIN2015-70560-R (BELI), P12–TIC–1867 (COPAS) and TIN2014-53986-REDT (RCIS)) the FPU scholarship program, granted by the Spanish Ministry of Education, Culture and Sports (FPU15/02980).

X. Liu et al. (Eds.): ICSOC 2018 Workshops, LNCS 11434, pp. 435–438, 2019.
https://doi.org/10.1007/978-3-030-17642-6_39

Automatic discovery of usage limitations requires the serialization of developers' knowledge regarding the SLA usage limitations based on the API structure and the implemented RESTful conversation [8]. This automatic discovery becomes crucial when there are frequent changes in the usage limitations of the external services or when the microservice architecture itself experiences changes. In this automatic discovery of usage limitations scenario, a key concept is the *boundary service* that corresponds to an API that has the closest interaction with the end-user by means of actions in the GUI. Specifically, each action in the GUI triggers what we coin as *root operations* (composed by a given path and an HTTP method) that are typically related to the user story. Commonly, as a facade, all the boundary services are deployed behind an API Gateway [7] that enhances the security of the infrastructure.

Given the frequent and multiple ways in which the third party APIs' usage limitations evolve and are likely to keep evolving, when building an application, we need to adapt our customer's expectations regarding the performance of the tool (i.e., how many operations can be generated over time and how long it will take to produce an operation). Therefore, to rapidly react to these changes, there is a need of automating the process to obtain the usage limitations that a provider can offer to its end-users based on certain optimal criteria in a microservice architecture, such as minimizing the requests made to the third-party providers. Specifically, the problem addressed is finding the usage limitations of a root operation induced by the rest of the services (internal and external) composing the microservice architecture. Despite the fact that knowing the usage limitations in the internal services can be useful, calculating the usage limitations in the root operations (i.e., the operations closer to the end users) is more valuable since they help the service provider to set its own usage limitations to end users and to understand the induced service level agreement he can offer to its users.

In this demonstration paper, we propose a tool that, given: (i) the external usage limitations (as derived from the purchased SLA) as well as the boundary service structure and root operation conversation; it translates the problem into a *constraint satisfaction optimization problem* (CSOP) to obtain the induced usage limitations for the specified root operation.

2 Using ELeCTRA to Calculate the Usage Limitations

In order to illustrate the problem, let us consider the following example: we have developed a report generation application for the internal evaluation of the researchers in our university. For this purpose, we needed to collect, per researcher, the quality indicators of his/her publications. After considering different bibliometric providers, we opted for the Scopus API [4] as it provides the set of publications as well as the CitesScore$^{\text{TM}}$ index, an impact factor index calculated by Elsevier. Specifically, our application depends upon four Scopus APIs. Per researcher, the following invocations are needed: (i) retrieving the simple list of publications; (ii) collecting the details regarding each publication; (iii) gathering the details concerning each publication's venue (journal or conference).

Each of these Scopus APIs present different usage limitations [6], so, depending on the input and the usage limitations of each API operation, the induced usage limitations for our end-users will vary. The question to be addressed is: *what is the maximum amount of researchers per week for which this report can be generated without over-passing the Scopus API usage limitations?*

We use ELeCTRA[1] to answer the question. First, we have to model the structural viewpoint of each Scopus RESTful service using the Open API Specification (OAS) [3], as well as the usage limitations by using the SLA4OAI Specification [5]. Each of these models should be publicly available through an URL. Finally, using the embedded editor in ELeCTRA, the x-conversation model is introduced to define the dependencies between the internal and external services, as well as the parameters needed to calculate automatically the usage limitations. This example is preloaded at ELeCTRA and can be accessed by selecting the *simple example*. ELeCTRA is composed of two different microservices (ELeCTRA-DOT and ELeCTRA-CSOP) and a user interface (ELeCTRA-UI) accessible through a web browser. This UI includes a textual editor so that a user can directly modify the *x-conversation* document and visualize the *OAS* and *SLA4OAI* models. When the user saves the model, two actions are carried out. On the one hand, an invocation to ELeCTRA-DOT, the graphical representation microservice, is performed. The *x-conversation* model is parsed and according to a set of rules, it is converted to a graph using the *dot* [1] notation. A png image is returned back to the UI. On the other hand, a request to ELeCTRA-CSOP, the CSOP model generator and solver microservice, is developed in order to calculate the usage limitations for the boundary operation.

ELeCTRA-DOT first represents all the RESTful requests (i.e., each pair of method and path) as the nodes of the graph by using the structural information present in the OAS model. Then, an edge between two operations is included if they are related in the *x-conversation* model. Next, this edge is labeled with the information about the number of invocations based on the x-conversation mode. Finally, for each operation of each service, the usage limitations information is retrieved by means of the SLA4OAI extension. ELeCTRA-CSOP, on the other hand, is the main microservice in this tool, since it is responsible for transforming the *x-conversation* model to a constraint satisfaction optimization problem in order to obtain the usage limitations for an operation. By using a set of rule constructs, the *x-conversation* is transformed into a set of parameters (e.g., the values of the usage limitations, the number of invocations from an operation to the next one), variables (e.g., the quota of the boundary operation to be maximized) and constraints (e.g., the sum of the requests made to a certain operation should not overpass the quota value for this operation). Specifically, this microservice uses the MiniZinc's [2] syntax, a language designed for specifying constrained optimization and decision problems over integers and real numbers. Once the problem has been successfully translated and validated, ELeCTRA-CSOP invokes the solver through the MiniZinc interface and gets the response back. Then, ELeCTRA-UI

[1] https://electra.governify.io.

receives the information and sends the updated usage plans to ELeCTRA-DOT to complete the remaining usage limitations. Finally, all this information is presented to the end user.

3 Instructions for the Organizers

This software tool is bundled into a v10.6.0 Node.js application, running in a Windows 10 system which should have installed (and added to the user PATH environment variable) Minizinc v2.1.7 and Graphviz v2.38.0.

Nevertheless, for the sake of simplicity and portability, authors have packed the entire app into a Docker Image[2]. Therefore, any x86_64 machine with Docker[3] will be able to run the application by typing: `docker run -p 8080:80 isagroup/governify-electra`.

Furthermore, to avoid any installation issue, the online web application is deployed at: https://electra.governify.io.

The demonstration video is available at: http://youtu.be/axbkDax1N9g.

References

1. Graphviz. https://graphviz.gitlab.io/_pages/doc/info/lang.html
2. Minizinc. http://www.minizinc.org/downloads/doc-latest/minizinc-tute.pdf
3. Open API Specification. https://www.openapis.org
4. Scopus API. https://dev.elsevier.com/api_docs.html
5. SLA4OAI Specification. https://github.com/isa-group/SLA4OAI-Specification
6. Gamez-Diaz, A., Fernandez, P., Ruiz-Cortes, A.: An analysis of RESTful APIs offerings in the industry. In: Maximilien, M., Vallecillo, A., Wang, J., Oriol, M. (eds.) ICSOC 2017. LNCS, vol. 10601, pp. 589–604. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69035-3_43
7. Gámez-Díaz, A., Fernández-Montes, P., Ruiz-Cortés, A.: Towards SLA-driven API gateways. In: JCIS (2015)
8. Ivanchikj, A., Pautasso, C., Schreier, S.: Visual modeling of RESTful conversations with RESTalk. J. Softw. Syst. Model. **17**(3), 1031–1051 (2018). https://doi.org/10.1007/s10270-016-0532-2

[2] https://hub.docker.com/r/isagroup/governify-electra.
[3] https://docs.docker.com/install/.

Governify for APIs: SLA-Driven Ecosystem for API Governance

Published in the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Tallinn, Estonia. August 2019.

Authors: *Antonio Gámez Díaz, Pablo Fernández Montes, Antonio Ruiz Cortés.*
DOI: *10.1145/3338906.3341176.*
Rating: *GGS class 1 (A+).*

Governify for APIs: SLA-Driven Ecosystem for API Governance

Antonio Gamez-Diaz
Universidad de Sevilla
Seville, Spain
antoniogamez@us.es

Pablo Fernandez
Universidad de Sevilla
Seville, Spain
pablofm@us.es

Antonio Ruiz-Cortés
Universidad de Sevilla
Seville, Spain
aruiz@us.es

ABSTRACT

As software architecture design is evolving to a microservice paradigm, RESTful APIs are being established as the preferred choice to build applications. In such a scenario, there is a shift towards a growing market of APIs where providers offer different service levels with tailored limitations typically based on the cost. In such a context, while there are well-established standards to describe the functional elements of APIs (such as the OpenAPI Specification), having a standard model for Service Level Agreements (SLAs) for APIs may boost an open ecosystem of tools that would represent an improvement for the industry by automating certain tasks during the development.

In this paper, we introduce *Governify for APIs*, an ecosystem of tools aimed to support the user during the SLA-Driven RESTful APIs' development process. Namely, an *SLA Editor*, an *SLA Engine* and an *SLA Instrumentation Library*. We also present a fully operational *SLA-Driven API Gateway* built on the top of our ecosystem of tools. To evaluate our proposal, we used three sources for gathering validation feedback: industry, teaching and research.

- **Website**: links.governify.io/link/GovernifyForAPIs
- **Video**: links.governify.io/link/GovernifyForAPIsVideo

CCS CONCEPTS

• **Information systems → RESTful web services**; • **Software and its engineering → Extra-functional properties**; **System description languages**.

KEYWORDS

RESTful APIs, SLA, OpenAPI Specification, SLA-driven APIs, API Gateways

ACM Reference Format:
Antonio Gamez-Diaz, Pablo Fernandez, and Antonio Ruiz-Cortés. 2019. Governify for APIs: SLA-Driven Ecosystem for API Governance. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3338906.3341176

1 INTRODUCTION

In the last decade, RESTful APIs are becoming a clear trend as composable elements that can be used to build and integrate software [8]. One of the key benefits this paradigm offers is a systematic approach to information modeling leveraged by a growing set of standardized tooling stack. In this context, the term of *API Economy* is being increasingly used to describe the movement of the industries to share their internal business assets as APIs [7, 11] not only across internal organizational units but also to external third parties; in doing so, this trend has the potential of unlocking additional business value through the creation of new assets [1, 7].

In order to be competitive in such a growing market of APIs, at least two key aspects can be identified: i) *ease of use* for its potential developers; ii) a flexible usage *plan* that fits their customer's demands.

Regarding the *ease of use* perspective, third-party developers need to understand how to use the exposed APIs so it becomes necessary to provide good training material but, unfortunately, API providers do not often write good documentation of their products [2]. Notwithstanding, during the last years, the *OpenAPI Specification*[1] (OAS), formerly known as *Swagger*, has become the *de facto* standard to describe RESTful APIs from a functional perspective providing an ecosystem of tools[2] that helps the developer in several aspects of the API development lifecycle.

Concerning the usage *plans* perspective, as APIs are deployed and used in real settings, the need for non-functional aspects is becoming crucial. In particular, the adoption of Service Level Agreements (SLAs) [9] could be highly valuable to address significant challenges that industry is facing, as they provide an explicit placeholder to state the guarantees and limitations that a provider offers to its consumers. Exemplary, these limitations (such as *quotas* or *rates*) are present in most common industrial APIs [3] and both API providers and consumers need to handle how they monitor, enforce or respect them with the consequent impact in the API deployment and consumption.

However, to the best of our knowledge, there is no widely accepted and open source tool that leverages from the functional model as well as the non-functional description to create usage plans including elements such as cost, functionality restrictions or limits and performing actual API governance in production.

In this paper, we introduce the *Governify for APIs* ecosystem, a set of tools which, starting from an OAS description, assist the user during the RESTful APIs' development process for the creation of usage plans (or SLAs) and performing seamlessly SLA-Driven API governance. Ultimately, we have evaluated our proposal in three different scenarios: teaching, research and industry.

[1]The latest version of the OpenAPI Specification is available at https://github.com/OAI/OpenAPI-Specification
[2]https://openapi.tools

The rest of the paper is structured as follows: Section 2 analyzes different alternatives to our proposal, Section 3 presents the *Governify for APIs* ecosystem and Section 4 outlines the evaluation that we performed. Finally, in Section 5, shows some final remarks and conclusions.

2 RELATED TOOLS

Despite ad-hoc solutions for regulating APIs have emerged, we focus on the so-called API Gateways [5], which have emerged to support API developers in the management of aspects such as consumer authentication, request throttling or billing. The increasing growth of APIs has resulted in a proliferation of API Gateways platforms that provide different levels of support for functional and non-functional aspects.

In order to spot these differences, we have considered[3] the publicly available information of 18 API Gateways. The conclusions:

1) Regarding the functional point of view: (i) Almost every API Gateway do allow the usage of the OAS for the functional definition of the API. (ii) Almost a half have an explicit (vendor-specific) model to define some parts of the configuration of the platform (e.g. endpoints, limitations, general information), but only one third allow importing/exporting. (iii) As stated by the authors in [10], the importance of OAS made it a key feature widely supported by the API Gateway provider.

Concerning the non-functional point of view: (i) Most API Gateways support quotas, against two thirds supporting rates. (ii) All API Gateways allow request as a metric, a third allows other predefined metric and only two API Gateways allow defining custom metrics. (iii) As pointed out in [3], API providers typically support limitations (quotas and/or rates) and they are usually defined over the number of requests with a minimal frequency supported that starts from minutely in the case of quotas, and secondly in the case of rates.

