Received: 23 January 2023 | Revised: 31 July 2023 | Accepted: 13 August 2023

DOI: 10.1002/spe.3260

RESEARCH ARTICLE

WILEY

Location-aware scalable service composition

Nicolás Pozas García | **Francisco Durán** | **Katia Moreno Berrocal** | **Ernesto Pimentel**

ITIS Software, University of Málaga, Málaga, Spain

Correspondence
Francisco Durán, ITIS Software, University of Málaga, Málaga, Spain.
Email: fdm@uma.es

Funding information
Spanish Government

Abstract

The problem of service composition is the process of assigning resources to services from a pool of available ones in the shortest possible time so that the overall quality of service is maximized. This article provides solutions for the composition problem that takes into account its scalability, services' locations, and users' restrictions, which are key for the management of applications using state-of-the-art technologies. The provided solutions use different techniques, including genetic algorithms and heuristics. We provide an extensive experimental evaluation, which shows the pros and cons of each of them, and allows us to characterize the preferred option for each specific problem. Since no solution dominates the others, we propose a decision tree, based on our results, to select the best composition algorithm in each situation.

KEYWORDS

genetic algorithms, quality decomposition, quality of service, service composition

1 | INTRODUCTION

Automatic service composition has turned key in the development of enterprise architectures for distributed systems. Technologies such as the cloud, the Internet of Things (IoT), or the cloud-fog-edge continuum have brought new challenges into the picture: (1) applications of hundreds, even thousands of services are being considered, and managing them manually has become impractical; (2) functionally equivalent services, with different quality of service (QoS) attributes, may be traded in services repositories, with the goal of optimizing the overall QoS, at a minimum cost, but the available offer makes the decision of where to deploy each service much harder; and (3) the distances between the components of an application, and between the front-end services and the users of the applications, as well as the quality of the communication channels being used, are key for providing the appropriate response times to the users, which in the context of service providers located around the world may suppose a significant difference.

Although the composition problem can be viewed from different perspectives, different nomenclatures, and in different domains, for us, the problem of service composition is the process of assigning resources to services from a pool of available ones. From all possible assignments, we are interested in finding the best possible solution, meaning that the overall QoS is optimal, or pseudo-optimal, in the shortest possible time, and all users' preferences (soft and hard) are satisfied. Given the above mentioned scenario, we provide solutions for the composition problem that takes into account its scalability, services' locations, and users' restrictions.

Abbreviations: GA, genetic algorithm; IoT, internet of things; QoS, quality of service; STD, standard deviation;

In service-oriented computing, it is common practice to arrange existing services into workflows.[1] The architecture of the service-based application to be composed will be provided as input, so that, not only service descriptions and constraints are considered, but also explicit functional dependencies between services. We assume a workflow description formalism like BPMN[2] or WS-BPEL,[3] although our techniques are applicable to other workflow notations, like CWL,[4] or OPMW,[5] from which we can extract functional dependencies.

We assume that a previous functional matchmaking has taken place. For each service, there will be an offer of candidate service providers, each with an estimate of its QoS attributes, specified as its service level agreement (SLA), which will be taken into account to compute the solution. To be able to consider communications' quality, we assume that the location and connection capacity of the providers will be as well available. Both attributes will be a key part of our work, to minimize latency times between service communications and to take into account potential bottlenecks. Even though this information is key to provide an optimal allocation in the cloud-fog-edge continuum, and even more if they are on the move, it may not always be available for all the services, and it can even change overtime. If not available, these values could be estimated empirically by performing actual invocations to the services.

QoS-aware service composition is a well-known NP-hard problem.[6] Therefore, while the problem can be solved using exact methods, like integer programming,[7] there is a consensus on the use of genetic algorithms (GA) as a good option for the composition problem because of its flexibility and easy adaptation to almost any optimization problem (see, e.g., works by McCall,[8] Sastry et al.,[9] and Katoch et al.[10]) In general, GA-based procedures provide an acceptable solution, mainly because in them time and precision can be traded. In other words, even if limiting its time of execution, they can still be able to find "acceptable" solutions.

We can also find in the literature solutions based on the use of heuristics (see, e.g., works by Mabrouk et al.[11] and Yuan et al.[12]), mainly based on the *utility* of each candidate provider for some global goal. In general, we may see the notion of utility as an estimate of the likelihood of each provider to be able to contribute towards a given goal. In other words, utility measures try to respond the question of how, and how much, each of the candidate providers of a service may contribute, for example, to a target global value, to given requirements, or to replaceability. For example, our goal may be to get a lower economic cost and a higher reliability. Using this type of heuristics, with the subsequent application of a GA to guarantee the satisfaction of user constraints, could give very satisfactory results in a very reasonable time (see, e.g., works by Mardukhi et al.[13] or Mabrouk et al.[11]). Indeed, the method may get not only solutions for a target goal, but also do it in a shorter time, since service providers may be grouped to significantly reduce the search space.

Let us get a taste of the complexity of the problem by experimentally comparing several of these alternative solutions. Let us consider a pure-genetic-algorithm solution (GA), a GA-utility solution (U), and a solution deciding on the best candidate by individually, or locally, considering each service (Express). Let us consider an application with 100 services, with 10 candidate providers for each service. Such a composition problem has a search space of 10^{100} possible combinations. The GA solution would start mutating and recombining individuals from its initial population through this search space until a suitable result is found. Depending on its meta-parameters, it would explore a bigger or smaller part of this search space, with a bigger or smaller chance of getting closer to a global optimum. A utility-based method with five utility degrees would have a search space of size 5^{100}, considerably smaller, although requiring some additional previous analysis. Finally, a method in which the service provider for each service is decided in isolation, just by taking the best candidate, would have a much lower complexity: $100 \times 10 = 10^3$. The question is then how much do we lose by speeding up the search?

We advance that our experiments show that the Express method works much faster, with a non-significant lost of accuracy in most cases. There are however several factors to take into account. First, it is not clear whether constraints can correctly be taken into account in the Express method, since there is no way to consider global constraints when deciding locally. Furthermore, the Express method may significantly depend on the application's architecture and the offer of providers, which may lead to some unpredictability. In Section 5, we will also see how the GA and utility-based methods behave. We will see how the utility-based methods have a much lower decision time, but require a significant preprocessing, which grows with the number of available service providers. On the other hand, as we will see, the number of providers is not that relevant for the other methods, since having a greater offer seems to simplify the problem of finding an acceptable candidate. This, in fact, is the key to understand the behavior of the utility-based solutions, which rely on the reduction of the number of candidate providers by grouping them in clusters. The utility-based methods may still be useful if the preprocessing can be reused.

In the rest of the article, we present in detail the above-mentioned alternative methods for service composition, with a focus on scalability, latency, and constraints. Specifically, we exploit the combination of GA, direct methods, and

utility-based heuristics. Our solutions consider the usual QoS attributes, namely cost, execution time, reliability, and availability, but also provides novel proposals to take into account channel-dependent attributes like latency and throughput. These two attributes have a significant importance in the cloud-fog-edge continuum, because of the potential impact of the distance between application's service providers, and the variety of available channels. To deal with latency and throughput, we propose a novel solution using the *location* and *bandwidth* of each service provider. Some of these solutions are able to handle global constraints (GA, U, and UM), and others do not (Express). The solution of the composition will be guided by the so-called *fitness* or *objective* functions, as a measure of the global aggregated QoS, which will be used to quantifying the quality of a solution for a given composition.

A discussion on these results is presented, focusing on their advantages and disadvantages, and bringing light to under what circumstances we would consider each of the proposed methods. An extensive experimentation has been carried out. Applications of up to 2000 services have been considered, with between 100 and 1000 candidate providers per service. As we will explain later, application structures, services, and service providers have been randomly generated. Some of those results are presented in this article and additional details may be found in the companion web site,[14] where implementations for all these solutions, together with scripts for the random generation of problems are also available. This extensive experimentation has allowed us to draw some conclusions on the pros and cons of each of the methods, and specifically on when each of them provides a better performance. This information is summarized in Section 5.3, in which a decision tree is provided.

The structure of the article is as follows. Section 2 presents a short discussion on the state of the art, focusing on different methods available to solve the service composition problem. Section 3 presents the common ground for the different methods, including the application model provided as input, the QoS attributes, the constraints, and the fitness/goal function used to quantify the quality of a solution and to compare composition solutions. Section 4 explains each of the provided solutions. Section 5 presents a series of experiments that show how the different solutions behave. Section 6 wraps up with some final conclusions and presents some lines of future work.

2 | RELATED WORK

In recent years, we have seen a development in distributed systems technologies, witnessing the appearance of new paradigms like cloud computing, mobile computing, the IoT, or the fog/edge computing. With these advances, new challenges have also emerged, and a large number of methods have been proposed for service composition. Even though most of them assume multiple existing web services arranged into workflows, many of them are merely based on the matching of input-output parameters of services. However, besides these parameters, other elements may affect the execution of services and their composition, such as global conditions, constraints, or service execution results.

Strunk's survey[6] summarizes, classifies and evaluates major research efforts on QoS-aware service composition. The survey concludes that, given the exponential time and costs of the composition problem, most approaches try to simplify the problem by either linearizing the objective function, considering local QoS maximization, not considering QoS constraints, considering single-objective optimization, or selecting a sub-optimal solution to the problem.

More recently, Arellanes and Lau[15] evaluate different service composition mechanisms for IoT systems, with a focus on their scalability. Instead of the usual horizontal/vertical scalability,[16] they focus, like us, on functional scalability, where scalability is studied in terms of the number of services composed. Even though Arellanes and Lou provide an interesting evaluation of composition mechanisms, supporting distributed dataflows, service-location transparency and other features, they propose an algebraic model able to match existing services. In our case, we assume that this match has previously resulted in an offer of candidate service providers for each of the services of an application. In the same vein, Chen et al.[17] propose the use of distributed collaborative filtering to select feedback using similarity rating of friendship, social contact, and community of interest relationships. Using these rates, they provide an adaptive filtering technique to determine the best way to combine direct and indirect trust to minimize convergence time and trust estimation bias. Horizontal/vertical scalability has been extensively studied, for example, in works by Calinescu et al.,[18] Coutinho et al.,[19] Xu and Helal,[20] Vakili and Navimipour,[21] and Cabre et al.[22]

In their 2022 paper, Razian et al.[23] present a literature review on the existing studies in service composition in dynamic environments, with a focus on the consideration of uncertainty. Given the variability of QoS values in real-world dynamic environments, the QoS estimation/prediction in service composition has become more challenging, since the QoS information available may be incomplete or unknown. The study reveals that the methods used to solve the problem of

composition under uncertainty by the works included in the study are probabilistic, machine-learning, fuzzy systems, and recommendations systems. According to this study, those works concerned about scalability use algorithms based on (meta-)heuristics. The study, however, points out the focus on service-specific attributes, without paying attention to their locations and latency: 61% of the analyzed works considers response time; availability, reliability, and throughput is considered in between 20% and 28% of them; cost, reputation, security, energy, and other attributes are considered in smaller percentages. One of the most promising works, regarding response time, might be the one by Parejo et al.,[24] where a hybrid approach that combines GRASP with Path Relinking[25] is used.

