Call Limit-Based Composite Service Selection

Karim Benouaret, Juba Agoun, Idir Benouaret, François Charoy

Call Limit-Based Composite Service Selection

Karim Benouaret†, Juba Agoun†, Idir Benouaret‡ and François Charoy§
†CNRS, LIRIS, Université Claude Bernard Lyon 1
Villeurbanne, France
‡CNRS, Univ. Grenoble Alpes
Grenoble, France
§CNRS, Inria, LORIA, Université de Lorraine
Villers-lès-Nancy, France
Email: †{karim.benouaret, juba.agoun}@liris.cnrs.fr, ‡idir.benouaret@univ-grenoblealpes.fr, §francois.charoy@loria.fr

***Abstract*—APIs allow companies to export, via the Internet, their skills and know-how, or even to open up new markets and new media for sale. But to fully exploit the advantages of these services, customers, mainly developers, must be equipped with tools giving the possibility of being able to assemble different services together. Fortunately, the notion of service composition is quite advanced, and different tools exist to compose services. However, as APIs with similar functionality are expected to be provided by competing providers, the key challenge is to find the most relevant compositions. This issue has been addressed in the context of QoS-based composite service selection. The downside, in practice, customers choose services based on the number of call limits. In this paper, we propose an approach to select the most relevant compositions based on the notion of call limit. Specifically, we show how the call limits of the individual services can be aggregated to obtain the call limits of a given composition. Then, we introduce the notion of minimal budget skyline, which comprises the most interesting compositions that fit within the customer's budget. In addition, we develop two algorithms, based on effective pruning strategies, to efficiently compute the minimal budget skyline. Finally, we present a thorough experimental evaluation of our approach.**

***Index Terms*—service composition, API, call limit, skyline**

I. Introduction

As the software world has evolved toward the idea of modularity, services – nowadays usually published in the form of Application Programming Interfaces (APIs) – are found in every modern application.

APIs represent a powerful opportunity for both customers (software developers) and providers (companies). Thanks to APIs, developers do not have to start from scratch when building their applications. They rather compose different existing services, thereby saving valuable time. On the other hand, by making available their APIs, organizations can create a new revenue stream by monetizing them [1].

With the proliferation of services and service providers, there can be multiple service providers competing to accomplish the same task. Therefore, the 2000s vision of selecting appropriate services that achieve good composition based on non-functional properties (NFPs) takes shape. Downside, what is happening now is different from what was imagined. The NFPs considered in current approaches; see e.g., [2]–[5], are Quality of Service (QoS) attributes (e.g., response time, throughput, etc.), where service providers undertake to respect the advertised values. However, in practice, it is up to the user to respect some rules and constraints. The use of a service requires accepting a service contract which contains terms that must be respected. We will then not talk about QoS, but about Term of Service (ToS) or usage policies. This includes, the form of registration, payment plans as well as call limits [6]. There are two different types of call limits: rate limits and quotas [7]–[9]. Rate limits describe limitations of use for a short period of time (e.g., 3 calls per second); their role is to protect servers and infrastructure against overload. Quotas describe usage limitations for a long time period (e.g., 100,000 requests per month) and are used to monetize APIs.

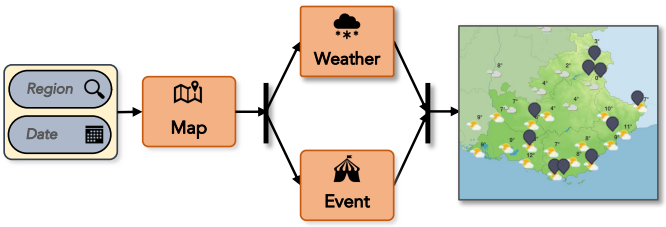

Fig. 1. Service Composition Example

In this work, we introduce the problem of call limit-based composite service selection whose goal is to find the most relevant service compositions – which fall within the developer's budget – in terms of call limits. The following example presents a typical scenario that clearly shows the different challenges involved in the call limit-based service composition problem.

A. Motivating Example

Consider a scenario where a software designer *Bob*, working for the French national tourist office, needs to create an application promoting social events. Specifically, users submit their requests to the system, specifying a region and a date. The system then returns a list of events displayed on a map. To help participants choose the appropriate events, the weather is also displayed on the map. To build this application, *Bob* chooses to compose existing APIs as shown in Figure 1.

First, a map service is called to get the desired portion of the map. Then, at the same time, a weather service and an

event service are invoked to respectively retrieve the weather and the events available in the desired region for a specific date. Finally, the results are displayed in the map part.

For these outsourced tasks, multiple services may be available providing the required functionality but with different call limits and prices (see Table I[1]). For example, map service s_{11} requires not to exceed 1 execution per second and 80K executions per month, and costs 8$, while map service s_{13} has an unlimited number of calls and costs 20$.

Now, assume that *Bob* has a budget of 30$. To select the compositions that fit within his budget, *Bob* must calculate the price of the possible combinations of services. In addition, *Bob* would like a composition with good request limits. Thus, he has a second task, which is to determine the call limits of the previously selected compositions from the call limits of the individual services. If the number of available services is large, performing an exhaustive search to find a satisfactory composition would be very painstaking. It is therefore important to set up an effective service composition framework that would identify and retrieve the most relevant service compositions in terms of both price and call limits.

B. Challenges and Contributions

In this paper, we address three major challenges about the call limit-based composite service selection.

Challenge 1: call limit aggregation. Given a set of services that participate in a composition, where each service has a limited number of authorized calls, our goal is to know the limitation of calls of the composition. In other words, *what are the calls limits of the composite service resulting from the orchestration of these different services?*

Contributions. At first glance, the problem seems simple, because the first idea that comes to mind is to convert the different call limits of the individual services into the same unit of time. Then simply aggregate the results according to the composition plan. However, this solution is not possible. For example, by converting the call limit of service s_{23} to month, we obtain 20K/d$\times$30 = 600K/M. But by carefully analyzing the two call limits, we observe that they are not quite equivalent. Indeed, in the second case, it is possible to invoke the service more than 20K times per day.

Rather than reducing the call limits to the same unit of time, we consider all time periods (s, m, d and M in our example). More specifically, we first fill the missing call limits of the individual services. For example, we calculate how many times can the service s_{11} be invoked per minute and per day. Then, we aggregate the obtained call limits to determine those of the compositions according to the composition plan. In our work, we study the aggregation of call limits in regards to four commonly used patterns, namely, “Sequence”, “AND”, “XOR” and “OR”.

[1]We consider four time periods: second, minute, day and month, which we denote by s, m, d and M, respectively

Challenge 2: selecting the most relevant compositions. To help the developer easily choose a composition among those that fit within his budget, it is crucial to select the most relevant ones. We are therefore interested in the following question: *what does it mean to be a relevant composition?*

Contributions. Returning to our example, observe that invoking a composition requires the invocation of all services (i.e., map, weather, event services). Hence, the call limits of the compositions are calculated as the minimum of those of the individual services. If we look at a particular unit of time, it is straightforward to determine the relevant set of compositions. For instance, consider the period *“month”*. It is clear that the compositions $\{s_{11}, s_{21}, s_{31}\}$ and $\{s_{11}, s_{22}, s_{31}\}$ are ideal since they are the cheapest (25$) and offer a maximum call limit for one month (70K). However, these compositions can only be invoked once per second. If several users try to use the future social events application simultaneously, only one user will succeed in having a result. It is thus important to maximize the call limits for all time units.

