

Navigating Rapid API: An Empirical Dive into the Service Market

by

Siddhi V Baravkar

**A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
(Computer and Information Science)
in the University of Michigan–Dearborn
2024**

Master's Thesis Committee:

**Assistant Professor Zheng Song, Chair
Assistant Professor Foyzul Hassan,
Professor Eli Tilevich, Virginia Polytechnic Institute and State
University**

Siddhi V Baravkar

siddhib@umich.edu

ORCID iD: 0009-0009-6590-4290

© Siddhi V Baravkar 2024

Dedication

I dedicate this work to everyone who has supported me throughout this journey. I want to express my deepest gratitude to my parents, Mr. Vijay Sadashiv Baravkar and Mrs. Sulbha Vijay Baravkar, for their unwavering support, selfless efforts, and for being my source of strength and motivation. Their contributions have helped me reach this stage and achieve my goals.

I want to dedicate this work to my pillar of support, my husband, Mr. Gaurav Pokharkar, who believed in me and pushed me to achieve something I never thought possible in my master's program. He listened to my progress and doubts every day without any complaint. Without his support, reaching this milestone would have been difficult. I extend my heartfelt appreciation to him, for his constant encouragement throughout this journey.

Moreover, I would like to dedicate this work to all my lab mates who helped me overcome obstacles along the way. Their guidance was truly invaluable.

Acknowledgements

I express my deepest gratitude to my thesis advisor, Dr. Zheng Song. I am grateful that I met him in May 2023 and had the opportunity to discuss his research topics and plans. Despite my lack of previous research experience, he had faith in my ability to successfully contribute to and complete this research. His guidance has been transformative, marking a significant milestone in my academic journey. This experience has not only enhanced my technical skills but also refined my research approach, enabling me to delve deeply into topics, identify key findings and gaps, and develop new ideas and strategies to achieve them.

I would also like to express my sincere gratitude to my thesis committee members, Dr. Eli Tilevich and Dr. Foyzul Hassan, for their invaluable feedback, insights, and guidance. This experience has been incredibly enriching, significantly improving both my technical skills and logical reasoning abilities.

Lastly, I extend my heartfelt gratitude to all my lab mates for their guidance and assistance. They were always there when I needed support, and I gained invaluable knowledge from them.

Table of Contents

Dedication	ii
Acknowledgements	iii
List of Figures	vi
List of Tables	vii
Abstract	viii
Chapter	
1 Introduction	1
1.1 The Service-Oriented Architecture (SOA)	1
1.2 The Evolution of Service Registries in SOA	1
1.3 RapidAPI Marketplace: The Ultimate New Generation Platform	2
1.4 Developers' Key Concerns and Challenges	4
2 Background and Motivation	5
2.1 Service Registry	5
2.2 Service Registry and How RapidAPI is Different	6
2.3 Service Selection Criteria	6
2.4 Empirical Study of Service Integration	7
2.5 Understanding Developers' Concerns	7
3 Methodology	9
3.1 Comparing API Registries	9
3.1.1 Comparing Two Generations of Service Registry	10
3.2 Understanding Web Service Selection and Integration	10
3.2.1 Data Collection and Labeling	10
3.3 Decoding Developers' Questions	19
3.3.1 Collecting, Labeling and Categorizing Developers' Questions	19
3.3.2 Developing an LLM Tool to Answer Developers' Questions	20
4 Results	24
4.1 Static Service Registry Vs Marketplace Workflow	24
4.1.1 Differences Between Two Generations of Service Registry	24
4.2 Data Analysis: Web Service Selection and Integration	26

4.2.1	Correlation between Service Facets and Usages	26
4.2.2	Selecting Services from Similar Options	28
4.2.3	How Developer Proficiency Impacts Service Integration	29
4.2.4	Service Facets Vs Usages by Eliminating Prototype Repositories	33
4.3	Addressing Developers' Concerns: Insights, Evaluation and Solution	35
4.3.1	Developers' Questions Taxonomy	35
4.3.2	Evaluation of Fine-tuned Automated Tool	40
5	Implications	42
6	Future Work and Directions	44
7	Conclusion	45
8	Publications	46
	References	47

List of Figures

3.1	Methodology Workflow	9
3.2	RapidAPI and GitHub Data Collection Workflow	11
3.3	RapidAPI and GitHub Data Collection	11
3.4	RapidAPI and GitHub Statistics	13
3.5	RapiAPI Service Distr	16
3.6	RapidAPI Distributions for Customized Metrics	16
3.7	Distribution of Developer's Proficiency Levels	17
3.8	Distribution of Programming Languages	18
4.1	Workflow of Traditional Service Discovery	25
4.2	Workflow of Service Marketplaces	25
4.3	Service Facets Vs Service Usages	27
4.4	RapidAPI Service Usage on GitHub	27
4.5	RapidAPI Facets Comparison for Selected Services and their Alternatives	30
4.6	RapidAPI Facets as per Developers' Proficiency	31
4.7	Error Handling Code Patterns	32
4.8	GitHub Key Leakages	33
4.9	Error Handling and Key Leakage Example	33
4.10	Service Facets Vs GitHub Service Usages (All Repositories and Only Matured Repositories)	34
4.11	API marketplaces user concerns	36
4.12	HTTP Error code ambiguity	37
4.13	Service Invocation mechanism	38
4.14	Key Leakage code snippet example	39
4.15	RapidAPI QoS	40

List of Tables

3.1	Statistics of API Registries	10
3.2	Data Collection from RapidAPI and GitHub	12
4.1	Issue distribution in GitHub, SO, and G2.com	35
4.2	Fine-tuned Model Accuracy for Test Dataset	40
4.3	Impacted APIs Identified from Test Data	41

Abstract

Service registry, a key component of the service-oriented architecture (SOA), aids software developers in discovering services that meet specific functionality requirements. Recent years have witnessed the transition from the traditional service registries to its successor, the *Service Marketplaces*, which has led to the widespread adoption of these web services. Service Marketplaces involve deeper engagement in the SOA software lifecycle and offer additional features, such as service request delegation and monitoring of services' Quality of Service (QoS).

However, through a comprehensive study of RapidAPI web services, the largest service marketplace, and their integration into GitHub-hosted applications along with analyzing developers' concerns posted on online Q&A forums related to service marketplaces, it was found that despite the extensive use of these web services, there is a lack of a systematic approach to guide service developers in creating appealing offerings. Additionally, many developers struggle with such a transition, leading to development inefficiencies and even security vulnerabilities.

This paper presents the first empirical study that:

- Provides a powerful avenue for better understanding integration developers' rationale for selecting services from a marketplace like RapidAPI and integrating them into applications (using the GitHub platform for this analysis).
- Highlights the challenges developers face with service marketplaces due to changes in the functioning of the service registry component of SOA.
- Offers a solution to help developers address challenges related to service marketplaces.

The article initially presents a detailed comparison between two generations of service registries to identify the root causes of developers' concerns related to the new generation service registry. In the next part, the article discusses data collected on over 16K RapidAPI services and 19K GitHub repositories that invoke these services, evaluating each based on metrics like latency, reliability, pricing, followers, aggregate ratings, community support, and provider support. The analysis explores how these metrics influence service popularity and usage on GitHub. By manually analyzing 800 repositories, developers' service selection preferences and integration patterns were identified, considering alternatives and features. Further, developers were classified by proficiency

levels to understand how expertise impacts service selection and integration strategies. Additionally, insights were refined by focusing on mature repositories, excluding those used for practice.

Finally, through manual labeling and analysis of developers' questions, a taxonomy of issues was developed, summarizing the impacts of the transition, and providing actionable suggestions for app developers, service providers, and marketplaces. We also fine-tune a Large Language Model (LLM) to answer similar questions and help extract critical information, such as service outages and key leakages.

