Semi-Automated Capacity Analysis of Limitation-Aware Microservices Architectures[⋆]

Rafael Fresno–Aranda[0000−0001−8480−5014][⋆⋆], Pablo Fernández[0000−0002−8763−0819], Amador Durán[0000−0003−3630−5511], and Antonio Ruiz–Cortés[0000−0001−9827−1834]

SCORE Lab, I3US Institute. Universidad de Sevilla. Seville. Spain
`{rfresno, pablofm, amador, aruiz}@us.es`

Abstract. The so-called API economy has popularised the consumption of APIs and their payment through pricing plans. This trend is very much in line with and particularly beneficial for systems with a microservices architecture that makes use of external APIs. In these cases, more and more often, the design of the system is based on the premise that its functionality is offered to its users within certain operating conditions, which implies that its architecture is aware of both the plans of external APIs and its capacity limits. We have coined these architectures as limitation-aware microservices architectures (LAMA). Furthermore, in case of a Software as a Service (SaaS) model implemented with a LAMA, the operating conditions would be explicitly grounded in the specific plans agreed with the SaaS customers. In such a context, to design a set of potential pricing plans for the LAMA and predict the expected operating conditions, it is necessary to determine the capacity limits that the architecture will offer and the cost of using external API. This is a tedious, time-consuming, and error-prone activity, so its automation would be of great value to software architects.

In this paper, we coin the term LAMA, describe the problem of automated capacity analysis of LAMAs, present a first approach to solving it by interpreting it as an optimisation problem implementable as a constraint satisfaction and optimisation problem and introduce three basic analysis operations from which any number of other operations can be modelled. Finally, a tooling support is also introduced.

Keywords: Capacity Analysis · Microservices Architectures · API · Services · QoS.

⋆ Grants RTI2018-101204-B-C21, RTI2018-101204-B-C22, PID2021-126227NB-C21, PID2021-126227NB-C22 funded by MCIN/AEI/10.13039/501100011033/ and "ERDF a way of making Europe"; grants P18-FR-2895, US-1264651 and US-1381595 funded by Junta de Andalucıa/ERDF, UE; and grant FPU19/00666 funded by MCIN/AEI/ 10.13039/501100011033 and by "ESF Investing in your future".

⋆⋆ Corresponding author

1 Introduction

The so-called *API economy* refers to an ecosystem of APIs used as business elements, where software system developers subscribe to and consume external APIs, while also providing their own APIs with their own pricing plans. WSO2 defines the API economy as *the ability for APIs to create new value and revenue streams for organisations through the use of APIs* [22]. Similarly, Capgemini defines it as *the ecosystem of business opportunities enabled by the delivery of functionality, data, and algorithms over APIs* [4]. Thus, the API economy has popularised the consumption of APIs and their payment through what are known as *pricing plans*, which describe their functionality, their capacity limits (aka limitations) and the price for using them.

The API economy is very much in line with and particularly beneficial for systems with a microservices architecture (MSA) [6] that uses external APIs. In an MSA, each microservice should have a well-defined API and make use of a standardized paradigm to integrate with the rest of microservices in the architecture and the external supporting services; usually, the most used paradigm used to define the interface and inter-operate is the RESTful paradigm.

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

Taking into consideration this common scenario in the industry, we aim to coin those architectures as *limitation-aware microservices architectures* (LAMA). In such a context, it is important to note that in the case of Software as a Service (SaaS) implemented with a LAMA, limitations play an even bigger role as the operating conditions would be explicitly grounded in the specific plans agreed with the SaaS customers. As an example, Fig. 1 shows a LAMA composed by three different services (S1, S2, S3) that make use of two external APIs (E1, E2) with different pricing plans. The LAMA customers have a pricing plan where they have to choose between the *Starter* plan and *Advanced* plan with different guaranteed operating conditions on the Requests Per Second (RPS) and a corresponding monthly price.