Computing the skyline [10] of the compositions that fit within the developer’s budget guarantees to include interesting compositions. This set comprises the non-dominated compositions. Informally, a composition dominates another, if the former is better than the latter considering both price and call limits. We refer to this set as the *budget skyline*. Different QoS-based service composition approaches leverage the notion of skyline to select the best compositions; see e.g., [11]–[13]. In this paper, we make the observation that after computing the call limits of the candidate compositions, there may be multiple equivalent compositions, i.e., with the same cost and call limits, and go one step forward by proposing the notion of *minimal budget skyline*, which consists of the non-dominated compositions, that fit within the developer’s budget, without redundancies. Clearly, this is a more manageable set than the budget skyline as it reduces the developer effort required to examine the compositions in order to select a desired one.

Challenge 3: computing the minimal budget skyline. A straightforward approach for computing the minimal budget skyline must go through the generation of all possible compositions. Clearly, this approach results in a huge computational cost. Thus, *can we devise efficient techniques to compute the minimal budget skyline efficiently?*

Contributions. We propose two pruning rules to tackle this problem. The first eliminates services that will result in compositions that do not fit into the developer’s budget. The second eliminates the dominated services from each set of functionally-equivalent services. The main goal of these pruning strategies is to reduce the number of compositions to generate. We incorporate these pruning rules into two different algorithms and study experimentally their efficiently.

The remainder of this paper is organized as follows. Section II formalizes and introduces the problem of call limit-based composite service selection. Section III describes our pruning techniques and algorithms. Section V reviews related

TABLE I
EXAMPLE OF SERVICES

Map			Weather			Event		
Service	Call limits	Price ($)	Service	Call limits	Price ($)	Service	Call limits	Price ($)
s_{11}	1/s, 80K/M	8	s_{21}	3/s, 70K/M	7	s_{31}	50/m, 70K/M	10
s_{12}	80/m, 60K/M	10	s_{22}	50/m, 70K/M	7	s_{32}	25/m, 80K/M	15
s_{13}	$+\infty$	20	s_{23}	20K/d	15	s_{33}	$+\infty$	30
s_{14}	1/s, 50k/M	8	s_{24}	1/s, 50K/M	10	s_{34}	50/m, 70K/M	12
s_{15}	25/m, 60K/M	12	s_{25}	1/s, 70K/M	12	s_{35}	25/m, 75K/M	15

work. Our experimental evaluation is reported in Section IV. Finally, Section VI concludes the paper.

II. PROBLEM DEFINITION

In this section, we formally describe the notion of call limits-based service composition and define the problem of minimal budget skyline.

A. Our Model

We model an abstract service composition as a tuple $\mathcal{A} = \langle \mathcal{S}, \mathcal{P} \rangle$. Where $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_m\}$ is a set of abstract services (also called service classes), and $\mathcal{P}$ is an execution plan that defines the invocation structure of the services in $\mathcal{S}$, which we denote by $\mathcal{P}(\mathcal{S})$. In this work we consider four atomic invocation operators.

- Sequence: $\mathcal{S}_i \rightarrow \mathcal{S}_j$ represents a composite service that performs the service $\mathcal{S}_i$ followed by the service $\mathcal{S}_j$;
- AND: $\mathcal{S}_i \cdot \mathcal{S}_j$ represents a composite service that performs both services $\mathcal{S}_i$ and $\mathcal{S}_j$;
- XOR: $\mathcal{S}_i \oplus \mathcal{S}_j$ represents a composite service that performs either service $\mathcal{S}_i$ or service $\mathcal{S}_j$;
- OR: $\mathcal{S}_i + \mathcal{S}_j$ represents a composite service that performs one or both of the services $\mathcal{S}_i$ and $\mathcal{S}_j$.

As an example, the execution plan of the abstract services involved in the social events application of our running example is Map $\rightarrow$ (Weather $\cdot$ Event).

Each abstract service $\mathcal{S}_i$ consists of all concrete services that deliver the same functionality; let $\mathcal{S}_i = \{s_{i1}, s_{i2}, \ldots, s_{in_i}\}$. Further, consider a set of time units $\mathcal{T} = \{t_1, t_2, \ldots, t_d\}$. We use $s_{ij}.t_\alpha$ to denote the call limit of service s_{ij} for time unit t_α; if the provider does not impose a call limit for a time unit t_α then we set $s_{ij}.t_\alpha$ to `null`. We define the call limits of a service s_{ij} as a vector $\ell(s_{ij}) = (s_{ij}.t_1, s_{ij}.t_2, \ldots, s_{ij}.t_d)$. For instance, the call limits of service s_{11} of our example is $\ell(s_{11})$ = (1/s, `null`, `null`, 80K/M). In addition, each service s_{ij} has a price, which we denote by $s_{ij}.p$ (e.g., $s_{11}.p$ = 8$).

A concrete service composition, which can be defined as an instantiation of the abstract service composition, is a tuple $c = \langle s_{1j_1}, s_{2j_2}, \ldots, s_{mj_m} \rangle$, where $s_{ij_i} \in \mathcal{S}_i$, $\forall i \in [1, m]$. We denote the set of all possible concrete compositions by $\mathcal{C}$. The price of a composition c, denoted by $c.p$ is the sum of prices of its individual services. Similar to services, we denote the call limit of a composition c for unit time t_α by $c.t_\alpha$, and its call limit vector by $\ell(c) = (c.t_1, c.t_2, \ldots, c.t_d)$. Next, we describe how we calculate the call limits of a composition.

B. Call Limit Aggregation

We propose a progressive aggregation strategy. In other words, we solve the problem for each invocation structure, and not by dealing with the global composition. This approach makes it possible to have a simpler view of the composition structure, but above all to simplify the aggregation of call limits by studying the problem on a case-by-case basis.

Consider an abstract composition $\mathcal{A} = \langle \mathcal{S}, \mathcal{P} \rangle$, where $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_m\}$. Assume a concrete composition $c = \langle s_{1j_1}, s_{2j_2}, \ldots, s_{mj_m} \rangle$, where $s_{ij_i} \in \mathcal{S}_i, \forall i \in [1, m]$. We define the following aggregation rules.

Aggregation Rule 1 (Sequence). *If $\mathcal{P}(\mathcal{S}) = \mathcal{S}_1 \rightarrow \mathcal{S}_2 \rightarrow \ldots \rightarrow \mathcal{S}_m$, i.e., the services are in sequence, then* $\ell(c) = (\min_{i=1}^{m}(s_{ij_i}.t_1), \min_{i=1}^{m}(s_{ij_i}.t_2), \ldots, \min_{i=1}^{m}(s_{ij_i}.t_d))$.

Aggregation Rule 2 (AND). *If $\mathcal{P}(\mathcal{S}) = \mathcal{S}_1 \cdot \mathcal{S}_2 \cdot \ldots \cdot \mathcal{S}_m$, i.e., the services are connected with the AND operator, then* $\ell(c) = (\min_{i=1}^{m}(s_{ij_i}.t_1), \min_{i=1}^{m}(s_{ij_i}.t_2), \ldots, \min_{i=1}^{m}(s_{ij_i}.t_d))$.

Aggregation Rule 3 (XOR). *If $\mathcal{P}(\mathcal{S}) = \mathcal{S}_1 \oplus \mathcal{S}_2 \oplus \ldots \oplus \mathcal{S}_m$, i.e., the services are connected with the XOR operator, then* $\ell(c) = (\sum_{i=1}^{m}(s_{ij_i}.t_1), \sum_{i=1}^{m}(s_{ij_i}.t_2), \ldots, \sum_{i=1}^{m}(s_{ij_i}.t_d))$.

