Client-Specific Homogeneous Service Composition at Runtime for QoS-Critical Tasks

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 customizing the composition strategy for different clients. In this paper, we introduce an approach that composes homogeneous services dynamically for each client, achieving the desired QoS for critical services 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 the QoS invoking a single service (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.

Keywords: Quality of Service · Service Composition · Runtime System

1 Introduction

Web services are widely used in modern software, providing diverse data and functionalities essential for various applications. Some of these data and functionalities are critical to an application's execution and user experience, such as real-time data processing in financial transactions or live updates in social media platforms. Given several functionally-equivalent services, developers face the problem of how to meet the stringent QoS requirements of their applications, in terms of latency, tail latency, and reliability.

The current state-of-the-practice approach is to select and invoke a skyline service (see Fig. 1 up left), i.e., a service that is not dominated by any other functionally-equivalent services. However, according to service statistics collected by service marketplaces [14] and an empirical study on developers' comments invoking web services [4], even skyline services sometimes fail to meet QoS requirements. To further improve QoS for critical tasks, previous works [5, 17, 18] have

explored the composition of homogeneous services, which involves simultaneously invoking multiple equivalent services and using the first response to continue the application's execution (Fig. 1 bottom left). While effective in enhancing QoS, this method significantly increases the number of service invocations, causing higher workloads on web servers and more network traffic. The additional invocation cost makes this approach impractical for large-scale deployment.

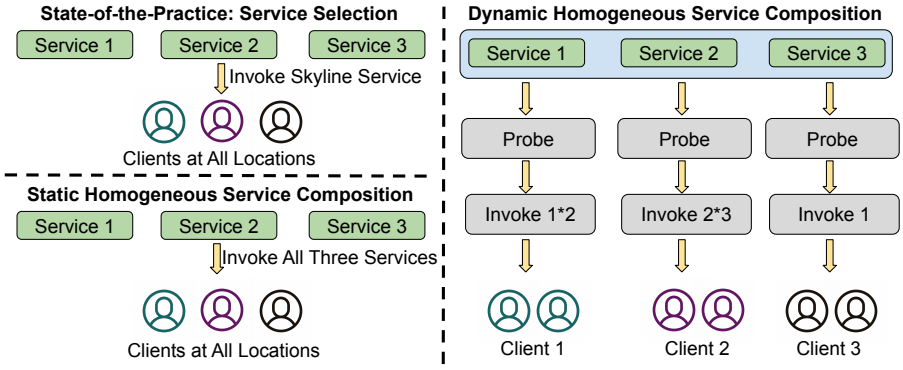


Fig. 1: Approaches for Satisfying the QoS Requirements of Critical Tasks

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 customizes service composition for end users at runtime. As demonstrated by the right subfigure in Fig. 1, for clients at different locations, our approach first probes the QoS of all homogeneous services, and calculates a client-specific composition strategy that best satisfies the developers' QoS requirements within a predefined invocation cost budget.

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 motivate 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.

2 Existing Approaches and Opportunities

In this section, we first summarize existing approaches for meeting the QoS requirements of critical tasks, and introduce our empirical study on the real-world performance of service invocations.

2.1 Meeting the QoS Requirements of Critical Tasks

The problem of QoS-based web service selection and composition has consistently received significant attention over the past two decades [1, 6]. As service-oriented architecture remains the primary method for accessing remote data and functionalities, critical tasks such as VR/AR, autonomous driving, and financial transactions demand stringent QoS requirements. Below, we introduce current solutions aimed at meeting these requirements.

Skyline Service Selection. Among a set of equivalent services, a service is considered a skyline service if no other service is better in all QoS attributes simultaneously. Developers can either hard code a pre-selected skyline service into their application [1] or rely on service gateways to select a service with optimal real-time QoS for composition [11]. This approach assumes that a selected skyline service can always satisfy the soft QoS requirements of an application [16]. However, this is not true for emerging applications with hard QoS requirements.

Static Homogeneous Service Composition. Homogeneous service composition has been explored in various contexts to enhance QoS. For example, some studies [17, 18] invoke multiple microservices in a way of speculative parallel fashion to improve the system reliability and execution time for applications in IoT environments. [5] parallely invoke multiple cognitive web services to improve accuracy (e.g., face recognition accuracy). Despite their benefits, these approaches simply invoking all services specified by static configurations, which lack flexibility and can lead to significant operational costs at runtime.

