Tailored Homogeneous Service Composition at Runtime to Enhance User-Perceived Performance

Zhengquan Li, Zheng Song
Department of Computer and Information Science, University of Michigan at Dearborn
Email: {zqli, zhesong}@umich.edu

***Abstract*—Web services are widely used in modern software, providing diverse data and functionalities. Some data and functionalities are critical to an application's execution and user experience, posing strict requirements on the Quality of Service (QoS) of their delivery (e.g., latency and reliability), which services often fail to meet. Previous studies show that composing homogeneous services, i.e., simultaneously invoking multiple services providing the same functionalities and returning the first response, can improve latency and reliability. However, this approach increases the workloads on cloud servers and causes additional network traffic, limiting its deployment at scale. Our empirical study reveals that services deliver varying QoS across different locations, making it possible to reduce the invocation cost by tailoring the composition strategy for different clients. In this paper, we introduce an approach that composes homogeneous services dynamically for each client, improving user-perceived QoS while minimizing the invocation costs. In particular, our approach first probes the QoS of all homogeneous services for a client, and then calculates an optimal composition strategy that satisfies the QoS requirements specified by App developers with minimum cost. We prototyped our approach as an Android library and tested it via both real-world experiments and simulations. The evaluation results show that our approach significantly improves QoS compared to invoking a single service with average best QoS across all locations (enhancing reliability to 100%, reducing average latency by 7% and tail latency by 35%) while incurring 50% less cost than static homogeneous composition, making it a useful tool for service-oriented applications.**

***Index Terms*—Quality of Service, Service Composition, Runtime System**

I. Introduction

Web services are widely used in modern software systems, providing diverse data and functionalities that are essential to many applications. Some of these services are critical to an application's execution and user experience. For example, applications that use interactive services—such as web search, financial trading, gaming, and social networks—rely on consistently low response times to attract and retain users [1]. When multiple functionally equivalent services are available, developers of these service-oriented applications face the problem of meeting the stringent QoS requirements of their applications, such as low latency, low tail latency, and high reliability.

The current state-of-the-practice approach used by application developers is *average-optimal service selection* (see Fig.1, top left), which selects and hardcodes the service with the best average performance among all functionally equivalent options into the application [2], [3]. However, according to service statistics collected by marketplaces such as RapidAPI [4], and an empirical study of developer comments on web service usage [5], these average-optimal services do not always deliver the best QoS performance in practice. To further improve QoS, previous works [6]–[8] have explored the *static composition of homogeneous services*, where developers hardcode multiple equivalent services into the application, and invoke all of them simultaneously at runtime, using the first response to proceed with the application's execution (Fig. 1, bottom left). While effective in enhancing QoS, this method—due to its strategy of simply invoking all specified services—significantly increases the number of service invocations, leading to higher workloads on all web servers and increased network traffic. The additional invocation cost scales with the number of available equivalent services, making this approach impractical for large-scale deployment.

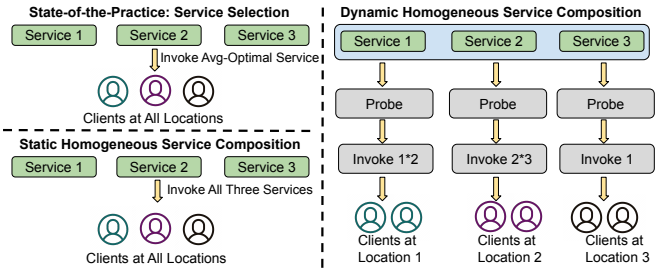


Fig. 1: Service Invocation Approaches for Improving User-Perceived Performance

By conducting an empirical study that involved invoking services from five global locations, we observed that most services exhibit significantly different QoS at various locations. Inspired by this observation, this paper introduces *a more cost-efficient homogeneous service composition approach, which dynamically customizes service composition for end users at runtime.* As demonstrated by the right sub-figure in Fig. 1, for clients at different locations, our approach first probes the QoS of all homogeneous services, then computes a client-specific composition strategy that determines which subset of services to invoke. This strategy aims to meet the developer's QoS requirements (e.g., latency and reliability) within a predefined invocation cost budget. By only invoking the subset services, our dynamic approach reduces the cost while preserving QoS performance compared to the static approach.

The main contributions of our paper are as below:

- We conducted a large-scale study of web service QoS by invoking services at five locations worldwide. Our study

confirmed that 1) Service QoS significantly varies by location, resulting in some users receiving unsatisfactory QoS, and 2) composing homogeneous service statically improves QoS, but causes unnecessary invocation costs, which motivates this work.
- We introduced an approach that customizes service composition strategies for individual clients for both QoS improvement and cost efficiency.
- We proposed a QoS estimation model for composition strategies, which ensures selecting a QoS-optimal strategy with less probing cost.
- We prototyped our approach and evaluated it with testbed experiments and trace-based simulations. The results indicate that 1) our QoS estimation model is more accurate than all baseline approaches; 2) on average, our approach improves reliability from 99% to 100%, reduces mean latency and tail latency by 7% and 35% respectively, while incurring 50% less cost than static homogeneous service composition.

The rest of this article is organized as follows: Section II discusses related work; Section III presents an empirical study identifying the challenges and opportunities in web service QoS; Section IV, and Section V explain the design and evaluation our approach, respectively; Section VII concludes the article. This article is a revised and extended version of a conference paper published in ICSOC'24 [9]. In particular, while the conference paper introduced the theoretical QoS estimation model for dynamical homogeneous composition, this article extends it by: a) presenting a large-scale empirical study on web service QoS to further motivate the problem and our solution; b) introducing a runtime system with an intuitive programming interface to facilitate adoption by application developers; and c) providing a more comprehensive evaluation of QoS improvements and system overhead.

II. Related Work

This section summarizes the existing works on the empirical measurement of web service QoS and the approaches used to optimize the service QoS.

A. Service QoS measurement Studies

The most recent large-scale web service QoS measurement was conducted 10 years ago [10], which investigated the failure probability, response time, and throughput of real-world web services. The study revealed that the performance of a service varied significantly across different invocation contexts, such as locations and times. The past ten years have witnessed the rapid growth of content distribution network (CDN) [11], which brings services closer to users. In this case, there is a lack of an updated and comprehensive investigation on the QoS of web services, which motivated our study. The blooming usage of RapidAPI also provides data insights into the QoS of services.

B. Service QoS Optimization

Web services face the problem of guaranteeing QoS for distributed end users as many applications require the service to consistently deliver fast, reliable performance for better user experience, thereby keeping their dominance among their competitors. The approaches to enhance the service QoS can be mainly categorized as server-side and client-side.

1) Server-side Approaches: The service providers strive to improve their service QoS in multiple ways: 1) by deploying their service on more CDNs [11] or edge servers [12], [13] that are closer to users to reduce network latency and failures caused by network congestion; 2) provisioning more computational resources on the backend, such as increasing the number of replicas or virtual machines to improve concurrency and reduce processing latency. These optimizations have proven to improve user-perceived performance in the past and remain the standard practice in commercial cloud and edge platforms [13]–[15].

However, such server-side optimizations have gradually reached a performance bottleneck. Once servers are sufficiently distributed and provisioned, further improvements bring only marginal latency gains. The remaining delays are dominated by last-hop conditions—for example, WiFi interference, local routing congestion, and device processing overhead—that are outside the visibility and control of the service provider. Moreover, the internal operation of the backend infrastructure is entirely opaque to application developers: they cannot determine which replica will serve a given request, nor influence how requests are scheduled or routed. Developers interact with these services only through published APIs, receiving aggregated QoS metrics such as average latency and reliability.

Consequently, while server-side methods enhance global efficiency, they cannot guarantee consistent quality for individual users. This limitation motivates the exploration of client-side approaches, where decision-making shifts from the service providers to the application developers, enabling greater flexibility in how services are utilized at the application level. These approaches are introduced below.

2) Client-side Approaches: When developing service-oriented applications, the developers face multiple equivalent services and need to carefully use them so that application users receive a better user experience.