Aggregation Rule 4 (OR). *If $\mathcal{P}(\mathcal{S}) = \mathcal{S}_1 + \mathcal{S}_2 + \cdots + \mathcal{S}_m$, i.e., the services are connected with the OR operator, then* $\ell(c) = (\max_{i=1}^{m}(s_{ij_i}.t_1), \max_{i=1}^{m}(s_{ij_i}.t_2), \ldots, \max_{i=1}^{m}(s_{ij_i}.t_d))$.

The output of a sequential or parallel (AND) invocation of m services is obtained by invoking all these services. Thus, the call limit of the composite service for each time unit is the minimum of the call limits of the involved services. For an exclusive (XOR) invocation of m services, the result is obtained by invoking one and only one of these m services. The invocation limit of the composite service for each period is then the sum of that of the concerned services. If the m services are connected with the OR operator, the output of the composition is obtained by invoking one or more of these services. Hence, the call limit of the composite service for each time unit is the maximum of the call limits of the involved services, since we can invoke several services simultaneously.

Consider the concrete composition $c = \langle s_{11}, s_{21}, s_{31} \rangle$ in our example. According to the composition plan Map $\rightarrow$ (Weather $\cdot$ Event), to calculate the call limit vector of c, our strategy applies first Aggregation Rule 2 on services s_{21} and s_{31}. Then, the obtained result is aggregated with the call limit vector of service s_{11} using Aggregation Rule 1.

TABLE II
EXAMPLE OF SERVICES WITH FILLED CALL LIMITS

Map

Service	Call limits	Price ($)
s_{11}	1/s, 60/m, 80K/d, 80K/M	8
s_{12}	80/s, 80/m, 60K/d, 60K/M	10
s_{13}	$+\infty$/s, $+\infty$/m, $+\infty$/d, $+\infty$/M	20
s_{14}	1/s, 60/m, 50/d, 50K/M	8
s_{15}	25/s, 25/m, 36K/d, 60K/M	12

Weather

Service	Call limits	Price ($)
s_{21}	3/s, 180/m, 70K/d, 70K/M	7
s_{22}	50/s, 50/m, 70K/d, 70K/M	7
s_{23}	20K/s, 20K/m, 20K/d, 600K/M	15
s_{24}	1/s, 60/s, 50K/d, 50K/M	10
s_{25}	1/s, 60/s, 70K/d, 70K/M	12

Event

Service	Call limits	Price ($)
s_{31}	50/s, 50/m, 70K/d, 70K/M	10
s_{32}	25/s, 25/m, 36K/d, 80K/M	15
s_{33}	$+\infty$/s, $+\infty$/m, $+\infty$/d, $+\infty$/M	30
s_{34}	50/s, 50/m, 70K/d, 70K/M	12
s_{35}	25/s, 25/m, 36K/d, 75K/M	15

TABLE III
EXAMPLE OF SERVICE COMPOSITIONS

Composition	Call limits	Price ($)
$c_1 = \langle s_{11}, s_{21}, s_{31} \rangle$	1/s, 50/m, 70K/d, 70K/M	25
$c_2 = \langle s_{11}, s_{21}, s_{32} \rangle$	1/s, 25/m, 36K/d, 70K/M	30
$c_3 = \langle s_{11}, s_{22}, s_{31} \rangle$	1/s, 50/m, 70K/d, 70k/M	25
$c_4 = \langle s_{11}, s_{22}, s_{32} \rangle$	1/s, 25/m, 36K/d, 70k/M	30
$c_5 = \langle s_{12}, s_{21}, s_{31} \rangle$	3/s, 50/m, 60K/d, 60K/M	27
$c_6 = \langle s_{12}, s_{21}, s_{32} \rangle$	3/s, 25/m, 36K/d, 60K/M	32
$c_7 = \langle s_{12}, s_{22}, s_{31} \rangle$	50/s, 50/m, 60K/d, 60k/M	27
$c_8 = \langle s_{12}, s_{22}, s_{32} \rangle$	25/s, 25/m, 36K/d, 60k/M	32
$\vdots$	$\vdots$	$\vdots$
$c_{125} = \langle s_{15}, s_{25}, s_{35} \rangle$	1/s, 25/m, 36K/d, 60k/M	39

However, to calculate the call limit vector of a concrete service composition, the missing call limits of the composed services must first be filled. For instance, the call limit of service s_{11} for both minute and day periods. It is worth noting that this cannot be done by a straightforward conversion. For example, service s_{11} has a minute limit of 1/s×60 = 60/m. But for unit time day, 60/m×60×24 = 86.4K/d exceeds the limitation authorized per month. Thus, the call limit of service s_{11} per day should be that of a month, i.e., 80K, since the customer can use the monthly quota in one day. Table II shows the call limits of the services of our running example after filling the missing values. Likewise, Table III depicts the call limits and the price of a portion[2] of the possible compositions after applying the aggregation rules.

C. Minimal Budget Skyline

For brevity, we focus our definitions on the compositions. However, they are valid on the services belonging to the same abstract service set.

[2]In total there are 125 compositions, we just show those that are necessary to understand the paper.

Definition 1 (Call Limit Preference)**.** *We say that a composition c is preferred over another composition c' with regard to call limit, denoted by $c \succeq_\ell c'$, iff c is as good as c' in all time units. Formally: $c \succeq_\ell c' \Leftrightarrow \forall\, t_\alpha \in \mathcal{T}, c.t_\alpha \geq c'.t_\alpha$.*

Definition 2 (Call Limit Strict Preference)**.** *We say that a composition c is strictly preferred over another composition c' with regard to call limit, denoted by $c \succ_\ell c'$, iff c is preferred over c' and additionally there exists a time unit for which c is better than c'. Formally: $c \succ_\ell c' \Leftrightarrow c \succeq_\ell c' \wedge \exists\, t_\beta \in \mathcal{T} : c.t_\beta > c'.t_\beta$.*

For example, composition c_1 is preferred over composition c_3, while it is strictly preferred over composition c_2.

Definition 3 (Dominance)**.** *We say that a composition c dominates another composition c', denoted by $c \succ c'$, iff c is as good as c' in both call limit and price, and c is strictly preferred over c' with regard to call limit or c is cheaper than c'. Formally: $c \succ c' \Leftrightarrow c \succeq_\ell c' \wedge c.p \leq c'.p \wedge (c \succ_\ell c' \vee c.p < c'.p)$.*

Definition 4 (Skyline)**.** *Given a set of compositions $\mathcal{C}$, the skyline of $\mathcal{C}$, denoted by Sky, comprises the set of compositions that are not dominated by any other composition. Formally: $Sky = \{c \in \mathcal{C} \mid \nexists\, c' \in \mathcal{C} : c' \succ c\}$.*

To illustrate these definitions, let us consider the Map services. Observe that service s_{11} dominates service s_{14} and service s_{12} dominates service s_{15}. However, the services s_{11}, s_{12}, s_{13} are not dominated. Thus, they form the skyline of the abstract service Map.