Based on these findings, we observe that, in spite all API Gateways spot similar features, the underlying model and concepts are different and each platform describes the configuration in a specific format, hindering, thus, the interoperability among providers. Consequently, organizations that face the transition from a certain API Gateway to a different one, they are required to perform a manual migration process and complex evaluation of the behavioral and vocabulary differences between the vendor-specific models of each API Gateway.

3 GOVERNIFY FOR APIS

3.1 Motivating Example

We will guide the explanation of the *Governify for APIs* ecosystem by means of a real-world RESTful API which needs to be governed; namely, the DBLP API, a service for retrieving bibliometric information in the Computer Science research area. It is a quite simple API that offers three GET endpoints for searching *authors*, *publications* and *venues* by introducing parameters in the query (e.g., *http://dblp.org/search/author/api?q=gamez+diaz*).

This API does not offer any explicit information of the non-functional properties or limitations besides of an entry in the FAQ[4] in natural language (sic. *you should always be fine when waiting for at least one or two seconds between two consecutive requests*). This lack hinders the creation of limitations-aware API clients because of the need for a prior human progressing of the aforementioned FAQ. Nevertheless, from the functional perspective, this API could be easily described by means of the OpenAPI Specification (c.f., resources on the website[5]). For the non-functional modeling, SLA4OAI[6] will be used. It is a vendor-neutral open-source proposal for describing APIs' aspects such as quotas and rates. It enables users to model the typical limitations that can be usually found in an API [3]. Note SLA4OAI is able to model a subset of terms that usually appear in an SLA; nevertheless, SLA can compliant with other SLA specifications, such as iAgree [6], including support for other metrics (e.g., availability) and concepts (e.g., penalties and rewards).

Starting from an API functionally defined with OAS, our goal will be to use *Governify for APIs* to regulate that API accordingly to its quota and rate limitations using a vendor-neutral specification.

3.2 Architecture

We have developed a Node.js set of tools, inspired in the microservice architectural style, packed as Docker public images[7] and deployed for a publicly online access. An overview of all the components is depicted in Figure 1. The source code and technical information are available at the supplementary website[5]. Specifically, we introduce the *SLA Editor* for hiding the complexity of the language to the end user. Next, we support two different enforces: gateway and standalone. The former, see Figure 3, is intended to be developers who want to regulate an API without modifying the source code. The latter is supposed to serve to developers who need a more fine-grained control by modifying the API code. We provide a *SLA-Driven API Gateway* for the first enforce and an *SLA Instrumentation Library* as part of a Node.js module for the latter. Both enforces are instrumented by the *SLA Engine*. Next, we describe in detail each tool.

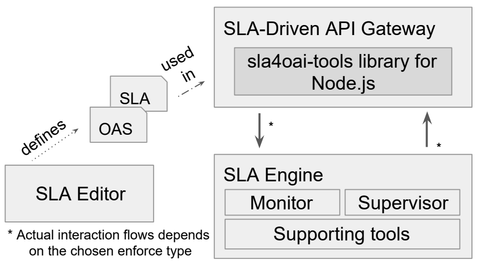


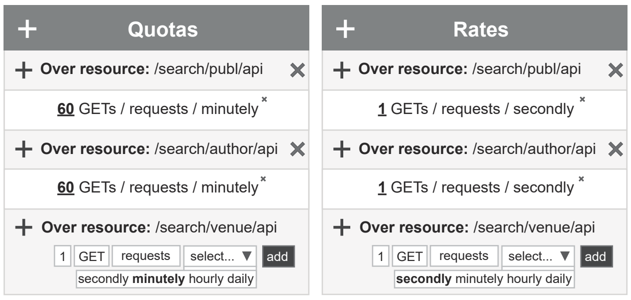
Figure 1: *Governify for APIs*' simplified architecture

3.3 SLA Editor

Governify for APIs provides an *SLA editor*[8], a user-friendly web-based text editor specifically developed for assisting the user during the modeling tasks, including auto-completion, syntax checking, and automatic binding (i.e., the changes in the UI are synchronized

[3]Avaliable at https://isa-group.github.io/2019-05-sla4oai-demo/files/api_gw.html
[4]https://dblp.org/faq/1474706
[5]https://links.governify.io/link/GovernifyForAPIs
[6]https://sla4oai.specs.governify.io
[7]https://hub.docker.com/u/isagroup
[8]https://designer.governify.io

with the underlying SLA4OAI textual model). Precisely, Figure 2 depicts this textual-visual binding. It is possible to create different plans (e.g., *free* and *pro*) with quotas and rates over specific metrics. Clicking in the + sign, the user is able to select the path and method (previously defined in the OAS document) for entering the value of the limitation. Note that other custom metrics besides *requests* can also be defined. Precisely, in the case of the DBLP API, after having modeled the functional part with OAS, the plan can be easily created. We added a made up professional plan just to show the modeling capabilities.



+ Quotas	+ Rates
+ **Over resource:** /search/publ/api ✕	+ **Over resource:** /search/publ/api ✕
60 GETs / requests / minutely *	1 GETs / requests / secondly *
+ **Over resource:** /search/author/api ✕	+ **Over resource:** /search/author/api ✕
60 GETs / requests / minutely *	1 GETs / requests / secondly *
+ **Over resource:** /search/venue/api	+ **Over resource:** /search/venue/api
1 GET requests select... ▼ add	1 GET requests select... ▼ add
secondly **minutely** hourly daily	**secondly** minutely hourly daily

Figure 2: Editing DBLP plans in the *SLA Editor* tool

3.4 SLA Engine

The SLA4OAI specification outlines the *Basic SLA Management Service*[9] (BSMS) defining the interaction flows and the endpoints */check* and */metrics*. In Figure 3 we focus on the *Gateway* enforce as it is a more complete scenario.

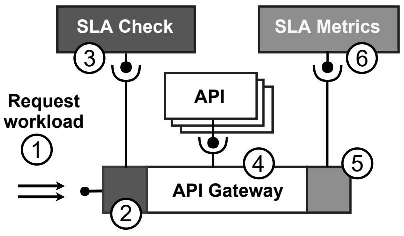


Figure 3: *Gateway* enforce defined in the SLA4OAI BSMS

First, requests will pass through the API Gateway until they are directed to the node that will serve it (step 1). Next, the API Gateway query the *SLA Check* API to determine if the request is authorized to develop the actual operation based on the appropriate SLA (step 2). Afterward, if it is authorized, the actual API is invoked and the response is returned (step 3). If it is not, a status code and a summary of the reason (as generated by the SLA check API) is returned (step 3). After the consumption ends (step 4), the metrics are sent to the *SLA Metrics* API (step 5), which is in charge of updating the status of the agreement with the new metrics introduced (step 6).

Since we stick to the SLA4OAI specification and it left open the implementation, our tooling for the*SLA Engine*, provides a concrete implementation of the BSMS, including also a particular way to handle SLA and users saving/retrieving tasks (*SLA Registry* and *Tenants*). Specifically, *Monitor*[10] is an implementation of the *Metrics* BSMS service and *Supervisor*[11], of the *Check* service.

The *Monitor* service exposes a POST operation in the route */metrics* for gathering the metrics collected from other different services. It can collect a set of basic metrics and send them to a data store for aggregation and later consumption. The metrics can be grouped in batches or sent one by one to fine-tune performance versus real-time SLA tracking.

The *Supervisor* service has a POST */check* endpoint for the verification of the current state of the SLA for a given operation in a certain scope. For each request, this service will evaluate the state of the SLA and will respond with a positive or negative response depending on whether a limitation has been overcome. In addition, this service also implements (outside the scope of the BSMS) these additional endpoints: GET/POST */tenants*, GET/POST */slas* and PUT/DELETE *slas/<id>* for managing both users (tenants and accounts) and SLA4OAI documents themselves.

3.5 SLA Instrumentation Library

Despite the fact that the BSMS defines the interaction flows between the endpoints, the concrete implementation of these interactions is left open. That is the way our aims to cover this lack. Specifically, we present an *SLA Instrumentation Library* for Node.js[12], which is a middleware (i.e., a filter that intercepts the HTTP requests and perform transformation if necessary) written for Express, the most used Node.js web application framework. This middleware intercepts all the inbound/outbound traffic to perform the BSMS flow. Throughout the Listing 1 we observe that it is necessary to import the library (line 3), to configure the endpoints of the services (lines 8 and 9) and finally register the middleware (line 12).

```
1  // Imports
2  const express = require("express");
3  const slaInstrumentationLib = require("sla4oai");
4
5  const app = express(); // Express init
6
7  // SLA4OAI init
8  const supervisorURL = {url: "supervisor.oai.governify.io"};
9  const monitorURL = {url: "monitor.oai.governify.io"};
10
11 // Express middleware registration
12 slaInstrumentationLib.register(app,supervisorURL,monitorURL);
```

Listing 1: Excerpt of the configuration of the SLA Instrumentation Library

3.6 SLA-Driven API Gateway

A more transparent way to implement the interaction flows defined is the BSMS is achieved by using a *Gateway* SLA enforce. Our tool, the *SLA-Driven API Gateway* is an open-source implementation to be deployed using any *SLA Engine*. Particularly, we provide an online preconfigured instance[13] using the aforementioned *SLA Instrumentation Library*. As depicted in Figure 4, API providers are only required to enter: (i) The real endpoint of their API; (ii) An

[9]https://sla4oai.specs.governify.io/operationalServices.html

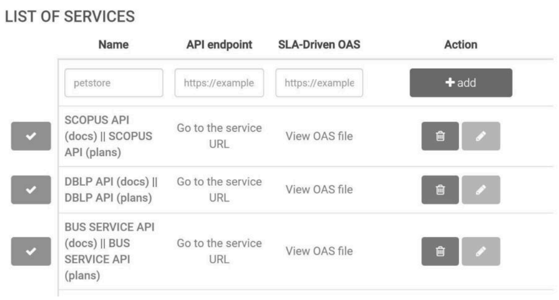
[10]https://monitor.oai.governify.io/api/v1/docs

[11]https://supervisor.oai.governify.io/api/v1/docs

[12]https://www.npmjs.com/package/sla4oai-tools

[13]https://gateway.oai.governify.io

URL pointing to the SLA4OAI document. Once an API is registered, the *SLA-Driven API Gateway* exposes a public and SLA-regulated endpoint, as well as a */plans* endpoint for a provisioning portal. It enables customers to purchase a plan, after that, this customer will get an API-key, acting as a bearer token for HTTP authentication to consume the SLA-regulated API.



LIST OF SERVICES

	Name	API endpoint	SLA-Driven OAS	Action
	petstore	https://example	https://example	+ add
✓	SCOPUS API (docs) || SCOPUS API (plans)	Go to the service URL	View OAS file	
✓	DBLP API (docs) || DBLP API (plans)	Go to the service URL	View OAS file	
✓	BUS SERVICE API (docs) || BUS SERVICE API (plans)	Go to the service URL	View OAS file	

Figure 4: Configuration UI of the SLA-Driven API Gateway

4 EVALUATION

We have performed a threefold qualitative evaluation in industry, teaching and research contexts.

Concerning the industrial evaluation, some OpenAPI Initiative members have expressed its interest in SLA4OAI, the SLA modeling proposal, and in promoting a working group for evolving and extending it. Indeed, in [4], we collaborated with people from Google, Paypal, AsyncAPI Initiative and Metadev for analyzing, starting from SLA4OAI, the status of SLAs and limitations in the industry. Furthermore, in spite of the fact the SLA4OAI extension and tools have not been widely announced nor promoted, we have disclosed the tooling ecosystem into the main public Node.js artifact repository (i.e., NPM) and this platform provides a set of analytics of usage since their publishing. Specifically, based on its data we observe that *SLA Instrumentation Library* has been downloaded and installed more than 600 times[14] while the *SLA Engine* was downloaded more installed than 1900 times.

Regarding the use of *Governify for APIs* in teaching, it has been extensively used in, at least, two undergraduate service-oriented related subjects. As students were required to create their own APIs[15], they also had to set the rate and quota limitations using *Governify for APIs*. Whereas we do not have any specific usage report, we collected useful information, issues and bugs derived from running in production.

As of the research context, we are validating our proposal (language and tools) in a national research network. Several members are exposing their research results by creating an API and applying limitations using *Governify for APIs* and SLA4OAI. Then, all these artifacts are being deployed in a central publicly available catalog[16].

[14] https://npm-stat.com/charts.html?package=sla4oai-tools
[15] https://github.com/gti-sos
[16] https://services.rcis.governify.io

5 CONCLUSIONS

In this work, we have presented the *Governify for APIs* ecosystem, a set of tools integrated aimed to support the user during the SLA-Driven RESTful APIs' lifecycle. Specifically, an *SLA Editor* and an *SLA-Driven API Gateway* on the top of an *SLA Engine* composed by an *SLA Monitor* and an *SLA Check* APIs.

We have evaluated our proposal in three different scenarios: teaching, research and industry, getting, therefore, a highly valuable source of information that will be used in the upcoming improvements. With *Governify for APIs* we prove that, with state-of-the-art tools, it is possible to improve lifecycle of SLA-Driven RESTful APIs, especially those problems derived from design and operation.

Specifically, (i) Complex usage plans with quota and rate limitations can be modeled with an OAS-compliant vendor-neutral format; (ii) The support of vendor-neutral initiatives paves the way for the interoperability between API Gateway providers; (iii) *Governify for APIs* left a publicly available ecosystem of open-source tools supporting the SLA-Driven RESTful APIs' lifecycle.

