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).

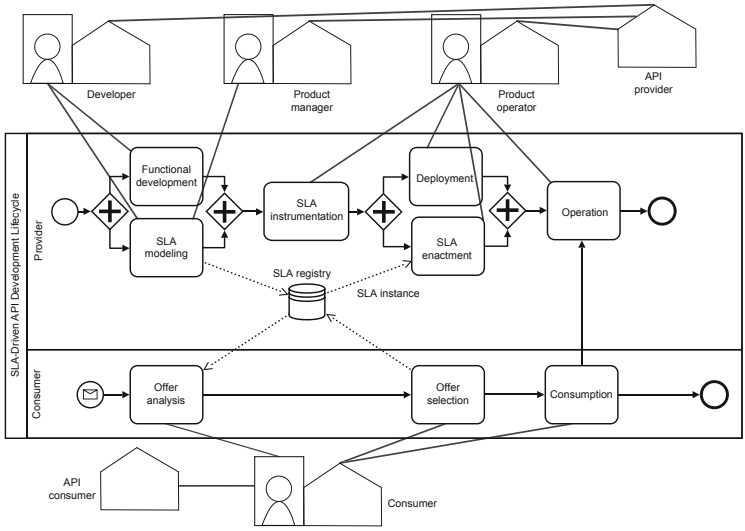

Fig. 1. SLA-Driven API development lifecycle

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.

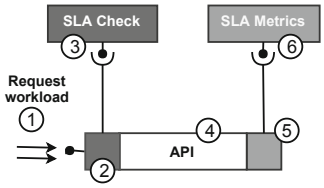

Fig. 2. Standalone SLA enforcing arch.

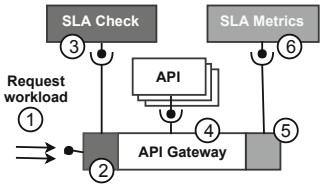

Fig. 3. Gateway SLA enforcing arch.

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