Definition 5 (Budget Skyline)**.** *Given a budget $\mathcal{B}$ and a set of compositions $\mathcal{C}$, the budget skyline of $\mathcal{C}$, denoted by $\mathcal{B}$-Sky, comprises skyline compositions whose price falls within the budget. Formally: $\mathcal{B}\text{-}Sky = \{c \in Sky \wedge c.p \leq \mathcal{B}\}$.*

Definition 6 (Equivalence)**.** *We say that a composition c is equivalent to another composition c', denoted by $c \equiv c'$, iff c and c' have the same call limit vector and the same price. Formally, $c \equiv c' \Leftrightarrow \ell(c) = \ell(c') \wedge c.p = c'.p$.*

Definition 7 (Minimal Budget Skyline)**.** *Given a budget $\mathcal{B}$ and a set of compositions $\mathcal{C}$, a set $\mathcal{B}$-Sky^* is called a non-redundant (or a minimal) budget skyline of $\mathcal{C}$ if the following statements hold.*

- $\nexists\, c, c' \in \mathcal{B}\text{-}Sky^* : c \equiv c'$;
- $\forall\, c \in \mathcal{B}\text{-}Sky, \exists\, c' \in \mathcal{B}\text{-}Sky^* : c \equiv c'$.

Now, let us go back to the compositions, and recall that the budget of *Bob* in our example is 30$. The non-dominated compositions that fit within this budget are c_1, c_3 and c_7. They thus form the budget skyline. Moreover, observe that the compositions c_1 and c_3 in budget skyline are equivalent. Hence, there may be two minimal budget skylines. One contains compositions c_1 and c_7 and the other comprises compositions c_3 and c_7. With the presence of a large number of compositions, by eliminating redundancies, the minimal budget skyline reduces effort required to examine the compositions in order to select a desired one.

We are now ready to state the problem of minimal budget skyline computation.

Problem Definition. Given a set of abstract services $\mathcal{S}$, connected with an execution plan $\mathcal{P}$, and a budget $\mathcal{B}$, compute a minimal budget skyline.

III. Minimal Budget Skyline Computation

In this section, we present two algorithms based on the following useful observations that increase efficiency.

Pruning Rule 1 (Price-Based Pruning)**.** *Let us denote by $\mathcal{S}_i.p^-$ the price of the cheapest service in service class $\mathcal{S}_i$. Assume a service $s_{ij} \in \mathcal{S}_i$. If $s_{ij}.p + \sum_{\mathcal{S}_k \neq \mathcal{S}_i} \mathcal{S}_k.p^- > \mathcal{B}$ then any composition containing service s_{ij} is out of budget.*

Proof. It is apparent since the price of a composition is the sum of the price of its component services. □

Pruning Rule 1 helps reduce the number of combinations to generate and examine by eliminating services that will result in compositions that are out of budget. In addition, it avoids computing the call limit vectors of these services. For example, regardless of the composition that contains the Map service s_{13}, it costs more than 30$, since 20+7+10 $>$ 30. Thus, service s_{13} can be discarded.

Pruning Rule 2 (Dominance-Based Pruning)**.** *Assume two services s_{ij} and s_{ik} belonging to the same service class $\mathcal{S}_i$ such that s_{ij} dominates s_{ik}. Consider two compositions $c = \langle s_{1j_1}, \ldots, s_{ij}, \ldots, s_{mj_m} \rangle$ and $c' = \langle s_{1j_1}, \ldots, s_{ik}, \ldots, s_{mj_m} \rangle$, which contain the same services with the exception s_{ij} and s_{ik}. Then, either c dominates c' or c and c' are equivalent.*

Proof. The fact that s_{ij} dominates s_{ik} means that s_{ij} is better than s_{ik} considering both call limit and price. Since the call limit aggregation rules and the additive function to compute the price of the compositions are monotone, c will necessarily be better than or at least equal to c'. □

Pruning Rule 2 helps further reduce the number of combinations to generate and examine. To exploit this pruning rule, we compose only the skyline services of each service class. For example, Map services s_{14} and s_{15} are not considered when generating the compositions since they are dominated.

A. Local Pruning Algorithm

Hereafter, we develop the *Local Pruning Algorithm* (LPA). LPA applies both pruning rules on the individual services, generates the possible compositions using the retained services, then computes a minimal budget skyline. This clearly reduces the number of generated compositions. The pseudocode of LPA is depicted in Algorithm 1.

For each service class $\mathcal{S}_i$ (loop in line 1), LPA eliminates services that will result in compositions that are out of budget based on Pruning Rule 1 (line 2). Then, the algorithm fills the missing call limit of the retained services (loop in line 3) and computes the skyline of $\mathcal{S}_i$ (line 5) to exploit Pruning Rule 2. After that, the possible compositions using the skyline services from each service class are generated, and only those that fall within the budget are retained (lines 6–7). Afterward, the call limit vector of each retained composition is computed according to the execution plan $\mathcal{P}$ using the call limit aggregation rules (loop in line 8). Finally, LPA eliminates redundant compositions at random, computes and returns the skyline of the non-redundant compositions set (lines 10–12).

Algorithm 1: Local Pruning Algorithm (LPA)

Input: abstract services $\mathcal{S}$, composition plan $\mathcal{P}$, budget $\mathcal{B}$
Output: minimal budget skyline $\mathcal{B}$-Sky^*

```
1  foreach S_i ∈ S do
2      S_i ← {s_ij ∈ S_i | s_ij.p + Σ_{S_k ≠ S_i} S_k.p⁻ ≤ B}
3      foreach s_ij ∈ S_i do
4          ℓ(s_ij) ← fill the missing call limits of s_ij
5      Sky_i ← compute the skyline of S_i
6  C ← compose Sky_1, Sky_2, ..., Sky_m
7  C ← {c ∈ C | c.p ≤ B}
8  foreach c ∈ C do
9      ℓ(c) ← compute the call limit vector of c w.r.t. P
10 C ← eliminate redundant compositions from C
11 B-Sky* ← compute skyline of C
12 return B-Sky*
```

Applying LPA to our example, services s_{13}, s_{23} and s_{33} are eliminated using Pruning Rule 1. The missing call limits of the remaining services are filled and Based on Pruning Rule 2, the algorithm retains only services s_{11} and s_{12} from the Map services, s_{21} and s_{22} from the Weather services, and s_{31} and s_{32} from the Event services. These services are then combined and first eight compositions (i.e., c_1 to c_8) in Table III are obtained. Compositions c_6 and c_8 are out of budget and are therefore eliminated. Then the call limit vector of the six remaining compositions is computed. Observe that, compositions c_1 and c_3 are equivalent. This is also the case of compositions c_2 and c_4. Thus, LPA eliminates at random redundancies and retains for example compositions c_1 and c_2. Thus, there are therefore four compositions, namely c_1, c_2, c_5 and c_7, that remain to be compared. The skyline of these compositions comprises compositions c_1 and c_7, which are returned as a minimal budget skyline.

B. Progressive Pruning Algorithm

In the following, we design our second algorithm termed *Progressive Pruning Algorithm* (PPA). PPA operates on the concept of *composition tree* to progressively compute a minimal budget skyline using both pruning rules.