ACKNOWLEDGMENTS

This work is partially supported by the European Commission (FEDER), the Spanish Government under projects BELI (TIN2015-70560-R) and HORATIO (RTI2018-101204-B-C21), and the FPU scholarship program, granted by the Spanish Ministry of Education, Culture and Sports (FPU15/02980). The authors would like to thank for their valuable technical contributions to Daniel Arteaga and Felipe Vieira da Cunha.

REFERENCES

[1] Michele Bonardi, Maurizio Brioschi, Alfonso Fuggetta, Emiliano Sergio Verga, and Maurilio Zuccalà. 2016. Fostering Collaboration Through API Economy: The E015 Digital Ecosystem. In *Proceedings of the 3rd International Workshop on Software Engineering Research and Industrial Practice (SER&#38;IP '16)*. ACM, New York, NY, USA, 32–38. https://doi.org/10.1145/2897022.2897026
[2] Forrester. 2015. *API Management Solutions , Q3 2014.* Technical Report. Forrester.
[3] Antonio Gamez-Diaz, Pablo Fernandez, and Antonio Ruiz-Cortes. 2017. An Analysis of RESTful APIs Offerings in the Industry. In *Service-Oriented Computing*, Michael Maximilien, Antonio Vallecillo, Jianmin Wang, and Marc Oriol (Eds.). Springer International Publishing, Cham, 589–604.
[4] Antonio Gamez-Diaz, Pablo Fernandez, Antonio Ruiz-Cortes, Pedro J. Molina, Nikhil Kolekar, Prithpal Bhogill, Madhuranjan Mohaan, and Francisco Méndez. 2019. The role of limitations and SLAs in the API industry. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, Tallin, Estonia. https://doi.org/10.1145/3338906.3340445
[5] Antonio Gámez-Díaz, Pablo Fernández-Montes, and Antonio Ruiz-Cortés. 2015. Towards SLA-Driven API Gateways. In *Actas de las XI Jornadas de Ingeniería de Ciencia e Ingeniería de Servicios*, Juan Manuel Murillo (Ed.), Vol. 201232273. Sistedes, Santander, 9. https://doi.org/10.13140/RG.2.1.4111.5609
[6] Antonio Gámez-Díaz, Pablo Fernández-Montes, and Antonio Ruiz-Cortés. 2018. Fostering SLA-Driven API Specifications. In *Actas de las XIV Jornadas de Ingeniería de Ciencia e Ingeniería de Servicios*, Manuel Lama (Ed.). Sistedes, Sevilla. https://doi.org/10.13140/RG.2.2.35748.53128
[7] Jose Maria Garcia, Pablo Fernandez, Antonio Ruiz-Cortes, Schahram Dustdar, and Miguel Toro. 2017. Edge and cloud pricing for the sharing economy. *IEEE Internet Computing* 21, 2 (3 2017), 78–84. https://doi.org/10.1109/MIC.2017.24
[8] Holger Harms, Collin Rogowski, and Luigi Lo Iacono. 2017. Guidelines for Adopting Frontend Architectures and Patterns in Microservices-based Systems. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 902–907. https://doi.org/10.1145/3106237.3117775
[9] C. Muller, A. Gutierrez Fernandez, P. Fernandez, O. Martin-Diaz, M. Resinas, and A. Ruiz-Cortes. 2018. Automated Validation of Compensable SLAs. *IEEE Transactions on Services Computing* (jan 2018), 1–1. https://doi.org/10.1109/TSC.2018.2885766
[10] Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino. 2018. An Analysis of Public REST Web Service APIs. *IEEE Transactions on Services Computing* (2018). https://doi.org/10.1109/TSC.2018.2847344
[11] W. Tan, Y. Fan, A. Ghoneim, M. A. Hossain, and S. Dustdar. 2016. From the Service-Oriented Architecture to the Web API Economy. *IEEE Internet Computing* 20, 4 (July 2016), 64–68. https://doi.org/10.1109/MIC.2016.74

A Pricing Modeling Framework for RESTful APIs Regulated with Limitations

Under revision - unpublished.

***Authors**: Antonio Gámez Díaz, Rafael Fresno Aranda, Pablo Fernández Montes, Antonio Ruiz Cortés.*

A Pricing Modeling Framework for RESTful APIs Regulated with Limitations

Antonio Gamez-Diaz, Rafael Fresno-Aranda, Pablo Fernandez and Antonio Ruiz-Cortes

Abstract—Today, APIs are regarded as a new form of business product, and ever more organizations are publicly opening up access to their APIs as a way to create new business opportunities in the so-called *API Economy*. A crucial activity in such a context is to define the set of *plans* (i.e., the *pricing*) that clearly depicts the functionality and performance being offered to clients. API providers usually define certain *limitations* in each instance of a plan (e.g., *quotas* and *rates*). For example, a free plan might be limited to having 100 monthly requests, and a professional plan to having 1000 monthly requests. Several proposals have emerged to model the functional part of the API (i.e., resources, responses, authentication, etc.). Specifically, the OpenAPI Initiative (OAI) has become the *de facto* standard in the industry. The alignment with standards is fostering the growth of an open ecosystem of tools aimed at automating both API development and operations (i.e., the DevOps paradigm). However, there is no proposal for modeling API *pricing* (including the *plans* and *limitations*), and this information is neither typically structured nor standardized. Therefore, answering questions regarding API limitations (e.g., determining whether or not a certain *pricing* is valid) is still a manual process with the consequent inconveniences (being tedious, time-consuming, error-prone, etc.). Additionally, the lack of standards-aligned specifications of API pricings and limitations hinders the creation of an open ecosystem of tools that can leverage this invaluable information so as to help in the DevOps cycle (e.g., limitation-aware testing, or automated gateway configuration). Specifically, this paper presents a *pricing modeling framework* that includes: (a) *Governify4APIs* model, a comprehensive and rigorous model of the concept of API *pricing* and its elements (e.g., *plans*, *limitations*, etc.). This model is accompanied by *SLA4OAI*, a serialization that extends the widely used OAI Specification; (b) a validity operation to validate the description of API *pricings*, with a tool set (*sla4oai-analyzer*) that has been developed to automate this operation and which provides both a stand-alone module and a RESTful API. Additionally, we performed a rigorous analysis of 244 real-world APIs, and the pricings of 32 of them were manually modeled to assess the expressiveness of the model and its serialization.

1 Introduction

Today, APIs are regarded as a new form of business product, and ever more organizations are publicly opening up access to their APIs as a way to create new business opportunities in this so-called API Economy. Indeed, this trend has been given a boost by the shift towards microservice architectures as the preferred choice for the construction of cloud-native Software as a Service. Since these architectures typically promote the deployment, isolation, and integration of components (i.e., microservices) by means of RESTful Web APIs (henceforth, for the sake of simplicity, APIs), they pave the way for easy connection to external APIs (as service consumers) or opening up internal APIs to the market (as service providers). Moreover, during the last few years, there has been a successful effort made for standardization (the Open API Specification[1]). Its core is the proposal of a standard specification for the functional part of the APIs (i.e., the resources and operations available). This specification has led to the creation of a rich open ecosystem of tools and techniques to help the industry in the development and evolution of APIs and microservice architectures.

In this context, from a non-functional perspective, defining business models and plans with API limitations, such as quotas or rates, has become crucial for the regulation of the behaviour expected from all participants and for a guarantee of a certain service level. However, the information regarding the limitations in APIs is neither structured nor standardized [1]. As a consequence, answering questions regarding limitations (e.g., calculating the number of requests attainable in a certain period) has to be a manual process with the consequent inconveniences (of being tedious, time-consuming, error-prone, etc.).

The objectives of the present study were fourfold: (i) to model and validate with rigour the concept of API limitation, and to study its properties; (ii) to provide a specific serialization of the model aligned with the Open API Specification (OAS); (iii) to define a validity operation for validating the description of API limitations; and (iv) to present a prototyping tool to automate this operation that can help users address API limitations.

The rest of this paper is structured as follows: Section 2 presents the motivation behind our proposal by discussing a real scenario, and we introduce the vocabulary used in the industry; Section 3 sets out a pricing model and a corresponding OAS-aligned serialization; Section 4 presents a validity operation; in Section 5 we validate our approach by reviewing 244 APIs in the industry to define a representative subset of 32 APIs that is modeled and analysed with our model, and we describe the tools developed to automate the analysis operation; Section 5.2 presents our prototyping tool; Section 6 describes related work; and Section 7 and Section 8 presents some conclusions and final remarks.

2 Pricing in the API Economy

In the API Economy world, API providers have to make sufficient information available for the consumer to get informed

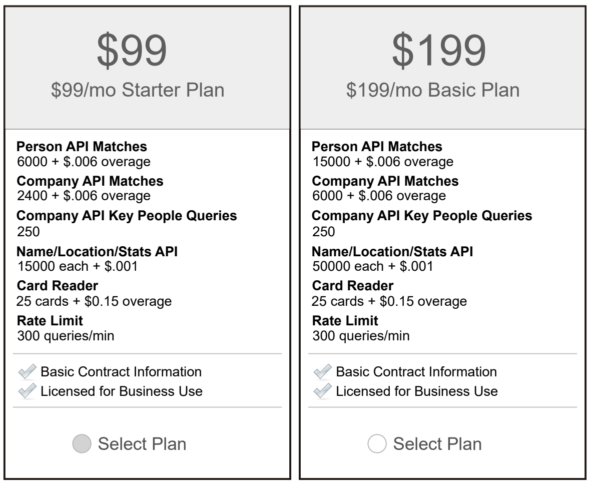
• *Authors are with the Universidad de Sevilla (Seville, Spain). Corresponding author e-mail: pablofm@us.es*

1. More details can be found at https://www.openapis.org

about their products. This includes information regarding the API itself (endpoints and methods), the *plans* that a user can subscribe to, and the associated *cost*. A *plan* includes information regarding the API's limitations (*quota* and *rates*) for each of its resources.

All this information is typically found in a section called *pricing* (however, cloud infrastructure providers tend to refer to it as an *offering*). Consequently, we shall henceforth consider a *pricing* to be a set of *plans* having an associated *cost*.

In order to illustrate these concepts, we present a real example: the FullContact API – a tool for managing and combining contacts from different sources (Gmail, social media, etc.). The API allows users to programmatically look up information and to match email addresses with publicly available information so as to enrich the contacts. Figure 1 depicts the pricing extracted from the *FullContact*[2] API.



$99 $99/mo Starter Plan	$199 $199/mo Basic Plan
Person API Matches 6000 + $.006 overage	**Person API Matches** 15000 + $.006 overage
Company API Matches 2400 + $.006 overage	**Company API Matches** 6000 + $.006 overage
Company API Key People Queries 250	**Company API Key People Queries** 250
Name/Location/Stats API 15000 each + $.001	**Name/Location/Stats API** 50000 each + $.001
Card Reader 25 cards + $0.15 overage	**Card Reader** 25 cards + $0.15 overage
Rate Limit 300 queries/min	**Rate Limit** 300 queries/min
✓ Basic Contract Information	✓ Basic Contract Information
✓ Licensed for Business Use	✓ Licensed for Business Use
● Select Plan	○ Select Plan

Fig. 1. Plans of the FullContact API.

This *pricing* example consists of two paid *plans* having a fixed price *cost* billed monthly. With respect to the *limitations*, for each *operation*, a *quota* is applied. For example, in the starter *plan*, only 6000 matches on Person are available. Nevertheless, an *overage* is defined, i.e., it is possible to surpass the *limit* by paying a certain amount of money, in this case, $0.006 per request. Regardless of the plan, a common *rate* of 300 queries per minute is applied.

In this context, several analytical challenges can arise since the API providers need to understand the plans in depth before taking further action. In particular, they should verify the validity of their plans (i.e., that there is nothing inconsistent).

Those challenges correspond to common questions on the API's pricing and plans that could be answered automatically with an appropriate model and analytical framework providing different analysis operations. The following two sections will detail the proposed model (Section 3) and the definition of a validity operation (Section 4).

2. https://www.fullcontact.com/developer. Accessed May 2019.

3 Pricing Model

In this section, we first present the Governify4APIs model (Section 3.1), then introduce SLA4OAI (Section 3.2), a specific textual serialization compatible with the OpenAPI Specification. Both sections will use the FullContact pricing described above as a running example.

3.1 The *Governify4APIs* model

The *Governify4APIs* is a model for API pricing, i.e., of each plan and the associated cost for a given API. It starts from the idea that each API resource (HTTP path and method) has a related set of limitations (quotas and rates) for each API plan.

Figure 2 depicts the entire *Governify4APIs* model. For the sake of clarity, we have split it into three areas: (i) in dark grey, *pricing, plans and cost*; (ii) in light grey, *limitations and limits*; (iii) in medium grey, *capacity*. In the following subsections, we will detail each part of the model with examples extracted from the FullContact API in Figure 1, considering each part: the plan area (Subsection 3.1.1), the limitations area (Subsection 3.1.2), and the capacity area (Subsection 3.1.3).

3.1.1 Pricing, plans, and cost

As depicted in the model (dark grey in Figure 2), a `Pricing` consists of a set of `Plans`. A `Plan` has a name and a `Cost` that defines the price charged to users so that they can access the service. In our example, the FullContact API has two `plans`: a *starter* and a *basic* `Plan`.