QoS-based Service Selection. Given a set of functionally equivalent services, developers can select among them based on various criteria, ranging from non-QoS factors such as the monetary cost of a single invocation and the quality of usage documentation to QoS-based metrics like reliability and latency [3]. In practice, developers typically choose the service with the best average QoS and hard-code it into the application [2], [3], as service providers generally publish QoS metrics as average values over all accepted historical invocations. For instance, if an application prioritizes low response time, the service with the lowest average latency is selected; if reliability is more important, the one with the highest average success rate is selected. This average-optimal approach assumes that the selected service will consistently deliver the best performance for the application [16]. However, as our empirical study shows, this assumption often breaks down, as some services may lack server deployments in certain regions, despite offering the best performance in most others.

Static Homogeneous Service Composition. Homogeneous service composition refers to invoking multiple functionally equivalent services simultaneously and using either one or multiple responses to fulfill the application's functionality, depending on the QoS metric the developer aims to optimize. This technique has been explored in various domains to enhance QoS.

For example, in the IoT domain, equivalent services are commonly available. A single functionality—such as detecting the presence of fire—can be implemented in multiple ways: (a) reading a temperature sensor to check if the indoor temperature exceeds a threshold; (b) reading data from a smoke sensor; or (c) applying image processing to camera footage. Since timely fire detection is critical for human safety, some studies [6], [8] propose invoking multiple equivalent implementations in parallel and using the first response to reduce detection time (i.e., speculative parallelism). In the web services domain, Bhatia et al. [7] invoke multiple cognitive web services and fuse their results to improve data-related QoS. Specifically, they invoke multiple face detection services, wait for all responses, and then select the result agreed upon by most services as the final output. This majority-voting mechanism improves the user-perceived accuracy and trustworthiness of the service.

Despite their benefits, these approaches simply invoke all services specified by static configurations at the application development stage, which lack flexibility and can lead to significant operational costs at the runtime stage. In contrast, our approach—dynamic homogeneous service composition—dynamically and strategically determines which subset of the specified equivalent services to invoke at runtime, aiming to maximize QoS improvement while minimizing invocation cost.

III. Motivation and Empirical Study

The motivation for dynamic homogeneous service composition arises from our extensive analysis of service statistics gathered from RapidAPI and the real-world QoS performance of service invocations. RapidAPI, the world's largest API hub, serves nearly three million developers with access to tens of thousands of APIs. Specifically, we first identified the widespread challenges of unmet QoS in web services. We then highlighted the opportunity for dynamically composing homogeneous services as a cost-effective approach to enhancing QoS for performance-critical tasks.

A. Service QoS Study

We begin by examining the service QoS statistics provided by RapidAPI. However, due to the lack of transparency in how RapidAPI collects and maintains this data, there is a possibility that the reported QoS metrics are based on outdated invocation history, which may not accurately reflect current service performance. To address this concern, we conducted our own empirical study by invoking the services and collecting live QoS measurements.

QoS Statistics of Services Measured by RapidAPI RapidAPI delegates end users' requests to services. It also measures the latency and successful rates of invoking services, and displays the statistical data on each service's web page. We developed a Python-based crawler and downloaded the web pages of all 5,784 active web services that provide QoS statistics. From each web page, the crawler obtained the statistic data at the service level (i.e., reliability) and average latency. We found 2862 services with statistical QoS data displayed in their web pages. Fig. 2 shows their latency and reliability distributions, where only 1627 services have an average latency of less than 1,000 ms and only 1,416 services have 99% or better reliability (i.e., the percentage of successful calls made to the service) [1].

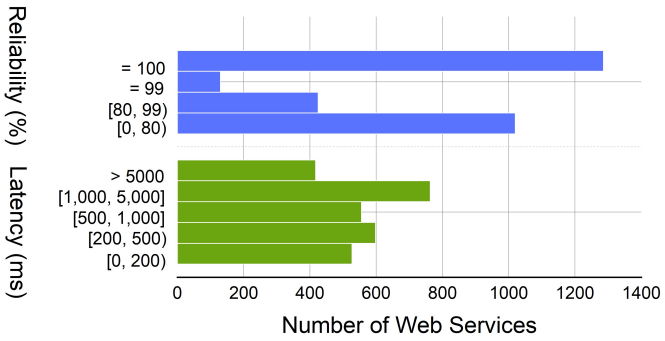


Fig. 2: Services QoS Distribution on RapidAPI

Empirical QoS Measurement The most recent large-scale web service QoS measurement, conducted 10 years ago [10], revealed significant performance variation across different invocation contexts like locations and times. Since then, the use of content distribution networks (CDNs) [11] has increased to reduce latency by bringing services closer to users. However, it is still unclear if CDNs consistently ensure service reliability and minimize end-to-end delays, which are crucial for a good user experience.

Methodology: Our study evaluated six types of tasks: weather forecasting, IP-to-location, face detection, language translation, flight data retrieval, and hotel data retrieval, to cover typical service usage scenarios [17]–[19]. For each task, we selected three homogeneous services, giving priority to those with lower subscription costs and higher popularity. We developed a Python program to invoke these services every 40 seconds from five global locations—Frankfurt, Tokyo, Sydney, Mumbai, and Michigan—over three continuous days. This process collected approximately 4,000 samples per service set, totaling 90 service invocation trace sets.

Results: Table I summarizes the average latency and tail latency (i.e., 95^{th} percentile latency) of all six sets of homogeneous services among five locations. We observe that 1) Among the 90 instances, many services exhibited both average and tail latencies in the range of several thousand milliseconds at certain locations. 2) On average, the tail latency was nearly twice as high as the corresponding average latency, reaching approximately 1,040 ms. Table II demonstrates the average reliability for all functionalities across all locations. We also found that most services exhibit unreliable service delivery,

[1] https://docs.rapidapi.com/docs/faqs

Task	Service	Germany		Japan		Australia		India		US	
		Avg. Lat.	Tail Lat.	Avg. Lat.	Tail Lat.	Avg. Lat.	Tail Lat.	Avg. Lat.	Tail Lat.	Avg. Lat.	Tail Lat.
Face Detection	inferdo	1574ms	1769ms	1735ms	1937ms	1623ms	2030ms	2081ms	2291ms	1761ms	1924ms
	microsoftFace	827ms	1252ms	1152ms	1573ms	1377ms	1851ms	1554ms	1820ms	1065ms	1365ms
	SmartClick	954ms	1314ms	1610ms	1975ms	1674ms	2003ms	1468ms	1900ms	1517ms	1841ms
IP to Location	IP_GEO	22ms	40ms	25ms	37ms	22ms	37ms	531ms	562ms	116ms	295ms
	IP_lookup	35ms	56ms	39ms	57ms	37ms	53ms	43ms	79ms	182ms	373ms
	IPGtolocation	651ms	907ms	636ms	796ms	803ms	989ms	934ms	1241ms	533ms	896ms
Weather Forecast	openWeatherMap	62ms	123ms	297ms	467ms	365ms	576ms	231ms	312ms	222ms	610ms
	Visual Crossing	397ms	439ms	716ms	783ms	904ms	1016ms	828ms	3773ms	296ms	797ms
	weatherbitio	374ms	419ms	705ms	753ms	902ms	951ms	808ms	856ms	248ms	575ms
Translation	Lecto Trans.	343ms	543ms	486ms	845ms	683ms	1028ms	590ms	873ms	350ms	538ms
	NLP Trans.	406ms	461ms	699ms	802ms	866ms	955ms	1166ms	1260ms	193ms	253ms
	Text Trans.	538ms	608ms	772ms	878ms	959ms	1074ms	1193ms	1363ms	289ms	359ms
Flight Info.	FlightRadar	3520ms	4088ms	4084ms	4634ms	2468ms	2962ms	6213ms	7551ms	2383ms	3090ms
	FlyTrips	309ms	375ms	1374ms	1491ms	1679ms	1765ms	1484ms	1632ms	823ms	1163ms
	TravelAdvisor	1128ms	2471ms	1172ms	2825ms	1312ms	3476ms	1580ms	3161ms	899ms	2655ms
Hotel Info.	Booking.com	238ms	769ms	652ms	1292ms	714ms	952ms	749ms	1149ms	635ms	1625ms
	Hotels	968ms	4284ms	1102ms	4635ms	1224ms	5810ms	1628ms	5242ms	887ms	4974ms
	Priceline	1925ms	3615ms	2333ms	3313ms	2249ms	3438ms	2482ms	4090ms	1983ms	3197ms
Avg. Performance	—	**792ms**	**1307ms**	**1088ms**	**1616ms**	**1103ms**	**1720ms**	**1420ms**	**2175ms**	**799ms**	**1473ms**