It is common practice to take as input of the composition problem a description of the architecture of the application, where the order of the nodes and their dependencies is made explicit. Although today other notations may be more active, the most frequently used notation for providing these descriptions is WS-BPEL[3]—see, for example, works by Siddiqui et al.,[26] Le et al.,[27] or Ouyang et al.[28] With WS-BPEL, control structures are defined as entities that regulate the execution flow, and depending on the job, more or less control structures will be used, such as conditions, sequential, iterative, parallel and so forth. Other works use BPMN[25] and similar workflow notations.

A wide range of solutions have been proposed for the composition problem. Resolution methods are usually divided in exact methods and heuristics-based methods. The use of other methods, like, for example, ant colonies,[29] dynamic programming,[30] or divide and conquer techniques,[31] have also been explored by different authors. A broader summary of the different alternatives can be found in Strunk's survey.[6] Among the exact methods, the most successful solutions are based on integer programming—see, for example, the work by Zeng et al.[32] Algorithms based on exact methods search for all possible compositions, and, for each of the compositions, the objective function is calculated. The composition with the best value for the objective function is selected as result. Typically, selection approaches can be based on local or global decisions. Depending on which approach is followed, local or global constraints might be considered. For instance, Yu et al.[30] propose a solution that uses a combinatorial model and a graph model. In it, the combinatorial model defines the problem as a multi-dimension multi-choice 0–1 knapsack problem; the graph model defines the problem as a multi-constraint optimal path problem. As pointed out, the execution time of exact methods grows exponentially as the input increases, becoming intractable.[30,32] The greatest benefit of these methods is that they provide exact solutions, guaranteeing global optimal solutions.

As an alternative to exact methods, some solutions that trade between execution time and precision have been proposed. By selecting service providers at the service level,[31,32] one can get good speed ups. One of such techniques is the use of the divide and conquer algorithm. However, as pointed out by Berbner et al.,[33] despite the great speed ups provided by the divide and conquer method, the use of this method does not give the best solution. Indeed, when looking at the problem globally, the greater the number of services, the more distant it is from the optimal one. Another problem with exact methods is their difficulties to establishing global restrictions. Yu and Lin[34] propose a method based on divide and conquer in which, to take into account global constraints, they propose making several runs of the algorithm, losing the speed that comes with the divide and conquer method.

Most of the proposals that use heuristics are based on GA.[35-39] Instead of exploring the entire candidate solutions space, as do exact methods, GA explore only part of this space, and such exploration is based on evolution. GA have one main weakness, since they are mainly based on statistics, there may be cases in which a local minimum is entered. When the GA is in this situation, it may be difficult to get out of it. A good selection of the hyper-parameters controlling the execution of the algorithm is key for getting the best results in the shortest times. These hyper-parameters basically control the type of mutations that the genes are subjected to and the criteria for deciding when the solution is good enough, that is, for stopping. Thanks to the fact that it significantly reduces complexity, and this time-precision trade, it is used by several authors. The biggest problem solutions based on GA have, as all heuristics-based methods, is that we cannot know how close to the optimal solution we are, not even if we already have the best solution. Furthermore, even with the best hyper-parameters, and though compromising some quality in the solutions, the growth of the execution times of GA is exponential, as is the search space. This means that, when scaling up, the quality of the solutions may get compromised if running under strict deadlines. With good hyper-parameters, as we will see in our experiments, this growth may be almost linear. One additional advantage of GA is that they are flexible enough to consider attributes of very different nature. As we will see in the coming sections, this flexibility has allowed us to consider channel-dependent attributes, like latency and throughput, in a rather efficient way.

Both single- and multi-objective algorithms are commonly used alternatives in the literature. Whilst a single-objective approach returns a single solution, the multi-objective one returns a set of solutions, each one achieving a different trade-off between the different objectives. Even though the multi-objective algorithms requires a posterior analysis, the single-objective approach relies on a weighted sum. The suitability of many-objective evolutionary algorithms to the

composition problem has been explored by different authors.[40-45] Suciu et al.[41] combine adaptive heuristics and the multi-objective algorithm. Moustafa and Zhang[42] use reinforcement learning to deal with the uncertainty characteristic inherent in open and decentralized environments. Trummer et al.[43] formally analyze complexity and precision guarantees. Ramírez et al.[45] consider nine QoS properties, namely response time, availability, reliability, throughput, latency, successability, compliance, best practices, and documentation. Yu et al.[44] propose an approach based on a reduced space searching strategy.

Alternative heuristic methods have been proposed based on the notion of *utility*. For example, Mabrouk et al.[11] propose an algorithm that uses a utility-like heuristic based on clustering. They calculate the utility of the providers and then clusterize to search for the composition. Yuan et al.[12] use the utility method with a fuzzy logic approach. Their work includes experiments with up to 1000 services, obtaining good execution times, which show the scalability of their proposal. However, these works use as quality attributes cost, execution time, availability, accuracy, and throughput. It is not clear how to extend their proposal to other attributes like latency.

In summary, what all above-mentioned works have in common is the search for a service provider for each service that maximizes a certain objective function. There are several ways of approaching the problem, but we can mainly differentiate between two types: exact and heuristic methods. Deshpande and Sharm[46] propose the use of a machine-learning classifier to choose between the most commonly used solutions, namely those based on exact methods (integer programming), on GA, and on ant colonies, depending on the complexity of the problem. Although the decision depends on multiple factors, in general, for problems with few services, the classifier recommends the exact methods,[30] while for more complex problems the classifier recommends the use of the solution based on GA,[36] rather than ant colony based algorithms.[29]

In most works where latency and throughput are dealt with References 11,47, and 39, these attributes are seen as numerical values associated to the service providers, that must be minimized or maximized, in the same way as the attributes of cost, response time and so forth are usually considered. Klein et al.[48,49] and Martini et al.[50] take into account communication channels by using a network model, what allows them to compute network QoS, including latency and transfer rate. Klein et al. use network models based on hash tables[48] and k-d trees.[49] Although they consider several other QoS attributes, the optimization is carried out for network QoS. In their 2014 paper,[49] they achieve a near-optimal latency for their service compositions, working under realistic network conditions. Instead of focusing on specific attributes, we treat all attributes uniformly, looking for solutions that maximize the aggregated QoS. Moreover, we use a simpler model, suitable for different techniques. As far as we know, no method has been proposed for dealing with latency apart from the above-mentioned ones, all of which use GA.

Utility approaches bring with them two main problems, the first of them is that since this is a heuristics-based method, as for GA, we cannot know with certainty if we have reached the best solution. The second problem is that all the attributes taken into account are service- and provider-dependent; they do not take into account attributes that depend on the communication channels between services.

Given the pros and cons of each of the most-frequently used methods, there are several key issues for which we cannot find a satisfactory answer in the literature. In the rest of the article, we present a general setting for scalable service composition with constraints, in which QoS attributes cost, execution time, reliability, availability, latency, and throughput are considered. As we will present in Section 5.3, a systematic strategy can then be defined in order to decide when using a method or another, depending on our specific needs.

3 | APPLICATION MODEL

The composition problem takes as input the application model and the offer of service providers on which to deploy the services. A workflow description of the application provides information on the services that are part of the application and the functional relationships between them, which is key for the aggregation of the QoS attributes as a global measure of the quality of the application. Since each of the services of our applications can be deployed on service providers at any location in the cloud-fog-edge continuum, and the quality of the communications is as important as the selection of providers, their locations, and the latency and throughput of the communication channels, must also be part of the inputs of the problem. The set of inputs is completed with a set of user's constraints, and with the relative importance of the different attributes when considering them together. We describe the details of these elements in the rest of this section.

3.1 | Service providers

Service providers are the services or infrastructure where services may be executed. In the cloud-fog-edge continuum, there are many types of providers, such as IaaS virtual machines, PaaS services, Arduino devices, Tomcat servers in on-site machines and so forth. We consider service providers as entities where we can run the services and for which we know their locations and the estimated values for their cost, response time, availability, reliability and so forth that is, their SLAs. The way in which we compute the different aggregate functions for each of these attributes will be discussed in Section 3.3.

Services must be executed as efficiently as possible, either individually or in combination with other services as described in the application's architecture. We assume that a previous matchmaking—in the line of those proposed by, for example, Wu and Wu,[51] Stefanic et al.,[52] and Kritikos and Plexousakis[53]—has been carried out, so that each service in an application will have a list of possible service providers where they can be deployed to be executed. Each service has a list of available and fully compatible providers to be executed, which may be different from the lists of the other services.

3.2 | Architectural patterns and composition quality

Given an architectural description of the application, and a set of providers for each of the services in it, the objective of the service composition problem is to obtain a composition with the best possible *quality*. But what does quality mean? By composition quality we mean a measure, normalized in the range [0, 1], with 0 the minimum possible quality and 1 the maximum one, that allows us to quantify this quality to compare different solutions. Each algorithm in this work has its own way of finding the best composition, but all solutions use the same measure of quality, what allows us to compare them. Although we use the same formula to evaluate a composition, each of the implemented methods has its own way of obtaining this value. We will discuss the objective function and the aggregation formulas in Section 3.3.

Although the basic elements of an application's architecture are its services, these will be grouped in recursive sub-structures. We call *component* the set of elements of which our architecture is composed, whether they are individual services, junction gates, separation gates, or sets of other components. The way in which the components are connected is usually defined by *architectural patterns*. The architectural patterns that we consider are the most common ones, namely the sequential, the parallel, the conditional, and the iterative patterns. These patterns are shown in Figure 1.

In the conditional and iterative patterns depicted in Figure 1, probabilities represent, respectively, the likelihood of executing a branch and the probability of making another iteration on the loop of services. Patterns and probabilities will be key to recursively calculate the global objective function using the aggregation functions presented in Section 3.3.