In order to design the LAMA pricing plans and predict the operating conditions, it is necessary to determine the capacity limits that the architecture will offer and the cost of using external APIs, analysing different scenarios; specifically, as motivating examples, we can identify three common situations: i) in order to articulate a strategic decision to define the LAMA pricing plan for a estimated scenario, we can ask about the baseline operational cost for such scenario (e.g. *Q1 - What is the cheapest operational cost for my LAMA in order to offer 2 RPS to 20 customers?*); ii) given a fixed relationship or a pre-existing ongoing contract with an external API, we could ask about the expected maximal operating conditions (e.g. *Q2 - Assuming we have a Basic plan and a Gold*

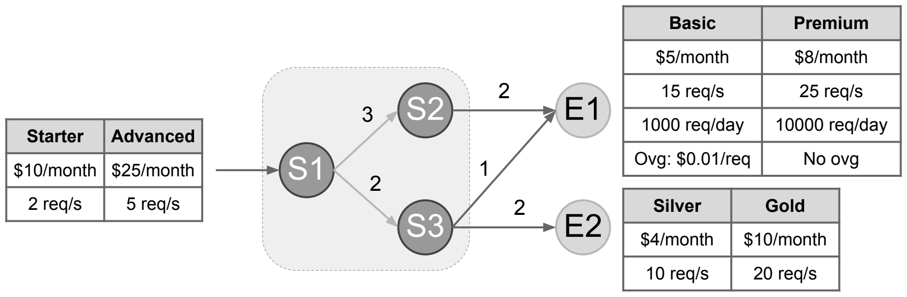

Fig. 1. A sample LAMA with 3 internal services and 2 external APIs, each of them with 2 pricing plans. Plan *Basic* has an overage cost. The LAMA offers its functionality through two pricing plans.

plan already contracted what is the maximal requests per minute (RPM) I can guarantee to all my 20 customers?); iii) in case we have a pre-existing budget limit for a given scenario, we could ask about the optimal combination of plans to be subscribed and the potential limitations I could guarantee to my customers with this combination (e.g. *Q3 - Assuming we have a monthly budget limit of $120 in my LAMA, which is the maximum RPS to each of 20 customers?*).

Beyond SaaS, it is important to note that those motivating examples can also be adapted for any LAMA, even if they represent ad-hoc systems inside a single organization and the number of customers is not relevant for the calculations, for example: *Q4 - What is the cheapest operational cost to guarantee a global operating condition of 50 RPS?*, *Q5 - Assuming we have a Basic plan and a Gold plan already contracted what is the maximal RPS I can guarantee as operating condition?* or, *Q6 - Assuming we have a monthly budget limit of $120 in my LAMA, which is the maximum RPS I can guarantee as operating condition?*. In fact, guiding the strategic decision of contracting external APIs and anticipating the different options of operating conditions of the system depending on the cost are critical aspects that could help software architects and DevOps of any LAMA.

These analysis questions deal with computer-aided extraction of useful information from a LAMA, which helps DevOps teams make certain decisions and detect potential issues. Trying to answer these questions even in a simple scenario leads to the conclusion that it is a tedious, time-consuming and error-prone activity. In addition, as the LAMA has some complexity, performing these analyses manually will be neither reliable nor cost-effective, and its automation would be of great value to software architects. We have coined this problem as the **problem of automated capacity analysis of LAMAs**. This capacity analysis comprises the automation of different *analysis operations* that arise from the **number of requests, the cost and the time**, which are the three essential cornerstones of this problem. This problem is novel and has not been fully explored in the literature. Existing work has always omitted at least one of the three essential dimensions of a LAMA. We think that this is because no existing proposal studies the relationship between them. The *QoS-aware service composition* problem is related to this topic, but does not fully address all the features of LAMAs, as described in Section 3.

In this paper, we coin the term LAMA, describe the problem of automated capacity analysis of LAMAs, and present an approach to solving it by interpreting it as an optimisation problem implementable as a constraint satisfaction and optimisation problem (CSOP). We also present an extensible catalogue of analysis operations and a public RESTful API that transforms a LAMA into an initial implementation using the MiniZinc modelling language [17] and provides solutions for basic analysis operations.

The remainder of this paper is structured as follows. In Section 2, we introduce and define the different concepts that serve as background. In Section 3, we explore related work and explain why their proposals are not useful in this situation. In Section 4, we provide an initial definition of this problem through a synthetic example. In Section 5 we present a preliminary approach and a possible model of the problem using a CSOP. In Section 6 we present a public RESTful API that provides solutions for three basic analysis operations. Finally, in Section 7 we summarise the conclusions of this paper and describe future work.