2.2 Empirical Study: A Large-Scale QoS Measurement

The most recent large-scale web service QoS measurement, conducted 10 years ago [22], revealed significant performance variation across different invocation contexts like locations and times. Since then, the use of content distribution networks (CDNs) [13] 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 usages scenarios [3, 7, 20]. 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,

	Service	Germany	Japan	Australia	India	US
FaceDet.	inferdo	1574 / 1769	1735 / 1937	1623 / 2030	2081 / 2291	1761 / 1924
	microsoftFace	827 / 1252	1152 / 1573	1377 / 1851	1554 / 1820	1065 / 1365
	SmartClick	954 / 1314	1610 / 1975	1674 / 2003	1468 / 1900	1517 / 1841
IP2Loc.	IP_GEO	22 / 40	25 / 37	22 / 37	531 / 562	116 / 295
	IP_lookup	35 / 56	39 / 57	37 / 53	43 / 79	182 / 373
	IPGtolocation	651 / 907	636 / 796	803 / 989	934 / 1241	533 / 896
Weather	openWeatherMap	62 / 123	297 / 467	365 / 576	231 / 312	222 / 610
	Visual Crossing	397 / 439	716 / 783	904 / 1016	828 / 3773	296 / 797
	weatherbitio	374 / 419	705 / 753	902 / 951	808 / 856	248 / 575
Trans.	Lecto Trans.	343 / 543	486 / 845	683 / 1028	590 / 873	350 / 538
	NLP Trans.	406 / 461	699 / 802	866 / 955	1166 / 1260	193 / 253
	Text Trans.	538 / 608	772 / 878	959 / 1074	1193 / 1363	289 / 359
Flight	FlightRadar	3520 / 4088	4084 / 4634	2468 / 2962	6213 / 7551	2383 / 3090
	FlyTrips	309 / 375	1374 / 1491	1679 / 1765	1484 / 1632	823 / 1163
	TravelAdvisor	1128 / 2471	1172 / 2825	1312 / 3476	1580 / 3161	899 / 2655
Hotel	Booking.com	238 / 769	652 / 1292	714 / 952	749 / 1149	635 / 1625
	Hotels	968 / 4284	1102 / 4635	1224 / 5810	1628 / 5242	887 / 4974
	Priceline	1925 / 3615	2333 / 3313	2249 / 3438	2482 / 4090	1983 / 3197
Avg.		**792 / 1307**	**1088 / 1616**	**1103 / 1720**	**1420 / 2175**	**799 / 1473**

Table 1: Latency Performance (in milliseconds) for All Six Sets of Homogeneous Services. Each cell presents Average Latency followed by Tail Latency.

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

Tokyo, Sydney, Mumbai, and Michigan—over continuous three days. This process collected approximately 4,000 samples per service set, totaling 90 service invocation trace sets.

Basic Results: Our findings, summarized in Tables 1 and 2, indicates that approximately 90% of the services had average and tail latencies exceeding 200ms—deemed unsatisfactory with tail latencies nearly twice the average, worsening user experiences. Moreover, most services failed to meet the minimum reliability standard of 99.9%, which is usually expected for service users. For instance, Amazon compensates users if AWS service availability drops below 99.9% [2]. Services involving complex cloud backends or large data transmissions, like face detection and translation, were particularly unreliable.

2.3 Observations and Opportunities

We further analyzed the measurement results and drew a couple of deeper observations that motivated our approach of composing homogeneous services for individual users.

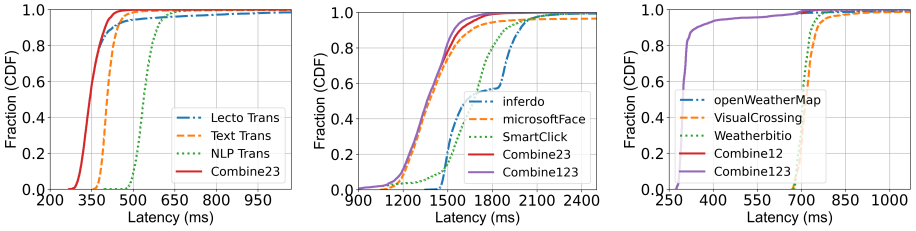
Observations: We first investigated the QoS performance of skyline services across different locations. For each task, given three services, the skyline service was selected as the one with the best average latency across all five locations. Our results show that skyline services do not consistently 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, skyline services could not consistently outperform their peers in some locations, risking QoS satisfaction.