This work is the first to provide a comprehensive analysis of developer behavior and challenges in service marketplaces, particularly RapidAPI. It offers valuable insights for improving service selection and integration, ultimately enhancing the efficiency and security of SOA-based applications. By providing actionable solutions and automating support through AI, this research has the potential to significantly improve the developer experience in modern service marketplaces.

CHAPTER 1

Introduction

1.1 The Service-Oriented Architecture (SOA)

SOA is an architectural design that facilitates communication between different services over a network, promoting reusability, scalability, and flexibility by enabling services to be deployed and managed independently [1]. These services can then be discovered by developers' to be integrated into their applications as needed, providing customized solutions for various personal or business requirements [2].

A key factor driving SOA's popularity is its compatibility with business models that require seamless integration of services from different vendors. In the computer industry, SOA has long been employed to integrate applications across different platforms (e.g., Windows, Linux, or macOS). Web services, such as REST APIs, are particularly well-suited for this, as they enable systems to communicate using standardized HTTP methods (GET, POST, etc.) [3]. A practical example of SOA in action is seen in e-commerce platforms, where services like payment processing, inventory management, and customer support are integrated through distinct web services.

1.2 The Evolution of Service Registries in SOA

Service-oriented architecture (SOA) has been widely adopted in modern software systems. It provides application developers a flexible and efficient framework for accessing remote domain-specific data and computational-intensive technologies hosted by third-party service providers. Examples of such services include face recognition, text generation, translation [4, 5], and querying for real-time ticket/flight/game information [6]. With the blooming of Machine Learning as a Service (MLaaS) [7], SOA will be even more important, with an estimated market size of 26.5 billion USD by 2027 [8].

To connect service providers and application developers, service registries serve as an important component of SOA. It stores various information about available services and helps developers discover services with the required functionalities [9]. It is observed that in recent years, the service registry has experienced a major migration in terms of its functionality, from traditional **static**

service registries to the new service **marketplaces**. For example, RapidAPI is the most widely used marketplace.

Unlike traditional service registries, which direct app developers to the websites of service providers, RapidAPI offers a streamlined, one-stop solution for app developers. On RapidAPI: 1) Integration developers can use keywords to search for services; 2) when invoking a service, the request is sent to the marketplace, which authenticates the credentials of the requests and serves as a middleman delegating the requests; 3) the marketplace further measures the number of requests, the latency and successful status of each request, and displays such metadata on each service's web page; 4) based on the billing strategies specified by service providers, the marketplace handles the billing for integration developers' service subscriptions; 5) the marketplace also provides discussion forums for integration developers and service providers to communicate directly.

These differences in features led to a comprehensive analysis, which aims to address the first research question, as outlined below.

- **RQ1:** What are the differences in workflow between traditional service registries and service marketplaces?

1.3 RapidAPI Marketplace: The Ultimate New Generation Platform

In the realm of web services, service selection represents a fundamental and enduring problem. This problem can be articulated as follows: when confronted with a collection of equivalent web services offering similar functionalities and usable interchangeably, how to identify and select the service that most effectively fulfills an application's functional requirements. Existing approaches solve this problem based on various facets, including latency, cost, reliability, community support, and trust [10, 11], [12]. These approaches frequently operate under the assumption that integration developers prioritize specific service facets when selecting services. However, the extent to which integration developers prioritize these facets remains unclear, leaving service providers uncertain about which aspects of their services require improvement.

Due to their convenience and utility, the SOA ecosystem makes extensive use of marketplaces, such as RapidAPI, and BaiduAPI, which attract 16K services, over 4 million users ¹, and 400 billion invocations per month [13]. These emerging platforms also enable large-scale empirical studies. Taking advantage of marketplaces as the source of relevant service metadata, while using open-source software repositories as the source of service invocation statistics. Juxtaposing the obtained information provides a powerful avenue for better understanding integration developers' rationale for selecting and integrating services.

¹<https://rapidapi.com/company/>

In this work, an empirical approach is taken to address the problem by conducting a large-scale study. Rather than surveying and interviewing integration developers, our study derives actionable insights by studying how developers select and integrate services in their source code. To execute our empirical study at a sufficiently large scale to derive meaningful insights, the following challenges had to be overcome: 1) how to identify service invocations within the source code, as developers often utilize various libraries with diverse functionalities; 2) how to distinguish the invoked services in the absence of a standardized format, such as differentiating service from other HTTP requests; 3) how to identify and extract relevant metadata associated with services.

Data was collected from 16K services with their metadata from RapidAPI, and 19K GitHub repositories that invoke these services. The analysis first focused on how different service facets impact their popularity as measured by RapidAPI and as measured by us from mining GitHub repositories, identifying the root causes for the differences. To pinpoint the exact reasons why developers select particular services, 800 services used in different repositories were randomly picked. These services were manually labeled whether alternate services could replace them by inspecting their call sites. Additionally, integration developers were divided into categories based on their proficiency levels and studied how they differed in service selection and integration.

This part of the study makes two primary contributions. Firstly, it introduces a novel methodology for empirically studying how integration developers select services, accompanied by the public release ² of all collected data for use by fellow researchers and practitioners. Secondly, through analysis of the gathered data, the article addresses the following research questions:

- **RQ2:** What are the common characteristics shared by services managed through RapidAPI, as well as the GitHub repositories and developers who utilize these services?
- **RQ3:** How do various service facets correlate with their usage patterns, as observed on both RapidAPI and GitHub platforms?
- **RQ4:** What factors primarily influence integration developers in selecting a service from a set of equivalent options with similar functionalities?
- **RQ5:** How does the proficiency level of developers influence their practices in selecting and integrating services?
- **RQ6:** How do various service facets correlate with their usage patterns in GitHub by eliminating prototype repositories?

²https://anonymous.4open.science/r/Web_Service-_Research-66B4/

1.4 Developers' Key Concerns and Challenges

However, these workflow differences are not widely known, which could potentially impact developers using the marketplaces. Our review of inquiries on StackOverflow(SO), a leading online question-and-answer platform for software developers, uncovered numerous instances where developers were puzzled about using RapidAPI. For example, one question that has been viewed 22K times asks about failing to access a free-to-use service via RapidAPI, despite possessing a valid key³. This confusion arises from a fundamental difference in how RapidAPI operates compared to traditional service registries. Traditionally, developers receive an invocation key only after subscribing to a specific service. Conversely, RapidAPI issues a universal key that grants access to all its services, yet mandates a subscription to individual services before they can be utilized. This discrepancy highlights a knowledge gap among application developers, who are accustomed to the conventional model of service registries. Bridging this gap could not only help service providers and application developers more efficiently use marketplaces but also provide marketplace owners insights to improve their platform design.

To the best of our knowledge, although previous studies [14, 15] have summarized developers' concerns and doubts when using traditional service registries, there hasn't been any research conducted on the challenges encountered by developers when using service marketplaces. The differences in their working mechanism require a dedicated study to understand the challenges faced by developers, especially those caused by the migration. The study will also generate insights for us to train an AI-based automated tool that assists developers and researchers in mitigating these challenges. In particular, this part of the article, seeks answers to the following research questions:

- **RQ7:** What are the developers' challenges, especially those due to migration from traditional service registry to service marketplace?
- **RQ8:** How can an AI-based automated tool assist developers and researchers in mitigating these challenges?

³<https://rb.gy/59mo95>

CHAPTER 2

Background and Motivation

This section outlines the background of our research, featuring a review of relevant literature in the field. It includes an overview of service registries and the concerns developers face in using them, insights into RapidAPI as a marketplace, an exploration of existing work on service selection, and a discussion of empirical study approaches for service integration, along with available datasets.

2.1 Service Registry

A service registry is an important component of the service-oriented architecture. It was originally designed to help developers search and find the services that fit their functionality requirements. Our definition of the two generations of service registries was inspired by [16], which was published almost ten years ago. They categorized the first generation as an information gateway between service providers and consumers, and the second generation as “integrated throughout the entire software life cycle.”