2 Background

2.1 Pricing Plans

API providers usually sell the functionality through multiple *pricing plans*, or simply *pricings* [12]. In the widely used case of RESTful APIs the functionality is defined as a set of endpoints (expressed as URLs) to resources that can be operated (with the HTTP directives). In such a context, a given API could have a number of endpoints and their specific pricing plans would specify the expected operating conditions for each endpoint. For the sake of clarity, from now on, in this paper we will assume an important simplification: each API only has a single endpoint and a single operational directive.

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

Pricing plans are usually paid through subscriptions, which tend to be billed monthly or yearly. The more expensive plans have less restrictive limitations. Some plans may also include an *overage* cost, that is, they allow clients to exceed their limitations for an additional fee (e.g. $0.01 per request beyond the imposed limitation in the example of Fig. 1).

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

2.2 Topology of a LAMA

A LAMA is an MSA with at least one external API that is regulated by a pricing plan, which includes, among other things, capacity limits and usage price.

As shown in Fig. 1, the topology of a LAMA can be represented as a DAG (directed acyclic diagram), where each dark node corresponds to an internal service in the LAMA and a light node corresponds to an external API. Each directed edge between nodes represents a consumption from one node to another: e.g. *microservice S1 consumes microservices S2 and S3, microservice S2 consumes external API E1, and microservice S3 consumes external APIs E1 and E2.* Each edge is labeled with the number of requests that are derived from the invocation of the consumer service: e.g. *each time the microservice S1 is invoked, the microservice S2 is consumed 3 times and microservice S3 is consumed 2 times.*

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

2.3 Capacity of an MSA

In general, *capacity* of an MSA refers to the maximum workload that it can handle, although there is no widely accepted definition for the term.

In our context, we consider the capacity of an entire MSA as the capacity of a given *entrypoint*. An entrypoint is the service of the MSA that is invoked first when a customer uses it, and then sends the appropriate requests to other services, which, in turn, may send further requests to more services.

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

3 Related Work

We are not aware of any existing proposal which analyses the capacity of an MSA with external APIs regulated by pricing plans. The most similar proposal, which has also been a major inspiration for us, is ELeCTRA by Gamez–Diaz et al. [10]. Based on the limitations of an external API (specified in its pricing) and the topology of an MSA with a single entrypoint, ELeCTRA computes the maximum values of the limitations that the entrypoint of the MSA will be able to offer to its users. Assuming that the topology of the MSA does not vary, these maximum values are determined solely by the values of the external API limitations, i.e., they are induced by them. This analysis of the limitations induced in an MSA is performed by ELeCTRA by interpreting the problem as a CSOP and using MiniZinc [17] as a solver.

Unfortunately, ELeCTRA's capabilities are insufficient to automatically analyse the capacity of a LAMA. Its main limitation is to consider that a pricing consists of a single constraint, or a single quota or a single limit. Thus, it is not possible to model prices, overage cost, or specify several limits (quotas and rates) in the same pricing. Consequently, none of the questions (Q1-Q6) raised in Section 1 could be solved with ELeCTRA.

Capacity analysis of LAMAs is also closely related to the *QoS-aware composition* problem. It tackles the selection of the best *providers* for different *tasks* in an architecture, based on QoS attributes offered by these providers that need to be optimised depending on user needs. This problem can be solved using search based techniques, using either Integer Linear Programming [23] or non-deterministic approaches [20, 3]. Drawing inspiration from these proposals, we could adapt their approaches to our problem, interpreting the LAMA as a composite service where the QoS attributes of each provider correspond to the attributes of a pricing plan. Nevertheless, this interpretation presents some limitations that does not allow a complete capacity analysis of a LAMA, in particular:

– No proposal considers capacity limitations of external providers, instead focusing on other attributes such as availability or response time.
– Each attribute needs an aggregation function that is used to select the best provider for each task. No aggregation function can model the exact semantics of rates and quotas, especially considering that they are defined over different time windows.
– No proposal defines analysis operations about capacity, time windows and cost.
– In previous work, a task could only be associated with a single provider. In our approach, there may be a need to use multiple API keys from the same provider to perform the same task.
– In real-world systems, a single provider can define multiple pricing plans for the same task. These plans differ in their cost, quotas, rates and other attributes. This was not the case in the previous approach, where a provider only had one plan for a task. Additionally, a LAMA might send multiple requests to the same provider.
– There might be cases where there is no solution to the problem, e.g. the LAMA is not able to serve enough requests to meet user requirements given the subscribed pricing plans. In previous approaches, this aspect was not taken into account.