We then examined the efficiency of the current static homogeneous service composition. Using the collected service invocation traces, we showed the latency improvement of invoking different services. For example, combining two services reduces average latency by 10% and 95th percentile latency by 24%, as shown in Fig.2a. Further adding a third service reduces tail latency more, as per Fig.2b. However, Fig.2c shows combining all three services fails to further reduce the latency of two services, which means that simply invoking all services does not always guarantee better QoS while incurring higher cost.



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

Fig. 2: Latency of Combined Invocations of Homogeneous Services

Observation 2

Static homogeneous service composition may incur unnecessary costs.

To find the efficient service combination for each user, it requires accurately estimate the resulting QoS for each combination. Existing method [8, 19] uses 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.

Observation 3

Existing QoS estimation model for compositions is inadequate.

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

3 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 [17], where $a*b$ denotes the workflow pattern of invoking two services, a and b, in parallel and using the first returned result.

3.1 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 to 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. 3.2).
- 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. 3.3).
- How to provide satisfactory QoS with minimum overhead? We establish a model that more accurately estimates the QoS of a composition with fewer probing requests (Sec. 3.4).

3.2 System Workflow

Traditional (heterogeneous) service composition [9] 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.

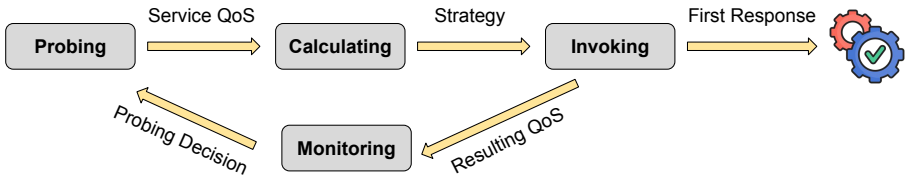


Fig. 3: The System Workflow of Our Approach

Fig.3 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.

3.3 Programming Model

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

Building a HomoService Fig. 4 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 outputs. For example, the configuration for the NLP Translation service (Fig. 5) 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 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. 4: Composing a Translation Service

Specifying QoS Requirements To meet diverse QoS requirements among applications, 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. Line 1-10 in Fig. 6 shows 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

```xml
<?xml version="1.0" encoding="UTF-8"?>
<serviceConfig>
    <serviceName>NLPTrans</serviceName>
    <serviceURL>http://nlp-trans.com/v1</serviceURL>
    <method>GET</method>
    <cost>0.0002</cost>
    <parametersMapping>
        <input name="text">text</input>
        <input name="from">srcLang</input>
        <input name="to">tgtLang</input>
    </parametersMapping>
    <responseMapping>
        <output name="translated.text">text</output>
    </responseMapping>
</serviceConfig>
```