The `Cost` may be very simple (e.g., assign a constant price to the `Plan`, e.g., *$99* or *$199* as in our example) or may depend on other properties. In this latter case, when the cost depends on a `Limitation`, we distinguish two costs: `OperationCost`, when an `Operation` is being charged for each time it is invoked; and `OverageCost`, when once a certain value of the `Limitation` has been reached (cf. Subsection 3.1.2), there start to be imposed charges per volume.

Either type of `Cost` can be periodic, defining a `Period` with an amount and a `TimeUnit`. In our example, the `Cost` of the *Starter* `Plan` is *99$* billed *monthly*, i.e., it has a *Period* with value 1 of the *TimeUnit* MONTH.

An `OperationCost` is frequent in pay-as-you-go payment models in which there is no monthly fixed `Cost` and the API consumer is only charged for, given a *requests* `metric`, the number of requests. In the model, this cost is associated with the operation by means of the Limitation. For example, a service might offer a `Plan` A in which each request can be charged at 0.10$ (volume: 1) and a `Plan` B where each pack of 1000 requests (volume: 1000) is charged at 75$. Depending on the client's needs, they might prefer `Plan` A or `Plan` B.

An `OverageCost` is usual when providers do not want to cut off the service once a `Limitation` has been reached, but want to continue providing it at a certain charge. Our example defines an overage when the `quota` values are reached: *each additional match after 6000 monthly matches is charged at $0.006.*

3.1.2 Limitations and limits

As depicted in the model (unshaded section in Figure 2), in order to carry out this regulation of the consumption of an API, each `Operation` in a `Plan` can be subject to `Limitations` on a `Metric`. The most frequent type of `Limitation`

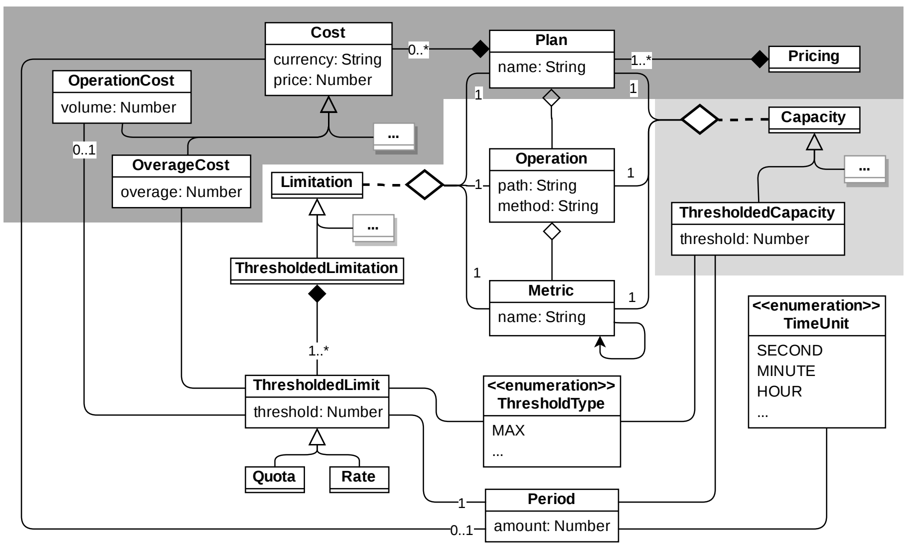


Fig. 2. *Governify4APIs* model for API pricing.

is the `ThresholdedLimitation` which establishes one or more `ThresholdLimits` on the number of `Metric` units in a `Period`. The `ThresholdType` is usually MAX (i.e., the `ThresholdLimit` would therefore represent the *maximum* number of `Metric` units). In defining their `Pricing`, `Limitations` allow providers to adjust the API's consumption to the platform's total `Capacity` (cf. Subsection 3.1.3).

An `Operation` is defined by the pair formed by HTTP method and path. For example, *GET /contacts* would represent the query operation on a collection of user-type objects. A common example of a `Metric` is the *number of requests*. Nonetheless, other metrics can be defined such as storage, bandwidth or CPU consumption.

In accordance with the implementation, i.e., the algorithm used to enforce the `Limitations`, we say that a `ThresholdedLimit` is a `Quota` if the computation of the number of metric units is done over a static window, i.e., in a *fixed* time window. For example, a *one-week static window* might be such that it always starts on Monday at 00:00 and ends on Sunday at 23:59, regardless of when the first metric unit is computed. On the contrary, if the time window is *sliding*, i.e., relative to the first metric unit computed, we say that the `ThresholdedLimit` is a `Rate`. For example, in a *one-week sliding window*, if the first metric unit were computed on Wednesday at 15:36:39, that window would close on the following Wednesday at 15:36:38.

Figure 3 illustrates graphically the differences between sliding and static windows. 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.

For example, if we use the *number of requests* as a `Metric`, and we want to prevent our users from making more than 4

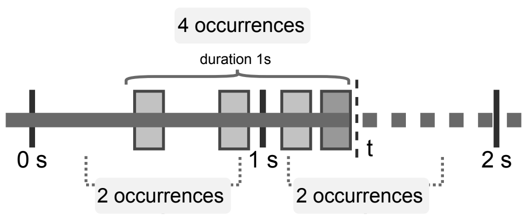


Fig. 3. Sliding (rates) vs static (quotas) windows.

requests per second, there are two different alternatives: a 1-second sliding time window with a limit of 4 requests, that opens after the first request and prevents more than 4 from being made during that second; or a 1-second static window with a limit of 2 requests, that could concentrate the first two requests at the end of the first second and the other two at the beginning of the next one.

In the industry, these limitations tend to follow definite patterns [1]. Specifically, `Quotas` tend to be defined over any metric and are measured in periods longer than an hour (e.g., daily, weekly, monthly or yearly), while `Rates` tend to be defined over the number of requests and are measured in shorter periods (e.g., secondly or minutely).

In our FullContact example, the *starter* plan has one `Rate` and four different `Quotas`. For example, the `Rate` is *300 requests in a 1-minute sliding window* and a `Quota` is *6000 matches in a 1-month static window*.

The model distinguishes two concepts: `ThresholdedLimitation` and `ThresholdedLimit`. A `ThresholdedLimitation` over a certain metric and operation establishes a fraction of the overall `Capacity` of the service. A `ThresholdedLimitation`, however, can be expressed in various ways, one of which is by defining a set of `ThresholdedLimits` that, within a time period, restrict the percentage of `Capacity` that consumers are allowed to use. For example, a `ThresholdedLimitation` on a certain operation can be defined as a set of ThresholdedLimits as follows: *30 requests every 1 week* and *1 request every 1 second*.

A different way to express a Limitation (as represented by the ellipsis "..." in the model) would be to use frequency distributions [2], so that referring to percentiles would allow the form of the distribution and its different attributes to be considered. For example, a percentile, such as 99.0 or 99.9 would show a plausible value in the worst case, while the 50th percentile would emphasize the typical case. In the present communication, however, we will not address limitations specified as frequency distributions.

3.1.3 Capacity

Finally, a crucial aspect that is not explicitly depicted in a pricing or a plan is the Capacity. This is an internal aspect that providers do not put out publicly. The Capacity of the service represents a subset of the constraints of the platform or system on which the service is being deployed. It is the result of having to satisfy mainly technical and budget criteria (e.g., CPU or memory, number of nodes of the cluster, etc.).

Estimating the service's Capacity is fundamental to defining the Pricing and analysing the Limitations. In particular, all the Limitations ought to be satisfied by the service, i.e., they must not exceed the service's Capacity.

As depicted in the model (medium grey in Figure 2), once the Capacity has been identified, it is specified as if it were a Limitation, i.e., the number of certain metric units in a given Period. Therefore, analogously to the Limitation, the ThresholdedCapacity has a threshold value and a ThresholdType (usually MAX) in a given Period of a TimeUnit.

A possible way to express the Capacity on the metric *request* is the *number of requests per second (RPS)* for each operation and plan. For example, a capacity of *10 000 RPS* in *GET /pets* in the *free plan* would mean that the entire set of free-plan users will be able to make 10 000 RPS. The Capacity can be different for each plan since different infrastructures may be used to provide a better level of service to the clients.

For example, an organization might have calculated, based on performance and stress tests, that its production cluster is able to accept 10 000 RPS. Consequently, if a limitation had been set of 10 requests per second per client, the theoretical number of concurrent requests would be 10 000/10 = 1000 concurrent clients.

A useful instrument when analysing Limitations is the *percentage of capacity utilization* or simply the *percentage of utilization (PU)*. Intuitively, this percentage directly determines whether or not a Limitation can be set because this will be impossible if the PU is greater than 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*:

In a first approximation, an API consumer could assume that, since 1 day is 86 400 seconds, for every second, they will have 43 200/86 400 = 0.5 requests. In this case, it is assumed a *uniform distribution* in which, little by little, the consumer will reach the 43 200 requests available in the day. This scenario corresponds to the *minimum PU*. But the ThresholdedLimitation states that for 1 day it is possible to make 43 200 requests, and in no case does it prevent the consumer from making all of them in a burst in the first instant of time. Indeed, in 1 second the consumer could make the whole set of 43 200 requests. This scenario implies a *burst distribution*, and corresponds to 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*.

Figure 4 illustrates different consumption scenarios for the same ThresholdedLimitation of *60 requests every 60 seconds*.

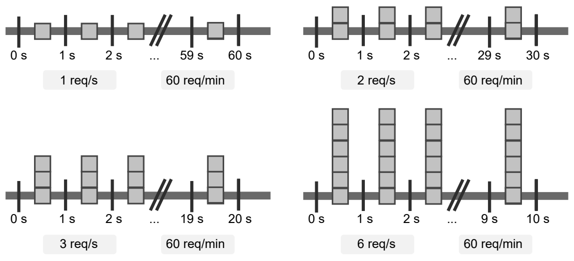


Fig. 4. Examples of different consumption scenarios for the same ThresholdedLimitation.

In a *uniform* consumption, 60 requests in 60 seconds would be equivalent to 1 request every 1 second. However, in a *burst* consumption, 2, 3, 6, or even a maximum of 60 requests could be made in 1 second. Therefore, to calculate the BPU in the limitation of *60 requests every 60 seconds*, we should take as a minimum value the *uniform* distribution of 1 request per second and as a maximum value the *burst* of 60 requests in 1 second in a 1 minute window.

3.2 SLA4OAI: A serialization for our model

The *Governify4APIs* model can be serialized to be aligned to a variety of API description specifications. Specifically, we propose SLA4OAI[3] [3], [4], 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).

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 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 a SLA4OAI document is the *plans* section. This describes different service levels, including the limitations set in the *quotas* and *rates*

3. https://github.com/isa-group/SLA4OAI-Specification

sections. In what follows, we shall detail some of the fields in a SLA4OAI file. Nevertheless, for a comprehensive description of the syntax, a JSON Schema document is available [5]. Further information is also available in the the specification's GitHub page [4].

As depicted in Listing 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 [4] (i.e., a protocol as part of the SLA4OAI proposal, beyond the scope of the present communication), 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 1. Main elements in SLA4OAI

Specifically, as depicted in Listing 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: FullContact
  sla: '1.0'
  type: plans
  api: ./fullcontact-oas.yaml
  provider: FullContact
infrastructure:
  supervisor: https://...
  monitor: https://...
availability: '2009-10-09T21:30:00.00Z'
```

Listing 2. Context, infrastructure and availability details in SLA4OAI

In the `Metrics` field, as depicted in Listing 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
matches:
  type: integer
  format: int64
  description: Number of matches
```

Listing 3. Metric details in SLA4OAI

The `Plans` section, as depicted in Listing 4, has the elements that will describe the plan-specific values – `quotas`, `rates`, and `pricing`.

4. https://github.com/isa-group/SLA4OAI-ResearchSpecification

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, a `period` with `amount` and a time `unit`, and the `scope` over which they should be enforced. As an extensible scope model, we propose two possible initial values (`tenant` or `account` as default). 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.

```
plans:
  Starter:
    pricing:
      cost: 99
      currency: USD
      period:
        amount: 1
        unit: month
    quotas:
      'v3/person.enrich':
        post:
          matches:
            - max: 6000
              cost:
                overage:
                  overage: 1
                  cost: 0.006
    rates:
      'v3/person.enrich':
        post:
          requests:
              - max: 10
                period:
                  amount: 1
                  unit: month
```

Listing 4. Plans details in SLA4OAI

4 Analysis

In this section, we propose an analysis framework to form a ground on which to reason about the pricing model presented. Consequently, this framework paves the way to exploiting the information contained in the model, and has been used to develop a validity analysis operation that could be useful in a real setting for both consumers and providers of APIs. As a foundation for the analysis operations, the first of the following subsections addresses the cornerstone of the analysis framework – the relationship between limitations and capacity. The subsequent subsections will detail and exemplify the list of analysis operations that have been defined.

4.1 Limits as percentages of capacity utilization

Since the capacity of the platform on which the service is deployed is not unlimited, the pricings should be defined to be compatible and coherent with that capacity. As an example, ensuring that the total capacity is sufficient for the potential use of the service defined in a particular plan should be analysed. Furthermore, we proposed (in Subsection 3.1.3) the notion that any given limitation corresponds to a Bounded set of Percentage of capacity Utilization (BPU) values derived from the potential usage scenarios a client could have for their consumption within the API while meeting its limitation.