Following this definition, we categorize *service marketplaces* as the second generation of a service registry, as they serve as a delegation between App developers and service providers. Three basic user roles interact with marketplaces:

- **Service Provider:** The service providers are individuals or organizations offering services to third-party users. They run the servers that host the services, which usually require authentication to be accessed. Service providers register their services in the service registry, to be used by App developers.
- **Application Developer:** App developers discover services by interacting with the service registry. They select the service that best fits their software’s requirements, develop service invocation logic, and add the authentication keys by which they pay for service usage.
- **End User:** End users are the consumers of the software, by whom the service requests are sent. End users usually have no knowledge and capability to modify the software they use.

2.2 Service Registry and How RapidAPI is Different

Traditional service registries serve as a bridge for integration developers to find services. RapidAPI, initially founded in 2015, is now the largest service registry. It takes a brand new role in service-oriented architecture as a service marketplace, which acts as a delegation between service providers and consumers. All service requests are sent to the delegation servers of RapidAPI, which further query the actual servers managed by service providers to obtain the results and return them to integration developers.

In particular, *the marketplace* acts as a '*One-stop-solution*' providing Service consumers with features like service discovery, QoS monitoring, API documentation, API service subscription, and billing all within a Platform as a Service (PaaS) offering, supporting the entire software development life cycle. [17]. *The service providers* share with the platform the invocation URL of their services, along with the documentation and example code for using the services and their price. *The integration developers* can discover the service and utilize it by subscribing to the service and managing billing on a platform itself [18].

RapidAPI consists of 49 different categories ¹ of services like Sports, Finance, Data, Entertainment, etc., and 522 *collections* ² represent a group of APIs sharing common characteristics. RapidAPI editors manually pick the services in each collection.

In RapidAPI, each service contains a cluster of endpoints that provide distinct functionalities. All the endpoints in a service share the same invocation URL and RapidAPI measures the QoS and the usages by services, not endpoints. Hence, in our data collection and analysis, we carry on the definition of services from RapidAPI and measure the performance and usages for each service, or say, a cluster of endpoints.

2.3 Service Selection Criteria

Selecting the appropriate service that aligns with the desired requirements is essential for developers to meet both functional and performance expectations. Numerous research studies [19–22] have been conducted to assist developers in recommending services or selecting optimal services, focusing primarily on evaluating or predicting QoS parameters such as availability, latency, reliability, etc. Other than QoS, cost [23] for invoking services, the level of community support for services [24, 25], and the quality of documentations [26] are also considered in service selection. Our empirical study considers all of the aforementioned criteria.

There are also other criteria for selecting services. For example, services' QoI (quality of information) [27], i.e., accuracy, updated frequency, and data completeness might be considered

¹<https://rapidapi.com/categories>

²<https://rapidapi.com/collections>

for service selection; the QoS perceived by end users may also be considered [28, 29]; the design patterns and anti-patterns of services may also be considered [30]. However, this article excludes these criteria from our study, as the QoI, end-user QoS, and design anti-patterns are hard to measure on a large scale.

2.4 Empirical Study of Service Integration

Understanding integration developers' preferences has been an important problem. There have already been multiple small-scale studies on this topic. For example, [31] surveyed less than 300 users on their opinions towards selecting one SMS service from 92 candidate services. Its result indicates that the service facets, ranking by their impact on the selection results, are 1) functionality of the service; 2) reliability; 3) cost; 3) developer support; and 4) latency. However, this rating for SMS service is ad-hoc and may not represent the general developer's preferences. Similarly, [30] interviewed 40 developers about their preferences over 3 sets of services, which still provides limited insights and might be biased. Due to the lack of proper methodology, large-scale studies on developers' preferences for selecting and integrating services in the wild are yet to be conducted.

2.5 Understanding Developers' Concerns

To the best of our knowledge, this paper is the first to study developers' questions about service marketplaces. The most related work studied developers' concerns related to traditional registries [14] by analyzing SO questions related to ProgrammableWeb and APIGuru. It outlines the taxonomy for traditional registries as "Authorization", "Function", "Documentation", and "Others". Correspondingly, a different taxonomy for the service marketplace is devised: the taxonomy includes the issues caused by the differences in their workflow (i.e., delegation), including security, usability, documentation, performance, etc.

Several works studied developers' concerns and patterns related to web service usage. Similar to our approach, many [15, 32, 33] analyzed developers questions posted on SO. Different from our approach that uses "RapidAPI" as keywords because all services need to be invoked via the uniform RapidAPI interface, they filtered the related posts either by pre-selecting services as keywords or by "service" hashtags. Some of their findings overlap with ours, such as QoS issues in services and issues related to API updates. Unlike these approaches, our work further reveals developers' issues caused by marketplaces.

Our paper explained how the shift from a traditional service registry to a service marketplace has confused developers, introduced a developers' questions taxonomy as well as developed an LLM-based AI-assisted tool to identify and address developers' concerns related to API marketplaces to ensure smoother service utilization. Similar ideas of automatically answering questions by

analyzing existing questions and answers have been widely explored [34–36], especially with the recent blooming of generative AI. This paper is the first to apply such a technique in the domain of web service, which makes it a useful tool for SOA developers and stakeholders.

CHAPTER 3

Methodology

This chapter outlines the methodology used to address our research questions. As illustrated in figure 3.1 we began by examining various service registry platforms and comparing their features in Sec. 3.1. Next, we discussed collecting and labeling data and statistics from RapidAPI and GitHub in Sec. 3.2. Finally, we explored developers' concerns regarding marketplaces by collecting, labeling and categorizing these concerns and proposed a LLM tool to mitigate these issues by training the labeled developers' concerns in Sec. 3.3 and evaluating its performance in Sec. 4.3.2.

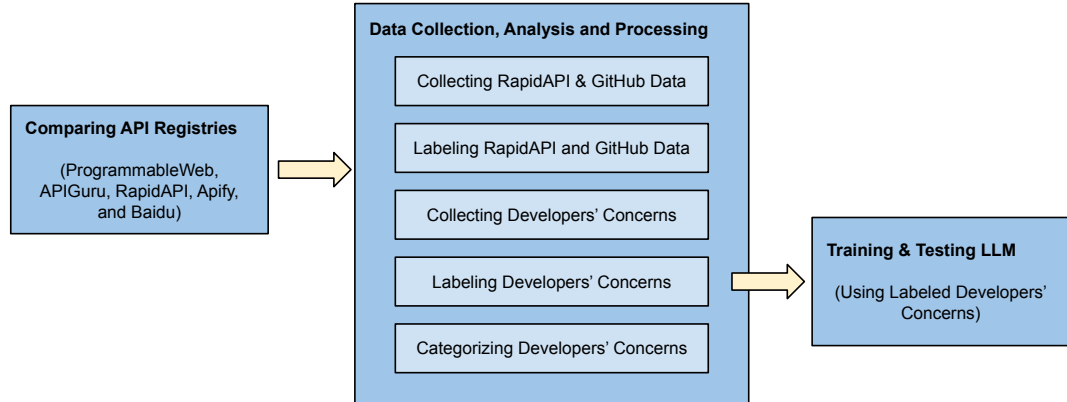


Figure 3.1: Methodology Workflow

3.1 Comparing API Registries

This section discusses the methodology for *RQI* which is related to the transition of service registry mentioned in Sec. 1.2. As illustrated by Figure 3.1, for *RQI*, the workflows of widely used service registries are discussed, categorized into two generations, and summarized the major differences in their workflows.

Features/Platforms	ProgrammableWeb	APIGuru	RapidAPI	Apify	Baidu
API Coverage	24000+ (before 2022)	2529	40000+	1558	730
Invocation Examples	No	No	19	3	6
QoS metrics	No	No	Yes	Yes	No
Request Delegation	No	No	Yes	Yes	Yes
Doc. by Service Providers	No	No	Yes	Yes	Yes
Generation	Static	Static	Marketplace	Marketplace	Marketplace

Table 3.1: Statistics of API Registries

3.1.1 Comparing Two Generations of Service Registry

The widely-used service registries were chosen, including ProgrammableWeb, APIGuru¹, RapidAPI, Apify², Baidu API³, as the targets of our study. Table 3.1 compares the statistics of these registries. RapidAPI and programmableWeb host more services than the other three.