Fig. 5: Example Configuration File for `nlptrans` Service

```
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. 6: Example of Specifying Utility Calculation and Defining Probing Trigger

($[0, +\infty]$), and, if so, selects the composition strategy with the highest utility. If any utility value is negative, the system defaults to invoking all services.

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. 6, the developer sets the probing interval as 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.

3.4 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 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 [8, 19] 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 [15] to model individual service's latency. We employ a piece-wise 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 Distribution) that require estimating parameters from the entire set of invocation samples, the distribution we choose is simple and only requires to acquire one additional latency statistical parameter, the minimum latency, in addition to the average latency.

After modeling each homogeneous service 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 of 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 $\mathbf{P}(L_s \leq x)$ by using $P(l_i \leq x) = F_i(x)$, where $F_i(x)$ can be 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 distributions models cannot achieve.

3.5 Reference Implementation

We implemented our runtime system as an Android library, facilitating easy integration into mobile Apps. our implementation 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.7). 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.

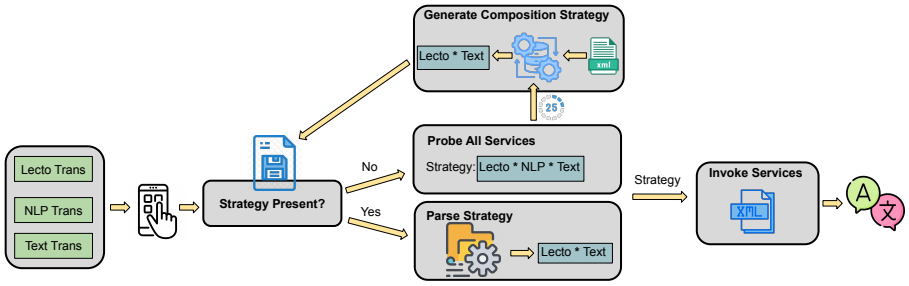


Fig. 7: Our Runtime Implementation

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, 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

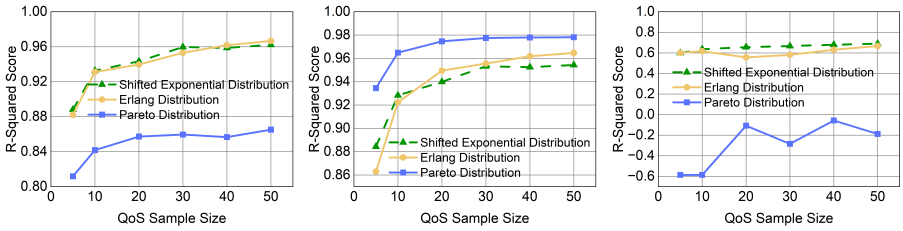
calculate the utility for each strategy using the formula above. The strategy with the highest utility value is considered the optimal composition strategy.

4 Evaluation

In this section, we evaluate the design of HomoService by answering the evaluation questions listed below:
EQ1: How accurate is our QoS estimation model? 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? Our results demonstrate that HomoService can reduce average latency by 7%, decrease tail latency by 35%, and increase reliability to 100% compared to skyline service selection. Additionally, it incurs 50% less cost compared to static homogeneous service composition.
EQ3: What are the HomoService's usage overheads? The CPU, memory, and battery usages of HomoService are acceptable on modern mobile phones.

4.1 Performance of our QoS Estimation Model

For Individual Services We use the real service latency traces collected in our empirical study, which contain 90 sets of service invocation samples. We compare our latency model which is based on Shifted Exponential distribution with two other distributions, namely the Erlang distribution and the Pareto distribution. Our evaluation measures how accurate the values predicted by these models match the actual values, in terms of the mean, median, and 95^{th} percentile latency [21]. We employ a statistical measure R^2 [12] (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.



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

Fig. 8: Sample Size's Impact on Modeling Fitness for Individual Services

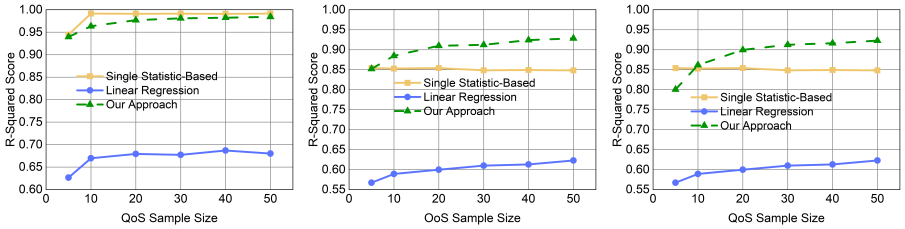
We adjusted the number of samples from 5 to 50 in increments of 10, randomly selecting samples for model fitting and repeating the process 400 times to calculate the average R^2. Fig. 8 shows the changes in R^2 for mean, median, and tail latency, respectively. We observe a clear trend that the Shifted Exponential distribution outperforms the other two distributions.

For Service Composition For service compositions, we further measure the estimation accuracy of our model on average latency and tail latency (90^{th}, 95^{th} percentile latency). We compare our approach with the following three baseline approaches: 1) Average Latency Based [8, 19], which estimates the latency of service composition as the minimum of the average latencies of all invoked services; 2) Single Statistic-Based, which further estimates tail latency as the minimum tail latency of individual services. For example, for three services with tail latencies T_1, T_2, T_3, the composition's tail latency is $T = \min(T_1, T_2, T_3)$; and 3) Linear Regression [10], which is trained using 80% of 90 sets of invocation traces and predicts average and tail latencies. In the above 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 3: R^2 of Actual and Estimated Latency Metrics