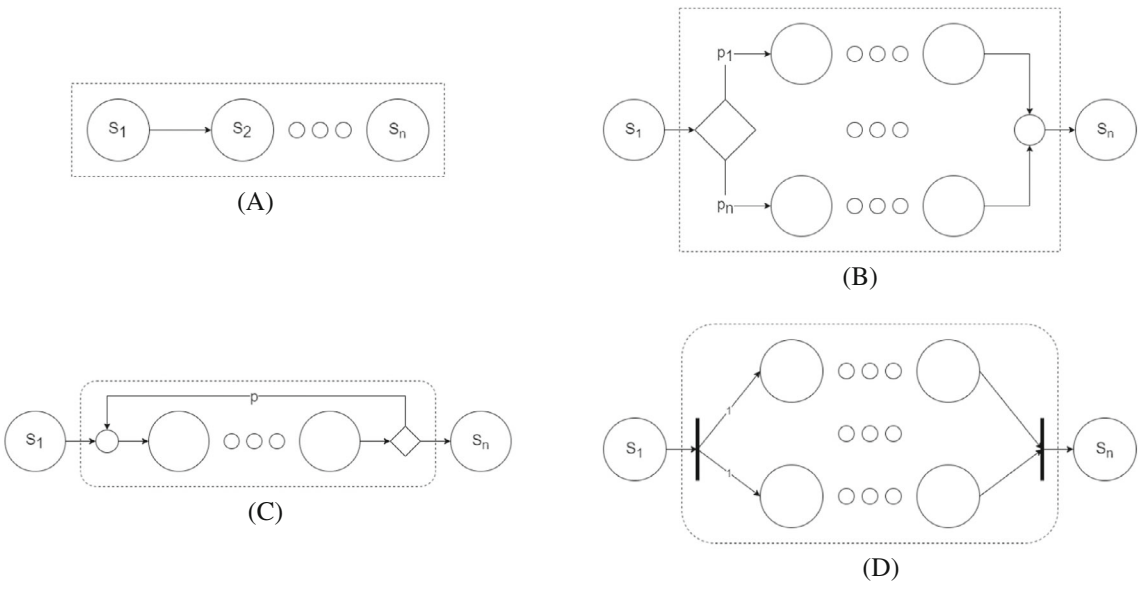

FIGURE 1 Architectural patterns. (A) Sequential. (B) Conditional. (C) Iterative. (D) Parallel.

Sequential pattern (Figure 1A). In this pattern, each component must wait for the previous component to perform its task. That is, the output from a component is the input of the next component in the sequence. For example, for summative QoS attributes, the goal is for each component to carry out its work as efficiently as possible, since the provider that offers the highest quality in isolation will be the best provider globally.

Conditional pattern (Figure 1B). This pattern has branches, each of which has a probability p_i of being executed, with $i = 1 \ldots n$, $0 \leq p_i \leq 1$, and $\sum_{i=0}^{n} p_i = 1$. To calculate the fitness function of each branch, as we will see in Section 3.3, we simply have to recursively calculate the aggregated value for that branch, and multiply it by its probability.

Parallel pattern (Figure 1C). In a parallel structure, all branches are executed concurrently. The services being executed in parallel must synchronize once completed. Therefore, for instance, its response time is given by the response time of the slowest branch. As we will see below, this is key, since, in this pattern, faster services may relax their response-time requirement in favor of improving other attributes. For example, we may have cases in which we decide to take a cheaper provider for a service because a faster, more-expensive one is possibly not going to improve the global fitness value. Something similar happens also with other attributes. For instance, for reliability, where if one branch of the pattern fails, we consider the whole corresponding component to have failed.

Iterative pattern (Figure 1D). This pattern represents a loop that have a probability p of repeating its execution, where $0 < p < 1$. Of course, the probability of getting out of the loop is then $1 - p$. Notice that $p = 0$ would mean that the body of the loop will never execute, and $p = 1$ an infinite loop.

3.3 | QoS attributes and their aggregation functions

We consider the following QoS attributes:

Cost. This attribute indicates the cost (in euros, €) per service invocation of the service. This attribute will be minimized.

Response time. The response time is measured as the time (in milliseconds, ms) that elapses between the call to a service and the output of that service. This attribute will be minimized.

Availability. Availability is measured as the rate (%) at which the resources and services of a system are accessible to end users, that is, available time with respect to total time. This attribute will be maximized.

Reliability. Reliability is measured as the rate (%) at which we expect the output from the execution of a service to be as expected. This attribute will be maximized.

Latency. Latency is measured as the time (in ms) it takes for data to get from the output of one component to the input of the next. This attribute will be minimized.

Throughput. Throughput is measured as the amount of information a channel is able to send from the output of one component to the input of the next in a given time (measured in megabytes per second, Mbps). The attribute will be maximized.

3.3.1 | Latency and throughput

Given the importance of latency and throughput for the cloud-fog-edge continuum, because of the potential impact of the distance between application's service providers, and the variety of available channels, we need a management of these attributes better than found in existing solutions. Specifically, if there is an offer of alternative providers for two services between which there is a communication, all possible combinations should be considered when choosing the best alternative. We cannot consider a latency value as something associated to a given provider, it also depends on where "the other side" is located.

To deal with latency and throughput, we take into account the *location* of each service provider. Such locations are provided as the actual latitude and longitude of a provider. Each provider also has a *bandwidth*, which determines the bandwidth the provider can work with. To be able to take into account in a systematic way the communications that may take place through gates, we have introduced components that act as logic gates and have their own providers. In other words, these components are responsible for joining several inputs into a single output (merge action) or an input into several outputs (split action), as routers or switches would do. These special providers only offer localization and bandwidth.

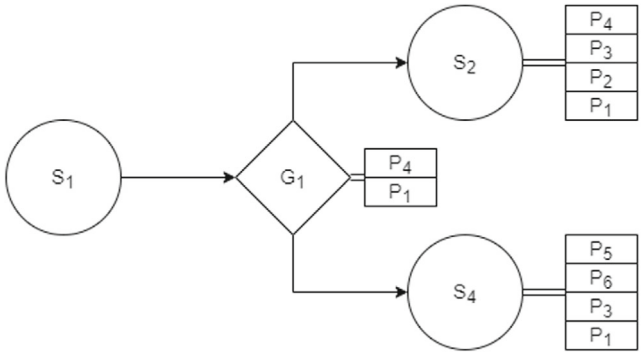

FIGURE 2 Example of latency estimation.

As each provider has a specific location, for latencies, we first need to calculate the distance between the different locations. Distances are approximated using the haversine formula, which determines the distance between two points on a sphere given their longitudes and latitudes.* Next, we multiply this value by a constant that represents the speed at which light travels through optical fiber. In our case, we have chosen to assume an estimated and uniform constant value of $1.5E-6$ as discussed in Poletti et al.'s paper.[54] Of course, this is a rough approximation. It is well known that geographic position and network latencies are not directly correlated. Alternatively, a system like Vivaldi[55] could have been used to estimate round-trip times (RTTs) between arbitrary nodes in a network.

Finally, the methods presented in Section 4 use latencies in two different ways. Once providers are chosen for the services, we can calculate distances and corresponding latency estimates. This is the case of the GA-based method (Section 4.2), in which the fitness function is calculated for specific assignments of providers to services. However, the methods taking decisions locally (Section 4.1) or based on heuristics (Section 4.3), require a measure of the latency associated to each service provider. We calculate this value of latency per provider by calculating the average of the latencies of such a provider with each provider of the services the service communicates with. Let us illustrate this by considering the situation in Figure 2, in which gate G_1 has two candidate providers (P_1 and P_4) and services S_2 and S_4 have, respectively, candidate providers $P_1, \ldots, P_4$ and P_1, P_3, P_5, P_6. And let us focus on providers P_1 and P_4 for gate G_1. To calculate an estimated latency for these providers we calculate the latency between each of these and each of the providers of S_2 and S_4, and calculate the average of these values. Specifically, for P_4, we calculate the average of the latencies between P_4 and $P_1, \ldots, P_4, P_1, P_3, P_5, P_6$, and store it in P_4. Then repeat the operation for P_1. Although a simple estimate, it has given good results, as will be seen in Section 5.

Like latency, throughput is an attribute that depends on several components. But in this case, in order to calculate the value of this attribute, we take the lowest bandwidth of all those obtained, that would be the bottleneck. To make testing easier, we have abstracted all possible combinations and created six levels of capability with the following values: (**L0**, 50 Mbps), (**L1**, 600 Mbps), (**L2**, 1.300 Mbps), (**L3**, 10.000 Mbps), (**L4**, 40.000 Mbps), and (**L5**, 160.000 Mbps). Furthermore, we assume that all bandwidths are compatible with each other, and that some technical solution is in place for connecting them.

Please, note that even though latencies and throughput are estimated using providers' location and bandwidth information, only constraints on global features are considered.

3.3.2 | Aggregation functions

Given an assignment of providers to services, the global values for each of the QoS attributes are calculated recursively using the architectural patterns presented in Section 3.2. Table 1 shows the aggregation functions used to calculate the values for the QoS attributes execution time, cost, reliability, and availability for the patterns in Figure 1.† Note that in the architectural patterns, probabilities represent the likelihood of executing a branch or another in a conditional pattern,

*To avoid recalculating the distance between locations continuously, all distances are calculated once and stored in a hash table.

†Similar aggregation patterns have been used by different authors, see, for example, works by Cardoso et al.,[37] Zeng et al.,[32] or Wan et al.[56]

TABLE 1 Aggregation functions for provider-dependent attributes.

QoS Attr.	Sequential	Conditional	Parallel	Iterative
Time (T)	$\sum_{i=1}^{n} T(t_i)$	$\sum_{i=1}^{n} p_i \cdot T(t_i)$	$\max_{i\in\{1\dots n\}} T(t_i)$	$\frac{T(t)}{1-p}$
Cost (C)	$\sum_{i=1}^{n} C(t_i)$	$\sum_{i=1}^{n} p_i \cdot C(t_i)$	$\sum_{i=1}^{n} C(t_i)$	$\frac{C(t)}{1-p}$
Reliability (R)	$\prod_{i=1}^{n} R(t_i)$	$\sum_{i=1}^{n} p_i \cdot R(t_i)$	$\prod_{i=1}^{n} R(t_i)$	$\frac{(1-p)\cdot R(t)}{1-p\cdot R(t)}$
Availability (A)	$\prod_{i=1}^{n} A(t_i)$	$\sum_{i=1}^{n} p_i \cdot A(t_i)$	$\prod_{i=1}^{n} A(t_i)$	$\frac{(1-p)\cdot A(t)}{1-p\cdot A(t)}$

or making another iteration on the loop of services or getting out of it. Note also that these probabilities are also used in the aggregation functions. As usual, these probabilities can be learnt from actual executions of the application or being estimated by experts.

The aggregated values for latency and throughput are also calculated recursively, but instead of following the nesting of the architectural patterns, for these attributes the procedure goes from the start event in the architectural description of the application following the communication channels. By considering services and gates as nodes, and assuming that $suc(x)$ denotes the set of successors of a node x, that $l_{x,y}$ is the latency between nodes x and y, and that $p_{x,y}$ is the probability of using the channel between x and y, the latency of a node x, denoted $L(x)$ is defined as follows.

$$L(x) = \begin{cases} 0, & \text{if } suc(x) = \emptyset, \\ \max\limits_{n\in suc(x)} (l_{x,n} + L(n)), & \text{if } suc(x) \neq \emptyset \wedge x \text{ parallel}, \\ \sum\limits_{n\in suc(x)} p_{x,n} \cdot (l_{x,n} + L(n)), & \text{otherwise.} \end{cases} \tag{1}$$

Basically, since the latency of a node following a node n is given by $l_{x,n} + L(n)$, if the node is a parallel gate, its latency is given by the maximum of its continuations, and otherwise by the weighted sum of its continuations. If the node has no successors, its latency is assumed 0.