In this context, the correspondence between limitations and BPU can be obtained by means of a normalization procedure that transforms the unit of the limitation to the capacity time unit, and then computing the minimum and maximum possible PUs. This procedure comprises just simple calculations, as is illustrated in the following example:

Consider a limitation with a limit of *43 200 requests / 1 day* and assume a total capacity of 50 000 RPS. Since all limitations should be expressed using the time unit of the capacity (second), the limitation is *43 200 requests / 86 400 seconds*. First, assume a *uniform* consumption, i.e., if in 1 day (86 400 seconds) there are 43 200 requests, in there will be 43 200/86 400 = 0.5 RPS. Given the value of the capacity, 50 000 RPS, the minimum PU is 0.5/50 000 = 0.000 01 = 0.001%. Now assume a single *burst* consumption, i.e., if a burst of 43 200 requests can occur during any 1 second window over 1 day. Given the value of the capacity, 50 000 RPS, the maximum PU is 43 200/50 000 = 0.864 = 86.4%. Therefore, the BPU of *43 200 requests / 1 day* subject to a capacity of 50 000 RPS is [0.001%-86.4%].

The calculation of the overall system capacity is a non-trivial procedure. It requires great technical effort in order to make a proper estimate. But, depending on the stage of development, even this will not always be feasible. In the present study, when the value of the system's capacity is unknown, we shall assign it the value of the highest capacity needed. To calculate this value, we shall assume uniform consumption after normalizing to the smallest time unit, and take the greatest value. The following is an example:

Consider the following two limitations: *1 RPS* and *100 RPW* (1 week, 604 800 s). In order to take the value of the highest capacity needed, we must first determine what the strongest limitation is. For this case, we normalize to the smallest unit, the second, $1\,RPS = 1$ and $100\,req/604\,800s = 0.000\,165\,RPS$, since $1 > 0.000\,165$ we have that the highest capacity needed is *1 RPS*. Therefore, we will take *1 RPS* as the value of the capacity. As a conclusion, it is worth noting that 1 RPS requires a higher capacity than 100 RPW, which only requires 0.000 165 RPS.

4.2 Pricing validity

We define the **validity** of a pricing as checking whether it is valid depending on a set of validity criteria. These include the absence of different types of conflict, for example, two limits within a limitation that cannot be met at the same time.

The validity of a *Governify4APIs* model is defined as certain validity criteria being met in each part of the model. In the model, a *pricing* has a set of *plans*, and these plans consist of *limitations*, each with its own *limits*. This hierarchy carries over to the validity operation. Hence, for example, a *pricing* will be valid, notwithstanding its satisfying other additional validity criteria, if all of its *plans* are also valid.

For solving validity conflicts, a **priority criterion** is required. For example, if two limits are defined with different values for a given metric and operation, which one should prevail over the other? In order to satisfy these requirements we assume henceforth the following default priority criteria: i) limitations with smaller periods over limitations with higher periods; ii) rates over quotas; iii) metric *number of requests* over any other metric. Notwithstanding, these criteria can be re-defined in other scenarios (e.g., metric *requests* is less important than the bandwidth in a certain business context).

We shall present the validity criteria in a hierarchy, starting from the fine-grained (VC1 - limits, VC2 - limitation) to the coarse-grained (VC3 - plan, VC4 - pricing) validity criteria. Each validity criterion comprises multiple validity subcriteria. Figure 5 gives an overview of this hierarchy of validity criteria. The details of each validity criterion are as follows:

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., there is no situation exceeding a *limit* with more priority while it is allowed by another *limit* with less priority (VC2.2); there are no *ambiguity conflicts* between any pair of its *limits*, i.e., two limits using the same period with different values (VC2.3) and there is no *capacity conflict*, i.e., the limitation does not surpass the associated *capacity* (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., two *limitations* on two related *metrics* (by a certain factor) cannot be met at the same time (VC3.2). If they happen to exist, the priority criteria will be used for determining which limit has to be prioritized.

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 one plan is less restrictive than the equivalent in another *plan* but the former *plan* is cheaper than the latter (VC4.2).

In order to understand these validity criteria, on the following subsubsections we will present examples of existence and absence of conflicts of each type.

4.2.1 Limit consistency conflict (VC2.2)

```
Capacity: 1000000 RPS
Limitations:
  Quota: 100 requests / 1 day
  Quota: 1000 requests / 1 week
```

Listing 5. Validity criterion VC2.2 (no limit consistency conflict)

An example of a situation where **there is no limit consistency conflict** can be observed in Listing 5. An inconsistency occurs when there is a possible situation exceeding a *limit* with more priority while it is allowed by another *limit* with less priority, according to the priority criteria hereinbefore mentioned. An example, using the size of the periods as the priority criterion, a conflict shall happen if the minimum PU of the limit with the longest period is less than the minimum PU of the limit with the shortest period.

The limit having the longest period is *1000 requests / 1 week* whose minimum PU is 1000/1 000 000 = 0.10%. The limit with the shortest period, *100 requests / 1 day*, has a minimum PU of 100/1 000 000 = 0.01%. Since $0.10\% \not< 0.01\%$, there is no conflict between these limits.

```
Capacity: 1000000 RPS
Limitations:
  Quota: 100 requests / 1 day
  Quota: 10 requests / 1 week
```

Listing 6. Validity criterion VC2.2 (limit consistency conflict)

On the other hand, in Listing 6 **there is a limit consistency conflict**. The limit with the longest period is *10 requests / 1 week* whose minimum PU is 10/1 000 000 =

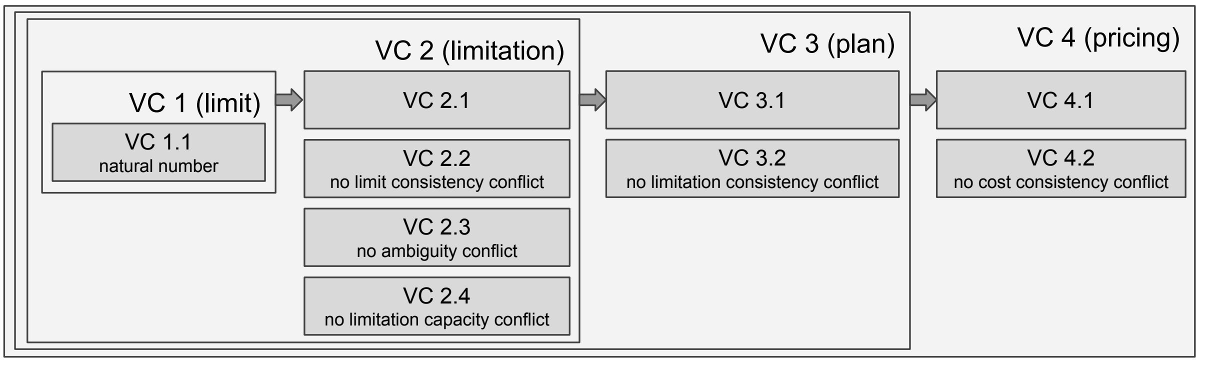


Fig. 5. Validity criteria hierarchy.

0.001%. The limit with the shortest period, *100 requests / 1 day*, has a minimum PU of 100/1 000 000 = 0.01%. Since 0.001% < 0.01%, there is a limit consistency conflict between these limits.

4.2.2 Ambiguity conflict (VC2.3)

```
Limitation:
  Limit: 1 request / 1 second
  Limit: 100 requests / 1 day
```

Listing 7. Validity criterion VC2.3 (no ambiguity conflict)

An example where **there is no ambiguity conflict** is presented in Listing 7, because the limits of the limitation use different periods, i.e., *1 second* and *1 day*.

```
Limitation:
  Limit: 1 requests / 1 second
  Limit: 100 requests / 1 second
```

Listing 8. Validity criterion VC2.3 (ambiguity conflict)

Conversely, in Listing 8 **there is a consistency conflict** because the limits of the limitation use the same period, i.e., *1 second*.

4.2.3 Capacity conflict (VC2.4)

```
Capacity: 100 requests / 1 second
  Limitations:
    Quota: 50 requests / 1 day
```

Listing 9. Validity criterion VC2.4 (no capacity conflict)

A possible situation where **there is no capacity conflict** is shown in Listing 9. First, we normalize using the unit of the capacity (i.e., seconds). Thus, there are *50 requests / 86 400s (1 day)*. Next, to calculate the BPU, we need both (i) the *minimum PU* (uniform distribution) and (ii) the *maximum PU* (burst distribution). For (i), if in 86 400 seconds there are 5 requests, in 1 second there will be 50/86 400 = 0.000 57 requests. The *minimum PU* is 0.000 57/100 = 0.0057%. For (ii), in 1 second there will be a burst of 50 requests. The *maximum PU* is 50/100 = 50%. Therefore, the BPU is [0.0057%,50%].

Since BPU is always less than 100%, there is no capacity conflict.

```
Capacity: 100 requests / 1 second (100 RPS)
Limitations:
  Quota: 200 requests / 1 day
```

Listing 10. Validity criterion VC2.4 (capacity conflict)

On the contrary, in Listing 10 **there is a capacity conflict**. First, we normalize using the unit of the capacity (i.e., seconds). Thus, there are *200 requests / 86 400s (1 day)*. Next, to calculate the BPU, we need both (i) the *minimum PU* (uniform distribution) and (ii) the *maximum PU* (burst distribution). For (i), if in 86 400 seconds there are 5 requests, in 1 second there will be 200/86 400 = 0.0023 requests. The *minimum PU* is 0.0023/100 = 0.000 23%. For (ii), in 1 second there will be a burst of 200 requests. The *maximum PU* is 200/100 = 200%. Therefore, the BPU is [0.000 23%,200%].

Since BPU is greater than 100%, there is a capacity conflict because of the *maximum PU*.

```
Capacity: 100 requests / 1 second (100 RPS)
Limitations:
  Quota: 200 requests / 1 day
  Rate: 99 requests / 1 second
```

Listing 11. Validity criterion VC2.4 (no capacity conflict)

Additionally, Listing 11 presents another example where **there is no capacity conflict**. First, we normalize using the unit of the capacity (i.e., seconds). Thus, there are *200 requests / 86 400s (1 day)* and *99 requests / 1s* Next, we calculate the BPU in each limitation as in other examples. The first limitation's BPU is [0.000 23%,200%].

Next, to calculate the BPU, we need both (i) the *minimum PU* (uniform distribution) and (ii) the *maximum PU* (burst distribution). For (i), the *minimum PU* is 99/100 = 99%. For (ii), in 1 second there will be a burst of 99 requests. The *maximum PU* is 99/100 = 99%. Therefore, the BPU is [99%,99%].

Now, we aggregate both BPUs: first, we get the maximum value of the minimum PU: *max(0.000 23%, 99%)=99%*. Next, we obtain the minimum value of the maximum PU: *min(200%,99%)=99%*.

Therefore, as a result, we got [99%,99%]. Given that it does not surpass the capacity, we state that there is no capacity conflict.

4.2.4 Limitation consistency conflict (VC3.2)

```
Limitation:
  Limit: 1000 KB / 1 month
Limitation:
  Limit: 1000 requests / 1 month
Relationship
1 request = 0.5 KB
```

Listing 12. Validity criterion VC3.2 (no limitation consistency conflict by a related metric)

On the one hand, in Listing 12 **there is no limitation consistency conflict by a related metric** because, if each request consumes 0.5 KB, in 1000 KB one would have at most $1000/0.5 = 2000$ requests. Given that $1000 < 2000$, the value of the limit on requests would not lead to any conflict.

```
Limitation:
  Limit: 1000 KB / 1 month
Limitation:
  Limit: 5000 requests / 1 month
(Relationship: 1 request = 0.5 KB)
```

Listing 13. Validity criterion VC3.2 (limitation consistency conflict by a related metric)

On the other hand, in Listing 13 **there is a limitation consistency conflict by a related metric** because, if each request consumes 0.5 KB, in 1000 KB one would have at most $1000/0.5 = 2000$ requests. Since $5000 > 2000$, one could never reach 5000 requests, and there is therefore a conflict deriving from the relationship between metrics.

4.2.5 Cost consistency conflict (VC4.2)

```
Plan 1:
  Limitation:
    Limit: 10 requests / 1 second
  Limitation:
    Limit: 100 requests / 1 day
  Cost: $10 / 1 month

Plan 2:
  Limitation:
    Limit: 100 requests / 1 second
  Limitation:
    Limit: 1000 requests / 1 day
  Cost: $100 / 1 month
```

Listing 14. Validity criterion VC4.2 (no cost consistency conflict)

An example where **there is no cost consistency conflict** can be observed in Listing 14, because any limitation in one of the plans is less restrictive than the equivalent in the other plan, but this other plan is also cheaper. In this example, plan 1 has stricter limitations and a lower cost than plan 2. The increase from 10 to 100 per-second requests, and from 100 to 1000 daily requests is also represented in the cost – from \$10 to \$100.

```
Plan 1:
  Limitation:
    Limit: 10 requests / 1 second
  Limitation:
    Limit: 100 requests / 1 day
  Cost: $10 / 1 month

Plan 2:
  Limitation:
    Limit: 1 requests / 1 second
  Limitation:
    Limit: 1000 requests / 1 day
  Cost: $1 / 1 month
```

Listing 15. Validity criterion VC4.2 (cost consistency conflict)

On the contrary, in Listing 15 **there is a cost consistency conflict** in the two plans' limitations and cost. While the decrease in per-second requests from 10 to 1 is indeed represented in the costs going from \$10 down to \$1, the increase from 100 to 1000 daily requests is in the contrary direction to the decrease in costs. There is therefore a cost inconsistency.