Table 3 presents the R^2 values for three latency metrics, comparing our approach to baselines using all trace set samples. Our method excels in average and 90% latency and matches the single-statistic-based approach in 95% latency.



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

Fig. 9: Sample Size's Impact on Modeling Fitness for Service Composition

We compared the performance of these methods with varying sample sizes, incrementing from 5 to 50 in steps of 10. We randomly select the required number of samples from each trace and repeat the procedure 400 times. As shown in Fig. 9, when the sample size reaches 30, our approach shows much better accuracy than other approaches, especially for tail latency. Overall, our method more accurately estimates composition latency, particularly with limited samples.

4.2 QoS Benefits and Cost Reduction of Using HomoService

We study how HomoService improves QoS and reduces cost by conducting both real-world testbed evaluation and simulations. In the testbed, we developed Android apps on Samsung Galaxy A53 phones (Octa-core CPUs, 6GB RAM) to invoke third-party services. For each trace captured for one task at one location,

we used Python to apply the optimization algorithm to identify the optimal strategy and invoke the services within the strategy.

QoS Parameters Configuration: For 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 QoS optimal service (minimum) to a maximum of K times the highest service cost, with K initially set at 2. Costs were sourced from the service corresponding Rapid page, normalized to integers, and used to generate budgets. We targeted a reliability of 99.99% and aimed for latencies 15% better than the QoS optimal service, with a set sample size of 25 for each application implementing the composition.

Performance Measured by Testbed: We developed six Android apps, one for each type of task, with three variants for each app. The first variant uses our system runtime, while the other two are: 1) Skyline, which invokes the service with the best average latency across five locations, and 2) Static Homogeneous Service Composition, which hardcodes and simultaneously invokes three homogeneous services. We tested each variant on three Android phones simultaneously for 24 hours in Michigan, USA, with a 40-second interval, recording end-to-end latency and average reliability.

	Reliability	**Cost**	**Latency ms (Avg. | Tail)**
Paradigm	Skyline/Static/Dynamic	Skyline/Static/Dynamic	Skyline/Static/Dynamic
Trans.	0.99/1.00/1.00	67/184/82	395/196/339 | 1511/300/1173
FaceDet	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
Avg.	**0.99/1.00/1.00**	**356/644/446**	**645/559/592 | 2022/1657/1864**

Table 4: Service QoS Performance of Real Apps at Michigan, US

Table 4 shows the QoS performance comparison of using three invocation paradigms across tasks. Observations include: 1) “Translation", “Face Detect", and “IP2Loc" services benefit the most from homogeneous service composition, with reducing 11% to 25% of average latency and 6% to 22% of tail latency compared to the corresponding Skyline services; 2) or all services, homogeneous service composition improves the reliability to 100%, both average latency and tail latency by 8%, as compared with the Skyline service; 3) dynamic composition costs 25% more than the Skyline 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 Skyline. Overall, the performance difference between HomoService and static homogeneous service compositions is minor, but the latter is much more expensive.

Performance Measured by Simulation: We ran 400 Python simulations and summarized the average results in Table 5, showing the QoS performance of three different invocation paradigms at five global locations. Findings include: 1) dynamic composition boosts QoS at each location, enhancing reliability to

	Reliability	Cost	Latency ms (Avg. | Tail)
Paradigm	Skyline/Static/Dynamic	Skyline/Static/Dynamic	Skyline/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	671/680/701 | 1389/1119/1250
US	0.99/1.00/1.00	356/645/468	661/550/591 | 2100/1621/1804
Avg.	**0.99/1.00/1.00**	**356/645/468**	**691/620/645 | 2568/1385/1677**

Table 5: Service QoS Performance at Five Locations Worldwide

100% and reducing average latency by 7% and tail latency by 35% compared to invoking skyline services; 2) Overall, dynamic composition costs 31% more than Skyline, 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.