TABLE I: The Latency Performance for All Six Sets of Homogeneous Services at Five Locations Worldwide

	FaceDet.	IP2Loc.	Weather	Trans.	Flight	Hotel
Service1	99.98	100	100	99.88	100	100
Service2	98.94	99.99	99.99	99.99	100	100
Service3	99.97	99.99	99.99	99.95	100	99.43
Avg.	99.63	99.99	99.99	99.94	100	99.81

TABLE II: The Average Reliability (%) for Homogeneous Services, with Service 1, 2, 3 Representing Different Services for Each Task Type.

with an average reliability for all services below three nines (99.9%). The services for face detection and translation are the least reliable among all six sets of services. The reason for this could be 1) the cloud backends of these cognitive services are more complex, and 2) a larger volume of data transmission (e.g., sending images for face detection) could also cause service failure.

Overall, service QoS—both latency and reliability—is highly variable and often insufficient, especially given that different applications have diverse QoS requirements. Improving user-perceived QoS is critical for many applications, as even delays of a couple of hundred milliseconds can significantly impact revenue [20], [21]. For example, Amazon reports that every 100ms of additional latency results in a 1% drop in sales [22], while the TABB Group estimates that a broker could lose up to $4 million in revenue per millisecond if its electronic trading platform is just 5ms slower than a competitor's [23]. Human-computer interaction studies similarly demonstrate that users are sensitive to small differences in operation delays [24]. Therefore, achieving better QoS has become a widely recognized and essential goal.

B. Observations and Opportunities

We further analyzed the measurement results and drew a few deeper observations that motivated our approach to composing homogeneous services for individual users to improve user-perceived QoS.

Observations: We first investigated the QoS performance of average-optimal services across different locations. For each task, given three services, the average-optimal service was selected as the one with the best average latency across all five locations. Our results show that average-optimal services do not consistently outperform their peers for most tasks. For example, in language translation, `Lecto Trans.` had the best latency in four locations but not in Michigan, USA.

Observation 1

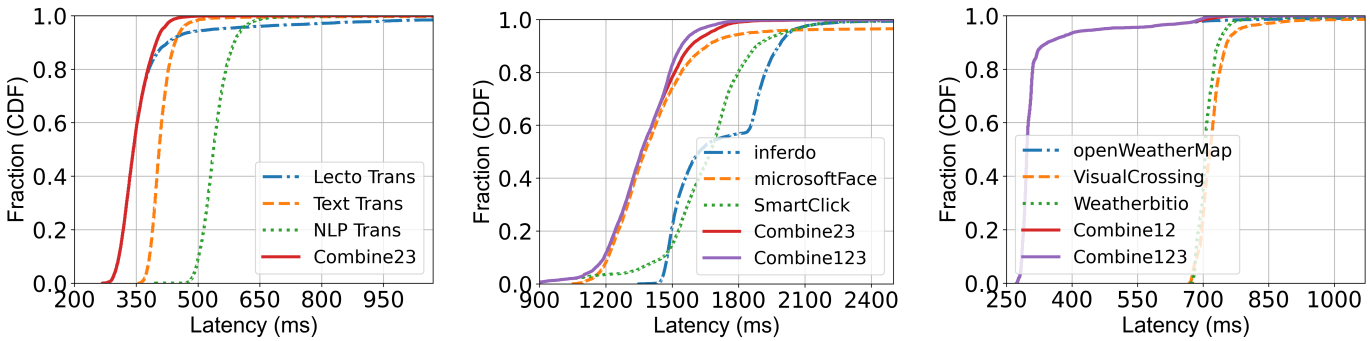
Due to the high variation in service QoS, average-optimal services could not consistently outperform their equivalent peers in some locations, risking user experience.

We then examined the efficiency of the current static homogeneous service composition. Using the collected service invocation traces, we showed the latency improvement achieved by invoking different services. For example, combining two services reduces average latency by 10% and 95th percentile latency by 24%, as shown in Fig.3a. Further adding a third service reduces tail latency more, as per Fig.3b. However, Fig.3c shows combining all three services fails to further reduce the latency compared to using two services, which means that simply invoking all services does not always guarantee better QoS while incurring higher cost.

Observation 2

Static homogeneous service composition may incur unnecessary costs.

To find the efficient service combination for each user, it requires accurate estimation of the resulting QoS for each combination. Existing methods [25], [26] use the minimum average latency among services as the composited latency, which is inaccurate: We observed that the average latency of a speculative parallel invocation can be even lower, as the fastest service may sometimes experience long tail latency, and the slower service might return the result sooner.



(a) Translation, Germany (b) Weather, Japan (c) FaceDet, Australia

Fig. 3: Latency of Combined Invocations of Homogeneous Services

Observation 3

Existing QoS estimation model for compositions is inadequate.

To better satisfy the QoS requirements, there is a pressing need to dynamically customize the service composition for individual users. To realize this approach, it demands a more accurate QoS estimation model.

IV. HomoService: Client-specific Service Composition

This section introduces our approach to composing homogeneous services for individual clients. To differentiate it from traditional web services, we refer to the composed service as "HomoService." We adopt the syntax for homogeneous service composition as described in [6], where $a * b$ denotes the workflow pattern of invoking two services, a and b, in parallel and using the first returned result.

A. Design Challenges

To the best of our knowledge, our work is the first attempt at client-specific homogeneous service composition. Our approach addresses these challenges:

- How can we support HomoServices generically for dissimilar application scenarios? We design a system workflow that requires no additional network infrastructure support and adapts to the fluctuations of service QoS (Sec. IV-B).
- How can developers build HomoServices considering their vastly dissimilar QoS requirements, such as hard/-soft requirements on latency, reliability, and cost? We design programming models that are both intuitive to use and flexible, ensuring high expressiveness (Sec. IV-C).
- How can we maximize the QoS improvements with minimum overhead? We establish a model that more accurately estimates the QoS of a composition with fewer probing requests (Sec. IV-D).

B. System Workflow

Traditional (heterogeneous) service composition [27] typically involves a service gateway that dynamically decides which services to invoke based on a client's QoS requirements and the services' historical QoS. Adapting this approach to support dynamic homogeneous service composition necessitates deploying distributed gateways in a fine-grained manner, which incurs additional costs. In contrast, our approach requires only modifying the service invocation workflow at the clients, eliminating the need for additional network infrastructure support.

Fig.4 shows our system workflow design, which includes four steps:

1) **Probing**: Before generating an optimal composition strategy, a client needs to probe the QoS of all homogeneous services several times. During the probing stage, all services are invoked simultaneously. The first returned result is used to continue the application's execution, while the QoS of the others (e.g., latency and reliability) is recorded.
2) **Calculating** the Optimal Strategy: After obtaining sufficient QoS samples, the client calculates an optimal strategy for future invocations. This strategy specifies a selected subset of homogeneous services.
3) **Invoking:** Following the strategy, the client invokes the selected services in parallel, uses the first returned result to continue the application's execution, and sends the QoS data to the monitoring module.
4) **Monitoring:** The monitoring module records the historical QoS of HomoService invocations. If there is a significant change in QoS, it disables the previously generated optimal strategy and triggers probing again.

C. Programming Model

Our programming model allows developers to intuitively build a HomoService, flexibly specify QoS requirements, and declaratively define when to probe.