5 Evaluation

In this section, we shall describe how we evaluated our proposal. In particular, the goal of the evaluation was to determine on the one hand the expressiveness of our model and whether this is enough for it to express a wide variety of real-world API pricings, and on the other which characteristics of SLAs the model is unable to express. Since we also seek to apply automated analysis techniques to solve the validity operation described in Section 4, we also need to model a variety of API limitations.

We aim to answer two main research questions (RQs), namely:

- **RQ1 - Expressiveness**. *Is the modeling language expressive enough to model real-world API pricings?* We validated our language based on the analysis of two datasets containing a total of 244 selected APIs, with multiple pricings.
- **RQ2 - Automation**. *Is it possible to automate the validation of API pricings?* Pricings should be valid and be devoid of inconsistencies between in their definition. We developed a tool to automate the analysis of API plans and solve the validity operation in 4.

5.1 RQ1 - Expressiveness

5.1.1 Analysing API limitations and pricing

For this analysis, we considered two earlier contributions: (i) our work in Gamez-Diaz et al. [1] in which we analysed a set of 69 APIs from two of the largest API directories; (ii) the work of Neumann et al. [6] in which the authors analysed a set of 500 APIs from the top most popular 4000 websites in the Alexa ranking [7]. The websites are ordered by their 1-month Alexa traffic rank, calculated using a combination of average daily visitors and page views over the preceding month.

We adapted and applied the process described in contribution [1] (i), screening the API repositories and applying the inclusion criterion described by the authors (*which includes more than 5000 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 2966 potential APIs to study. For a 90% confidence level and a 15% margin of error [8], 30 APIs ought to be selected.

In contribution [6] (ii), the authors analysed 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 analyse 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 analysed.

Combining the 30 APIs extracted according to [1] and the 217 from the dataset in [6] and removing duplicates left a dataset of 244 APIs – the *Governify4APIs dataset*. Table 1 presents the overall picture of the analysis that was carried out.

TABLE 1
Main numbers of our Governify4APIs dataset.

APIs from Gamez-Diaz (N=2966)	30 (90% conf., 15% error)
APIs from Neumann (N=217)	217
Total APIs after removing dups.	244
APIs having pricing	152 out of 244
Manually modeled APIs (N=244)	32 (85% conf., 11% error)

We analysed 244 APIs in regard to two main types of attributes: *limitations* and *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 Governify4APIs dataset comprises 244 APIs, only 152 of them, the 62.3%, present a pricing or a plan. Consequently, the analysis of pricing is limited to this reduced dataset. Table 2 presents some results of the analysis.

TABLE 2
Results of the analysis in real-world APIs.

	N=244	N=152
Has limitations	95.5%	94%
Has quotas	55.7%	67.5%
Has rates	84%	76.8%
Has quotas and rates	44.3%	50.3%
Simple cost (e.g., monthly price)	57.4%	92.1%
Has a pay-as-you-go cost model	9.8%	15.8%
Includes overage cost	4.9%	7.9%

Limitations analysis: Most APIs (95.5%) have limitations in terms of quotas (55.7%) or rates (84.0%). Almost half use a combination of the two (44.3%). These limitations are usually rather simple (e.g., monthly requests for quotas and secondly requests for rates). However, a minority tend to have a higher level of expressivity. For example, they use the information from the HTTP request – from query parameters (2.5%) to other low-level aspects of the HTTP message such as headers, body, etc. (10.2%). A marginal number of APIs allow consumers to exceed the limitation value once or many times per month (1.2%).

Pricing analysis: the vast majority of the APIs (92.1%) include a simple (e.g., monthly) cost. Nonetheless, they may have operation costs (15.8%) or include overage costs (7.9%). Finally, a minority have purchasable add-ons or extras (6.6%) or their pricing is calculated based on the number of users (11.8%).

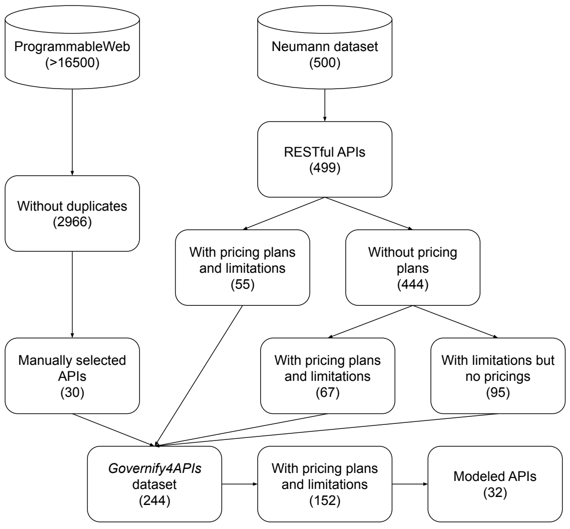

ProgrammableWeb (>16500)
Neumann dataset (500)
RESTful APIs (499)
Without duplicates (2966)
With pricing plans and limitations (55)
Without pricing plans (444)
Manually selected APIs (30)
With pricing plans and limitations (67)
With limitations but no pricings (95)
Governify4APIs dataset (244)
With pricing plans and limitations (152)
Modeled APIs (32)

Fig. 6. Diagram of the database filtering process, starting from the Gamez and Neumann datasets.

Given these 32 APIs, we analysed the different metrics included in the documentation of each of them. We found a total of 129 metrics, although different providers may name a same metric with different names (e.g. requests per second and transactions per second). 57.36% of these metrics are domain independent (such as requests, storage or users), while 42,64% are dependent (such as emails, documents or invoices). We grouped the 129 metrics in different categories based on their similarity, resulting in 15 categories. The most populated one is *requests/TU*, including 41 metrics. The second one is *AI* (Artificial Intelligence, with 19 metrics, as some of the analysed 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.

Recently, we analysed 95 different APIs from the Governify4APIs dataset to find new plans and metrics. We found some interesting information about these APIs. The most common names for pricing plans are *Basic*, *Premium*, *Enterprise* and *Professional*. We also found that the most common number of plans in an API is 4, closely followed by 3. Additionally, it seems that more expensive plans tend to have more SLAs, such as high availability or 24/7 customer support. More information about this analysis can be found in dataset D02 in [9].

5.1.2 Modeling API pricings

In this work, we seek to perform automated operations through analysers that solve the validity operation regarding pricings, as were discussed in Section 4. Nevertheless, before using any analyser or tool, these pricings have to be modeled. This subsection describes a validation of our Governify4APIs 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 an errorfree curated list of 32 API pricings with the model in subsection 3.2, which represents the variability found in the industry.

As noted above, we analysed different attributes of the Governify4APIs dataset of 244 APIs. The next step would be to write the OAS and SLA4OAI specification of every API so that it can be passed to the automated analyser. However, since this is a time-consuming task, we selected a representative subset with a confidence level of 85% and an error margin of 11%, resulting in in a subset of 32 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.

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 32 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*.

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., *OpenWeatherMap*.

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 case, i.e., the cost per single selectable unit (e.g., the total cost in a plan with 3 users, the minimum amount).

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, months...) use fixed windows, and limits with shorter periods (e.g. seconds, minutes...) use sliding windows. This decision is based on the research in [1].

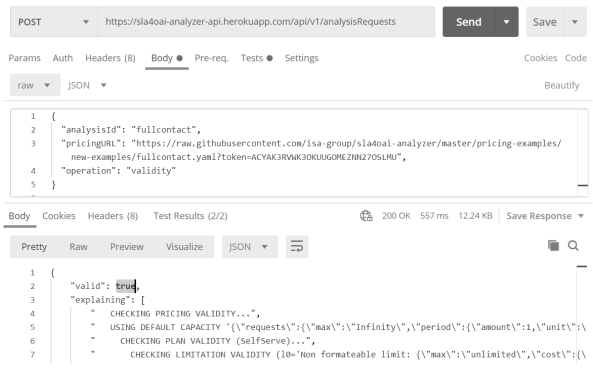
Open issues:

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.

5.2 RQ2 - Automation

The main operation regarding API limitations is *Validity* (cf. Section 4). This operation, in order to be useful for practitioners, need to be automated by means of a specific tool. To this end, we have developed *sla4oai-analyzer*, an initial version of a publicly available command-line tool [10]. Once installed, given a SLA4OAI file, the command *sla4oai-analyzer -o <operation> -f <myFile.yaml>* will initiate the validity analysis for this file.



https://sla4oai-analyzer-api.herokuapp.com/api/v1/analysisRequests

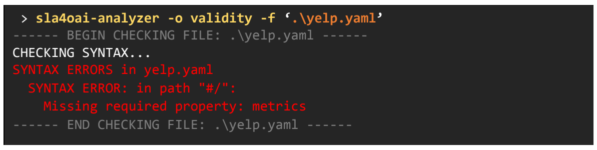
```
{
  "analysisId": "fullcontact",
  "pricingURL": "https://raw.githubusercontent.com/isa-group/sla4oai-analyzer/master/pricing-examples/
    new-examples/fullcontact.yaml?token=ACYAK3RVWK3OKUUGOMEZNN27OSLMU",
  "operation": "validity"
}
```

200 OK 557 ms 12.24 KB

```
{
    "valid": true,
    "explaining": [
        "   CHECKING PRICING VALIDITY...",
        "   USING DEFAULT CAPACITY '{\"requests\":{\"max\":\"Infinity\",\"period\":{\"amount\":1,\"unit\":\
        "     CHECKING PLAN VALIDITY (SelfServe)...",
        "       CHECKING LIMITATION VALIDITY (l0='Non formateable limit: {\"max\":\"unlimited\",\"cost\":{\
```

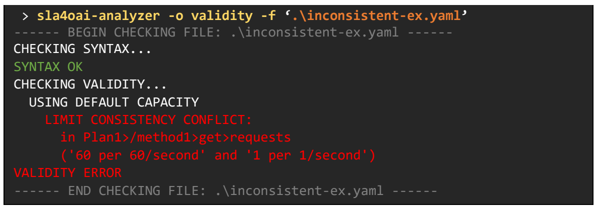
Fig. 7. Simple UI for the *sla4oai-analyzer* API.

In order to be integrated into the Governify framework, this tool is also available as an API [11]. Furthermore, we provide a Postman documentation[5] with 32 examples of invocations of the validity operation using this API. Figure 7 is a screenshot of a simple UI for using the API while analysing the validity of the FullContact pricing.



```
 > 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 ------
```