4.3 Overheads of Using HomoService

We also evaluated the overhead of using HomoService on resource-limited mobile devices with a "Language Translation" app. We compared CPU usage, memory utilization, and energy consumption of our implementation against two other baseline approaches, running each app variant on a fully charged phone with identical settings for three hours. The results show that: 1) our approach increased CPU usage by at most 2% compared to invoking the skyline service and required about 30MB of additional memory, which is minimal given modern mobile phones' capabilities; 2) our approach consumed 30mAh more energy compared to the "Skyline" approach, while invoking all services consumed 44mAh. 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.

5 Conclusion

In this paper, we introduced an approach that dynamically composes homogeneous services for each client, achieving the desired QoS for 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.

References

1. Alrifai, M., Skoutas, D., Risse, T.: Selecting skyline services for QoS-based web service composition. In: WWW'10. pp. 11–20 (2010)

2. AWS: Amazon compute service level agreement (May 2022), https://aws.amazon.com/compute/sla/
3. Azzam, S., Al-Kabi, M., Alsmadi, I.: Web services testing challenges and approaches (March 2012), https://shorturl.at/svPZ0
4. Baravkar, S., Zhang, C., Hassan, F., Cheng, L., Song, Z.: Decoding and answering developers' questions about web services managed by marketplaces. In: 2024 IEEE International Conference on Software Services Engineering. IEEE (2024)
5. Bhatia, A., Li, S., Song, Z., Tilevich, E.: Exploiting equivalence to efficiently enhance the accuracy of cognitive services. In: IEEE CLOUDCOM'19. pp. 143–150
6. Chattopadhyay, S., Banerjee, A.: Qos-aware automatic web service composition with multiple objectives. ACM Transactions on the Web **14**(3), 1–38 (2020)
7. Ding, C., Zhou, A., Liu, Y., Chang, R.N., Hsu, C.H., Wang, S.: A cloud-edge collaboration framework for cognitive service. IEEE Transactions on Cloud Computing **10**(3), 1489–1499 (2022). https://doi.org/10.1109/TCC.2020.2997008
8. Hiratsuka, N., Ishikawa, F., Honiden, S.: Service selection with combinational use of functionally-equivalent services. In: 2011 IEEE International Conference on Web Services. pp. 97–104. IEEE, Washington DC, USA (2011)
9. Lemos, A.L., Daniel, F., Benatallah, B.: Web service composition: a survey of techniques and tools. ACM Computing Surveys (CSUR) **48**(3), 1–41 (2015)
10. Mann, G., Sandler, M., Krushevskaja, D., Guha, S., Even-Dar, E.: Modeling the parallel execution of black-box services. In: HotCloud (2011)
11. Moussa, H., Gao, T., Yen, I.L., Bastani, F., Jeng, J.J.: Toward effective service composition for real-time soa-based systems. SOCA **4**, 17–31 (2010)
12. Ozer, D.J.: Correlation and the coefficient of determination. Psychological bulletin **97**(2), 307 (1985)
13. Pathan, A.M.K., Buyya, R., et al.: A taxonomy and survey of content delivery networks. Grid computing and distributed systems laboratory, University of Melbourne, Technical Report **4**(2007), 70 (2007)
14. rapidAPI: RapidAPI - the next-generation API platform (2015), https://rapidapi.com/hub
15. ReliaWiki: The exponential distribution (2017), https://reliawiki.org/index.php/The_Exponential_Distribution
16. Rosario, S., Benveniste, A., Haar, S., Jard, C.: Probabilistic qos and soft contracts for transaction-based web services orchestrations. IEEE Transactions on Services Computing **1**(4), 187–200 (2008)
17. Song, Z., Li, Z., Tilevich, E.: A meta-pattern for building qos-optimal mobile services out of equivalent microservices. Service Oriented Computing and Applications **18**(2), 109–120 (2024)
18. Song, Z., Tilevich, E.: Equivalence-enhanced microservice workflow orchestration to efficiently increase reliability. In: 2019 IEEE International Conference on Web Services (ICWS). pp. 426–433. IEEE (2019)
19. Song, Z., Tilevich, E.: 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). pp. 530–540. IEEE (2020)
20. Wu, Z., Madhyastha, H.: Understanding the latency benefits of multi-cloud webservice deployments. ACM SIGCOMM Computer Communication Review **43**, 13–20 (04 2013). https://doi.org/10.1145/2479957.2479960
21. Zhang, W., He, J.: Modeling end-to-end delay using pareto distribution. In: ICIMP'2007. pp. 21–21. IEEE (2007)
22. Zheng, Z., Zhang, Y., Lyu, M.R.: Investigating QoS of real-world web services. IEEE transactions on services computing **7**(1), 32–39 (2012)