A composition tree is a tree representation of the execution plan $\mathcal{P}$. There are two types of nodes: operator nodes and service nodes. An operator node is a node that contains one of the invocation symbols "SEQ", "AND", "XOR" or "OR". A service node contains one of the abstract services in $\mathcal{S}$. The root of a composition tree, which we denote by $\mathcal{R}$, is always an operator node except in the trivial case where the set of abstract services $\mathcal{S}$ contains a single abstract service; intermediate nodes are also operator nodes, whereas the leaves of a composition tree are always service nodes. Note that a

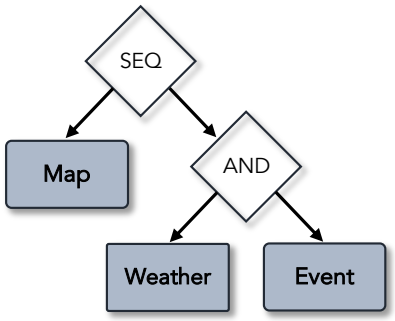

Fig. 2. Composition Tree Example

Algorithm 2: Progressive Pruning Algorithm (PPA)

Input: abstract services $\mathcal{S}$, tree root $\mathcal{R}$, budget $\mathcal{B}$
Output: minimal budget skyline $\mathcal{B}$-Sky^*

1 **if** $\mathcal{R}$ *is a service node* $\mathcal{S}_i$ **then**
2 $\mathcal{S}_i \leftarrow \{s_{ij} \in \mathcal{S}_i \mid s_{ij}.p + \sum_{\mathcal{S}_k \neq \mathcal{S}_i} \mathcal{S}_k.p^- \leq \mathcal{B}\}$
3 **foreach** $s_{ij} \in \mathcal{S}_i$ **do**
4 $\ell(s_{ij}) \leftarrow$ fill the missing call limits of s_{ij}
5 $Sky_i \leftarrow$ compute the skyline of $\mathcal{S}_i$
6 **return** Sky_i
7 **else**
8 $\mathcal{C} \leftarrow$ compose LPA$(\mathcal{S}, \mathcal{R}_l, \mathcal{B})$, LPA$(\mathcal{S}, \mathcal{R}_r, \mathcal{B})$
9 $\mathcal{S}' \leftarrow$ involved abstract services in $\mathcal{C}$
10 **if** $\mathcal{S}' = \mathcal{S}$ **then**
11 $\mathcal{C} \leftarrow \{c \in \mathcal{C} \mid c.p \leq \mathcal{B}\}$
12 **else**
13 $\mathcal{C} \leftarrow \{c \in \mathcal{C} \mid c.p + \sum_{\mathcal{S}_k \notin \mathcal{S}'} \mathcal{S}_k.p^- \leq \mathcal{B}\}$
14 **foreach** $c \in \mathcal{C}$ **do**
15 $\ell(c) \leftarrow$ compute the call limit vector of c w.r.t. $\mathcal{R}$
16 $\mathcal{C} \leftarrow$ eliminate redundant compositions from $\mathcal{C}$
17 $\mathcal{B}$-$Sky^* \leftarrow$ compute skyline of $\mathcal{C}$
18 **return** $\mathcal{B}$-Sky^*

composition tree can be either a non-binary tree or a binary tree. Assume for example a sequence of three abstract services $\mathcal{S}_i$, $\mathcal{S}_j$ and $\mathcal{S}_k$. This can be represented by a root node "SEQ" having three children $\mathcal{S}_i$, $\mathcal{S}_j$ and $\mathcal{S}_k$ or by a root node "SEQ" having two children $\mathcal{S}_i$ and an intermediate operator node "SEQ", which in turn has two children $\mathcal{S}_j$ and $\mathcal{S}_k$. In the following, we consider binary composition trees. Figure 2 depicts the composition tree of our running example.

The main idea of PPA is to apply the pruning rule not only on services but also on sub-compositions to further prune the search space. In our example, after applying the pruning rules on the abstract services, i.e., once the skylines of the Map, Weather and Event services are computed, the Weather and Event services are combined to generate sub-compositions on which the pruning rules will be applied. Then the result is combined with the Map services to generate the final compositions on which a minimal budget skyline will be computed. The pseudocode of PPA is shown in Algorithm 2.

PPA proceeds as follows. For a given composition tree node $\mathcal{R}$, initially the root node, the algorithm checks whether it is a service node or an operator node (line 1). If $\mathcal{R}$ is a service node, say $\mathcal{S}_i$, then, similar to LPA, PPA eliminates services that will result in compositions that are out of budget based on Pruning Rule 1 (line 2), fills the missing call limit of the retained services (loop in line 3) then computes and returns the skyline of $\mathcal{S}_i$ (lines 5–6) to exploit Pruning Rule 2. Otherwise, i.e., $\mathcal{R}$ is an operator node, then PPA generates the possible compositions/sub-compositions $\mathcal{C}$ from the call of PPA for both left and right children of $\mathcal{R}$ (line 8), denoted by $\mathcal{R}_l$ and $\mathcal{R}_r$, respectively. Then, the algorithm computes the set $\mathcal{S}'$ containing the involved abstract services in $\mathcal{C}$ (line 9). This set is useful to utilize Pruning Rule 1 on sub-compositions. If $\mathcal{S}' = \mathcal{S}$, i.e., the compositions in $\mathcal{C}$ contain a service from all service classes, then only those that fall within the budget are retained (lines 10–11); else, i.e., the compositions in $\mathcal{C}$ are sub-compositions, then PPA applies Pruning Rule 1 by considering the price of the sub-compositions and the price of the cheapest service in each non-involved service class $\mathcal{S}_k$ (lines 12–13). After that, the call limit vector of each retained composition/sub-composition is computed according to node operator $\mathcal{R}$ using the call limit aggregation rules (loop in line 14). Afterward, PPA eliminates redundant compositions/sub-compositions at random, computes and returns the skyline of the non-redundant compositions/sub-compositions set (lines 16–18). This process will be repeated until all nodes are examined. At the end, a minimal budget skyline is returned.

TABLE IV
WEATHER–EVENT SUB-COMPOSITIONS

Sub-composition	Call limits	Price ($)
$c'_1 = \langle s_{21}, s_{31} \rangle$	3/s, 50/m, 70K/d, 70K/M	17
$c'_2 = \langle s_{21}, s_{32} \rangle$	3/s, 25/m, 36K/d, 70K/M	22
$c'_3 = \langle s_{22}, s_{31} \rangle$	50/s, 50/m, 70K/d, 70k/M	17
$c'_4 = \langle s_{22}, s_{32} \rangle$	25/s, 25/m, 36K/d, 70k/M	22

Let us apply PPA on our running example. Similar to LPA, PPA retains services s_{11} and s_{12} from the Map services, s_{21} and s_{22} from the Weather services, and s_{31} and s_{32} from the Event services. Then the retained Weather and Event services are combined to generate the sub-compositions listed in Table IV along with their call limit vectors and princes. As adding 8$ (i.e. the minimum price of Map services) to the price of any sub-composition will not exceed the budget, in this example, no sub-composition is eliminated by Pruning Rule 1. However, observe that there is a dominant sub-composition, which is c'_3. The algorithm thus combines only c'_3 with the skyline Map services s_{11} and s_{12}, and generates the compositions c_3 and c_7 in Table III. As c_3 and c_7 are not equivalent and do not dominate each other. As c_3 and c_7 are not equivalent and do not dominate each other, they are returned as a minimal budget skyline – recall that in our example there are two minimal budget skyline sets $\{c_1, c_7\}$ and $\{c_3, c_7\}$. From this example, we can see that PPA generates fewer compositions than LPA. But we need to confirm the superiority of PPA by an experimental evaluation.

IV. Experimental Evaluation

In this section, we present an experimental evaluation of our approach. In particular, we conduct two sets of experiments. First, we examine the size of the budget skyline ($\mathcal{B}$-Sky), i.e., the conventional skyline applied to our problem, and minimal budget skyline ($\mathcal{B}$-Sky^*). With fewer compositions, customers will choose from fewer alternatives, which is desired. As $\mathcal{B}$-Sky^* is theoretically smaller than $\mathcal{B}$-Sky, we would like to see how much smaller is $\mathcal{B}$-Sky^* compared to $\mathcal{B}$-Sky in practice. In the second set of experiments, we investigate the performance of our proposed algorithms (LPA and PPA). As a baseline, we consider the adaptation of existing works [11]–[13] to our problem. This line of work uses a pruning technique similar to Pruning Rule 2 for QoS-based service composition. Roughly speaking, this baseline algorithm, which we denote by BSA, looks like LPA without Pruning Rule 1.