For throughput we only need to consider services. The global throughput at a node x is computed as the minimum of the throughput of that node and the throughputs of all possible paths from a such a node. By denoting t_x the throughput of a provider currently assigned to node x, the throughput of a node x, denoted $Th(x)$, is defined as follows.

$$Th(x) = \begin{cases} \mathrm{t}_x, & \text{if } suc(x) = \emptyset, \\ \min(\mathrm{t}_x, \min\limits_{n\in suc(x)} Th(n)), & \text{otherwise.} \end{cases} \tag{2}$$

3.4 | Weights and relevance of QoS attributes

It is not always the case that all attributes are equally relevant. If we only consider cost and response time, we may decide, for example, that we are willing to pay a bit more for having a better execution time. In such a case, we may give, for example, a weight of 0.7 to the response time and 0.3 to the cost. These weights indicate that 70% of the decision will be influenced by the response time, and the remaining 30% by the cost. These weights will be taken into account when calculating the objective (or fitness) function.

Assuming that Q is the set of QoS attributes, q_i is the global aggregated value for each attribute i, and that W_i, a value in the range $[0, 1]$, with $\sum_{i\in Q} W_i = 1$, is the weight for attribute i, Equation (3) shows the usual fitness function in this type of problems.

$$f = \sum_{i\in Q_{\max}} (W_i \cdot q_i) + \sum_{i\in Q_{\min}} (W_i \cdot (1 - q_i)), \tag{3}$$

where $Q_{\max}$ and $Q_{\min}$, with $Q = Q_{\max} \cup Q_{\min}$, are the sets of QoS attributes to maximize or minimize, respectively. This fitness function is defined in the range of $[0, 1]$, and our goal is to take the individual (composition) with the highest f value.

3.5 | Constraints

Constraints may be provided to be taken into account in the resolution of the composition. In our work, constraints are established globally, on specific QoS attributes. Without loosing generality, we assume that the syntax used to express constraints on attributes includes three basic operators which are *less than*, *greater than*, and *in range*. For example, a global constraint like $((50 < COST < 1000) \wedge (AVAILABILITY > 0.8))$ means that the global application cost must be between 50€ and 1000€, and the availability must be greater than 80%.

We support both soft and hard constraints:

Soft constraints. Soft constraints have an associated weight that indicates the penalty applied to the total quality of the composition. If a solution does not satisfy it, it is not discarded, but, as we will see below, a penalization is applied. The fitness function f is adjusted depending on the number of failed constraints as shown in Equations (4) and (5), where W_c is the penalty for missing constraints.

$$n_c = 1 - \frac{\text{failedConstraints}}{\max(\text{totalConstraints}, 1)}, \tag{4}$$

$$f_c = (n_c \cdot W_c) + ((1 - W_c) \cdot f). \tag{5}$$

The penalty W_c can be adjusted to make the constraints more or less influential in the search for the objective function.

Hard constraints. Hard constraints allow avoiding undesired solutions, since their satisfaction is mandatory. If any of these constraints is breached, the solution will have the worst possible quality (0), and therefore very likely discarded.

4 | METHODS FOR COMPOSITION

This section presents the four different solutions for service composition, namely the Express method, the (pure) genetic algorithm GA, and two alternative methods based on utilities, U and UM. The pros and cons of each algorithm, as well as the details of operation and implementation, will be presented in each of the corresponding sections.

4.1 | The Express method (Express)

This algorithm is the simplest of the three ones presented in this work. It basically consist in the following procedure:

1. We calculate the maximum and minimum values of each of the QoS attributes of the providers of each component.
2. For each candidate provider of each service, we obtain the sum of each of the values of the different QoS attributes, normalized between the previous maximum and minimum values, and multiply it by its weight. In other words, we locally calculate the fitness function for individual services.
3. Given these values, we then simply take the candidate provider with the highest value for each service.

With these simple steps, we obtain the best provider on a service-by-service basis. Note that in addition to the values for QoS attributes cost, response time, availability and reliability, we calculate the estimates for latency and throughput as explained in Section 3.3.

Limitations. We will see in Section 5 that this algorithm returns good results in very short times. However, the fact that service providers are selected in isolation prevents the selection of optimal solutions in all cases, specifically, it cannot handle correctly parallel-pattern subproblems, global constraints, or attributes that depend on the selection for other services, like latency and throughput. If hard constraints are to be considered, the method may give unacceptable answers.

4.2 | The genetic algorithm

GA provides a very flexible and highly configurable method, which allows us to adapt it to virtually any problem. As we will see in the coming sections, we use our GA solution for service composition as a reference method, since it provides

the more general solution, allowing us to consider simultaneously all QoS attributes, including concrete measures for latency and throughput, constraints and so forth.

The starting point of a GA is a random population, seen as genes, that will improve as new genes are generated. Each allele of the gene encoding is the index of a service and the value that is entered in it will be the service provider for that service. The selection of the best specimens are chosen by measuring them using an objective function. Beginning with a sufficiently varied initial population, by repeatedly applying *mutation*, *crossover* and selection operations will lead to a pseudo-optimal solution.

Limitations. As we explained in Section 1 and will further discuss in Section 5, a GA algorithm may present scalability problems as the search space increases. However, even if the execution time grows as the number of services and the number of providers grow, since the search space grows exponentially, with an appropriate representation and hyper-parameters the complexity of the GA solution may grow polynomially. As we will see in Section 5, in our case, the growth for our experiments is almost linear. This is basically part of the time-precision trade proper of GA-based solutions, where we can stop the search as soon as an acceptable solution is found. In some cases, GA may have problems to leave a local optimal to find a better one. The mutation and recombination steps used, and their parameters, as well as the criteria used to stop the procedure are key to adjust the procedure.

4.3 | A method based on utility (U and UM)

The resolution by utility, as we explained in the introduction, is a heuristics-based method. The utility-based method proposes finding a pseudo-optimal solution, reducing the complexity of the problem to solve, with the goal of reducing the computation time. It does so by creating clusters of service providers in accordance to the utility they offer given the requirements of its service. A GA is then in charge of finding the best assignment. In this way, although a GA is in charge of providing a solution, it does so on a smaller search space, also being able to deal with user-imposed constraints.

The three stages of the method are described in the following subsections.

4.3.1 | Pre-calculation or clusterization

The utility of a service provider is a value assigned to each of the available providers of each component. If a same provider is available for two different components, it may get different utility values for each of them, since it represents how good each particular provider is for a chosen target global goal. In this work we consider two different utilities, one that only depends on the QoS attributes (UM) and another one that depends also on the number of providers of each type (U), with the goal of improving the eventual replaceability. In UM, for each provider, a weighting is made between the degree of a certain quality of component attribute and how many providers reach that degree.

Degree of component quality. For each provider and attribute, the values are grouped in discrete degrees. The degrees are divided in clusters of similar spans between the minimum and maximum values of the corresponding attribute. The ranges of the degree of utility are inherent to each component, depending on its smallest and greatest values. By considering these clusters, the number of considered values is reduced, but this reduction also results in a lost of information. As we will see in the following, identifying the right number of degrees is key for the optimal operation of the method.

Degree of probability. The degree of probability is the number of providers of the particular component that are able to meet a certain range. As the range becomes more challenging, that is, closer to the minimum in case of negative attributes or maximum in case of positive ones, fewer providers will be able to reach that range.

Normalized degree of component quality. The degree of component quality is normalized in the interval (0,1]. The normalized degree of component quality will be used to compare each provider against the rest of the component's providers.

The *normalized degree of component quality* is multiplied by the *degree of probability*, resulting in a table in which each row will be a specific component, with a particular QoS attribute and whose columns will be the grades. This generates a weighting between the number of providers available and how good the value is. Since cluster sizes balanced, the decision to choose a provider does depend both on the value for an individual provider and the number of providers that can reach such a value.

TABLE 2 Codification of the genome.

C_1			...	C_n		
k_1	…	k_m	…	k_1	…	k_m
$ut_{1,1}$	…	$ut_{1,m}$	…	$ut_{n,1}$	…	$ut_{n,m}$

As already explained, the utility method allows the user to specify constraints. Constraints have to be normalized, using the formula described in the previous section. The normalized value of these are considered as the utility required by the user for a certain attribute.

Let us finish this section by saying that, if the providers offer of a component changes, only the tables where the information about that component was stored have to be recalculated.

4.3.2 | GA-based search

The pre-computed values (Section 4.3.1) will be repeatedly used during the GA-search of the best solution. The gene is encoded as follows. As depicted in Table 2, each component takes one index of the gene for each considered QoS attribute. For example, if we have 3 components and 3 QoS attributes, the gene will have length 9. The genome is filled with the indexes of the quality degree matrix degrees.

The utility values are calculated out of the values in the genes representing each candidate solution. Specifically, the utility estimate for the QoS attribute i, denoted Ut_i, is calculated as the average of the alleles of the gene corresponding to such QoS attribute of each component. Going back to our previous example, having 3 components and 3 QoS attributes, for attribute number 1 we take the average of the values in positions 1, 4, and 7 of the genome. The aggregated utility for the QoS attribute i is then just the weighted average of these estimated utilities—see Equation (6), where we assume a genome G, with $ut_{j,i}$ the value for attribute i of component j, and C the set of components.

$$Ut_i = \frac{\sum_{j \in C} ut_{j,i}}{|C|}. \tag{6}$$

To determine whether a given genome satisfies given constraints, the estimation is compared with the normalized value of the constraints. The quality of a given composition is calculated as a fitness function, which will allow the GA to select the best candidates from each population.

Equation (7) is used to calculate such fitness function, where f is 1 if a genome is valid and 0 if the genome utility does not satisfy the indicated constraint, W_i is the weight given by the user to the corresponding QoS attribute, and QoS is the set of QoS attributes. We consider a genome as feasible when all the QoS attributes satisfy the imposed constraint. In case of failure to fulfil any restriction, a penalty will be applied to the fitness. If the fitness is negative, the genome is discarded.

$$\text{penalty}_G = \sum_{i \in \text{QoS}} W_i * f. \tag{7}$$

4.3.3 | Selection of the best provider

Once the GA stops, a composition must be generated out of the best genome. To choose the best composition we will have to go through each of the components and assign them to the provider whose quality value is closest to the referent of that cluster. To do so, we just have to seek the expected utility at the component level. For each service quality attribute and component, the averages of the genome degrees are obtained, resulting in a value known as the expected utility for the component. For the provider assignment, the provider's QoS attributes will be normalized, with respect to the available providers of such a component. The utility of a provider is then calculated as the weighted sum of such values.