Further, the features of these registries were explored by using them, both as service providers listing services and as App developers searching for services and invoking them. For ProgrammableWeb which has been retired, its features were explored by reading papers and tutorials. Initially, they were categorized into two groups: static service registries and service marketplaces, distinguishing them based on their common patterns, w.r.t. 1) delegating end-user requests; 2) measuring and displaying the QoS of services; 3) allowing service providers to customize documentation; and 4) providing example codes for invoking services in various programming languages. Sec. IV further analyzes the differences in their workflows in detail.

3.2 Understanding Web Service Selection and Integration

This section delves into our methodology for collecting and labeling data from RapidAPI and Github to answer *RQ2* mentioned in Sec. 1.3.

3.2.1 Data Collection and Labeling

Fig. 3.2 shows the flow for the data collection, in which *Selenium web driver* [37] was used to crawling and MySQL database for storage. This data collection process was repeated for two rounds over 8 months, to further understand how services' performance and usage change.

¹<https://apis.guru/>

²<https://apify.com/>

³<https://apis.baidu.com/>, in Chinese

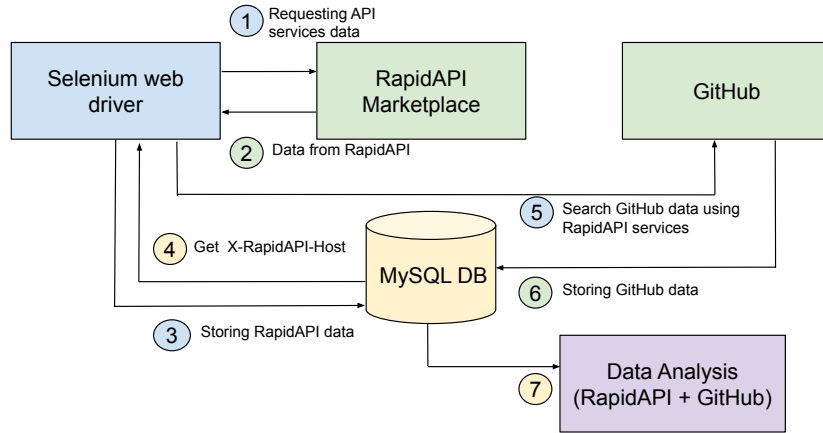
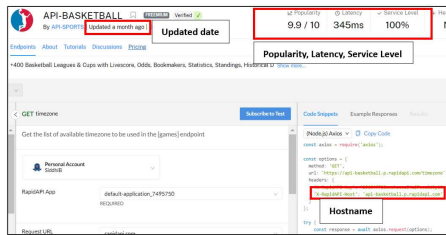
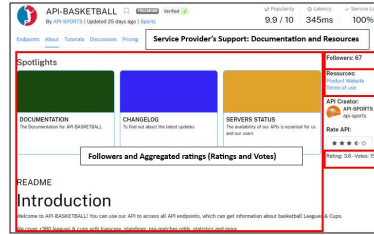


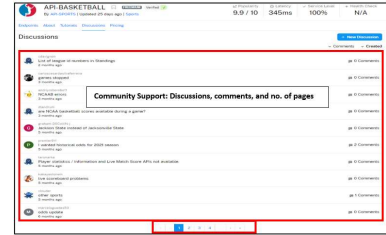
Figure 3.2: RapidAPI and GitHub Data Collection Workflow



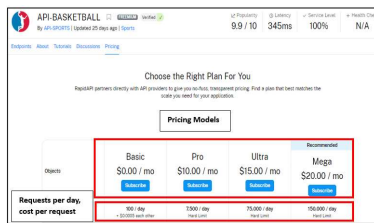
(a) Rapid API metrics



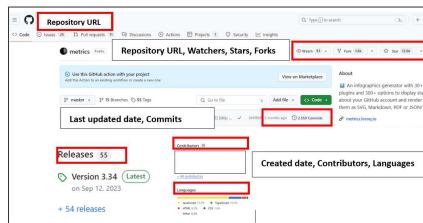
(b) RapidAPI Support (Provider)



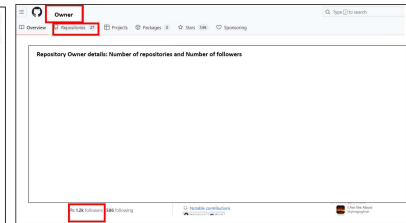
(c) RapidAPI Discussions



(d) RapidAPI Pricing



(e) GitHub Repository Details



(f) GitHub Owner Details

Figure 3.3: RapidAPI and GitHub Data Collection

3.2.1.1 Collecting Service Metadata from RapidAPI

Initially, we obtained the full list of web services from all 49 RapidAPI categories ⁴. For each web service in the list, we crawled its detail page. We obtained metrics that include popularity, latency, service level, pricing, hostname, updated date, guiding documents, and discussion forums, as indicated by Fig. 3.3. The popularity, latency, service level, and updated date along with hostname (in Fig. 3.3a) are directly measured by RapidAPI, as it delegates service requests. The guiding documents (in Fig. 3.3b) are also specified by service providers to guide integration developers, additionally we collected followers, ratings and votes. For the discussion forums (in Fig. 3.3c), we collected pages of discussions and the number of replies. The pricing (in Fig. 3.3d) is set by service providers and may have different formats. For example, developers can set the price for the monthly subscription, the quota (hourly, daily, or monthly request limits), and the per-request cost if the total requests exceed the quota. We further collected the collections ⁵, which are similar services manually grouped by editors.

3.2.1.2 Collecting Services Usages from Github

Invoking services from RapidAPI always requires specifying *X-RapidAPI-Host* of the service, which has a similar format of “service name”+“p.rapidapi.com,” with “service name” being unique to each service. For example, the RapidAPI host for “Bitcointy” service is “community-bitcointy.p.rapidapi.com.” Hence, repositories searched on Github for repositories using the *X-RapidAPI-Host* of services as the keyword, as specified by steps 4 and 5 in Fig. 3.2. We collected each repository’s metadata, including repository URLs, stars, forks, watchers, commits, contributors, languages, repository created and updated date (Fig. 3.3e) and its owner (Fig. 3.3f). We also collected each owner’s repository count and followers to measure their proficiency.

3.2.1.3 Statistics of Collected Data:

Platforms / Data collection	Round 1	Round 2
RapidAPI services	16,613	16,395
RapidAPI services used in GitHub	1,500	1,677
Repositories using RapidAPI services	10,317	19,733

Table 3.2: Data Collection from RapidAPI and GitHub

We conducted two rounds of data collection, in July 2023 and March 2024, to understand how RapidAPI services and their usage changed in 8 months. Table 3.2 displays the data collection

⁴<https://rapidapi.com/categories>

⁵<https://rapidapi.com/collections>

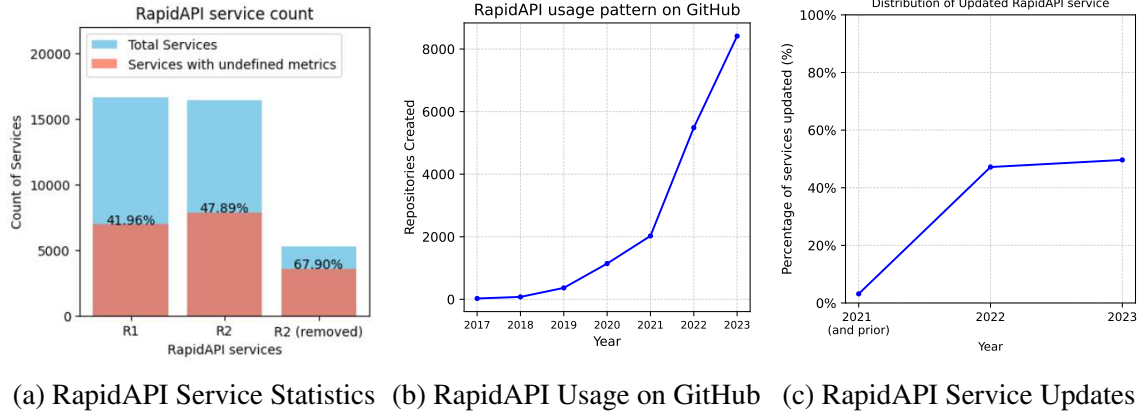


Figure 3.4: RapidAPI and GitHub Statistics

statistics gathered from RapidAPI and GitHub across two rounds. We observe that 31.60% services available in the first round were removed from RapidAPI at the time of our second round collection, with a similar amount of services added. The utilization of RapidAPI services on GitHub experienced a surprising increase of 93.6%, from 10K repositories to 19K repositories.