1) Building a HomoService: Fig. 5 demonstrates how to compose a translation HomoService to convert English text "hello world" into French using our programming interface. Line 2 reads the XML configuration files for each homogeneous service, and Line 4 invokes the HomoService. The XML file specifies a service's endpoint URL, cost per request, and mappings from user inputs to URL parameters and service

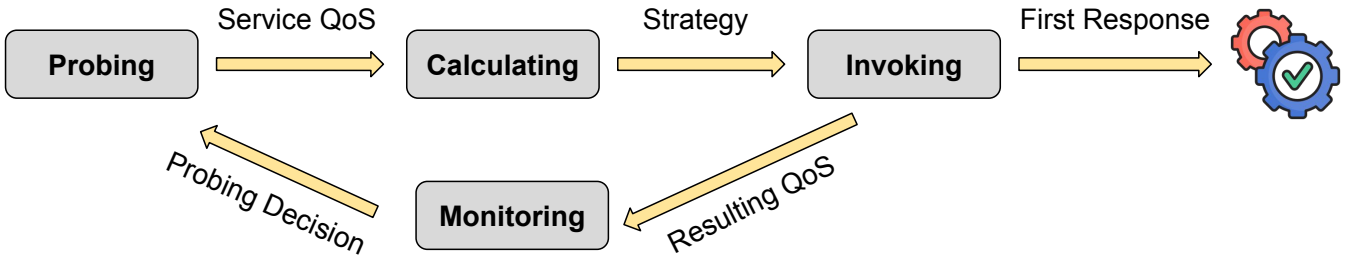


Fig. 4: The System Workflow of Our Approach

outputs. For example, the configuration for the NLP Translation service (Fig. 6) details the service name (`NLPTrans`), endpoint URL ('http://nlp-trans.com/v1'), and cost ($0.0002 per request). The file sets the HTTP method to `GET` and maps input parameters for the text (`text`), source (`from`), and target languages (`to`) into the service URL. The URL for a request would thus be: http://nlp-trans.com/v1?text=hello%20world&from=en&to=fr. The `responseMapping` section instructs on how to parse the service's JSON response to extract the translated text.

```
1 //Creating Homogeneous Services
2 HomoService Trans = new HomoService("lectotrans",
      "nlptrans", "texttrans");
3 //Invoking HomoService to translate ``hello world''
      into French
4 String result = Trans.invoke("hello world", "en",
      "fr");
```

Fig. 5: Composing a Translation Service

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <serviceConfig>
3      <serviceName>NLPTrans</serviceName>
4      <serviceURL>http://nlp-trans.com/v1</serviceURL>
5      <method>GET</method>
6      <cost>0.0002</cost>
7      <parametersMapping>
8          <input name="text">text</input>
9          <input name="from">srcLang</input>
10         <input name="to">tgtLang</input>
11     </parametersMapping>
12     <responseMapping>
13         <output
               name="translated.text">text</output>
14     </responseMapping>
15 </serviceConfig>
```

Fig. 6: Example Configuration File for `nlptrans` Service

2) Specifying QoS Requirements: As different applications have different QoS requirements, we provide an interface for developers to define how to generate the optimal composition strategy. The developers need to write a Java lambda function to calculate the utility value of each service composition strategy. Lines 1-10 in Fig. 7 show an example. The developers obtain the estimated QoS values by calling corresponding functions. The utility is then calculated by dividing reliability by the product of latency, tail latency, and cost, aiming to maximize reliability while minimizing latency and cost. Our runtime system takes this specified lambda function, checks if the utility value is in the valid range ($[0, +\infty]$), and, if so,