Let us consider the following example. Table 3 shows the utility degree matrix of an application with two components, for the attributes of cost (C) and response time (T), and four utility degrees (d_1 … d_4). Assume that the GA has returned [1, 2, 3, 1] as the best genome. This means that for the cost attribute of the first component we must look for a degree

TABLE 3 Utility degree matrix.

		d_1	d_2	d_3	d_4
C_1	C	0.15	0.45	0.75	1.0
	T	0.3	0.55	0.8	1.0
C_2	C	0.15	0.45	0.75	1.0
	T	0.3	0.55	0.8	1.0

of utility 1, for the response time attribute of the first component we must look for a degree of utility 2, and so on. Our objective is to go through the different providers and compare their standardized quality attributes, and choose the one that comes closest to these values. If we look at the table, for cost, the degree of utility 1 corresponds to the representative 0.15 and the degree of utility 2 to the representative 0.45. These providers will make the best composition.

4.3.4 | Adaptation for special attributes and goal

As explained in previous sections, the utility degrees depend on the number of components and the values they can achieve within the component. As explained in Section 3.3, this presents a challenge when dealing with values that depend on two or more components like latency and throughput. As also explained there, we use certain assumptions and heuristics to be able to handle latencies and throughputs in the most convenient way. For latency, following an approach very similar to the one used for utilities, we use a scale between the number of providers and the best values they can reach. In the cases of latency and throughput, it is done by transferring the value of the channel to the component provider itself. As explained in Section 3.3.1, the average of the latencies between a provider and all its potential following providers is calculated.

In the UM case, we do not take into account the number of providers able to satisfy the quality values, that is, the probability table, and therefore, the quality table becomes directly the utility table of this method. This is an optimistic approach, since eliminating the probability table is equivalent to multiplying the quality values by 1, that is, assuming that all providers are equally available. This small modification significantly reduces the pre-computation time, but the reason for considering it is to show that we can consider other utility criteria, and that being more focus on quality, it may show better results when comparing just by looking at it.

Limitations. Utility-based methods promise speed ups at the expense of loosing some precision. As we will see in Section 5, the qualities of the solutions are quite poor, whilst the execution times are not better than for other ones. Nevertheless, this claim requires two clarifications. First, even though the total execution time is not as good as one might expect, it is really competitive if we do not consider the pre-computation time. Note that the results of the pre-computation can be re-used if the application and providers do not change, which may occur in some cases. In fact, this situation may be more common than we think. Consider, for example, a factory or a hospital, where we have a fix number of available resources, and decisions for reconfiguration need to be taken rapidly. On the other hand, note that whilst the UM method uses the fitness as only utility target, U uses both fitness and number of providers. This means that U does not optimize for fitness, which justifies its poor numbers on the fitness values obtained. The UM method helps understanding this, since it uses fitness as only utility. Lastly, note that the performance of the pre-computation depends on the number and type of attributes considered. In our case, the use of latency and throughput requires the analysis of the multiple providers to get the estimations for each of them. Works using classical attributes report lower preprocessing times (see, e.g., works by Mardhuki et al.[13] or Mabrouk et al.[11]).

5 | EXPERIMENTS

This section presents some experimental results to evaluate and compare the different solutions presented in the previous sections. Additional experiments and results, as justification of the selection of the used hyper-parameters is available at the companion web site.[14]

All the experiments shown in this section have been repeated ten times. Here we show both average values and their standard deviations (STDs). In addition to results for the methods presented in the previous sections, we have executed

some of the experiments on a purely-random solution that we have named *RND*. This composition method consists in randomly assigning one of the available providers to each component. This simple method will serve as a reference for the execution time and fitness of the other methods. For the methods requiring pre-computation, in most cases, we will explicitly depict the time required for the pre-computation and the computation of the composition itself. Specifically, in most of the graphs where the execution time is shown, each of them will include a shadow of the same color as its line. The height of this shadow defines the pre-computation time required for each method. That is, the part below the shadow represents the execution time of the algorithm, the height of the shadow represents the pre-computation time, and the opaque line at the top of the shadow is the total execution time of the method. For those methods using GAs, two stopping criteria have been established, namely that they reach 180 s of execution, not counting the pre-computation time, or that a convergent population has been found. Lastly, we have seen in the previous sections that users may specify certain parameters, like attributes' weights (Section 3.4) or constraint penalties (Section 3.5). These parameters are of course relevant for the final results, but not for the experimental comparison presented in this section. As stated in Section 3.2, all solutions use the same measure of quality, that is, they all use the same formula to evaluate a composition. Regarding constraints penalties, they are used in the same way by the GA and U-UM methods.

All details on the dataset, hyper-parameters and additional details on the results can be found in the companion web site.[14]

5.1 | Experiments without user constraints

With the aim of getting a first glance on the scalability of the methods, Figure 3 shows the execution times and overall quality (fitness) for each of the considered methods, for different numbers of services and candidate providers. In these charts, Express represents the Express method (Section 4.1), GA the GA method (Section 4.2), U is the utility-based method targeting both availability and global QoS and UM is the utility-based method targeting only global QoS (see Section 4.3). Applications of up to 2000 services have been considered, with 100 and 1000 candidate providers per service. For example, for the U solution they are denoted (U, 100) and (U, 1000). To run these experiments, the applications' architectures and the QoS values of the candidate providers were randomly generated. The figure depicts charts for both the average execution times and their STD. Since the execution time depends on the complexity of the problem, given by the number of services to be composed, the architectures, and the QoS values, the charts not only show a significant variability, we can observe ups and downs in the charts as the number of services increase. To show more significant results, each experiment was repeated 10 times, the same ones for the different methods, and average values are depicted.

Although we will discuss in some depth the settings and the results from experiments like this one, let us assume for now that the parameters for the different methods are optimized, and let us focus on the main key observations regarding our discussion on scalability. First, observe that these experiments do not consider user's constraints, since they would

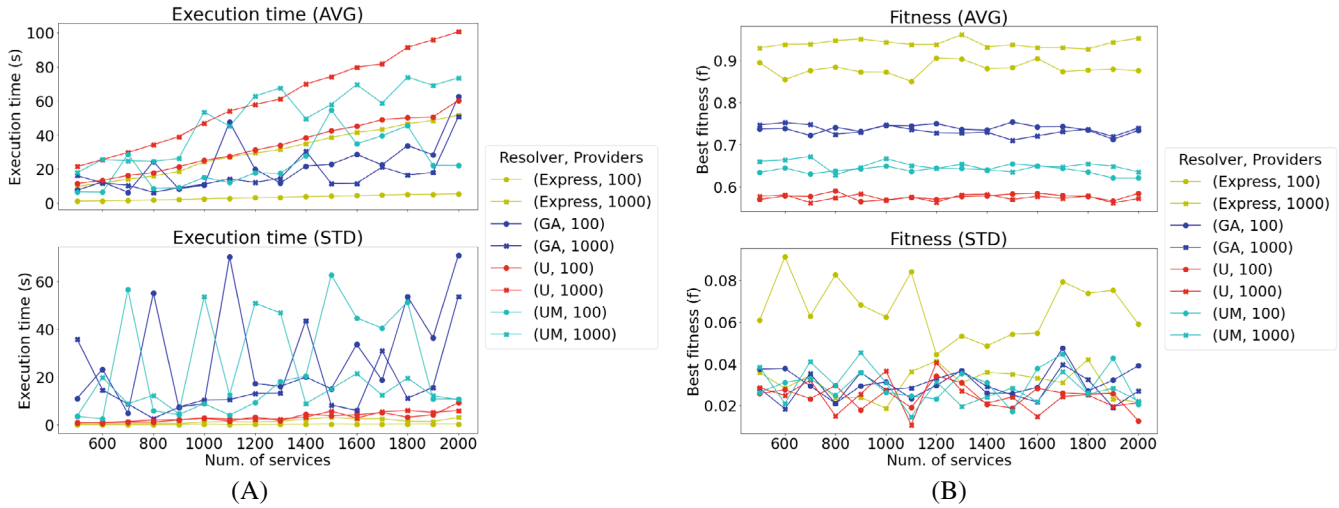

FIGURE 3 Scalability in the different methods. (A) Execution time. (B) Fitness.

rule out some of the methods. The chart at the top of Figure 3B shows the average fitness values for the experiments using the different methods for problems of different sizes. We can observe that the best fitness values are obtained with the Express method, followed by the GA-based method, and with the worst results for the utility-based ones. We can also observe that only for the Express method the number of service providers is significant. The charts in Figure 3A show the average and STD for the execution times for these experiments. Our first observation in this case is that the growth for all of them is more or less linear, with some peaks due to the influence of the application's structure of the genetic-algorithm-based solutions, that is, GA, U, and UM (see STD chart). The Express method provides, as expected, the best execution times. The second best is, despite its variability, GA.

If we had to take a decision on which method to use by looking at these charts, we would clearly take the Express one. However, as pointed out before, there are a few factors to take into account. First, it is not clear whether constraints can correctly be taken into account in the Express method, since there is no way to consider global constraints when deciding locally. Even though, in average, the Express method presents better fitness values, it also presents a greater variability (high STD values). This basically means that even though it gives good results in many problems, there are others in which they are bad: depending on the architecture and the offer of providers, local decisions will be advisable or not.

The times shown in Figure 3 for the utility-based methods include both the preprocessing and the processing times. The figure shows how the combined values grow with the number of service providers. We will see below that the utility-based methods have a much lower decision time, but the preprocessing is significant, and grows with the number of service providers. Indeed, the number of providers is not that relevant for other methods, since having a greater offer seems to simplify the problem of finding an acceptable candidate.

The observations on Figure 3 are reinforced by the charts in Figure 4, which show the execution time and fitness for the composition of 500 services changing the number of service providers up to 1000. These experiments still do not consider user constraints. We can clearly observe that the increase in the number of providers has a somewhat more noticeable impact on the execution time required for methods with a pre-computation or brute force. In terms of fitness, it can be seen that higher values are obtained, but this is only an effect caused by having a wider range of providers to choose from.