Figure 3.4a illustrates the statistical overview of RapidAPI services. In round 1, we gathered data from 16,613 services, the metrics of 41.96% of which turned undefined. Undefined metrics occur whenever the number of developers using a service is insufficient for RapidAPI to collect enough information about the service’s popularity, latency, and service level. Similarly, in round 2, data from 16,395 services were collected, with 47.89% of them featuring undefined metrics. Round 2 shows a 5.94% increase in services with undefined metrics, as compared to round 1. Among the 31.60% services removed from round 2, 67.90% exhibit undefined metrics.

Additionally, we analyzed the creation dates of these GitHub repositories to examine the usage patterns of RapidAPI services over recent years and updated time of each RapidAPI service to retrieve the APIs actively supported by the service providers. Our analysis indicates a noteworthy rise in the creation of GitHub repositories integrating RapidAPI services from 2017 to 2023. Fig. 3.4b illustrates that developers are increasingly relying on external APIs to enhance the functionality of their applications. By analyzing the updated time of each RapidAPI service we found that approximately only 50% of services are actively taken care by the service providers as observed in Fig. 3.4c. This research offers valuable insights for optimizing user-friendliness and fostering smoother usage patterns to accommodate an expanding user base.

3.2.1.4 Labeling RapidAPI Service Facets

The service metadata provided by RapidAPI provides numeric values for each service's latency, service level, and popularity ⁶. We further define how to label three additional facets, including pricing, support from service developers, and support from the community, as detailed below.

- **Popularity:** The popularity metric is associated with the invocations for a particular service. It is depicted by the RapidAPI platform on a scale of 0 to 10 with a granularity of 0.1, where a higher score indicates greater service utilization. The value is calculated by RapidAPI using two factors: how many developers are using each service and how many requests are sent within a certain period. However, RapidAPI's calculation algorithm is not publicly accessible.
- **Latency:** Noting that RapidAPI delegates all service requests, latency is measured as the average time (ms) it takes for a delegation server to receive a response from the API server for all calls within the last 30 days.
- **Service Level:** The service level metric indicates server reliability through successful service calls. On the RapidAPI platform, the service level is depicted on a scale of 0% to 100% with a granularity of 1%, where a higher score indicates greater service reliability.
- **Last updated:** The facet refers to the most recent date and time when an API was updated or modified. This information is crucial for developers as it helps them stay informed about changes, enhancements, bug fixes, or new features introduced to the API.
- **Followers:** The followers feature is associated with the number of users that follow specific APIs to stay informed about changes, updates, and announcements regarding the APIs. A larger number of followers could suggest that more users either utilize the particular API or are keen on staying informed about its updates. RapidAPI displays this follower count for each API. The value can range from zero to any positive integer.
- **Pricing:** RapidAPI enables service providers to specify various price models, including basic, pro, ultra, and mega, each with a distinctive pricing structure for every service. Initially, we classified the services into three categories: cost-free (services with more than 100 requests per day), limited cost-free (services with fewer than 100 requests per day), and paid services. For the limited cost-free and paid categories, we calculated a "cost per request" by 1) choosing the price model with the lowest cost, as most integration developers may choose the lowest cost; 2) dividing the cost by number of requests allowed per period. As an illustration, in the case of the service depicted in Figure 3.3d, we opt for the basic model due to its lower cost as

⁶<https://docs.rapidapi.com/docs/faqs>

it allows 100 requests per day. However, for example this model would have allowed for 24 requests per day, falling short of our set threshold of 100 requests per day. Consequently, we will consider the cost per request as \$0.0005. In cases where the cost per request parameter is unspecified, we have evaluated it by $\frac{cost}{Requestsperday}$.

- **Support from service providers:** This metric involves information about the documentation support provided by the service providers on the marketplace to assist the developers in utilizing their services. Using documentation content and resource links (see Fig. 3.3b) as parameters, we set a threshold value of documentation word count of at least 100 words, as a minimum of 100 words can adequately convey necessary details. In particular, we categorized the service providers' support as: 1) *good*, if a service has at least a 100-word count for the documentation as well as a resource link; 2) *average*, if a service has either one but not both; and 3) *poor*, if neither is available.
- **Support from the community:** This metric provides information regarding whether the integration developers of a service are forming a community to help each other. For each service, we consider how many discussion threads have at least one reply, as answering questions forms a healthier community than purely asking questions. In particular, we categorized the service community support as: 1) *good*, if 70% of discussions have replies; 2) *average* for 70% to 10%, and 3) *poor* for 10% to 0%.
- **Aggregated Ratings:** Aggregated ratings is a derived facet obtained by multiplying the ratings and number of votes facets for that API. RapidAPI ratings provide users with insights into the quality and performance of APIs available on the platform typically based on overall user experience such as user reviews, reliability, speed, documentation quality, etc. RapidAPI presents ratings for each API on a scale from 0 to 5 and the votes as number of users who have rated the API. By multiplying ratings and votes values, we created a weighted measure as aggregated ratings that considers not only the average rating but also the extent to which API is used.

Fig. 3.5 shows the service distribution based on popularity, latency, and service level, with the blue line denoting services collected in round 1, yellow for round 2, and green for services included in round 1 but removed in round 2. Note that here we excluded the services with undefined metrics. We observe that 1) the service level and popularity of services in round 1, round 2, and removed are almost identical; 2) over 70% of services removed from round 2 have high latency, making the average latency of services in round 2 smaller than round 2; 3) however, many services still suffer from poor QoS, i.e., over 20% services with reliability lower than 98% and over 25% of services with latency higher than 1,000 ms; 4) Almost 95% of services have followers count ≤ 25 .

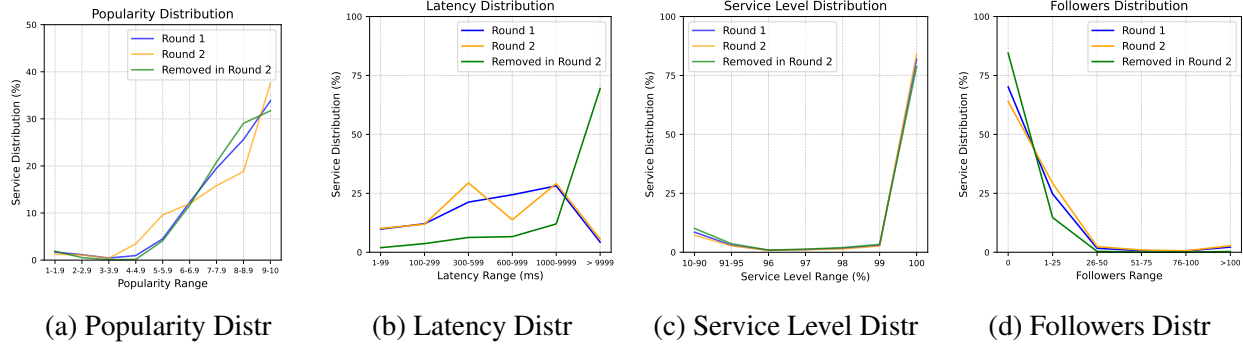


Figure 3.5: RapiAPI Service Distr

One of our unexpected findings was that many services provided insufficient QoS. A commonly shared understanding is that the QoS of services is bound by SLA (a service-level agreement). Hence, we further studied the documentation of all services, looking for the keywords “SLA” and “service level agreement.” To our surprise, among all the services collected in round 2, only 15 services mentioned SLA variations in their documentation and only 3 had legitimate SLA documents.