```
0  //Defining the utility calculation
1  Function<CompositionStrategy, Double>
       utilityFunction = s -> {
2          double latency = s.getEstimatedLatency();
3          double tailLatency =
               s.getEstimatedTail();
4          double reliability =
               s.getEstimatedReliability();
5          double cost = s.getEstimatedCost();
6          double utility = reliability / (latency
               * tailLatency * cost);
7          return utility;
8  };
9  //Applying the defined utility formula
10 Trans.applyUtility(utilityFunction);
11 //Defing when to trigger probing
12 Trans.probeAfterRuns(100) // or
       Trans.probeAfterQoSDecrease(30)
```

Fig. 7: Example of Specifying Utility Calculation and Defining Probing Trigger

selects the composition strategy with the highest utility. If any utility value is negative, the system defaults to invoking all services.

3) Defining Probing Trigger: We provide two schemes for defining when to trigger probing, allowing developers to choose based on their preferences or the QoS changes of services. The first scheme, "probeAfterRuns," triggers probing after a specified number of invocations, allowing developers to set a fixed interval for probing. As shown in Line 12 of Fig. 7, the developer sets the probing interval to 100 runs. The second scheme, "probeAfterQoSDecrease," involves recording the average QoS of the previous $n = 20$ invocations. If the QoS of a new service invocation differs from this average by a specified percentage, probing is triggered. An example of using this scheme is Trans.probeAfterQoSDecrease(30), which sets the tolerance gap for QoS difference at 30%. This scheme is sensitive to QoS changes in services, ensuring timely adjustment of the composition strategy.

D. Composition QoS Estimation Model

Given an optimization goal set by a developer, an effective way of identifying the optimal composition strategy is to traverse all possible composition strategies. This process involves: 1) Modeling QoS for individual homogeneous services and estimating QoS for their compositions; 2) Evaluating all feasible strategies against cost and reliability constraints to select the one with the optimal latency satisfaction index. The rationale for exhaustive search is its manageable scope: if we

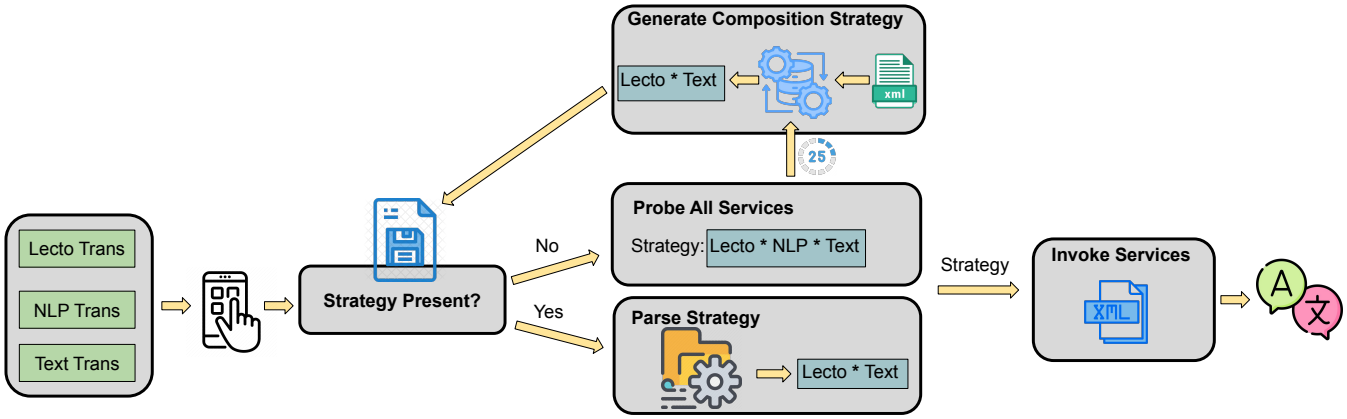


Fig. 8: Our Runtime Implementation

have 10 homogeneous services, only $2^{10} = 1,024$ possible strategies exist. The cornerstone of successfully finding the optimal strategy is accurately estimating the QoS of each service composition.

For a service $i \in \mathcal{I}$, we use c_i, r_i, and l_i to denote its cost, reliability, and latency. Given a set of homogeneous services $\mathcal{H}_s$ of a composition strategy s, their speculative parallel invocation succeeds when any service succeeds, and fails when all constituent services fail. Existing approaches [25], [26] estimate the QoS for their speculative parallel invocation as: 1) $C_s = \sum_{i \in \mathcal{H}_s} c_i$; 2) $R_s = 1 - \prod_{i \in \mathcal{H}_s} (1 - r_i)$; 3) $L_s = \min(l_i), \forall i \in \mathcal{H}_s$. We found that while the cost and reliability estimations are accurate, the latency estimation is not.

Accurate Latency Estimation with Low Overhead Achieving high accuracy in QoS estimation is challenging, and doing so with low overhead makes it even more difficult. Higher estimation accuracy usually requires more QoS samples, increasing probing requests. To address this, we model service latency as a distribution rather than a single value, enhancing the accuracy of estimating composition latency even with limited QoS samples. The distribution is calculated from the recorded QoS data of the service. In particular, this paper adopts the Shifted Exponential Distribution [28] to model an individual service's latency. We employ a piecewise function, $F_i(x)$, to represent the Cumulative Distribution Function (CDF) of service i's latency, where x denotes a latency value.

$$F_i(x) = \begin{cases} 0, & x < t \\ 1 - e^{-\frac{1}{m-t}(x-t)}, & x \geq t \end{cases} \tag{1}$$

The distribution parameters, t and m, correspond to the service's **minimum latency** and **average latency**, respectively. In contrast to other service latency distributions (e.g., Erlang and Pareto) that require estimating parameters from the entire set of invocation samples, the distribution we choose is simple and only requires acquiring one additional latency statistical parameter, the minimum latency, in addition to the average latency.

After modeling each homogeneous service's latency, we calculate the resulting latency distribution for a composition. Recall that we use $l_i, \forall i \in \mathcal{H}_s$ to denote the latency of a service i. We use $\mathbf{P}\big(L_s \leq x\big)$ to denote the probability that the latency L_s of a composition strategy s is less than x. Assuming $l_i, \forall i \in \mathcal{H}_s$ are independent, we have:

$$\begin{aligned} \mathbf{P}\big(L_s \leq x\big) &= \mathbf{P}\big(\min(l_i) \leq x\big), \forall i \in \mathcal{H}_s \\ &= 1 - \mathbf{P}\big(\min(l_i) > x\big), \forall i \in \mathcal{H}_s \\ &= 1 - \prod_{i=1}^{\mathcal{H}_s} \mathbf{P}\big(l_i > x\big) \\ &= 1 - \prod_{i=1}^{\mathcal{H}_s} (1 - \mathbf{P}\big(l_i \leq x\big)) \end{aligned} \tag{2}$$

We can calculate the resulting latency CDF $\mathbf{P}\big(L_s \leq x\big)$ by using $P(l_i \leq x) = F_i(x)$, where $F_i(x)$ is given by Eq. 1. After calculating the latency distribution, we can further calculate other statistics of interest, such as the average latency (i.e., L_s) or tail latency (i.e., T_s) in a closed-form expression, which other service distribution models cannot achieve.

E. Reference Implementation

We implemented our runtime system as an Android library, facilitating easy integration into mobile Apps. Our implementation[2] contains approximately 620 lines of Java code. We further use "Language Translation" as an example to demonstrate how the library is implemented and integrated into an application (Fig.8). Initially, when a translation request arises, and no pre-calculated strategy exists, all three translation services are invoked simultaneously, and the first returned result is used. This probing continues until a sufficient number of QoS data samples are collected, specified as 25. Based on this data, the system calculates an optimal composition strategy (e.g., combining only `Lecto` and `Text`), which is then saved. Future requests bypass the probing phase and use the pre-determined strategy, enhancing QoS by avoiding less efficient service combinations.

QoS Optimization Goal Utilizing the programming interface for specifying QoS requirements, we implemented the following QoS optimization goal. We treat the developer's reliability and latency requirements as hard and soft constraints,

[2]https://github.com/zqli-sketch/HSC

respectively. Let $\hat{C}$, $\hat{R}$, $\hat{L}$, and $\hat{T}$ denote the developer's per-invocation budget, minimum reliability, desired average latency, and desired tail latency. Let $\mathcal{I} = i = 1, 2, 3, ...I$ represent a set of homogeneous services, and $\mathcal{S} = s = 1, 2, 3, ...S$ represent all possible composition strategies. For a strategy s, let L_s, R_s, T_s, and C_s denote its latency, reliability, tail latency, and cost. The utility calculation formula of this optimization goal can be noted as:

$$\text{Utility}(s) = \frac{L_s + \hat{L}}{L_s} \cdot \frac{T_s + \hat{T}}{T_s} \quad \text{where } R_s \geq \hat{R} \text{ and } C_s \leq \hat{C} \tag{3}$$

Here, $\frac{L_s+\hat{L}}{L_s} * \frac{T_s+\hat{T}}{T_s}$ represents the latency satisfaction index, ranging from $(1, +\infty)$. To determine the optimal strategy, our system runtime first ensures each strategy meets the reliability and cost constraints: $R_s \geq \hat{R}$ and $C_s \leq \hat{C}$. Then, it calculates the utility for each strategy using the formula above. The strategy with the highest utility value is considered the optimal composition strategy.

V. Evaluation

This section presents how we evaluate HomoService and the experimental results. Specifically, we begin by assessing the accuracy of our QoS estimation model, followed by an analysis of the cost-efficiency in improving client-perceived QoS. Lastly, we evaluate the overhead imposed on mobile clients by our approach. The evaluation questions, along with the key findings, are as follows:

EQ1: How accurate is our QoS estimation model?