There are some interesting observations on Figures 3 and 4 worth pointing out. In global terms, it can be observed that the Express method offers the best fitness quality in a reasonable time, followed by the GA method and the UM method. GA and UM show STDs much higher than the other methods. This is due to their dependency on the architecture of the application. More interesting is to see that although the total execution times for U and UM increase, the times without their pre-computation actually decrease. This also is observed for the GA method, and is due to the increase of the offer: as the number of providers grow, it is easier to find an acceptable solution. The number of alternative providers significantly affects the Express method, which need to choose the best candidate, even locally. The RND method shows a near-zero

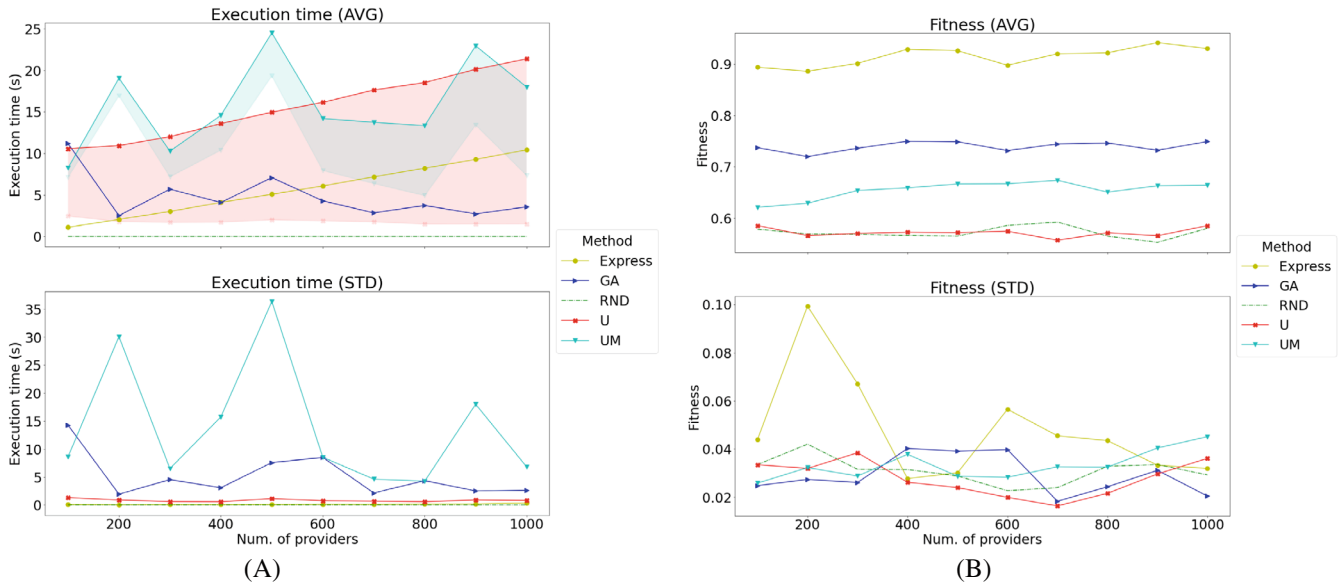

FIGURE 4 (A) Execution times and (B) fitness values for the different methods with 500 services when increasing the number of providers.

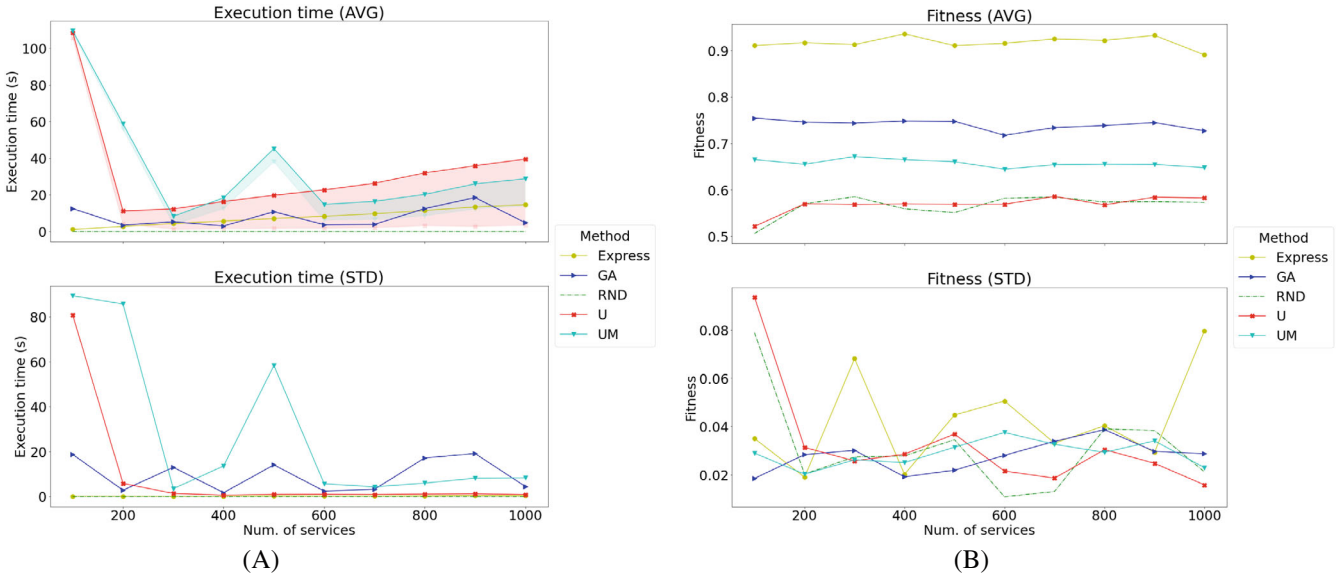

FIGURE 5 (A) Execution times and (B) fitness values for the different methods when increasing the number of services, with 500 providers per service.

execution time. Including the results for U in this chart is a bit unfair, since, as pointed out in Section 4.3, it is not looking for the best fitness. It is interesting to see, however, how its computation time grows globally, but not if we only look at the composition itself and leave out the pre-computation time.

Figure 5 shows a different view of the same experiments. In this case, we compare the methods as the number of services increases with a fixed number of providers per component: 500. Figure 5 shows the same trend as Figure 4, where we can see a linear increase in execution time and quite similar fitness values. In this case we also observe a significant variability for GA and UM. The part due to the pre-computation for the utility-based methods is again quite significant.

5.2 | Experiments with user constraints

We present in this section results for experiments similar to those in the previous section, but considering user constraints. Since the problems are created randomly, we also need a way to generate constraints randomly. A constraint is created for each QoS attribute. We use two different types of constraints, one more restrictive than the other. In both cases, we use a penalty weight of 80%. Please, note that the penalty weight is specified by the user. As explained above, the GA and U-UM methods apply constraints in the same way, and therefore the value does not affect their experimental comparison.

Figure 6 shows the results for the first type of constraints. If it is a positive attribute, the global value of the composition must be greater than 70% of the maximum possible value. If it is a negative attribute, the global value of the composition must "be." less than 130% of the minimum possible value. For example, if the cost values are in the range [100,500], the constraint would be "COST *smaller than* 130." These are the most common constraints, where we impose a minimum or maximum value for a specific attribute to be considered valid.

The results depicted in Figure 6 show that the Express method is significantly better than the others. Note that the most common constraints are somehow redundant with respect to the optimization goals themselves. If we are looking for the best candidate, it most possibly provides a value that satisfies the constraint. Note however that since the rest of the methods consider all candidates equally plausible, they are penalized in the fitness values of the best solutions provided. The execution times are reduced in all cases, since the constraints help discarding solutions very quickly.

In our last experiment we consider a more challenging type of constraint. Figure 7 shows the results for experiments in which the constraints are "maliciously" generated. In this case, for a positive QoS attribute, it must be fulfilled that the overall value of the composition must be less than 70% of the maximum possible value. If it is a negative attribute, the overall value of the composition must be greater than 130% of the value of the minimum possible. For our same example, if the cost values are in the range [100,500], the constraint would be "COST *greater than* 130" Note that this constraint

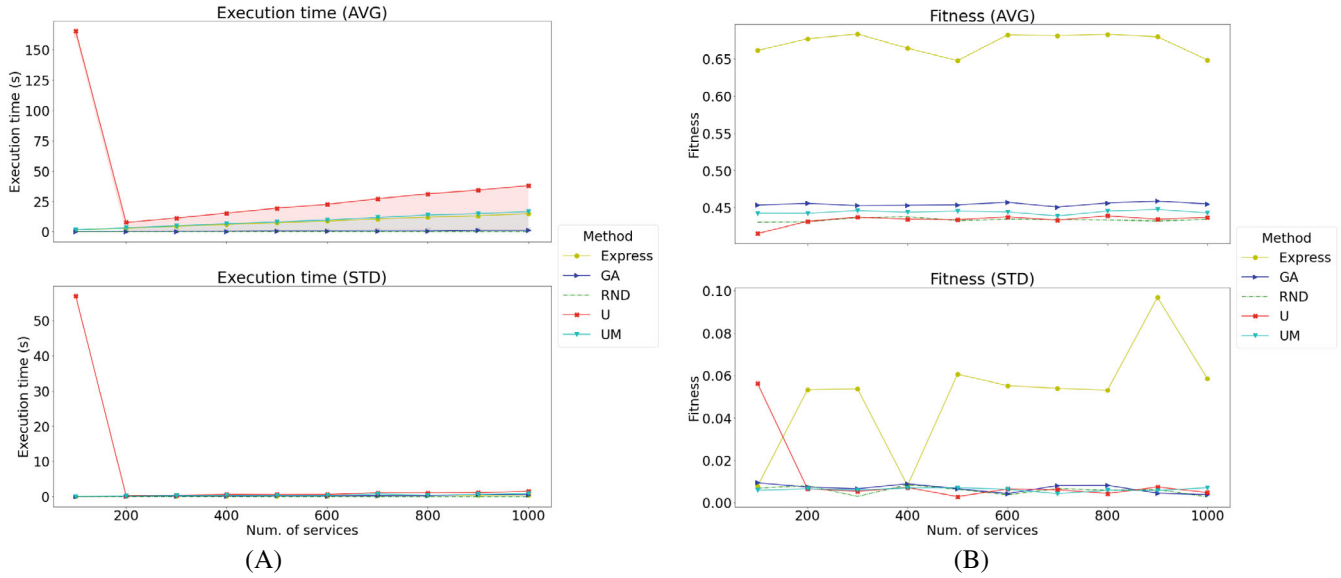

FIGURE 6 (A) Execution times and (B) fitness values for the different methods with common constraints.

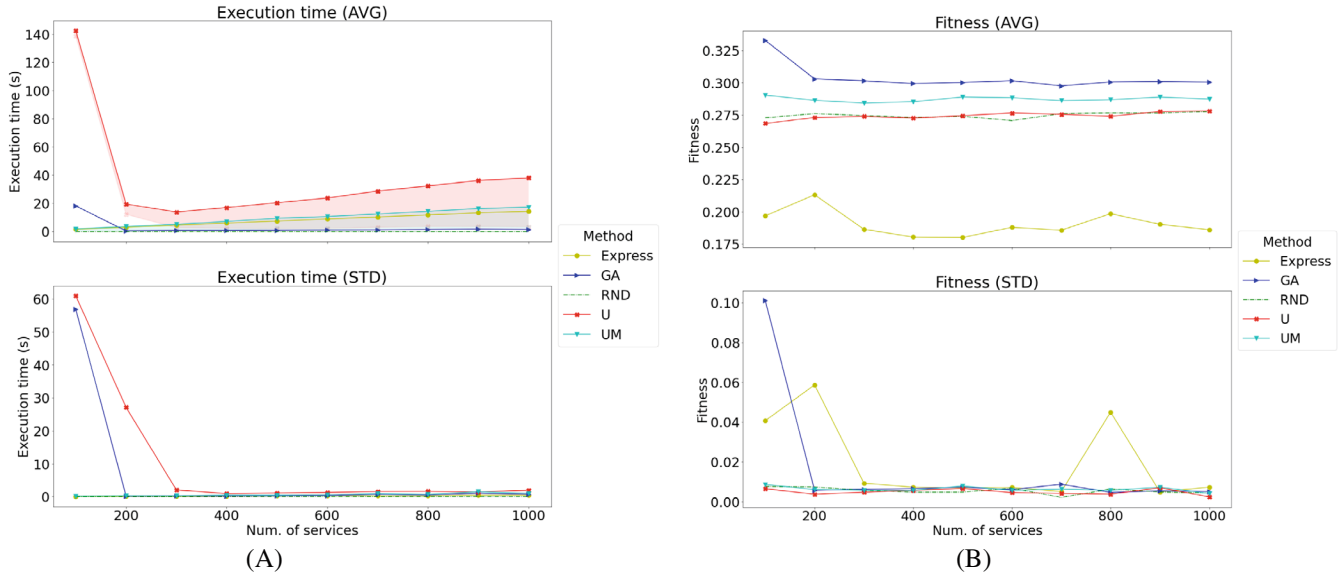

FIGURE 7 (A) Execution times and (B) fitness values for the different methods with uncommon constraints.

forces taking the "wrong" provider. Figure 7 shows similar values for all methods but Express, for which very poor fitness values are obtained.