4 Capacity Analysis of LAMAs

The capacity of a LAMA refers to the maximum workload that it can handle over a given period of time and at a maximum cost, without exceeding any of the external limitations derived from subscribed pricing plans. This definition is in line with the *capacity and performance management practice* in ITILv4 [15].

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

– *Metrics.* This dimension addresses the metrics (bounded to a scale) that have an impact on the capacity or are constrained by external APIs. In this paper we focus on a single metric, *number of requests*, that is the most widely used metric in the industry [11] and is constrained and limited in most commercial API pricing plans. It is important to note that the metric should always be bounded to a particular scale. In the case of *number of requests*, we could have different time scales such as Requests Per Second (RPS) or Requests Per Hour (RPH).
– *Temporality.* This dimension represents the temporal boundaries for the capacity to be analyzed. In this context, the same LAMA could have different capacities depending on the time period when it is calculated. These boundaries are typically linked with the desired operating conditions or, in case of a SaaS, the defined pricing plans. As an example, in Fig. 1, since both LAMAs pricing plans and external APIs plans are stated in terms of months, the appropriate temporal boundaries for the capacity analysis should only address a monthly perspective. However, in a more realistic setting there could be scenarios where different external APIs have different plan periods and consequently, the capacity analysis should combine multiple temporal perspectives involved.
– *Cost.* This dimension takes into account the derived costs from the infrastructure operation and the cost derived from the contracted plans with the different external APIs. In the example of Fig. 1, multiple options are possible, depending on the number of plans contracted; we assume that it is possible to contract multiple times a particular plan as this is the norm in the real API market.

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

When dealing with real-world architectures, the number of internal services and external APIs is considerably high, and thus there is a great number of plans and possible combinations. Additionally, when defining the pricing plans to be offered to the LAMA customers, it is fundamental to know the limitations derived from the usage by the external APIs together with its associated cost. In fact, these costs will be part of the operational costs of the LAMA, and are essential when analysing the OpEx (Operational Expenditures) [1] for the desired operating conditions in general, and to have profitable pricing plans in the case of a SaaS LAMA.

5 Automated Capacity Analysis

Automated LAMA capacity analysis deals with extracting information from the model of a LAMA using automated mechanisms. Analysing LAMA models is an error-prone and tedious task, and it is infeasible to do it manually with large-scale and complex LAMA models. In this paper we propose a similar approach to that followed in other fields, i.e., to support the analysis process from a catalogue of analysis operations (analysis of feature models [2, 5], service level agreements [16, 19, 18] and Business Process [21]).

In this sense, all the analysis operations we have faced so far can be interpreted in terms of optimal search problems. Therefore, they can be solved through Search Based Software Engineering (SBSE) techniques, similarly to other cloud engineering problems [13]. We tackle this problem as a Constraint Satisfaction and Optimisation Problem (CSOP), where, *grosso modo*, the search space corresponds to the set of tuples $(Requests, Time, Cost)$ that conform valid operating conditions of the LAMA. The objective function is defined on the variable that needs to be optimised in each case: requests, time or cost.

5.1 Formal Description of LAMAs

The primary objective of formalising a LAMA is to establish a sound basis for the automated support. Following the formalisation principles defined by Hofstede et al. [14], we follow a transformational style by translating the LAMA specification to a target domain suitable for the automated analysis (*Primary Goal Principle*). Specifically, we propose translating the specification to a CSOP that can be then analysed using state-of-the-art constraint programming tools.

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

Due to lack of space, we summarised the most relevant aspects of mapping a LAMA into a CSOP in our supplementary material [9]. The following paragraphs explain how the different elements and relationships of a LAMA are translated into a CSOP, mentioning the different variables and parameters that are needed to model the problem:

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

req_{E_i} that denote the request served by internal services and external APIs respectively must be greater than or equal to 0.