A. Evaluation Setup

Since there is not any sizable API composition test case that is in the public domain and that can be used for experimentation purposes, we implemented a service generator. The generator takes as input a real-life composition model, e.g., the model in Figure 1, and produces for each abstract service a set of synthetic services and their associated synthetic call limits and prices. The generation is controlled by three parameters: (1) the number of abstract services; (2) the number of services per abstract service; and (3) the number of time units. Each service may have between 0 and 3 call limits, with 0 standing for unlimited number of calls. As in reality, it is more likely that a service enforces two call limits (a rate limit and a quota). In addition, to be even closer to reality, the generation is made so that the distribution of the call limits, when converted to time unit "day", follows the distribution obtained in [6] (see Figure 13 in the paper). The price of each service ranges between 5$ and 70$. It is realistic to assume that the price is typically anti-correlated with call limits, i.e., services with good call limits are more likely to be expensive. Our service takes this fact into account.

In each experiment, we investigate the effect of one problem parameter, while we set the remaining ones to their default values. Table V displays the parameters under investigation, their examined and default values.

The data generator and the algorithms, i.e., BLA, LPA and PPA were implemented in Python 3, and all experiments were conducted on a 3.0 GHz Intel Core i7, with 64 GB of RAM swapping to an SSD of 512GB with 210GB free, running Windows. Reported values are averages of 10 executions on 10 different data generation.

TABLE V
Parameters and Examined Values

Parameter	Examined Values	Default
# abstract services (m)	3, 4, 5, 6, 7	5
# services per abstract service (n)	100, 150, 200, 250, 300	200
# time units (d)	3, 4, 5, 6, 7	5
Budget per abstract service (b)	5, 10, 15, 20, 25	15

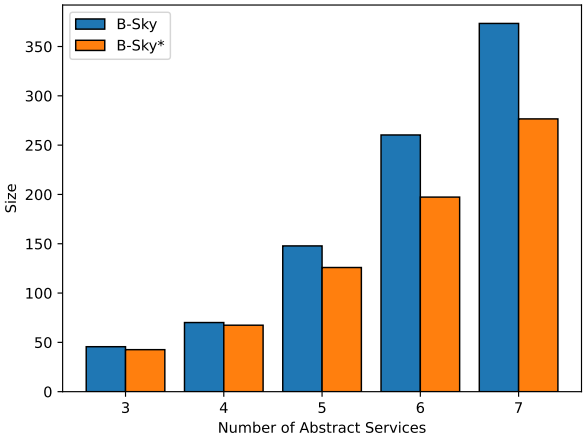

Fig. 3. Size vs number of abstract services m

B. $\mathcal{B}$-Sky Size vs $\mathcal{B}$-Sky^* Size

In the first set of experiments, we compare the size of the budget skyline and that of the minimal budget skyline varying m, n, d and b. The results are shown is Figure 3, Figure 4, Figure 5 and Figure 6, respectively. A general observation is that the size of $\mathcal{B}$-Sky^* is smaller than the size of $\mathcal{B}$-Sky. Smaller size reduces the effort required to manually examine the selected compositions to choose the most suitable one.

As depicted in Figure 3, the size of both skyline variants increases with the increase of m. This is because, the number of possible compositions increases exponentially with the increase of m. Therefore more compositions have chances not to be dominated.

Figure 4 shows that the size of $\mathcal{B}$-Sky and $\mathcal{B}$-Sky^* increases with higher n since when n increases the number of candidate compositions increases significantly. Therefore, similar to the last experiment, more compositions have chances not to be dominated.

Observe in Figure 5 that both $\mathcal{B}$-Sky and $\mathcal{B}$-Sky^* have higher size as d increases. The reason is that, with the increase of d, the dimensionality of the call limit vectors of the compositions becomes significant. Therefore, a composition has better opportunity not to be dominated in all time units.

Figure 6 depicts the size of $\mathcal{B}$-Sky and $\mathcal{B}$-Sky^* varying b. As expected, the size of both variants increases with the increase of b. This is because with more budget more compositions fit within the budget.

C. Scalability of $\mathcal{B}$-Sky^* Algorithms

In this set of experiments, we compare the algorithms BLA, LPA and PPA in terms of execution time. The results varying m, n, d and b are depicted in Figure 7, Figure 8, Figure 9 and Figure 10, respectively.

As shown in Figure 7, the performance of BLA and LPA deteriorates when m increases. However, PPA remains efficient. For example, when $m = 7$ PPA is more than four orders of