As we can see, depending on the characteristics of the application, better results are obtained with certain methods. This point will be discussed in Section 5.3. It is also interesting to note that the execution times are mostly influenced by the number of services of the application, and not so much by their providers.

5.3 | Discussion and method selection

The discussion in the previous sections leads us to the conclusion that not all methods work well in all situations, and that there is no method that is better than the others in all cases. In fact, even those methods that show a better behavior

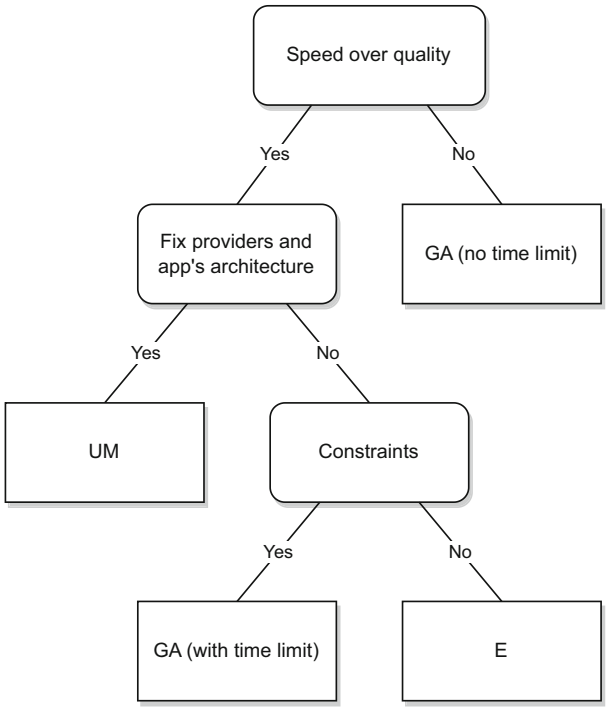

FIGURE 8 Algorithm recommendation.

regarding execution time and fitness values, may still have a significant variability, which may mean that they work better or worst depending on the specific architecture and providers offer. If we want to have a general rule of thumb, applicable in most cases, we would pick two main decision criteria, namely constraints and speed requirements.

Our first observation is that the most important factor to decide is whether we need to take into account constraints, and what type of constraints we have. With no constraints, or with regular constraints, any of the methods may provide acceptable solutions. With special constraints, however, the Express method is to be ruled out. The decision then is basically on how much time there is to make a decision. If the set of providers is fix, and the parameters do not need to be changed, and therefore the preprocessing for utility-based methods can be reused, then the method to choose may be U, UM or Express, depending on the target goal and precision requirements. Of course, other utility targets may be considered for specific situations, we may, for example, need to maximize reliability, or a combination of attributes, in which case specific utility-based solutions may be useful. With no constraints, we can pick a method that works with the components in isolation, since they offer a sufficiently good result in a smaller time. If the problem has (special) constraints, it would be ideal to use methods that have a complete view of the problem to avoid loss of information unless we want a quick solution.

These conclusions are summarized in the decision tree in Figure 8. If our problem has constraints, depending on whether these are special or not (see Section 5.2), we would choose a method that uses GA, either UM, U, or GA. If the constraints are not special, Express and GA offer the best relation execution speed–fitness. If no constraint is involved, the decision is on whether speed or quality is the most-relevant factor. If speed is key, then the Express method is clearly the recommended one. GA, UM, and U can still offer good fitness values, but with no guarantees for all cases. If having a good fitness is important, GA is possibly the one to use.

Let us close this discussion by insisting on the idea that these methods are quite configurable, and that almost any of the methods may get a better result for a specific problem. In the experiments carried out in this work we have chosen different configurations in which an assessment is made between the execution time and the quality of the solution obtained, but the optimal configuration always depends on the problem to be solved and the objective we have in mind. For example, if we are using GA, U, or UM, and our goal is to have a good composition regardless of the execution time, we should configure our method to perform a more exhaustive exploration of the problem by increasing the mutation of the GA. If, on the other hand, our goal is to obtain an acceptable solution in a very short execution time, we will move to an approach of exploiting the best providers found at any given time by decreasing the mutation.

The fact that the structure of applications is randomly generated allow us to analyze the general behavior of the different solutions, also related to their scalability, however, for specific architectures, one could also find specific solutions performing better than those presented here.

6 | CONCLUSIONS

We have presented several alternative methods for service composition, with a focus on scalability, latency, and constraints. Specifically, we have presented methods that exploit the combination of GA, direct methods, and utility-based heuristics. Whilst GA is purely a genetic algorithm, the utility-based methods U and UM are based on the idea of utility, carrying on a preprocessing of the problem before invoking the GA, and a post-processing to transform its output into an assignment of providers.

Our framework covers all usual QoS attributes, including cost, execution time, reliability, availability, latency, and throughput. The managing of the channel-dependent attributes is based on novel proposals, which allow us to handle all attributes in a uniform way, and thus provide support for their use by different methods. Some of these solutions are able to handle constraints (GA, U, and UM), and others do not (Express). The solution of the composition is guided by the so-called fitness function, as a measure of the global aggregated QoS, which is used to quantifying the quality of a solution for a given composition. Implementations for all these solutions, together with an exhaustive set of experiments, are available at the companion web site.[14] Here we have presented a selection of the most relevant experiments. Moreover, a discussion on these results is presented, focusing on their advantages and disadvantages, and bringing light to under what circumstances we would consider each of the proposed methods.

GA seem to give quite good results with proper parameter tuning. Moreover, these algorithms offer a lot of versatility to add new aggregation functions, new features, and new ways of working with new QoS attributes as seen in the case of latency and throughput.

We have seen that in the experiments the utility algorithms in general do not provide the best results, since they are not designed for optimizing fitness. In these algorithms, the reduction of the search space and the probability that the providers can satisfy the imposed values are the priority. This type of algorithm can be especially useful if we are looking for fast replacements between different providers in the same cluster or grouping, or targeting a different global goal, in which case a specific utility is to be defined.

Finally, one method that has really surprised us is the simplest to implement. It consists of taking the best provider for each of the components by looking at them individually. As we have insisted on several times, this method has many limitations, such as working with constraints, with attributes that depend on several components, cannot offer the best composition and so forth. However, it is a simple method, fast, and in most cases, especially without constraints, it offers acceptable solutions, in many cases very good ones, in a very short execution time. This type of solutions may be particularly useful for the migration of applications,[57] where a quick response is critical for the good operation.

Handling attributes that depend on several providers poses interesting challenges in most methods. We have proposed ways to deal with them through heuristics and extra calculations. By working with them more explicitly, we can make better estimates and offer certain features. For example, we may provide a better placing, by locating the maximum number of services in the same provider or in locations close to each other, or with respect to the user. This kind of information and data is especially relevant when working on the continuum.

As future work, there are two main lines of research we plan to develop. The first one is the improvement of the computation for utility-based methods. The use of network models already available in the literature may speed up the preprocessing required for attributes like latency and throughput, but also others that involve the specification of the neighbors of a service for the calculation of the required estimations. The second line consists in the development of an intelligent system that decides, for a given problem, not only the recommended method, but also the hyper-parameters to use. This recommendation system, with the appropriate integration, may allow a management system to deploy and automatically manage complex applications in the best possible way.

There are many other issues that we plan to study in the near future. For example, the model for throughput should be able to take into account the possibility of splitting the inputs. Also, we have observed that the architecture of the applications, the attributes considered, and many other factors also affect the performance of the different methods. We would like to understand these factors, and how this information may be included in a recommendation system.

AUTHOR CONTRIBUTIONS

All authors of the article have made substantial contributions to conception and design of the proposal, have been involved in the development of the algorithms and their implementation, and have participated in the design and implementation of the experiments.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for carefully reading the manuscript. Their comments have been of great help in improving its quality and clarity. This work has been partially supported by Spanish Government projects TED2021-130666B-I00 and PID2021-125527NB-I00. Funding for open access charge: Universidad de Málaga / CBUA.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in Companion web site to LocASSC at http://services-composition-web.s3-website-us-east-1.amazonaws.com/.

ORCID

Francisco Durán https://orcid.org/0000-0001-5864-8094
Ernesto Pimentel https://orcid.org/0000-0002-7125-8434