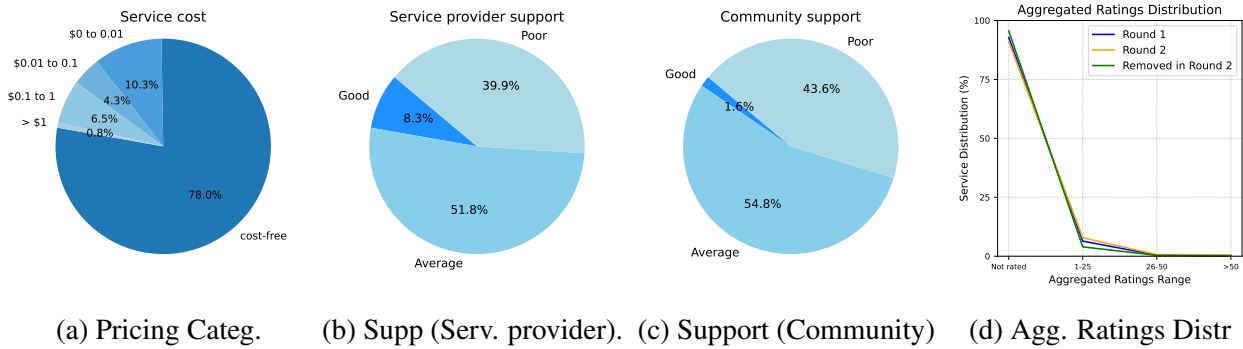


Figure 3.6: RapidAPI Distributions for Customized Metrics

Fig. 3.6 shows the service distributions based on our customized metrics, i.e., cost per request, service developer’s support, and community support. We observe that: 1) most services fall into the cost-free category, but still more than 10% services charge for more than 1 cent per request; 2) over 60% service developers provide good or average support for their integration developers; 3) only very few services have an activity community; and 4) Over 95% of services have no ratings.

3.2.1.5 Labeling GitHub Developers’ Proficiency

The developer proficiency level may impact the service selection as per their preferences. In particular, to measure the proficiency of a developer who owns a GitHub repository that integrates RapidAPI services, we consider the following three aspects: i) the number of repositories owned,

ii) the number of forks on the repository that invoke RapidAPI services, and iii) the number of followers.

As the resulting metric has multiple dimensions, we applied the following rules to classify developers into three levels:

- **Skilled Developers:** If all three aspects fall in the top 20% of their respective counts.
- **Average Developers:** If any 2 aspects fall in the top 20% of their respective counts.
- **Novice Developers:** If only one or no metric falls in the top 20% of their respective counts.

Fig. 3.7 shows the distribution of developers' proficiency levels under our definition. We observe that less than 25% of developers that integrate services in their software are classified as skilled and average, with a majority of users fall into the novice category.

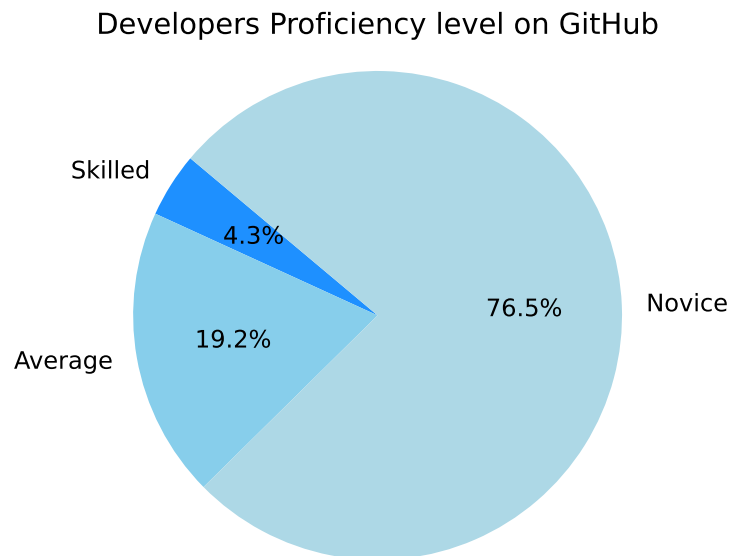


Figure 3.7: Distribution of Developer's Proficiency Levels

3.2.1.6 Labeling GitHub Service Usages

We further measured the main purposes of the repositories using services. We analyzed the primary programming language used in each repository, which is provided by Github, and categorized the purposes of the repository by the primary language, as listed below:

- Web Development, with languages being Javascript, Typescript, HTML, CSS, Jupyter Notebook, EJS, Svelte, Vue, ASP.NET, etc.

- General Purpose Scripting, with languages being Python, Ruby, PHP, Perl, Groovy, Lua, etc.
- Mobile App Development, with languages being Swift, Kotlin, and Dart.
- Backend Systems, with languages being C Sharp, Java, Go, Rust, C++, Scala, Haskell, etc.
- Database, with languages being SQL, HCL, and PLSQL.
- Document and Template Languages, with languages being Tex, XSLT, and Roff.
- Others, with languages not in the above categories.

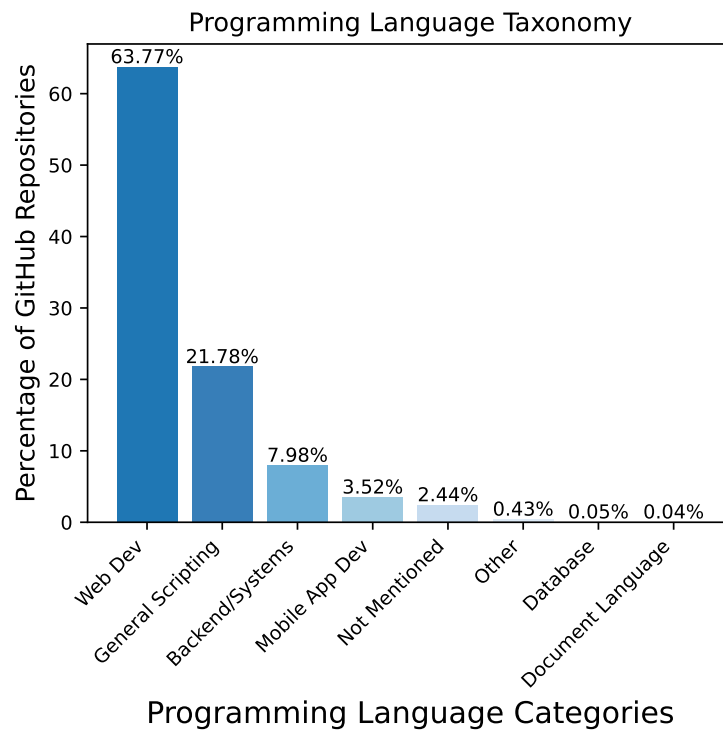


Figure 3.8: Distribution of Programming Languages

3.2.1.7 Labeling and Eliminating GitHub Prototype Repositories

We observed that some Github repositories were created by developers learning how to use RapidAPI services. Many learners create a repository, upload their example code, invoke a service, and never update the repository again. Removing such repositories from our analysis may help derive more accurate insights about service usage patterns.

Hence, we examined the GitHub repositories and divided them into two categories: All repositories and Matured repositories, depending on their activity levels. The activity level on GitHub

was determined by analyzing the number of commits and the involvement of contributors in each repository. In particular, we eliminated these prototype repositories with fewer than 5 commits and only 1 contributor.