Findings: Our results show that the accuracy of our model is similar for average latency and higher for tail latency, as compared with baseline approaches.

EQ2: How much does HomoService improve QoS and reduce invocation costs?

Findings: Our results demonstrate that HomoService can reduce average latency by 8%, decrease tail latency by 35%, and increase reliability to 100% compared to average-optimal service. Additionally, it incurs 50% less cost compared to static homogeneous service composition.

EQ3: What are the HomoService's usage overheads?

Findings: The CPU, memory, and battery usages of HomoService are acceptable on modern mobile phones.

A. Performance of our QoS Estimation Model

We first evaluated the fitness of the latency model for individual services. Then, we assessed the model's accuracy in estimating composite latency.

For Individual Services We use the real service latency traces collected in our previous study, which contain 90 sets of service invocation samples, to evaluate the fitness of our latency model. We compare our model, which is based on Shifted Exponential distribution, with two other distributions– namely, the Erlang and Pareto distributions. Our evaluation measures how closely the values predicted by these models match the actual values, in terms of the mean, median, and 95^{th} percentile latency [29]. We employ a statistical measure R^2 [30] (or the coefficient of determination) to evaluate the goodness of fit of a regression model from predicted latency to actual latency. R^2 score usually ranges from 0 to 1, with a higher score indicating a better fit of the model to the data.

We first measure the fitness of these models if a large volume of samples is provided. For each of the 90 sets of invocation samples, we estimate the distribution parameters for each of the three distributions. We then calculate R^2 for mean, median, and tail latency (i.e., 95^{th} percentile) as $R^2 = 1 - \frac{\sum_i (\hat{y}_i - y_i)^2}{\sum_i (y_i - \bar{y})^2}$, where i denotes a set of invocation samples, $\hat{y}_i$, y_i, and $\bar{y}_i$ represent the predicted value, actual value, and the average of all actual values respectively. Table 1 shows the results, from which we observe that the Shifted Exponential distribution is a good fit for modeling individual service latency. Compared with the Erlang distribution, the Shifted Exponential distribution exhibits slightly inferior tail latency. We still choose to use Shifted Exponential distribution for its simplicity (i.e., it only needs to know the average and minimum latency of all invocation samples), as well as its superior performance for small sample sizes.

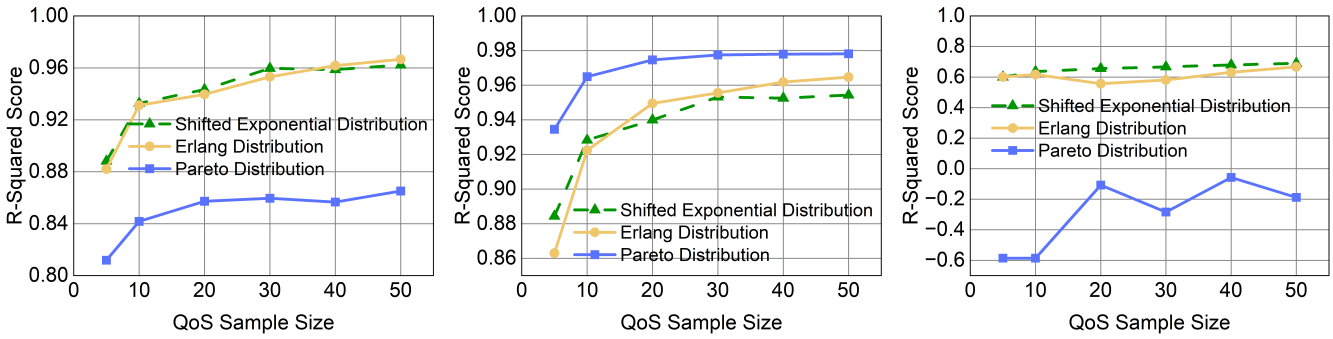
	Shifted Exponential Distribution	Erlang Distribution	Pareto Distribution
Means	0.9983	0.9985	0.8444
Medians	0.9785	0.9707	0.9812
Tails	0.8041	0.9019	-0.6448

TABLE III: R^2 between the Actual and Predicted Means, Medians, and Tails of Individual Services

Next, we measure the fitness of these models when given a small volume of samples, as it is practical to make the composition decision at runtime by invoking each service a limited number of times. We vary the number of invocation samples in increments of 10, ranging from 5 to 50. For each sample size, we randomly select invocation samples for model fitting, repeat the process 400 times, and calculate the average R^2 for these 400 repetitions. Fig. 9 demonstrates the changes in R^2 for mean, median, and tail latency, respectively. We observe a clear trend that the Shifted Exponential distribution outperforms the other two distributions.

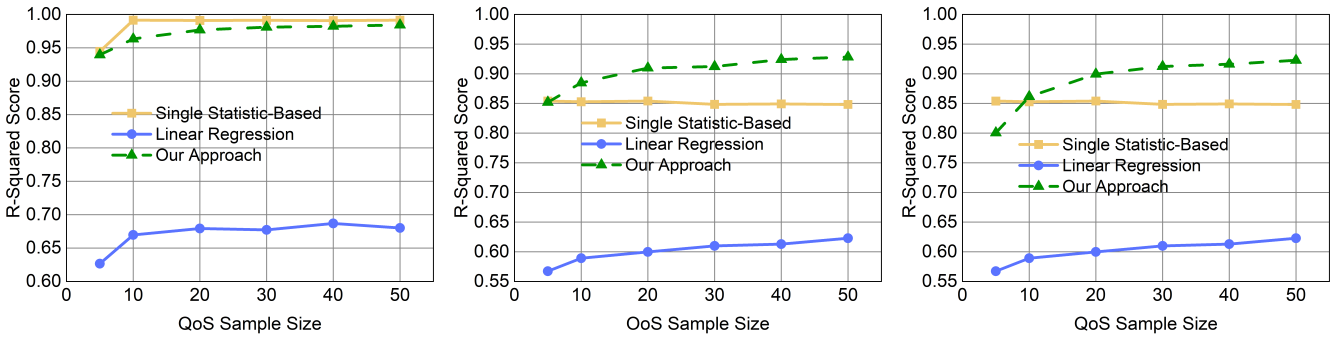
For Service Composition For service compositions, we further measure the estimation accuracy of our model for average latency and tail latency (90^{th}, 95^{th} percentile latency). We compare our approach with the following baseline approaches:

- **Average Latency Based [25], [26]:** This is the approach adopted in existing studies. It estimates the latency of a composition as the minimum value of the average latency of all invoked services.
- **Single Statistic-Based:** We extend the idea of using the minimum value of average latency to tail latency. We measure the tail latency of individual services and use the minimum value as the tail latency of their composition. For instance, for three homogeneous services with tail latency values T_1, T_2, T_3, the tail latency of their composition is calculated as $T = \min(T_1, T_2, T_3)$.
- **Linear Regression:** To make a fair comparison, we propose a linear regression-based approach. We use 80% of the 90 sets of invocation traces as the training set to train a linear regression model for predicting the average and tail latency values for their composition. This method has demonstrated its effectiveness in estimating the overall



(a) R^2 for Avg. Latency (b) R^2 for Mdn. Latency (c) R^2 for Tail Latency

Fig. 9: Sample Size's Impact on Modeling Fitness for Individual Services



(a) Average Latency (b) 90% Tail Latency (c) 95% Tail Latency

Fig. 10: Sample Size's Impact on Modeling Fitness for Service Composition

service latency when considering all the latencies of RPC calls [31]. In the aforementioned example, the linear regression model takes T_1, T_2, T_3 as inputs and outputs the estimated resulting tail latency for a composition strategy that combines three services.

	Average	90%	95%
Average Latency-Based	0.9620	N.A.	N.A.
Single Statistic-Based	0.9620	0.8995	0.9546
Linear Regression	0.6944	0.6819	0.6492
Our Approach	0.9858	0.9433	0.9371

TABLE IV: R^2 of Actual and Estimated Latency Metrics

Table IV summarizes the R^2 values for different latency metrics for our approach and the baseline approaches, which are calculated based on all samples in a trace set. We observe that our approach outperforms all other approaches in average and 90% latency, and is comparable with the single-statistic-based approach in 95% latency.

We further compare the performance of these approaches when the sample size is smaller. We vary the number of invocation samples in increments of 10, ranging from 5 to 50. We randomly select the required number of samples from each trace, and repeat the procedure 400 times. As shown in Fig. 10, when the sample size reaches 30, our approach shows significantly better accuracy than other approaches, especially for the tail latency. Overall, our approach is more accurate in estimating the composition latency compared with other approaches, particularly when the sample size is limited.

B. QoS Benefits and Cost Reduction of Using HomoService

We then study how HomoService improves QoS and reduces cost by conducting both real-world testbed evaluation and simulations.

QoS Parameters Configuration When using HomoService, parameters like service invocation cost, required reliability, latency, and sample size are crucial. We developed a random budget generator to evaluate our solution under various cost constraints. The budget range is defined from the cost of the average-optimal service (minimum) to a maximum of K times the highest service cost, with K initially set at 2. Costs were sourced from the corresponding service's RapidAPI page, normalized to integers, and used to generate budgets. We targeted a reliability of 99.99% and aimed for latencies 15% better than the average-optimal service, with a set sample size of 25 for each application implementing the composition.

1) Performance Measured by Testbed: We first describe the configuration of our testbed and the baseline approaches used for comparison, followed by an analysis of the experimental results.

Configuration and Deployment We developed six Android applications in Java, each corresponding to a specific task selected from our empirical study in Section III-A. For each task, we used three equivalent services, and retrieved their endpoint URLs from their respective pages on the RapidAPI website. Each application was implemented in three variants, corresponding to different service invocation paradigms:

QoS Metric	Reliability			Cost			Avg Latency (ms)			Tail Latency (ms)		
Paradigm	**Avg-Optimal**	**Static**	**Dynamic**	**Avg-Optimal**	**Static**	**Dynamic**	**Avg-Optimal**	**Static**	**Dynamic**	**Avg-Optimal**	**Static**	**Dynamic**
Translation	0.99	1.00	1.00	67	184	82	395	196	339	1511	300	1173
Face Detect	1.00	1.00	1.00	10	13	13	1241	1071	1098	1648	1537	1555
IP2Loc	0.99	1.00	1.00	23	96	44	213	143	159	937	712	768
Weather	1.00	1.00	1.00	2000	3500	2495	283	254	265	1204	987	1073
Flight	1.00	1.00	1.00	20	23	23	880	846	848	1958	1741	1754