- **Requests served by internal services**. Each internal service in the LAMA S_i must serve all requests sent to it by every other service S_j, denoted as $req_{S_j S_i}$. Thus, for each internal service there is a constraint $req_{S_i} = \sum_{j=1}^{n} req_{S_j S_i} \cdot req_{S_j}$.
- **Requests served by external APIs**. Each external API E_i must serve all requests sent to it by the internal services S_j, denoted as $req_{S_j E_i}$. External APIs do not send requests between them. Thus, for each external API there is a constraint $req_{E_i} = \sum_{j=1}^{n} req_{S_j E_i} \cdot req_{S_j}$. Additionally, the total number of served requests is the sum of the requests sent to each plan below its limitations, $limReq_{ij}$, and the requests sent over the limitations, $ovgReq_{ij}$. This differentiation in two variables helps us obtain the number of overage requests more easily. Thus, for each plan P_{ij} of external API E_i there is a constraint $req_{E_i} = \sum_{j=1}^{n} limReq_{ij} + ovgReq_{ij}$. Furthermore, no requests can be sent using a plan with no keys, so for each plan there is a constraint $limReq_{ij} > 0 \rightarrow keys_{ij} > 0$. Also, no overage requests can be sent if there are no requests below limitations, so for each plan there is another constraint $ovgReq_{ij} > 0 \rightarrow limReq_{ij} > 0$.
- **Quota of each pricing plan**. The number of requests served by each external API E_i must not exceed any quota q_{ij}, defined over a time unit qu_{ij}. Multiple keys $keys_{ij}$ for each plan may be obtained. For each external API E_i and each of its respective plans P_{ij}, there is a constraint $limReq_{ij} <= keys_{ij} \cdot q_{ij} \cdot \lceil time/qu_{ij} \rceil$.
- **Rate of each pricing plan**. The number of requests served by each external API E_i must not exceed any rate r_{ij}, defined over a time unit ru_{ij}. Note that rates need to account for the time unit of the quota, as the rate is reset at the beginning of each unit. Therefore, for each external API E_i and each of its respective plans P_{ij}, there is a constraint $limReq_{ij} - qu_{ij} \cdot \lfloor time/qu_{ij} \rfloor <= keys_{ij} \cdot r_{ij} \cdot \lceil time \mod qu_{ij}/ru_{ij} \rceil$. If a plan has no no quota, the constraint is simplified to $limReq_i <= keys_{ij} \cdot r_{ij} \cdot \lceil time/ru_{ij} \rceil$.
- **OpEx of each external API**. The cost of each external API E_i is the sum of the subscriptions to each plan P_{ij} plus overage costs. For each external API E_i, there is a constraint $OpEx_i = \sum_{j=1}^{n} keys_{ij} \cdot cost_{ij} + ovg_{ij} \cdot ovgReq_{ij}$.
- **Total OpEx**. The total cost of the LAMA is the sum of the cost of each external API. There is a constraint $OpEx = \sum_{i=1}^{n} OpEx_i$.

5.2 Analysis Operations

We propose a catalogue of three analysis operations that leverage the formal description of LAMAs to automatically extract helpful information. Analogous analysis operations have been defined in the context of the automated analysis of feature models [2], service level agreements [16, 19, 18] and in the area of MSAs [10] (we may remark that it is not our intention to propose an exhaustive set of analysis operations as that would exceed the scope of this paper). For the description of the operations as CSOPs, we will refer to the input specification of a LAMA `L` and a variable `v`. Additionally, we will use the following auxiliary operations:

- `map(L)`. This operation translates a LAMA specification `L` to a CSP following the mapping described in Section 5.1 and more detailed in [9].
- `minimize(CSP, v)`. This standard CSOP-based operation returns a solution for the input `CSP` (if any) with the minimum value of variable `v`.
- `maximize(CSP, v)`. Same that prior operation but with the maximum value of variable `v`.

In what follows, we present three basic analysis operations, and, for the first operations identified in Section 1, we provide an explanation of how it is mapped to a CSOP from the corresponding basic operation.

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

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

With this operation we can answer question Q2 (*Assuming we have a Basic plan and a Gold plan already contracted what is the maximal RPM I can guarantee to all my 20 customers?*) in Section 1 resulting in 5.6 RPS to each customer:

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

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

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

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

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

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

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

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

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

6 Validation

In order to verify that our proposal can be exploited in a useful way, we have developed a tooling support that partially supports it. Specifically, we have developed a RESTful API that provides our 3 basic capacity analysis operations, and a deepnote notebook that shows the Python implementation of the 6 analysis questions posed in Section 1. This tooling support can and should be understood as a minimal but solid proof of concept.