Fig. 8. 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 ------
```

Fig. 9. Tool running the validity operation with errors.

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). Figure 9 depicts a consistency conflict detected by this tool, caused by a modeling mistake.

As illustrations of some outputs of the tool, Figure 8 shows a pricing with *syntax errors* and Figure 9 a *consistency conflict*.

5.3 Threats to validity

We need to analyse the various validity threats that may have influenced our work, and the ways in which we tried to mitigate them.

5.3.1 Internal validity

These threats refer to the factors that introduce bias in our work and affect its utility. In our case, the main threat is the subjective and manual review process of the documentation of 32 different APIs. As a result, some limitations might have been overlooked and omitted in our models. To mitigate this threat, we checked each API multiple times and made the appropriate changes when necessary, recording each change, taking screenshots of their websites and saving their URLs. Moreover, some APIs updated their documentation over the span of time of writing this paper, so we updated their corresponding models. In some cases, pricing plans were removed from the API website, so we used the *Wayback Machine* tool [6] to retrieve older versions of these pages.

5. https://documenter.getpostman.com/view/683324/TVKEYHv8
6. https://archive.org/web/

5.3.2 External validity

This refers to the extent to which we can generalise from the results of our work. One threat is representativity. The APIs were extracted from two sources – 217 valid APIs from the Neumann dataset and 2966 APIs replicating the extraction in Gamez-Diaz. Since this is too large a number to be analysed manually, we opted to select a representative sample with a 90% confidence level and 15% margin of error – i.e., 32 APIs. This means that our model may not generalise to the rest of the APIs in the dataset. To mitigate this threat, we selected APIs from a wide range of domains, and some of them are popular and extensively consumed by a large number of users.

Another threat is that whereas our model supports the majority of the attributes analysed, it does not support some of them. With that in consideration, we concluded that we are able to model 84% (N=244) of the APIs regarding limitation attributes and 89.3% (N=152) regarding pricing attributes, meaning that we are confident enough on the ability to generalise our model. Those APIs that can be modeled regarding both limitation and pricing attributes comprise 76.2% of the overall Governify4APIs dataset.

Finally, our proposal has not yet been validated with other API consumers and providers. This means that SLA4OAI might not be as usable or useful as we intended. To mitigate this, we provide a JSON schema [5] to help understand the specification. Additionally, we proposed the addition of SLA4OAI within the OpenAPI Specification.

6 Related Work

Our goal has been to analyse the current proposal for defining API pricings. As far as we know, we are the first ones to present a proposal that integrates with the OpenAPI Specification. Certain related proposals have focused on Web services while others allow the definition of entire Service-Level Agreements (SLAs). We have analysed the most prominent academic and industrial proposals that aim at defining SLAs in both traditional Web services and cloud scenarios in order to outline their scope and limitations. Specifically, in Table 3, we consider 7 aspects to analyse in each SLA proposal, namely: **F1** determines the format in which the document is written; **F2** indicates whether the target domain is that of Web services; **F3** indicates whether it can model more than one offering (e.g., different operations of a Web service); **F4** indicates whether it allows modeling hierarchical models or overriding properties and metrics; **F5** indicates whether temporal concerns can be modeled (e.g., in metrics); **F6** indicates whether there exists a tool assisting users to model with that proposal; **F7** indicates whether there exists a tool or framework to enact the SLA.

In [6], the authors analysed more than 500 publicly available APIs to identify the different trends in the current industrial landscape. In [1], we analysed a set of 69 real APIs in the industry to characterize the variability of their offers, drawing a number of invaluable conclusions about real-world APIs such as: (i) most APIs provide different capabilities depending on the tier or plan that the API consumer is willing to pay for; (ii) usage limitations are a common aspect that all APIs describe in their offers; (iii) limitations on API requests are the commonest, including quotas over static time periods (e.g., *1.000 requests each natural day*) and rates for dynamic

TABLE 3
Analysis of SLA models.

Name	**F1**	**F2**	**F3**	**F4**	**F5**	**F6**	**F7**
ysla [12]	YAML		✓	✓	✓	✓	✓
SLAC [13]	DSL					✓	✓
CSLA [14]	XML		✓			✓	
L-USDL Ag. [15]	RDF	✓	✓		†	✓	
rSLA [16]	Ruby	✓		✓	✓		✓
SLAng [17]	XML	✓					
WSLA [18]	XML	✓	✓		✓		
SLA* [19]	XML	✓	✓		✓		
WS-Ag. [20]	XML	✓	✓	✓	†		

† Supported with minor enhancements or modifications.

time periods (e.g., *3 requests per second*); (iv) offers can include a broad number of metrics over other aspects of the API that may be domain-independent (such as the number of results returned or the size of the request in bytes) or domain-dependent (such as the CPU/RAM consumption during the request processing or the number of different resource types). Based on these conclusions, we identified the need for non-functional support in the API development life-cycle and the high level of expressivity present in the API offers.

Some of our previous studies in this line include: constructing a proof of concept to highlight the importance of the automated analysis of limitations [21]; presenting an initial ecosystem of tools to support the SLA4OAI proposal [22]; ascertaining the importance of the role of SLAs in APIs in an industrial context with high-level OpenAPI partners such as PayPal or Google Apigee [3]; work on an initial SLA4AOI proposal [4].

Based on the comparison of the different SLA models, we would draw the following conclusions. (i) None of the specifications provides any support for or alignment with the OpenAPI Specification. (ii) Most of the approaches provide a specific syntax for RDF/XML (some, however, lack such a syntax), but there is no explicit support for YAML or JSON serializations. (iii) While many proposals are complete, others leave some parts open for practitioners to implement. (iv) A number of proposals aim to model Web services but are focused on traditional SOAP Web services rather than RESTful APIs, so that they do not address the modeling standardization of the RESTful approach in which the concept of a resource is clearly unified (a URL), and the amount of operations is limited (to HTTP methods such as GET, POST, PUT and DELETE). This lack of support for RESTful modeling prevents the approach from having a concise and compact binding between its functional and non-functional aspects. (v) They have insufficient expressivity to model such limitations as quotas and rates for each resource and method, and with full management of the temporality (static or sliding time windows and periodicity) present in the typical industrial API SLAs. (vi) Most are designed to model a single offer, and usually lack support for hierarchical modeling or overriding properties and metrics (F4). In such a context, they cannot model a set of tiers or plans that yield a complex offer which maintains coherence. Instead, they rely on a manual process which is typically error prone. (vii) Finally, the ecosystem of tools proposed in each approach (when indeed such an ecosystem exists) is extremely limited, and is aimed solely at being a prototype. Apparently, neither are they integrated into a developer community nor is there evidence of their use by practitioners in the industry.

There has also been a proposal of a model-driven approach to defining API service licenses and an API SLA analyser system which utilizes the proposed license model to uncover SLA violations in real-time [23]. Other work has sought to determine whether one can know how to price SaaS by summarizing existing knowledge from different research areas and SaaS pricing practice [24], and [25] presents a three-level productization model for different phases of SaaS businesses.

To the best of our knowledge, while there does exist previous work on analysing pricing in general, there has been no work focused on relating API pricing and limitations (i.e., quotas, rates).

7 Future Work

In [26], the authors distinguish five types of incoming workloads: *static*, *periodic*, *once-in-a-lifetime*, *unpredictable*, and *continuously changing*. If we introduce the concept of temporality in the pricing, i.e., to consider that certain plans have a determined temporal validity (e.g., day/night plan), the operations have to be adapted to consider this temporality. Joining temporality with workload models, one could automate the management of this type of advanced scenarios which require infrastructures that are dynamic (e.g., instances that start or stop and have a variable cost).

Furthermore, alert systems can be defined which notify users when certain percentages of consumption of the limitations are reached, so that they can take this situation into account and adjust their consumption accordingly.

With respect to the tool, as it is just a proof of concept, it lacks various features. (i) It is a command-line tool and an API, and is not really useful for end-users. (ii) The implementation of *cost conflicts* is too naive as it only supports simple cost values (but neither overages nor operation costs).

In our model, we identified two open issues that should be addressed: (i) extend the model to incorporate parameters that are limited to a certain range of allowed values instead of a maximum value; (ii) expand the overage concept to establish a limit on the overage itself.

8 Conclusions

In this paper, we have proposed a model derived from the plurality of business and pricing models that rigorously captures the nature of a limitation. In doing so, we have studied both the limitations and the pricing plans of a set of 244 APIs using two different datasets. We then presented an errorfree curated list of 32 API pricings with a formal validated model that represents the variability found in the industry. The Governify4APIs dataset used in the analysis is publicly available as it could be a useful resource for both academics and practitioners.

During the process of modeling the curated subset of 32 APIs from the entire Governify4APIs dataset of 244 analysed APIs, we can extract some lessons that we learned which might be useful for practitioners responsible for modeling API pricing or API limitations. (i) When creating the initial version of the model, many parts were inadvertently missed. This was found to be crucial to having a strong syntax validation process. (ii) With respect to metrics, once we had defined the limitations, we often forgot about defining the metric in its corresponding section. Furthermore, after modeling several plans, it was common to reuse the metric definition by cropping and pasting. This resulted in a growing section of unused metrics.

While our *Governify4APIs* model was serialized to be compatible with the OpenAPI Specification (SLA4OAI), it can easily be exported to other API descriptions (such as RAML, API Blueprint, I/O Docs, WSDL, or WADL).

We have identified a common query that arises from the limitations in APIs: their validity. We found out that this operation can be automated, giving as a result a tool that can help both customers and providers in dealing with API limitations.

Since our work is aligned to open standards such as the OpenAPI Initiative, it can pave the way to creating an open ecosystem of tools for automating the development process, taking into account API limitations – limitations-aware testing or API clients, for example.

In conclusion, this communication has presented (i) *Governify4APIs*, a rigorous agnostic model of API pricing, and *SLA4OAI*, a specific serialization aligned with the OpenAPI Specification, (ii) an analysis operation in API pricing automated by *sla4oai-analyzer*, (iii) a rigorous analysis of 244 APIs and a curated publicly available [9] dataset of 32 API pricing models; and (iv) an initial tool prototype to automate the operation.

Acknowledgments

This work has been partially supported by FEDER/Ministerio de Ciencia e Innovación – Agencia Estatal de Investigación under project HORATIO (RTI2018101204-B-C21), the Andalusian Government under project EKIPMENT-PLUS (P18-FR-2895), and the FPU scholarship program granted by the Spanish Ministry of Education, Culture, and Sports (FPU15/02980 and FPU19/00666). The authors wish to thank Daniel Arteaga and Felipe Vieira da Cunha for their invaluable technical contributions.

References

[1] A. Gamez-Diaz, P. Fernandez, and A. Ruiz-Cortes, "An analysis of restful apis offerings in the industry," in *ICSOC*. Springer, 2017, pp. 589–604.

[2] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems.* 9781491929124, 2016.

[3] A. Gamez-Diaz, P. Fernandez, A. Ruiz-Cortes, P. J. Molina, N. Kolekar, P. Bhogill, M. Mohaan, and F. Méndez, "The role of limitations and slas in the api industry," in *ESEC/FSE*, ser. ESEC/FSE 2019. ACM, 2019.

[4] A. Gamez-Diaz, P. Fernandez, and A. Ruiz-Cortes, "Automating sla-driven api development with sla4oai," in *ICSOC*. Cham: Springer, 2019.

[5] A. Gamez-Diaz, R. Fresno-Aranda, P. Fernandez, and A. Ruiz-Cortes, "Sla4oai json schema," Jul. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5118599

[6] A. Neumann, N. Laranjeiro, and J. Bernardino, "An Analysis of Public REST Web Service APIs," *IEEE TSC*, 2018.

[7] Alexa, "The top 500 sites on the web," https://www.alexa.com/topsites, 2020, [Online; accessed June-2020].

[8] L. Isserlis, "On the value of a mean as calculated from a sample," *Journal of the Royal Statistical Society*, vol. 81, no. 1, pp. 75–81, 1918.

[9] A. Gamez-Diaz, R. Fresno-Aranda, P. Fernandez, and A. Ruiz-Cortes, "isa-group/2021-paper-pricing-modeling-framework- for-apis-dataset," Jul. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4697208

[10] ——, "isa-group/sla4oai-analyzer," Jul. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5146288

[11] ——, "isa-group/sla4oai-analyzer-api," Jul. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5146290

[12] R. Engel, S. Rajamoni, B. Chen, H. Ludwig, and A. Keller, "ysla: Reusable and configurable slas for large-scale sla management," in *CIC*, 2018, pp. 317–325.

[13] R. B. Uriarte, F. Tiezzi, and R. D. Nicola, "Slac: A formal service-level-agreement language for cloud computing," in *UCC*, ser. UCC '14. IEEE Comp. Soc., 2014, pp. 419–426.