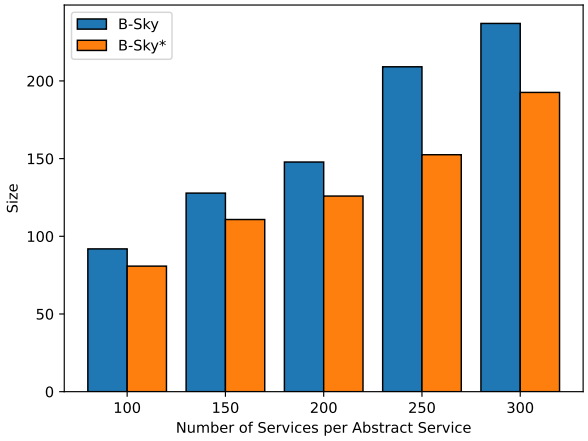

Fig. 4. Size vs number of services per abstract service n

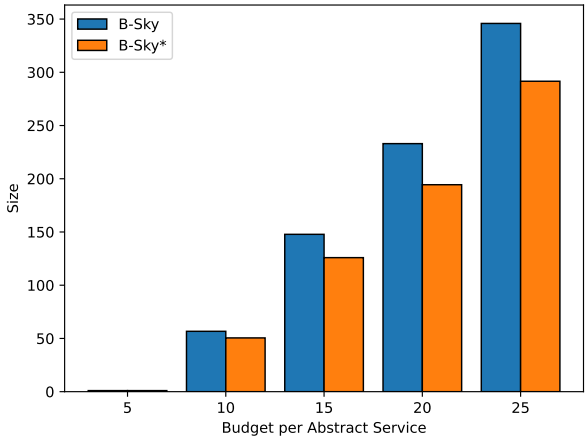

Fig. 6. Size vs budget per abstract service b

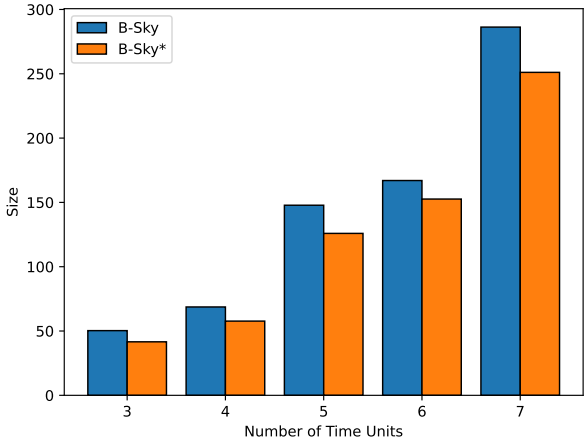

Fig. 5. Size vs number of time units d

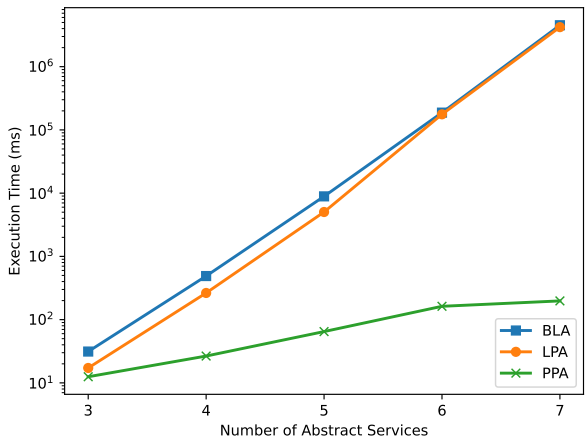

Fig. 7. Execution time vs number of abstract services m

magnitude faster than BLA and LPA. The reason is that BLA and LPA need to examine a huge number of compositions when m is large, while PPA eliminates progressively the non-relevant sub-compositions. Another important observation is that LPA slightly outperforms BLA since it employs an additional pruning rule, i.e., price-based pruning.

Figure 8 shows that the execution time of all algorithms increases as n increases. It is to be expected since more services lead to more combinations to examine. Similar to the last experiment, PPA outperforms LPA, which in turn outperforms BLA for the same reason.

Figure 9 draws the execution time of the three algorithm varying d. observe that the execution time of all algorithms increases with higher d. This is due to three facts. First, the algorithms require more time to fill the missing call limits of services and to compute the call limit vectors of the compositions. Second, the cost of dominance checks increases. And more importantly, as discussed previously, the dimensionality of the call limit vectors of the compositions becomes significant and a composition has better opportunity not to be dominated in all time units. Therefore the algorithms need to perform more dominance checks. In comparison, PPA is constantly two orders of magnitude faster than LPA, which in turn is faster BLA.

Observe in Figure 10 that the performance of all algorithms deteriorates with higher b. It is to be expected since with more budget more compositions fit within the budget. Therefore, the algorithms need to examine more compositions. Another important observation is that, compared to LPA, BLA equalizes when b exceeds 20$. This is because, with a high budget, the price-based pruning strategy becomes less effective. In comparison, still, PPA outperforms LPA, which in turn outperforms BLA.

To sum up, PPA is the clear winner. This demonstrates the benefits resulting from the application of our pruning techniques progressively on the sub-compositions instead of

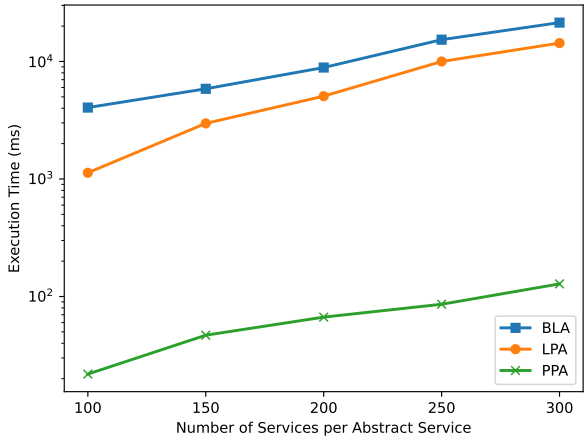

Fig. 8. Execution time vs number of services per abstract service n

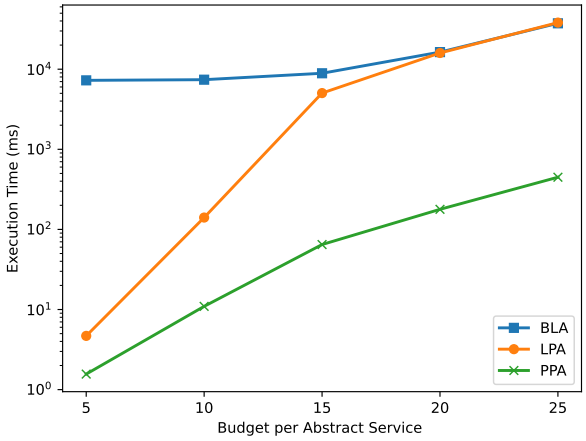

Fig. 10. Execution time vs budget per abstract service b

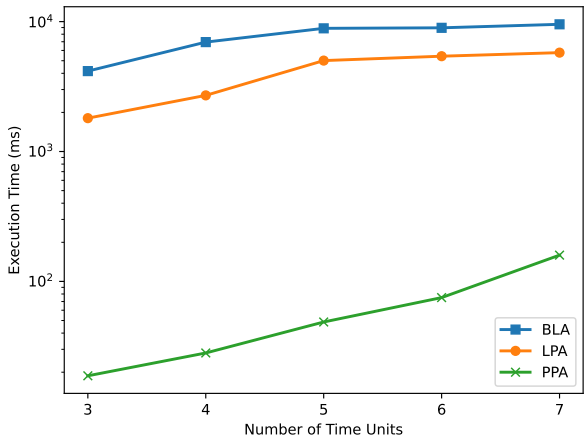

Fig. 9. Execution time vs number of time units d

applying them locally. We can also see a slight superiority of LPA compared to BLA. This shows the importance of the price-based pruning technique.

V. Related Work

Our proposal can be related to previous work on API analysis and QoS-based composite service selection. Below we provide an overview on works done in both topics.

A. API Analysis

There is a line of work that empirically analyzes the characteristics and features of APIs. We summarize the results on the call limit. In [7], the authors analyze a set of 69 RESTful APIs of XaaS offering. They distinguish different kinds of limitations: (1) limits over requests; (2) limits over storage; (3) limits over resources; (4) limits over transaction size; and (5) limits over other metrics. The results show that more than 80% of rate limits and quotas are defined over number of requests. In addition, this kind of rate limits and quotas are classified among the three most important API concepts in [9]. This motivated us to focus our work on limits over requests, however, our definitions and algorithms can be easily adapted to other types of limitations. The work in [14] presents an analysis of the top-20 RESTful services listed on programmableweb[3]. Only three of the examined services, i.e., 15% do not express any usage limits. Another study [15] on 222 Web APIs founds that 89% of them state and implement limitation on the number of invocations. The salient analysis [6] on 500 public REST Web APIs indicates that 56.4% of them provide information about the existence or absence of call limits. Only 12.8% of the 500 APIs state that they do not enforce any call limits.

This works only provide statistics on the call limits of APIs, but do not deal with the aggregation of call limits or the selection of the best service compositions as we have done. However, we relied on this works for the generation of our data to be close to reality.

B. QoS-based Composite Service Selection

QoS-based composite service selection has received much attention over the past two decades. We review the most outstanding works. In [3], the authors introduce a linear programming-based selection model to find the optimal components services. In [2], the authors consider an extended linear programming model that can fulfill constraints at runtime through adaptive reoptimization under varying QoS characteristics. The work in [4] investigates a combinatorial model and a graph model for the QoS-based service composition problem and introduces a heuristic algorithm for each model. In [5], the authors propose a hybrid approach that combines global optimization with local selection to find a close-to-optimal selection efficiently. The proposed solution consists of two steps: first, they use mixed integer programming to find the

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

optimal decomposition of global QoS constraints into local constraints. Second, they use distributed local selection to find the best web services that satisfy these local constraints.