Hotel	1.00	1.00	1.00	20	53	24	858	848	848	4874	4669	4866
Average	0.99	1.00	1.00	356	644	446	645	559	592	2022	1657	1864

TABLE V: Service QoS Performance of Real Apps with Different Invocation Paradigms at Michigan, US

- **Avg-Optimal:** This variant invokes the service with the best average latency among the three equivalent services across the five locations.
- **Static:** This variant employs Static Homogeneous Service Composition, where all three equivalent services are hardcoded and invoked simultaneously. The application uses the first response returned to continue execution.
- **Dynamic:** This variant uses our proposed approach, *HomoService*, which dynamically determines the optimal subset of services to invoke. We used the same three equivalent services and composed them via our programming interface. The composition was configured with target reliability, average latency, and tail latency requirements as described previously.

We tested each variant on three Samsung Galaxy A53 phones (Octa-core CPU, 6GB RAM), running simultaneously for 24 hours in Michigan, USA. Each variant was executed at 40-second intervals. For the variant using dynamic service composition, the cost requirement was varied in each run based on the random budget generator. We recorded both the end-to-end latency and the average reliability for all variants.

Results Table V shows the QoS performance comparison among three invocation paradigms across tasks. Observations include: 1) “Translation", “Face Detect", and “IP2Loc" services benefit the most from homogeneous service composition, reducing average latency by 11–25% and tail latency by 6–22% compared to the average-optimal services; 2) Considering all services, homogeneous service composition improves reliability to 100% and reduces both average and tail latencies by 8% compared with the average-optimal service; 3) Dynamic composition costs 25% more than the average-optimal service, whereas static composition costs 80% more without significant QoS benefits, except for "Translation," where it reduces tail latency to 300ms, outperforming our solution due to its higher cost, nearly 3x that of invoking an average-optimal service. Overall, the performance difference between HomoService and static homogeneous service compositions is minor, but the latter is much more expensive.

2) Performance Measured by Simulation: To evaluate the performance of our approach at scale, we used real service QoS traces collected in our previous study. These traces cover all six tasks and were gathered from five locations worldwide, comprising 90 sets of service invocation samples. For each task at each location, we used Python to apply our optimization algorithm to identify the optimal composition strategy and simulate service invocations accordingly (i.e., **Dynamic**). Consistent with the testbed evaluation, we also implemented the **Avg-Optimal** paradigm and the **Static** homogeneous composition paradigm as baselines. Due to the randomness introduced by the cost budget generator, we ran the Dynamic simulation 400 times. In each run, the cost requirement was varied according to the generator, following the same configuration as in the testbed experiments.

Results For each task at each location, we computed the average latency and reliability after completing the designated simulation invocations under each invocation paradigm. We then averaged the results across all six tasks to obtain the overall average latency and reliability for each paradigm. The aggregated results for all locations are summarized in Table VI. Findings include: 1) Dynamic composition boosts QoS at each location, enhancing reliability to 100% and reducing average latency by 8% and tail latency by 35% compared to invoking the average-optimal services considering all locations; 2) Overall, dynamic composition costs 31% more than invoking average-optimal services, whereas static composition costs 80% more but only slightly improves latency (3% average, 11% tail). In summary, HomoService offers the most cost-effective improvement in service reliability, latency, and budget efficiency.

Sensitivity Analysis We next investigate how the system parameter, the budget upper bound parameter K, affects the performance of HomoService. We conducted simulations with 400 runs for each configuration; larger values of K means higher cost budget.

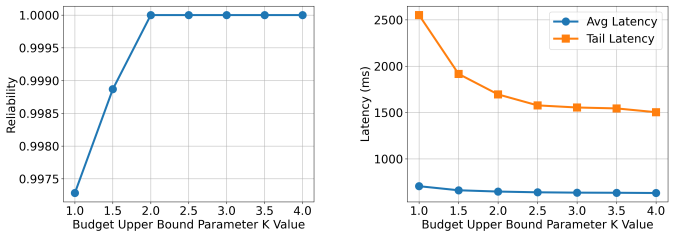


Fig. 11: Cost Budget’s Impacts on QoS Benefits

As shown in Fig. 11, increasing the cost budget from 1 to 2 significantly improves reliability, reaching the maximum of 100%. Beyond this point, reliability remains unchanged. In terms of latency, increasing the cost budget has a greater impact on tail latency than on average latency. Similar to the reliability trend, increasing the budget initially leads to a sharp reduction in tail latency, followed by a plateau. These observations suggest that allocating a higher budget to invoke more services does not necessarily yield proportional

QoS Metric	Reliability			Cost			Avg Latency (ms)			Tail Latency (ms)		
Paradigm	**Avg-Optimal**	**Static**	**Dynamic**	**Avg-Optimal**	**Static**	**Dynamic**	**Avg-Optimal**	**Static**	**Dynamic**	**Avg-Optimal**	**Static**	**Dynamic**
India	0.99	1.00	1.00	356	645	467	841	711	744	3094	1392	1773
Japan	0.99	1.00	1.00	356	645	467	500	432	448	2882	1111	1500
Australia	0.99	1.00	1.00	356	645	469	783	730	743	3379	1683	2059
Germany	0.99	1.00	1.00	356	645	468	701	671	680	1389	1119	1250
US	0.99	1.00	1.00	356	645	468	661	550	591	2100	1621	1804
Average	0.99	1.00	1.00	356	645	468	697	619	641	2568	1385	1677

TABLE VI: Service QoS Performance of Different Invocation Paradigms at Five Locations Worldwide

benefits, highlighting the low cost-efficiency of traditional static homogeneous service composition.

C. Overheads of Using HomoService

While the QoS benefits of homogeneous service composition are evident for end users, it is also important to measure the overhead of supporting it on resource-limited mobile devices. We used the "Language Translation" App to compare the CPU usage, memory utilization, and energy consumption of our framework against the other two baseline approaches. We ran each variant of the app continuously for three hours on a fully charged phone configured with identical system settings, including screen brightness, network connectivity, and background applications.

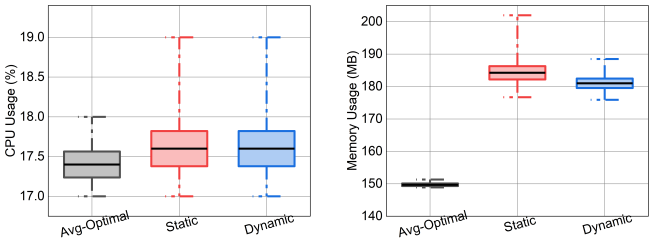


Fig. 12: Phone's CPU/Memory Usage Comparison

CPU and Memory Consumption Fig. 12 shows the CPU and memory usage for three service invocation paradigms. Both "Dynamic" and "Static" incurred less than 1% on average and 2% at peak for CPU usage. For memory usage, our solution used approximately 30MB more memory. Considering the GB-level memory capacities and speedy processors of modern mobile phones, the additional CPU and memory overhead is acceptable.

Power Consumption We also recorded the phone's power consumption over the three-hour test period. We found that the consumption for "Avg-optimal", "Static" and "Dynamic" was 669mAh, 712mAh, and 699mAh, respectively. We can see that invoking all services incurred higher power consumption due to issuing more HTTP requests than our approach.

In summary, our approach incurs acceptable overhead for clients. Static homogeneous service composition leads to higher costs and greater resource consumption, underscoring the efficiency of dynamic composition for enhancing QoS.

VI. Applicability

Many server-side mechanisms—such as CDNs, edge computing, load balancing, and redundancy strategies—have been widely adopted to improve QoS. However, server-side optimization is approaching a bottleneck, as further improvements often require significant investment in infrastructure and resources. In contrast to these well-established server-side techniques, homogeneous service composition optimizes QoS on the client side. This client-centric approach offers a promising yet underexplored direction for enhancing user-perceived performance. Although it may involve invoking additional services, application developers or providers typically operate within a defined cost budget for each round of service invocation. Within this budget, invoking multiple services to improve QoS can be a worthwhile investment, as better user experiences can lead to tangible business benefits, such as increased user retention and market share [32].