3.3 Decoding Developers’ Questions

This section discusses data collection and the methodology for *RQ7* and *RQ8* which comprises two research questions mentioned in Sec. 1.4. As illustrated by Figure 3.1, for *RQ7*, analyzed and labeled the questions posted on various Q&A forums, and created a taxonomy. Lastly, for *RQ8*, an LLM-based tool is developed by fine-tuning and evaluating its effectiveness on a test dataset.

3.3.1 Collecting, Labeling and Categorizing Developers’ Questions

To answer *RQ7*, we collected data from online Q&A platforms and categorized them. In particular, we used these platforms’ search interfaces to find data related to our study.

3.3.1.1 Data Collection

The two most popular online Q&A platforms for App developers are SO ⁷ and the issue panel of GitHub repositories [38]. We also included data from G2 ⁸, a platform for developers to publish reviews related to software and services. Among the three marketplaces we mentioned earlier, we chose to only study questions related to RapidAPI and Apify, as Baidu API is targeted for Chinese software developers and we cannot find questions about it in English-speaking online communities.

To get the most recent questions related to our topic, we used “RapidAPI” and “Apify” as keywords to search on these online communities. Altogether, we collected approximately 2000 random questions.

3.3.1.2 Data Categorization and Labeling

We conducted a manual examination of each question to gain a comprehensive understanding of developers’ doubts and concerns. Three of the authors, each with 3 to 5 years of industrial software development experience, individually reviewed each question and identified recurring patterns. Beyond just summarizing developers’ doubts and concerns, we also inspected the questions for technical issues overlooked by developers and delved into the root causes.

As the next step, we conducted a group discussion to reach an agreement on the patterns, root causes, and impacts of developers’ questions. We applied majority voting to resolve the

⁷stackoverflow.com

⁸g2.com

disagreement in our opinion. We assigned names to each category of questions and labeled the questions themselves accordingly, aligning them with the commonly observed issues.

Upon categorizing the data, it was clear that some questions—such as those about coding issues with various HTTP libraries and frameworks, project ticket trackers created by users to track their work progress, and discussions not pertinent to API marketplaces—did not align with the focus of our research. To determine the relevance of a question, we employed a straightforward criterion: would a developer encounter the same issue using a traditional registry? If the answer was affirmative, we deemed the question irrelevant and excluded it from our analysis. This filtering process resulted in a refined dataset comprising 103 records.

Among all the questions we collected, a majority of them (95%) are related to simple coding, e.g., “*Can’t use RapidAPI with Retrofit (HTTP client for Android and Java)...*” Given that many software developers first encounter HTTP requests early in their careers [39], the prevalence of fundamental coding inquiries suggests that a substantial number of RapidAPI users are novice software engineers. This observation underscores how marketplaces like RapidAPI empower beginning developers to incorporate complex functionalities into their applications, democratizing access to sophisticated software development tools.

3.3.2 Developing an LLM Tool to Answer Developers’ Questions

After categorizing the data, it was discovered despite the variety of ways developers pose questions, the underlying issues often fall into a few distinct categories. This revelation led to answering *RQ8* by utilizing the categorized questions, along with their identified root causes and proposed solutions, to fine-tune a Large Language Model (LLM). The enhanced LLM can now use the insights from this analysis to answer developers’ questions with greater accuracy. Additionally, it has the capability to sift through developers’ inquiries to track performance problems related to specific APIs and detect potential key leakages. For instance, if the refined model spots a question highlighting performance issues with a certain API, it can flag this to the attention of either the service provider or the platform, facilitating a more proactive approach to issue resolution.

3.3.2.1 Data for Training the LLM Tool

As 95% of all questions are general questions related to coding, we picked 7 questions from the ones we removed, representing different types of coding-related issues, and added them back to form 110 questions as inputs for fine-tuning.

Following the openAI guides ⁹, all 110 questions are put into a single file, to be fed into the LLM fine-tuning interface. Each record in our dataset is structured according to a “role-completion”

⁹<https://platform.openai.com/docs/guides/fine-tuning/preparing-your-dataset>

format, which is divided into three distinct components: system role, user role, and assistant role. The system role is specifically designated as "Act as an API marketplace expert to address user concerns effectively," positioning it to function in an assistant capacity. Within this framework, the inquiries posed by developers are categorized under the user role, illustrating the perspective from which questions are asked. On the other side, the role of the "Assistant" is assigned to encapsulate the expected LLM outcomes, encompassing both actionable recommendations in response to the developers' queries and pertinent information aimed at API marketplaces and service providers. This role-completion method is designed to simulate a realistic interaction scenario, where the LLM adopts the role of an informed assistant, providing targeted advice and insights based on the user's (developer's) inquiries and the predefined system role's expertise.

3.3.2.2 Training the LLM tool

As we mentioned earlier, we expect the fine-tuned model to 1) answer developers' questions automatically without requiring human developers to get involved; and 2) extract useful information from these questions. Hence, for each of the questions in the training set, we created answers (assistant role) following the procedure given below:

- Manually labeling each question: in particular, we labeled the following information for each question: 1) what category/subcategory it belongs to; 2) which service is mentioned; and 3) if an invocation key is exposed.
- Adding solutions to the answer: we created general suggested solutions for each category and subcategory of questions, and formed the answers using these suggestions. For example, for the example question given in the introduction section and other questions with the same root cause, we give the same suggestion of "the problem is caused by RapidAPI requires to subscribe to a service before invoking it, even if the service is free. To solve this problem, login to RapidAPI and subscribe to the service you are using."
- Adding information to be extracted to the answer, including its service and whether an invocation key is exposed, using the Chain-of-Thought (COT) prompts.

CoT prompts generate intermediate reasoning steps before arriving at a final answer, mimicking human problem-solving processes. It enhances the model's ability to tackle complex questions by making its reasoning transparent and interpretable, thereby improving the accuracy and reliability of its outputs. When generating inputs, we follow the CoT mechanism by giving the logic behind how we reason the information from questions, which helps LLM accurately extract information from differently formatted developers' questions. To demonstrate the difference between prompting with and without CoT, we give two examples below.

1) Without CoT:

Listing 3.1: Without CoT

```
User: Authentication never works... fetch(https://covid-19-data.p.rapidapi.com
/totals, {method: GET, headers: { x-rapidapi-key: keyHiddenInManuscript, x
-rapidapi-host: covid-19-data.p.rapidapi.com}}).then(response => {console.
log(response);}).catch(err => {console.error(err);}); will return 403 {
message: You are not subscribed to this API
```

```
Assistant: This is a Usability issue because it mentions in the user question
- You are not subscribed to this API and 403. \n Please subscribe to the
API. The API is X-RapidAPI-Host: covid-19-data.p.rapidapi.com. Please hide
the key to avoid security issues. The code-practice is not good as X-
RapidAPI-Key keyHiddenInManuscript, is exposed causing security issue.
```

2) With CoT:

Listing 3.2: With CoT

```
User: Authentication never works... fetch(https://covid-19-data.p.rapidapi.com
/totals, {method: GET, headers: { x-rapidapi-key: keyHiddenInManuscript, x
-rapidapi-host: covid-19-data.p.rapidapi.com}}).then(response => {console.
log(response);}).catch(err => {console.error(err);}); will return 403 {
message: You are not subscribed to this API.}
```

```
Assistant: The user question has a usability issue, noting that the user has
an error You are not subscribed to this API. The problem is caused by
RapidAPI requiring to subscribe to a service before invoking it, even if
the service is free. To solve this problem, log in to RapidAPI and
subscribe to the service you are using. In the analysis, it is recognized
that the X-RapidAPI-Host is covid-19-data.p.rapidapi.com, and it is noted
negatively that the X-RapidAPI-Key is exposed as keyHiddenInManuscript,
indicating wrong security practices, User must hide the API key. The
exposed key is a 50-character mix of numbers and lowercase letters, which
is a security concern. However, the main concern is regarding API
subscriptions. To address this, it is suggested user should generate a new
, valid API key. Additionally, it's advisable to double-check the header
parameters and API subscription to ensure they are correctly configured.
These actions could potentially identify the API subscription problem,
addressing the usability category issue effectively.
```

In particular, we chose gpt-3.5-turbo-1106 among all other models due to its large token limit of 16K (1 token is 4 words) for each training input stream. Being able to preserve more question

details enhances contextual comprehension and optimizes model performance. The runtime for fine-tuning 110 records is almost 15 minutes with an epoch size of 3. We run the fine-tuning two rounds, one without CoT and one with CoT. The cost for two rounds of training is minimal, i.e., around \$10.