Smart LAMA [8], [1] is a public RESTful API that supports, among others, various endpoints which transform a LAMA into a CSOP using the MiniZinc modelling language. In particular, for the scope of this paper, we will focus on the three main endpoints that provide solutions to the three analysis operations described in the previous section.

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

- `/maxRequests`. This operation returns the maximum number of requests that the LAMA is able to serve per unit of time without exceeding any external limitation. It supports some query parameters: `OpEx` is used to specify a maximum total budget that can be spent to subscribe to the different pricing plans; `time` can be used to specify the unit of time in which the operation is calculated (indicated as the number of seconds, e.g. a minute is represented as 60); and `K-<API>-<Plan>` is used to indicate a specific number of subscriptions to a plan (e.g. `K-E1-Basic=1` means that there is 1 subscription to plan *Basic* of API E1). Note that there will be no restriction to the total cost of the LAMA if no OpEx or specific subscriptions are indicated.
- `/minCost`. This operation returns the minimum cost to serve a certain number of requests, which is specified using the `reqL` query parameter, over a certain time window, specified through the `time` query parameter. Both parameters are required. Obtaining the minimum cost implies obtaining the optimum combination of subscriptions to the pricing plans, which is also included within the MiniZinc output and may be extracted if needed.
- `/minTime`. This operation returns the minimum time (in number of seconds) in which the LAMA can serve a certain number of requests, which is specified using the `reqL` query parameter. This endpoint also supports the `OpEx` and `K-<API>-<Plan>` parameters, which work exactly as described above.

[1] Available at https://smart-lama-api-beta.herokuapp.com/api/v2.

The formal description of a LAMA used in the notebook is a JSON-based language which we named *LAMA-DL*. Due to lack of space, we will not explain how to transform a LAMA into LAMA-DL. Nonetheless, we believe that the example included in the notebook using the LAMA in Fig. 1 is self-explanatory and contains all supported elements.

Note that, by default, the API is set up to assume that no overage requests should be used. To enable the use of overage requests, all operations support the `useOvg` query parameter, which should be explicitly set to `true`. Only pricing plans with overage costs may have overage requests.

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

To validate Smart LAMA, we developed an online Deepnote notebook [7]. It contains wrappers that take a formal description of a LAMA as input, send the appropriate request to the API using the corresponding query parameters, and return the solution provided by MiniZinc, including the final values of all internal variables after solving the CSOP. The notebook includes a complete example based on the LAMA in Fig. 1 and shows how to use the API to solve each of the 6 different analysis questions introduced in Section 1. Note that the notebook has *Execute* access, meaning that its cells can be executed but not edited. However, it can be duplicated and then edited.

Some examples of API calls included in the notebook [7] are the following:

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

7 Conclusions and future work

In this paper, we presented the problem of automating the capacity analysis of microservices architectures in situations where the MSA consumes external APIs that define pricing plans. We introduced the concept of *limitation-aware microservices architecture* (LAMA) and explored the different dimensions involved in the capacity analysis of a LAMA. We listed 6 analysis questions, which we determined to be derived from three basic analysis operations that allow the definition of any number of other operations. We presented a public API that transforms a LAMA into a proof-of-concept implementation of these operations using MiniZinc, and evaluated it in a synthetic LAMA example. We are confident that our proposal will prove useful to DevOps teams who need to deal with issues related to capacity analysis of LAMAs.

We are aware that our tooling support is partial and therefore incomplete, but it shows the real possibility of answering questions in less time than if it were done manually. In this sense, we are working on using notation to describe both the topology and the pricing closer to some of the available technology.