REFERENCES

1. Wang P, Ding Z, Jiang C, Zhou M, Zheng Y. Automatic web service composition based on uncertainty execution effects. *IEEE Trans Serv Comput*. 2016;9(4):551-565. doi:10.1109/TSC.2015.2412943
2. ISO/IEC. International Standard 19510, Information Technology—Business Process Model and Notation (BPMN): V 2.0; 2013. https://www.omg.org/spec/BPMN/2.0/
3. Barreto C, Bullard V, Erl T, et al. WS-BPEL: web services business process execution language version 2.0. Standard. OASIS; 2007.
4. Crusoe MR, Abeln S, Iosup A, et al. Methods included: standardizing computational reuse and portability with the common workflow language. *Commun ACM*. 2022;65(6):54-63. doi:10.1145/3486897
5. Garijo D, Gil Y, Corcho O. Abstract, link, publish, exploit: an end to end framework for workflow sharing. *Future Gener Comput Syst*. 2017;75:271-283. doi:10.1016/j.future.2017.01.008
6. Strunk A. QoS-aware service composition: a survey. *2010 Eighth IEEE European Conference on Web Services (ECOWS)*. IEEE; 2010:67-74.
7. Zeng L, Benatallah B, Dumas M, Kalagnanam J, Sheng QZ. Quality driven web services composition. *Proceedings of the 12th International Conference on World Wide Web (WWW)*. ACM; 2003:411-421.
8. McCall J. Genetic algorithms for modelling and optimisation. *J Comput Appl Math*. 2005;184(1):205-222. doi:10.1016/j.cam.2004.07.034
9. Sastry K, Goldberg D, Kendall G. *Genetic Algorithms*. Springer; 2005:97-125.
10. Katoch S, Chauhan SS, Kumar V. A review on genetic algorithm: past, present, and future. *Multimed Tools Appl*. 2021;80(5):8091-8126. doi:10.1007/s11042-020-10139-6
11. Ben Mabrouk N, Beauche S, Kuznetsova E, Georgantas N, Issarny V. QoS-aware service composition in dynamic service oriented environments. In: Bacon JM, Cooper BF, eds. *Middleware*. Springer; 2009:123-142.
12. Yuan Y, Zhang W, Zhang X, Zhai H. Dynamic service selection based on adaptive global QoS constraints decomposition. *Symmetry*. 2019;11(3):403. doi:10.3390/sym11030403
13. Mardukhi F, NematBakhsh N, Zamanifar K, Barati A. QoS decomposition for service composition using genetic algorithm. *Appl Soft Comput*. 2013;13(7):3409-3421. doi:10.1016/j.asoc.2012.12.033
14. Pozas García N, Durán F, Moreno Berrocal K, Pimentel E. Companion web site to location-aware scalable service composition; 2023. http://services-composition-web.s3-website-us-east-1.amazonaws.com/
15. Arellanes D, Lau K. Evaluating IoT service composition mechanisms for the scalability of IoT systems. *Future Gener Comput Syst*. 2020;108:827-848. doi:10.1016/j.future.2020.02.073
16. Sotiriadis S, Bessis N, Amza C, Buyya R. Elastic load balancing for dynamic virtual machine reconfiguration based on vertical and horizontal scaling. *IEEE Trans Serv Comput*. 2019;12(2):319-334. doi:10.1109/TSC.2016.2634024
17. Chen I, Guo J, Bao F. Trust management for SOA-based IoT and its application to service composition. *IEEE Trans Serv Comput*. 2016;9(3):482-495. doi:10.1109/TSC.2014.2365797
18. Calinescu R, Grunske L, Kwiatkowska M, Mirandola R, Tamburrelli G. Dynamic QoS management and optimization in service-based systems. *IEEE Trans Softw Eng*. 2011;37(3):387-409. doi:10.1109/TSE.2010.92
19. Ferreira Coutinho E, Carvalho Sousa dFR, Leal Rego PA, Goncalves Gomes D, Souza N. Elasticity in cloud computing: a survey. *Ann Télécommun*. 2015;70(7-8):289-309. doi:10.1007/s12243-014-0450-7
20. Xu Y, Helal A. Scalable cloud-sensor architecture for the Internet of Things. *IEEE Internet Things J*. 2016;3(3):285-298. doi:10.1109/JIOT.2015.2455555
21. Vakili A, Navimipour NJ. Comprehensive and systematic review of the service composition mechanisms in the cloud environments. *J Netw Comput Appl*. 2017;81:24-36. doi:10.1016/j.jnca.2017.01.005

22. Cárdenes Cabré JA, Precup D, Sanz R. Horizontal and vertical self-adaptive cloud controller with reward optimization for resource allocation. *2017 International Conference on Cloud and Autonomic Computing (ICCAC)*. IEEE; 2017:184-185.
23. Razian MR, Fathian M, Bahsoon R, Toosi AN, Buyya R. Service composition in dynamic environments: a systematic review and future directions. *J Syst Softw*. 2022;188:111290. doi:10.1016/j.jss.2022.111290
24. Parejo JA, Segura S, Fernández P, Ruiz-Cortés A. QoS-aware web services composition using GRASP with path relinking. *Expert Syst Appl*. 2014;41:4211-4223. doi:10.1016/j.eswa.2013.12.036
25. Festa P, Resende MG. *Grasp: An Annotated Bibliography*. Springer; 2002:325-367.
26. Siddiqui H, Bajwa IS, Tabassum D, Ramzan S. Producing BPEL models from text. *2016 Sixth International Conference on Innovative Computing Technology (INTECH)*. IEEE; 2016:471-477.
27. Le DN, Nguyen NS, Mous K, Ko RKL, Goh AES. Generating request web services from annotated BPEL. *2009 IEEE-RIVF International Conference on Computing and Communication Technologies: Research, Innovation and Vision for the Future (IEEE-RIVF)*. IEEE; 2009:1-8.
28. Ouyang C, Verbeek E, van der Aalst WM, Breutel S, Dumas M, ter Hofstede AH. Formal semantics and analysis of control flow in WS-BPEL. *Sci Comput Progr*. 2007;67(2):162-198. doi:10.1016/j.scico.2007.03.002
29. Zhang W, Chang C, Feng T, Jiang H. QoS-based dynamic web service composition with ant colony optimization. *34th Annual Computer Software and Applications Conference*. IEEE; 2010:493-502.
30. Yu T, Zhang Y, Lin KJ. Efficient algorithms for web services selection with end-to-end QoS constraints. *ACM Trans Web*. 2007;1(1):6-es. doi:10.1145/1232722.1232728
31. Pozas García N, Durán F. On the scalability of compositions of service-oriented applications. In: Hacid H, Kao O, Mecella M, Moha N, Paik H, eds. *International Conference on Service-Oriented Computing (ICSOC)*. Lecture Notes in Computer Science. Springer; 2021:449-463.
32. Zeng L, Benatallah B, Ngu A, Dumas M, Kalagnanam J, Chang H. QoS-aware middleware for web services composition. *IEEE Trans Softw Eng*. 2004;30(5):311-327. doi:10.1109/TSE.2004.11
33. Berbner R, Spahn M, Repp N, Heckmann O, Steinmetz R. Heuristics for QoS-aware web service composition. *2006 IEEE International Conference on Web Services (ICWS'06)*. IEEE; 2006:72-82.
34. Yu T, Lin KJ. Service selection algorithms for composing complex services with multiple QoS constraints. In: Benatallah B, Casati F, Traverso P, eds. *International Conference on Service-Oriented Computing (ICSOC)*. Lecture Notes in Computer Science. Springer; 2005:130-143.
35. Mirjalili S. *Genetic Algorithm*. Springer; 2019:43-55.
36. Canfora G, Di Penta M, Esposito R, Villani ML. An approach for QoS-aware service composition based on genetic algorithms. *Conference on Genetic and Evolutionary Computation (GECCO)*. ACM; 2005:1069-1075.
37. Cardoso J, Sheth A, Miller J, Arnold J, Kochut K. Quality of service for workflows and web service processes. *J Web Semant*. 2004;1(3):281-308. doi:10.1016/j.websem.2004.03.001
38. Fan S, Peng K, Yang Y. Large-scale QoS-aware service composition integrating chained dynamic programming and hybrid pruning. In: Jin H, Wang Q, Zhang LJ, eds. *International Conference on Web Services (ICWS)*. Lecture Notes in Computer Science. Vol 10966. Springer; 2018:196-211.
39. Fan S, Ding F, Guo C, Yang Y. Supervised web service composition integrating multi-objective QoS optimization and service quantity minimization. In: Jin H, Wang Q, Zhang LJ, eds. *International Conference on Web Services (ICWS)*. Lecture Notes in Computer Science. Vol 10966. Springer; 2018:215-230.
40. Wada H, Suzuki J, Yamano Y, Oba K. E^3: a multiobjective optimization framework for SLA-aware service composition. *IEEE Trans Serv Comput*. 2012;5(3):358-372. doi:10.1109/TSC.2011.6
41. Suciu M, Pallez D, Cremene M, Dumitrescu D. Adaptive MOEA/D for QoS-based web service composition. In: Middendorf M, Blum C, eds. *Evolutionary Computation in Combinatorial Optimization*. Springer; 2013:73-84.
42. Moustafa A, Zhang M. Multi-objective service composition using reinforcement learning. In: Basu S, Pautasso C, Zhang L, Fu X, eds. *Service-Oriented Computing*. Springer; 2013:298-312.
43. Trummer I, Faltings B, Binder W. Multi-objective quality-driven service selection—a fully polynomial time approximation scheme. *IEEE Trans Softw Eng*. 2014;40(2):167-191. doi:10.1109/TSE.2013.61
44. Yu Y, Ma H, Zhang M. F-MOGP: a novel many-objective evolutionary approach to QoS-aware data intensive web service composition. *Congress on Evolutionary Computation (CEC)*. IEEE; 2015:2843-2850.
45. Ramírez A, Parejo JA, Romero JR, Segura S, Ruiz-Cortés A. Evolutionary composition of QoS-aware web services: a many-objective perspective. *Expert Syst Appl*. 2017;72:357-370. doi:10.1016/j.eswa.2016.10.047
46. Deshpande N, Sharma N. Composition algorithm adaptation in service oriented systems. In: Muccini H, Avgeriou P, Buhnova B, et al., eds. *Software Architecture*. Springer; 2020:170-179.
47. Alrifai M, Risse T. Combining global optimization with local selection for efficient QoS-Aware service composition. *Proceedings of the 18th International Conference on World Wide Web*. ACM; 2009:881-890.
48. Klein A, Ishikawa F, Honiden S. Towards network-aware service composition in the cloud. *Proceedings of the 21st International Conference on World Wide Web*. ACM; 2012:959-968.
49. Klein A, Ishikawa F, Honiden S. SanGA: a self-adaptive network-aware approach to service composition. *IEEE Trans Serv Comput*. 2014;7(3):452-464. doi:10.1109/TSC.2013.2
50. Martini B, Paganelli F, Cappanera P, Turchi S, Castoldi P. Latency-aware composition of virtual functions in 5G. *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*. IEEE; 2015:1-6.

51. Wu J, Wu Z. Similarity-based web service matchmaking. *2005 IEEE International Conference on Services Computing (SCC'05)*. IEEE; 2005:287-294.
52. Stefanic P, Kochovski P, Rana OF, Stankovski V. Quality of service-aware matchmaking for adaptive microservice-based applications. *Concurr Comput*. 2021;33(19):e6120. doi:10.1002/cpe.6120
53. Kritikos K, Plexousakis D. Mixed-integer programming for QoS-based web service matchmaking. *IEEE Trans Serv Comput*. 2009;2(2):122-139. doi:10.1109/TSC.2009.10
54. Poletti F, Wheeler NV, Petrovich MN, et al. Towards high-capacity fibre-optic communications at the speed of light in vacuum. *Nat Photonics*. 2013;7(4):279-284. doi:10.1038/nphoton.2013.45
55. Dabek F, Cox R, Kaashoek MF, Morris RT. Vivaldi: a decentralized network coordinate system. *ACM SIGCOMM 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. ACM; 2004:15-26.
56. Wan C, Ullrich C, Chen L, Huang R, Luo J, Shi Z. On solving QoS-aware service selection problem with service composition. *2008 Seventh International Conference on Grid and Cooperative Computing*. IEEE; 2008:467-474.
57. Carrasco J, Durán F, Pimentel E. Live migration of trans-cloud applications. *Comput Stand Interfaces*. 2020;69:103392. doi:10.1016/j.csi.2019.103392

How to cite this article: Pozas García N, Durán F, Moreno Berrocal K, Pimentel E. Location-aware scalable service composition. *Softw Pract Exper*. 2023;53(12):2408-2429. doi: 10.1002/spe.3260