Another stream of work leverages the concept of skyline. The work in [11] identify a set of representative skyline services that best represent all the trade-offs of different QoS attributes, so that it is possible to find compositions with a high utility score. In [12], the authors propose indexing services to compute the skyline compositions efficiently. In [16], the authors model preferences on QoS attributes using fuzzy linguistic predicates, which are transformed into numerical weights. Then, the authors present a hybrid evolutionary algorithm to find a set of preferred skyline solutions heuristically. The authors in [13] develop efficient algorithms that allow to find the skyline compositions from the skyline of abstract services instead of considering all possible service compositions. The work in [17] focuses on computing the skyline compositions in the presence of QoS correlations. Different pruning techniques are investigated to accelerate the computing process.

Contrary to these QoS-based works, ours is based on the concept of call limit. Moreover, as shown in our experimental evaluation, our algorithms outperform the adaptation of these works to our problem.

VI. Conclusion

In this paper, we introduced the problem of call limit-based composite service selection. We first showed how the call limits of the compositions can be computed from the call limits of the individual services depending of the composition plan. We then proposed the notion of minimal budget skyline which comprises the most relevant compositions and developed two algorithms based on effective pruning techniques for the minimal budget skyline computation problem. Our experimental evaluation showed that the minimal budget skyline can eliminate a large number of candidate compositions that are considered redundant. Moreover, we investigated the scalability of our algorithms and found that they outperform the adaptation of existing methods to our problem.

An interesting future direction is to deal with the issue of incremental updates. For example, if a new service is added or a service dynamically changes its cost or call limits, the algorithms must include techniques to find the correct result without restarting from scratch.

References

[1] R. Koçi, X. Franch, P. Jovanovic, and A. Abelló, "Classification of changes in API evolution," in *Proceedings of the IEEE International Enterprise Distributed Object Computing Conference (EDOC)*. Paris, France: IEEE, 2019, pp. 243–249. [Online]. Available: https://doi.org/10.1109/EDOC.2019.00037

[2] D. Ardagna and B. Pernici, "Adaptive service composition in flexible processes," *IEEE Transactions on Software Engineering (TSE)*, vol. 33, no. 6, pp. 369–384, 2007. [Online]. Available: https://doi.org/10.1109/TSE.2007.1011

[3] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "Qos-aware middleware for web services composition," *IEEE Transactions on Software Engineering (TSE)*, vol. 30, no. 5, pp. 311–327, 2004. [Online]. Available: https://doi.org/10.1109/TSE.2004.11

[4] T. Yu, Y. Zhang, and K. Lin, "Efficient algorithms for web services selection with end-to-end qos constraints," *ACM Transactions on the Web (TWEB)*, vol. 1, no. 1, p. 6, 2007. [Online]. Available: https://doi.org/10.1145/1232722.1232728

[5] M. Alrifai and T. Risse, "Combining global optimization with local selection for efficient qos-aware service composition," in *Proceedings of the International World Wide Web Conference (WWW)*, J. Quemada, G. León, Y. S. Maarek, and W. Nejdl, Eds. Madrid, Spain: ACM, 2009, pp. 881–890. [Online]. Available: https://doi.org/10.1145/1526709.1526828

[6] A. Neumann, N. Laranjeiro, and J. Bernardino, "An analysis of public REST web service apis," *IEEE Transactions on Services Computing (TSC)*, vol. 14, no. 4, pp. 957–970, 2021. [Online]. Available: https://doi.org/10.1109/TSC.2018.2847344

[7] A. Gamez-Diaz, P. Fernandez, and A. R. Cortés, "An analysis of restful apis offerings in the industry," in *Proceedings of the International Conference on Service-Oriented Computing (ICSOC)*, ser. Lecture Notes in Computer Science, E. M. Maximilien, A. Vallecillo, J. Wang, and M. Oriol, Eds., vol. 10601. Madrid, Spain: Springer, 2017, pp. 589–604. [Online]. Available: https://doi.org/10.1007/978-3-319-69035-3_43

[8] M. Stocker, O. Zimmermann, U. Zdun, D. Lübke, and C. Pautasso, "Interface quality patterns: Communicating and improving the quality of microservices apis," in *Proceedings of the European Conference on Pattern Languages of Programs (EuroPLoP)*. Irsee, Germany: ACM, 2018, pp. 10:1–10:16. [Online]. Available: https://doi.org/10.1145/3282308.3282319

[9] A. Gamez-Diaz, P. Fernandez, A. Ruiz-Cortés, P. J. Molina, N. Kolekar, P. Bhogill, M. Mohaan, and F. Méndez, "The role of limitations and slas in the API industry," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, (ESEC/FSE)*, M. Dumas, D. Pfahl, S. Apel, and A. Russo, Eds. Tallinn, Estonia: ACM, 2019, pp. 1006–1014. [Online]. Available: https://doi.org/10.1145/3338906.3340445

[10] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, D. Georgakopoulos and A. Buchmann, Eds. Heidelberg, Germany: IEEE Computer Society, 2001, pp. 421–430. [Online]. Available: https://doi.org/10.1109/ICDE.2001.914855

[11] M. Alrifai, D. Skoutas, and T. Risse, "Selecting skyline services for qos-based web service composition," in *Proceedings of the International World Wide Web Conference (WWW)*, M. Rappa, P. Jones, J. Freire, and S. Chakrabarti, Eds. Raleigh, North Carolina, USA: ACM, 2010, pp. 11–20. [Online]. Available: https://doi.org/10.1145/1772690.1772693

[12] Q. Yu and A. Bouguettaya, "Multi-attribute optimization in service selection," *World Wide Web : Internet and Web Information Systems (WWW)*, vol. 15, no. 1, pp. 1–31, 2012. [Online]. Available: https://doi.org/10.1007/s11280-011-0121-9

[13] ——, "Efficient service skyline computation for composite service selection," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 25, no. 4, pp. 776–789, 2013. [Online]. Available: https://doi.org/10.1109/TKDE.2011.268

[14] D. Renzel, P. Schlebusch, and R. Klamma, "Today's top "restful" services and why they are not restful," in *Proceedings of the International Conference on Web Information Systems Engineering (WISE)*, ser. Lecture Notes in Computer Science, X. S. Wang, I. F. Cruz, A. Delis, and G. Huang, Eds., vol. 7651. Paphos, Cyprus: Springer, 2012, pp. 354–367. [Online]. Available: https://doi.org/10.1007/978-3-642-35063-4_26

[15] F. Bülthoff and M. Maleshkova, "Restful or restless - current state of today's top web apis," in *The Semantic Web: ESWC 2014 Satellite Events*, ser. Lecture Notes in Computer Science, V. Presutti, E. Blomqvist, R. Troncy, H. Sack, I. Papadakis, and A. Tordai, Eds., vol. 8798. Anissaras, Crete, Greece: Springer, 2014, pp. 64–74. [Online]. Available: https://doi.org/10.1007/978-3-319-11955-7_6

[16] X. Zhao, L. Shen, X. Peng, and W. Zhao, "Finding preferred skyline solutions for sla-constrained service composition," in *Proceedings of the IEEE International Conference on Web Services (ICWS)*. Santa Clara, CA, USA: IEEE Computer Society, 2013, pp. 195–202. [Online]. Available: https://doi.org/10.1109/ICWS.2013.35

[17] Y. Du, H. Hu, W. Song, J. Ding, and J. Lu, "Efficient computing composite service skyline with qos correlations," in *Proceedings of the IEEE International Conference on Services Computing (SCC)*. New York City, NY, USA: IEEE Computer Society, 2015, pp. 41–48. [Online]. Available: https://doi.org/10.1109/SCC.2015.16