[14] Y. Kouki, F. A. de Oliveira, S. Dupont, and T. Ledoux, "A language support for cloud elasticity management," in *CCGrid*. IEEE, may 2014, pp. 206–215.

[15] J. M. García, P. Fernández, C. Pedrinaci, M. Resinas, J. Cardoso, and A. Ruiz-Cortés, "Modeling Service Level Agreements with Linked USDL Agreement," *IEEE TSC*, vol. 10, no. 1, pp. 52–65, 1 2017.

[16] S. Tata, M. Mohamed, T. Sakairi, N. Mandagere, O. Anya, and H. Ludwig, "rsla: A service level agreement language for cloud services," in *CLOUD*. IEEE, June 2016, pp. 415–422.

[17] D. D. Lamanna, J. Skene, and W. Emmerich, "SLAng: A language for defining service level agreements," in *FTDCS*, vol. 2003-Janua, 2003, pp. 100–106.

[18] H. Ludwig, A. Keller, A. Dan, and R. King, "A service level agreement language for dynamic electronic services," in *WECWIS*. IEEE Comput. Soc., 2002, pp. 25–32.

[19] K. T. Kearney, F. Torelli, and C. Kotsokalis, "SLA * An abstract syntax for Service Level Agreements," in *GRID*. IEEE, 10 2010, pp. 217–224.

[20] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu, "Web Services Agreement Specification (WS-Agreement)," Open Grid Forum, Tech. Rep., 2004.

[21] A. Gamez-Diaz, P. Fernandez, C. Pautasso, A. Ivanchikj, and A. Ruiz-Cortes, "Electra: Induced usage limitations calculation in restful apis," in *ICSOC Workshops*. Springer, 2019, pp. 435–438.

[22] A. Gamez-Diaz, P. Fernandez, and A. Ruiz-Cortes, "Governify for apis: Sla-driven ecosystem for api governance," in *ESEC/FSE*, ser. ESEC/FSE 2019. ACM, 2019.

[23] M. Vukovic, L. Zeng, and S. Rajagopal, "Model for service license in api ecosystems," in *Service-Oriented Computing*. Springer, 2014, pp. 590–597.

[24] A. Saltan, "Do we know how to price saas: A multi-vocal literature review," in *IWSiB*. ACM, 2019, p. 7–12. [Online]. Available: https://doi.org/10.1145/3340481.3342731

[25] T. Yrjönkoski and K. Systä, "Productization levels towards whole product in saas business," in *IWSiB*, ser. IWSiB. New York, NY, USA: Association for Computing Machinery, 2019, p. 42–47. [Online]. Available: https://doi.org/10.1145/3340481.3342737

[26] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, *Cloud Computing Patterns*. Springer, 2014.

Antonio Gámez-Díaz is a predoctoral researcher at the University of Sevilla, where he received a BSc degree and an MSc degree in Software Engineering in 2016, with a competitive predoctoral fellowship (FPU) granted by the Spanish government. His research interests are focused on Service-Oriented Computing. Due to the precarious situation of science in his country, he left the Academia in 2021 and he is working at VMware, where he contributes in a number of cloud-native open source projects. Contact him at antoniogamez@us.es; https://personal.us.es/agamez2.

Rafael Fresno-Aranda is a predoctoral researcher at the University of Sevilla. He received a BSc degree in 2019 and an MSc degree in Software Engineering in 2020. He recently got a competitive predoctoral fellowship (FPU), granted by the Spanish government. His research focuses on Service-Oriented Computing, including the analysis of microservices architectures and RESTful APIs. He has contributed to open source tools and utilities, and has collaborated with the University of California Berkeley to develop an auditor framework for software engineering students. Contact him at rfresno@us.es.

Pablo Fernández is an associate professor at the University of Sevilla, Spain, and a member of the ISA Research Group. His current research is focused on the automated governance of organizations based on service level agreements and commitments. He has been the lead architect for several PPP projects in scenarios of public administrations and major firms. Contact him at pablofm@us.es; https://www.isa.us.es/members/pablo.fernandez

Prof. Dr Antonio Ruiz-Cortés is a Full Professor and head of the Applied Software Engineering Group at the University of Seville (Spain). His current research focuses on service-oriented computing, business process management, testing and software product lines. He is an associate editor of Springer Computing, recipient of the Most Influential Paper of SPLC (2017) and VAMOS award (2020), and elected member of the Academy of Europe. Contact him at aruiz@us.es; https://www.isa.us.es/members/antonio.ruiz.

Bibliography

[1] Mike P. Papazoglou and Willem-Jan van den Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16(3):389–415, 2007. ISSN 0949-877X. doi: 10.1007/s00778-007-0044-3. (page 5).

[2] Dimitrios Georgakopoulos and Michael P. Papazoglou. *Service-Oriented Computing*. The MIT Press, Cambridge, Mass, 2008. ISBN 0262072963. doi: 10.5555/1522450. (page 5).

[3] Tony Mauro. Microservices at Netflix: Lessons for architectural design, Feb 2015. URL `https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/`. (page 5).

[4] Bruno Costa, Paulo F. Pires, Flávia C. Delicato, and Paulo Merson. Evaluating REST architectures—Approach, tooling and guidelines. *Journal of Systems and Software*, 112:156–180, 2016. ISSN 0164-1212. doi: 10.1016/j.jss.2015.09.039. (page 5).

[5] Mansour Kavianpour. SOA and Large Scale and Complex Enterprise Transformation. In Bernd J. Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *Service-Oriented Computing – ICSOC 2007*, pages 530–545, Berlin, Heidelberg, 2007. Springer. ISBN 9783540749745. doi: 10.1007/978-3-540-74974-5_5. (page 5).

[6] Stephan Aier, Philipp Offermann, Marten Schönherr, and Christian Schröpfer. Implementing Non-functional Service Descriptions in SOAs. In Dirk Draheim and Gerald Weber, editors, *Trends in Enterprise Application Architecture*, TEAA, pages 40–53, Berlin, Heidelberg, 2007. Springer. ISBN 9783540759126. doi: 10.1007/978-3-540-75912-6_4. (page 5).

[7] Sam Newman. *Building microservices: designing fine-grained systems*. O'Reilly Media, Sebastopol, CA, 2015. ISBN 9781491950357. (page 5).

[8] Wei Tan, Yushun Fan, Ahmed Ghoneim, M. Anwar Hossain, and Schahram Dustdar. From the Service-Oriented Architecture to the Web API Economy. *IEEE Internet*

Computing, 20(4):64–68, 2016. ISSN 1941-0131. doi: 10.1109/MIC.2016.74. (page 6).

[9] Michele Bonardi, Maurizio Brioschi, Alfonso Fuggetta, Emiliano Sergio Verga, and Maurilio Zuccalà. Fostering Collaboration through API Economy. In *Proceedings of the 3rd International Workshop on Software Engineering Research and Industrial Practice*, SER&IP, page 32–38, New York, USA, 2016. Association for Computing Machinery. ISBN 9781450341707. doi: 10.1145/2897022.2897026. (page 6).

[10] Antonio Parejo, Sebastián García, Enrique Personal, Juan Ignacio Guerrero, Antonio García, and Carlos León. OpenADR and Agreement Audit Architecture for a Complete Cycle of a Flexibility Solution. *Sensors*, 21(4):1204, Feb 2021. ISSN 1424-8220. doi: 10.3390/s21041204. (page 6).

[11] H. Ludwig, A. Keller, A. Dan, and R. King. A service level agreement language for dynamic electronic services. In *Proceedings of the 4th IEEE International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems*, WECWIS, pages 25–32, Newport Beach, USA, 2002. IEEE Computer Society. ISBN 0769515673. doi: 10.1109/WECWIS.2002.1021238. (page 7).

[12] D. D. Lamanna, J. Skene, and W. Emmerich. SLAng: A language for defining service level agreements. In *Proceedings of the IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, volume 2003 of *FTDCS*, pages 100–106, San Juan, USA, 2003. ISBN 0769519105. doi: 10.1109/FTDCS.2003.1204317. (page 7).

[13] Keven T. Kearney, Francesco Torelli, and Constantinos Kotsokalis. SLA*: An Abstract Syntax for Service Level Agreements. In *Proceedings of the 11th IEEE/ACM International Conference on Grid Computing*, GRID, pages 217–224, Brussels, Belgium, 2010. IEEE Computer Society. ISBN 9781424493470. doi: 10.1109/GRID.2010.5697973. (page 7).

[14] Alain Andrieux, Karl Czajkowski, A. Dan, Kate Keahey, H. Ludwig, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web Services Agreement Specification (WS-Agreement). Technical report, Open Grid Forum, 2004. (page 7).

[15] Rafael Brundo Uriarte, Francesco Tiezzi, and Rocco De Nicola. SLAC: A Formal Service-Level-Agreement Language for Cloud Computing. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, UCC, pages 419–426, Washington, USA, 2014. IEEE Computer Society. ISBN 9781479978816. doi: 10.1109/UCC.2014.53. (page 7).

[16] Yousri Kouki, Frederico Alvares de Oliveira, Simon Dupont, and Thomas Ledoux. A language support for cloud elasticity management. In *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, CCGrid, pages 206–215, Chicago, USA, 2014. IEEE Computer Society. ISBN 9781479978816. doi: 10.1109/CCGrid.2014.17. (page 7).

[17] S. Tata, M. Mohamed, T. Sakairi, N. Mandagere, O. Anya, and H. Ludwig. rsla: A service level agreement language for cloud services. In *Proceedings of the IEEE 9th International Conference on Cloud Computing*, CLOUD, pages 415–422, San Francisco, USA, 2016. IEEE Computer Society. doi: 10.1109/CLOUD.2016.0062. (page 7).

[18] José María García, Pablo Fernández, Carlos Pedrinaci, Manuel Resinas, Jorge Cardoso, and Antonio Ruiz-Cortés. Modeling Service Level Agreements with Linked USDL Agreement. *IEEE Transactions on Services Computing*, 10(1):52–65, 2017. ISSN 19391374. doi: 10.1109/TSC.2016.2593925. (page 7).

[19] R. Engel, S. Rajamoni, B. Chen, H. Ludwig, and A. Keller. ysla: Reusable and Configurable SLAs for Large-Scale SLA Management. In *Proceedings of the IEEE 4th International Conference on Collaboration and Internet Computing*, CIC, pages 317–325, Philadelphia, USA, 2018. IEEE Computer Society. ISBN 9781538695029. doi: 10.1109/CIC.2018.00050. (page 7).

[20] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *ITIL® Foundation, ITIL 4 edition*. TSO, The Stationery Office, 2019. ISBN 9780113316076. (page 8).

[21] Antonio Gámez-Díaz, Pablo Fernández, and Antonio Ruiz-Cortés. An Analysis of RESTful APIs Offerings in the Industry. In Michael Maximilien, Antonio Vallecillo, Jianmin Wang, and Marc Oriol, editors, *Service-Oriented Computing – ICSOC 2017*, pages 589–604, Cham, 2017. Springer International Publishing. ISBN 9783319690353. doi: 10.1007/978-3-319-69035-3_43. (pages 19, 42, 44).

[22] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc, 2016. ISBN 9781491929124. (page 20).

[23] Antonio Gámez-Díaz, Pablo Fernández, Antonio Ruiz-Cortés, Pedro J. Molina, Nikhil Kolekar, Prithpal Bhogill, Madhuranjan Mohaan, and Francisco Méndez. The role of limitations and SLAs in the API industry. In *Proceedings of the 27th ACM Joint*

European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, pages 1006—-1014, New York, USA, 2019. Association for Computing Machinery. ISBN 9781450355728. doi: 10.1145/3338906.3340445. (pages 22, 42, 44).

[24] Antonio Gámez-Díaz, Pablo Fernández, and Antonio Ruiz-Cortés. Automating SLA-Driven API development with SLA4OAI. In Sami Yangui, Ismael Bouassida Rodriguez, Khalil Drira, and Zahir Tari, editors, *Service-Oriented Computing – ICSOC 2019*, pages 20–35, Cham, 2019. Springer International Publishing. ISBN 9783030337025. doi: 10.1007/978-3-030-33702-5_2. (pages 22, 23, 43, 45).

[25] Antonio Gámez-Díaz, Rafael Fresno-Aranda, Pablo Fernández, and Antonio Ruiz-Cortés. Sla4oai json schema, 2021. (page 23).

[26] Antonio Gámez-Díaz, Pablo Fernández, and Antonio Ruiz-Cortés. Governify for APIs: SLA-Driven ecosystem for API governance. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 1120—-1123, New York, USA, 2019. Association for Computing Machinery. ISBN 9781450355728. doi: 10.1145/3338906.3341176. (pages 31, 43, 45).

[27] Antonio Gámez-Díaz, Pablo Fernández, Cesare Pautasso, Ana Ivanchikj, and Antonio Ruiz-Cortés. ELeCTRA: Induced Usage Limitations Calculation in RESTful APIs. In Xiao Liu, Michael Mrissa, Liang Zhang, Djamal Benslimane, Aditya Ghose, Zhongjie Wang, Antonio Bucchiarone, Wei Zhang, Ying Zou, and Qi Yu, editors, *Service-Oriented Computing – ICSOC 2018 Workshops*, pages 435–438, Cham, 2019. Springer International Publishing. ISBN 9783030176426. (pages 31, 35, 43, 45, 101).

[28] Antonio Gámez-Díaz, Rafael Fresno-Aranda, Pablo Fernández, and Antonio Ruiz-Cortés. isa-group/sla4oai-analyzer, 2021. (page 35).

[29] Antonio Gámez-Díaz, Rafael Fresno-Aranda, Pablo Fernández, and Antonio Ruiz-Cortés. isa-group/sla4oai-analyzer-api, 2021. (page 35).

[30] Antonio Gámez-Díaz, Pablo Fernández, and Antonio Ruiz-Cortés. SLA-Driven Governance for RESTful Systems. In Lars Braubach, Juan M. Murillo, Nima Kaviani, Manuel Lama, Loli Burgueño, Naouel Moha, and Marc Oriol, editors, *Service-Oriented Computing – ICSOC 2017 Workshops*, pages 352–356, Cham, 2018. Springer International Publishing. ISBN 9783319917641. doi: 10.1007/978-3-319-91764-1_30. (page 45).

[31] Antonio Gámez-Díaz, Pablo Fernández, and Antonio Ruiz-Cortés. Fostering SLA-Driven API Specifications. In Manuel Lama, editor, *Actas de las XIV Jornadas de Ingeniería de Ciencia e Ingeniería de Servicios*, JCIS, Sevilla, Spain, 2018. Sistedes. doi: http://hdl.handle.net/11705/JCIS/2018/011. (page 45).

[32] Antonio Gámez-Díaz, Pablo Fernández, and Antonio Ruiz-Cortés. Towards SLA modeling for RESTful APIs. In Guadalupe Ortiz, editor, *Actas de las XIII Jornadas de Ingeniería de Ciencia e Ingeniería de Servicios*, JCIS, Tenerife, Spain, 2017. Sistedes. doi: http://hdl.handle.net/11705/JCIS/2017/010. (page 45).

[33] Antonio Gámez-Díaz, Pablo Fernández, and Antonio Ruiz-Cortés. Towards SLA-Driven API Gateways. In Juan Manuel Murillo, editor, *Actas de las XI Jornadas de Ingeniería de Ciencia e Ingeniería de Servicios*, JCIS, Santander, Spain, 2015. Sistedes. doi: http://hdl.handle.net/11705/JCIS/2015/005. (page 45).

[34] Javier Troya, José Antonio Parejo, Sergio Segura, Antonio Gámez-Díaz, Alfonso E. Márquez Chamorro, and Adela del Río Ortega. Flipping Laboratory Sessions in a Computer Science Course: An Experience Report. *IEEE Transactions on Education*, 64(2):139–146, 2021. doi: 10.1109/TE.2020.3016593. (page 46).

[35] Javier Troya, Sergio Segura, José Antonio Parejo, Adela del Río Ortega, Antonio Gámez-Díaz, and Alfonso E. Márquez Chamorro. Flipping Laboratory Sessions: An Experience in Computer Science. *IEEE Revista Iberoamericana de Tecnologías del Aprendizaje*, 15(3):183–191, 2020. ISSN 1932-8540. doi: 10.1109/RITA.2020.3008132. (page 46).

[36] Javier Troya, Sergio Segura, José Antonio Parejo, Adela del Río Ortega, Antonio Gámez-Díaz, and Alfonso E. Márquez Chamorro. Invirtiendo las clases de laboratorio en Ingeniería Informática: Un enfoque ágil. In Óscar Cánovas Reverte, Jesús García Molina, Pedro Enrique López de Teruel Acolea, and Antonio Ruiz Martínez, editors, *Actas de la XXV Edición de las Jornadas sobre la Enseñanza Universitaria de la Informática*, pages 15–22, Murcia, 2019. JENUI. (page 46).

[37] M. Broy. Toward a mathematical foundation of software engineering methods. *IEEE Transactions on Software Engineering*, 27(1):42–57, 2001. ISSN 1939-3520. doi: 10.1109/32.895987. (page 100).

[38] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. *Cloud Computing Patterns*. Springer, 2014. ISBN 9783709115671. doi: 10.1007/978-3-7091-1568-8. (page 101).