Our approach, *HomoService*, accurately identifies optimal composition strategies that maximize QoS benefits within a given budget, using only a limited number of samples. Supported by an intuitive programming interface, it allows developers to easily customize invocation strategies for users at different locations.

VII. Conclusion and Future Work

In this paper, we introduced an approach that dynamically composes homogeneous services for each client, achieving the desired QoS for performance-critical services while minimizing invocation costs. Unlike static homogeneous service composition, which invokes all services to improve QoS at high costs, our approach customizes the optimal composition strategy to balance QoS benefits and invocation costs for each end user. We implemented a system prototype of our approach and conducted comprehensive evaluations. The experimental results demonstrate the efficacy and applicability of our approach, making it a valuable tool for service-oriented application developers.

In future work, we plan to: 1) Extend the current QoS optimization objective, which primarily focuses on system-related performance metrics such as latency and reliability, to also incorporate data-related quality factors from service responses. Designing a mathematical model that jointly optimizes system-level and data-level performance (e.g., accuracy) while achieving a proper trade-off between them remains a non-trivial challenge; 2) Investigate collaborative strategies for QoS learning. At present, each user probes the services independently and stores the measured QoS locally to determine its own optimal strategy. Future research will explore mechanisms that allow users to share anonymized QoS observations, thereby reducing probing overhead and accelerating convergence across clients.

ACKNOWLEDGEMENT

This research is supported by NSF through grant 2104337.

REFERENCES

[1] M. E. Haque, Y. H. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley, "Few-to-many: Incremental parallelism for reducing tail latency in interactive services," *ACM Sigplan Notices*, vol. 50, no. 4, pp. 161–175, 2015.

[2] Z. Li and Z. Song, "Poster: Service polymorphism: Enhancing web service performance by serving clients dissimilarly," in *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2024, pp. 1444–1445.

[3] S. Baravkar, O. Pellegrini, P. Gaikwad, E. Tilevich, and Z. Song, ""how can i be of service?"—a comprehensive analysis of web service integration practices," in *2024 IEEE International Conference on Web Services (ICWS)*. IEEE, 2024, pp. 1206–1216.

[4] rapidAPI, "RapidAPI - the next-generation API platform," 2015. [Online]. Available: https://rapidapi.com/hub

[5] S. Baravkar, C. Zhang, F. Hassan, L. Cheng, and Z. Song, "Decoding and answering developers' questions about web services managed by marketplaces," in *2024 IEEE International Conference on Software Services Engineering*. IEEE, 2024.

[6] Z. Song, Z. Li, and E. Tilevich, "A meta-pattern for building qos-optimal mobile services out of equivalent microservices," *Service Oriented Computing and Applications*, vol. 18, no. 2, pp. 109–120, 2024.

[7] A. Bhatia, S. Li, Z. Song, and E. Tilevich, "Exploiting equivalence to efficiently enhance the accuracy of cognitive services," in *IEEE CLOUDCOM'19*, pp. 143–150.

[8] Z. Song and E. Tilevich, "Equivalence-enhanced microservice workflow orchestration to efficiently increase reliability," in *2019 IEEE International Conference on Web Services (ICWS)*. IEEE, 2019, pp. 426–433.

[9] Z. Li, L. Cheng, and Z. Song, "Client-specific homogeneous service composition at runtime for qos-critical tasks," in *International Conference on Service-Oriented Computing*. Springer, 2025, pp. 87–95.

[10] Z. Zheng, Y. Zhang, and M. R. Lyu, "Investigating QoS of real-world web services," *IEEE transactions on services computing*, vol. 7, no. 1, pp. 32–39, 2012.

[11] A.-M. K. Pathan, R. Buyya *et al.*, "A taxonomy and survey of content delivery networks," *Grid computing and distributed systems laboratory, University of Melbourne, Technical Report*, vol. 4, no. 2007, p. 70, 2007.

[12] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[13] Akamai Technologies, "Akamai intelligent platform overview," https://www.akamai.com, accessed: 2025-10-20.

[14] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[15] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[16] S. Rosario, A. Benveniste, S. Haar, and C. Jard, "Probabilistic qos and soft contracts for transaction-based web services orchestrations," *IEEE Transactions on Services Computing*, vol. 1, no. 4, pp. 187–200, 2008.

[17] Z. Wu and H. Madhyastha, "Understanding the latency benefits of multi-cloud webservice deployments," *ACM SIGCOMM Computer Communication Review*, vol. 43, pp. 13–20, 04 2013.

[18] S. Azzam, M. Al-Kabi, and I. Alsmadi. (2012, March) Web services testing challenges and approaches. Researchgate. [Online]. Available: https://shorturl.at/svPZ0

[19] C. Ding, A. Zhou, Y. Liu, R. N. Chang, C.-H. Hsu, and S. Wang, "A cloud-edge collaboration framework for cognitive service," *IEEE Transactions on Cloud Computing*, vol. 10, no. 3, pp. 1489–1499, 2022.

[20] J. Brutlag, "Speed matters for google web search," http://services.google.com/fh/files/blogs/google_delayexp.pdf, 2009, accessed: 2025-06-17.

[21] S. Souders, "Velocity and the bottom line," http://radar.oreilly.com/2009/07/velocity-makingyour-site-fast.html, 2009, accessed: 2025-06-17.

[22] GigaSpaces, "Amazon found every 100ms of latency cost them 1% in sales," https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales, 2009, accessed: 2025-06-17.

[23] Digital Realty, "The cost of latency," https://www.digitalrealty.com/resources/articles/the-cost-of-latency, n.d., accessed: 2025-06-17.

[24] W. D. Gray and D. A. Boehm-Davis, "Milliseconds matter: An introduction to microstrategies and to their use in describing and predicting interactive behavior." *Journal of experimental psychology: applied*, vol. 6, no. 4, p. 322, 2000.

[25] N. Hiratsuka, F. Ishikawa, and S. Honiden, "Service selection with combinational use of functionally-equivalent services," in *2011 IEEE International Conference on Web Services*. Washington DC, USA: IEEE, 2011, pp. 97–104.

[26] Z. Song and E. Tilevich, "Win with what you have: QoS-consistent edge services with unreliable and dynamic resources," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2020, pp. 530–540.

[27] A. L. Lemos, F. Daniel, and B. Benatallah, "Web service composition: a survey of techniques and tools," *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, pp. 1–41, 2015.

[28] ReliaWiki, "The exponential distribution," 2017. [Online]. Available: https://reliawiki.org/index.php/The_Exponential_Distribution

[29] W. Zhang and J. He, "Modeling end-to-end delay using pareto distribution," in *ICIMP'2007*. IEEE, 2007, pp. 21–21.

[30] D. J. Ozer, "Correlation and the coefficient of determination." *Psychological bulletin*, vol. 97, no. 2, p. 307, 1985.

[31] G. Mann, M. Sandler, D. Krushevskaja, S. Guha, and E. Even-Dar, "Modeling the parallel execution of black-box services." in *HotCloud*, 2011.

[32] W. Zhang, C. K. Chang, T. Feng, and H.-y. Jiang, "QoS-based dynamic web service composition with ant colony optimization," in *2010 IEEE 34th Annual Computer Software and Applications Conference*. IEEE, 2010, pp. 493–502.

Zhengquan Li is a PhD candidate in Computer and Information Science department at the University of Michigan - Dearborn. His research interests include but are not limited to distributed systems, middleware for web service optimization, and ML for edge. Before joining his PhD jouney, he received the Double-Degree of Computer Science from the Joint-Program between Central China Normal University (CCNU) and University of Wollongong Australia (UOW). He received the Distinguished Paper Award from IEEE ICDCS in 2024.

Zheng Song is an Assistant Professor in the Computer and Information Science Department at the University of Michigan - Dearborn. He received his second PhD in Computer Science (with a focus on distributed systems and software engineering) from Virginia Tech USA in 2020, and his first PhD (with a focus on wireless networking and mobile computing) from Beijing University of Posts and Telecommunications China in 2015. He worked as a software engineer at Sina for one year after he graduated with a Master's degree from China Agriculture University in 2009. He received the Best Paper Award from IEEE Edge in 2019, the NSF CRII award in 2021, and the Distinguished Paper Award from IEEE ICDCS in 2024.