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

References

1. Andreo, S., Calà, A., Bosch, J.: OpEx Driven Software Architecture A Case Study. In: Proceedings of the 15th European Conference on Software Architecture (ECSA) (Companion) (2021)
2. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: A literature review. Information systems **35**(6), 615–636 (2010)
3. Canfora, G., Di Penta, M., Esposito, R., Villani, M.L.: A lightweight approach for QoS-aware service composition. In: Proc. 2nd International Conference on Service Oriented Computing (ICSOC'04)-short papers. Citeseer (2004)
4. How to build an API economy for your enterprise - Capgemini. https://www.capgemini.com/2020/12/how-to-build-an-api-economy-for-your-enterprise/ ([nd]), accessed on 2022-06-13
5. Durán, A., Benavides, D., Segura, S., Trinidad, P., Ruiz-Cortés, A.: Flame: a formal framework for the automated analysis of software product lines validated by automated specification testing. Software & Systems Modeling **16**(4), 1049–1082 (2017)
6. M. Fowler - Microservices. https://martinfowler.com/articles/microservices.html ([nd]), accessed on 2022-06-13
7. Fresno-Aranda, R., Fernández, P., Durán, A., Ruiz-Cortés, A.: Notebook for automated analysis of a basic LAMA, https://bit.ly/smart-lama-api
8. Fresno-Aranda, R., Fernández, P., Ruiz-Cortés, A.: Smart LAMA API: Automated Capacity Analysis of Limitation-Aware Microservices Architectures. In: Proceedings of the XVII Jornadas de Ingeniería de Ciencia e Ingeniería de Servicios (In press). SISTEDES (2022)

9. Fresno-Aranda, R., Fernández, P., Durán, A., Ruiz-Cortés, A.: Semi-Automated Capacity Analysis of Limitation- Aware Microservices Architectures (Supplementary Material) (Aug 2022), https://doi.org/10.5281/zenodo.7025641
10. Gamez-Diaz, A., Fernandez, P., Pautasso, C., Ivanchikj, A., Ruiz-Cortes, A.: ELeCTRA: Induced Usage Limitations Calculation in RESTful APIs. In: 16th International Conference on Service-Oriented Computing (ICSOC). pp. 435–438. Springer (2018)
11. Gamez-Diaz, A., Fernandez, P., Ruiz-Cortes, A.: An analysis of RESTful APIs offerings in the industry. In: 15th International Conference on Service-Oriented Computing (ICSOC). pp. 589–604. Springer (2017)
12. Gamez-Diaz, A., Fernandez, P., Ruiz-Cortés, A., Molina, P.J., Kolekar, N., Bhogill, P., Mohaan, M., Méndez, F.: The role of limitations and SLAs in the API industry. In: Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). pp. 1006–1014 (2019)
13. Harman, M., Lakhotia, K., Singer, J., White, D.R., Yoo, S.: Cloud engineering is search based software engineering too. Journal of Systems and Software **86**(9), 2225–2241 (2013)
14. ter Hofstede, A.H., Proper, H.A.: How to formalize it?: Formalization principles for information system development methods. Information and Software Technology **40**(10), 519–540 (1998)
15. ITIL Capacity Management. https://www.smartsheet.com/content/itil-capacity-management ([nd]), accessed on 2022-06-13
16. Martín-Díaz, O., Ruiz-Cortés, A., Durán, A., Müller, C.: An approach to temporal-aware procurement of web services. In: 3rd International Conference on Service-Oriented Computing (ICSOC). pp. 170–184. Springer (2005)
17. MiniZinc constraint modeling language ([nd]), https://www.minizinc.org/, accessed on 2022-06-13
18. Müller, C., Gutierrez, A.M., Fernandez, P., Martín-Díaz, O., Resinas, M., Ruiz-Cortés, A.: Automated validation of compensable SLAs. IEEE Transactions on Services Computing **14**(5), 1306–1319 (2018)
19. Müller, C., Resinas, M., Ruiz-Cortés, A.: Automated analysis of conflicts in WS-Agreement. IEEE Transactions on Services Computing **7**(4), 530–544 (2013)
20. Parejo, J.A., Segura, S., Fernandez, P., Ruiz-Cortés, A.: QoS-Aware web services composition using GRASP with Path Relinking. Expert Systems with Applications **41**(9), 4211–4223 (2014)
21. del Río-Ortega, A., Resinas, M., Cabanillas, C., Ruiz-Cortés, A.: On the definition and design-time analysis of process performance indicators. Information Systems **38**(4), 470–490 (2013)
22. How to Make the API Economy a Reality - WSO2. https://wso2.com/choreo/resources/how-to-make-the-api-economy-a-reality/ ([nd]), accessed on 2022-06-13
23. Zeng, L., Benatallah, B., Ngu, A.H., Dumas, M., Kalagnanam, J., Chang, H.: QoS-aware middleware for web services composition. IEEE Transactions on software engineering **30**(5), 311–327 (2004)