3.3.2.3 Test Dataset Collection for Evaluation

To test the performance of our fine-tuned LLM, we further retrieved additional SO questions from a historical dataset hosted on Google BigQuery for test dataset generation. Different from our training dataset which was partially collected from the search interface of SO, we can query all SO questions using BigQuery without being limited to the first 500 results as in the search interface. We removed the overlapping data and still got around 1,000 questions left, from which we selected the most recent 100 records for testing. We applied these 100 questions to our fine-tuned model and recorded the responses. As the ground truth, we further manually labeled their categories/subcategories, as well as other information like the impacted service names and key leakages.

CHAPTER 4

Results

This chapter provides answers to our research questions *RQ1*, *RQ3* to *RQ8*. It covers the workflow differences between two generations of service registries in Sec. 4.1, data analysis for web service integration, and selection criteria in Sec. 4.2. Additionally, it addresses developers' concerns resulting from the transition in service registry workflows providing some insights and evaluation of the proposed LLM tool designed to address and mitigate these challenges faced by developers in Sec. 4.3.

4.1 Static Service Registry Vs Marketplace Workflow

This section provides a detailed answer to the *RQ1* mentioned in Sec. 1.2, followed by actionable suggestions to web service stakeholders, including service providers, App developers, and marketplaces. Below we first briefly summarize the answers to the RQs.

- *RQ1*: Marketplace differs from the static registry in terms of service subscription, service registration, service invocation, billing, and service QoS monitoring.

4.1.1 Differences Between Two Generations of Service Registry

Fig. 4.1 and 4.2 describe the workflows of the traditional service registry and service marketplace, respectively. In both figures, we use *green* to denote the workflow of service selection and *orange* to denote service invocation. We observe that the workflows are very similar. For service selection, service providers first register their services, and App developers search for services. App developers then subscribe to the service they select and distribute the invocation key with software to end users. For service invocation, end users use the invocation key to send service requests.

4.1.1.1 Workflow Comparison

We compare the workflow of two generations in detail.

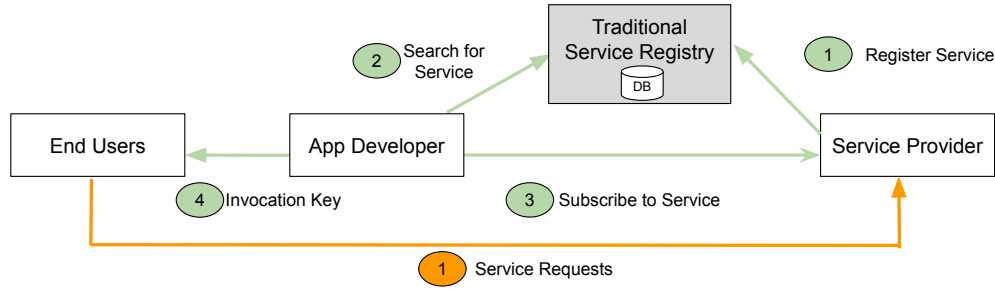


Figure 4.1: Workflow of Traditional Service Discovery

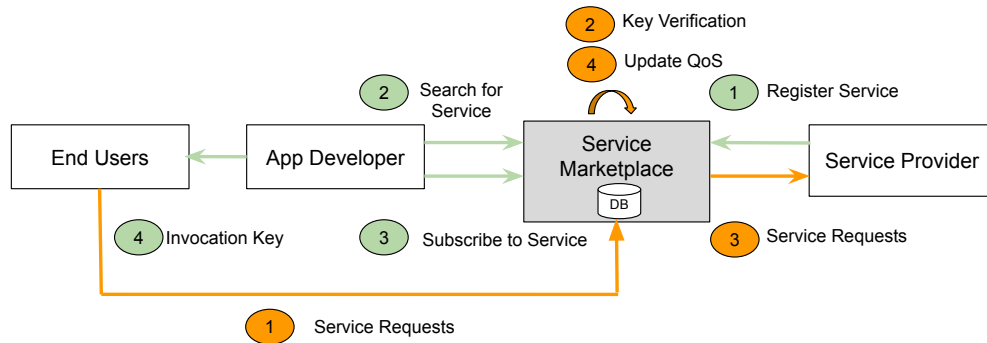


Figure 4.2: Workflow of Service Marketplaces

1. **Service Subscription:** Service subscription is the main difference in the service selection workflow between both generations. While both generations display service information on the registry, the traditional registry redirects developers to the service provider's website, for further purchasing and subscription. In contrast, marketplaces provide a one-stop experience for service selection, the service developers are supposed to fill in all descriptive information about their services on the marketplaces, and the App developers will subscribe to the service on the marketplace and also pay to the marketplace.
2. **Service Registration:** In a service marketplace, the process of subscribing to APIs differs from that of a service registry. Developers are required to subscribe to the necessary API directly within the marketplace itself. Therefore, the marketplace must provide comprehensive information. Without such information, users may not be directed to the service provider's website
3. **Service Invocation:** The service invocation requests for traditional service registries are sent to the service provider's servers, but for service marketplace requests are sent to servers owned by marketplaces. The marketplace then delegates the request from the end users to the service providers.

4. **Billing:** In a traditional service registry, service providers handle billing before a request is invoked. However, the billing process differs in marketplaces like RapidAPI. Here, users can subscribe to different APIs offered on the platform and are billed on the marketplace itself according to the pricing plans established by the API providers. Charges are determined by factors such as the volume of API requests and the specific endpoints accessed.
5. **Service QoS monitoring:** QoS monitoring is introduced in the service marketplace and involves tracking various parameters such as popularity, latency, and service level. When the end user invokes a service, they send a request to the service marketplace along with an API key. The service marketplace then verifies the API key and forwards the request to the service provider. Following this, it proceeds to update the Quality of Service (QoS).

In summary, the static service registry helps developers find services but does not play a role in the subsequent service invocation, while the service marketplace is involved throughout the entire software lifecycle of SOA applications. The distinction in workflow between traditional service registries and service marketplaces, particularly in the service marketplace's role as an intermediary between App developers and service providers, has raised concerns among developers. With millions of developers now relying on service marketplaces, it's imperative to address and resolve these commonly faced issues so that platform providers can cultivate trust with users, promote favorable user experiences, and cultivate enduring relationships that will strengthen the platform's success and expansion.

4.2 Data Analysis: Web Service Selection and Integration

This section details the data analysis we employed to address *Research Questions 3 to 6*.

4.2.1 Correlation between Service Facets and Usages

First, we aim to answer *RQ3*. The popularity of RapidAPI is indicated by both the number of developers utilizing the service and the frequency of invocations made to those services. We correlated this metric by examining the number of repositories utilizing each service on GitHub, terming it as the average repositories per service to evaluate the usage of each RapidAPI service on GitHub.

Fig. 4.4 illustrates the relationship between the average RapidAPI popularity of all services and the average number of repositories on GitHub utilizing these RapidAPI services. We observe that as RapidAPI popularity increases, so does the number of average repositories on GitHub. This correlation indicates that GitHub can serve as a suitable platform for studying the RapidAPI usage patterns. However, for the services used in more than 40 repositories, we observe a slight decrease



Figure 4.3: Service Facets Vs Service Usages

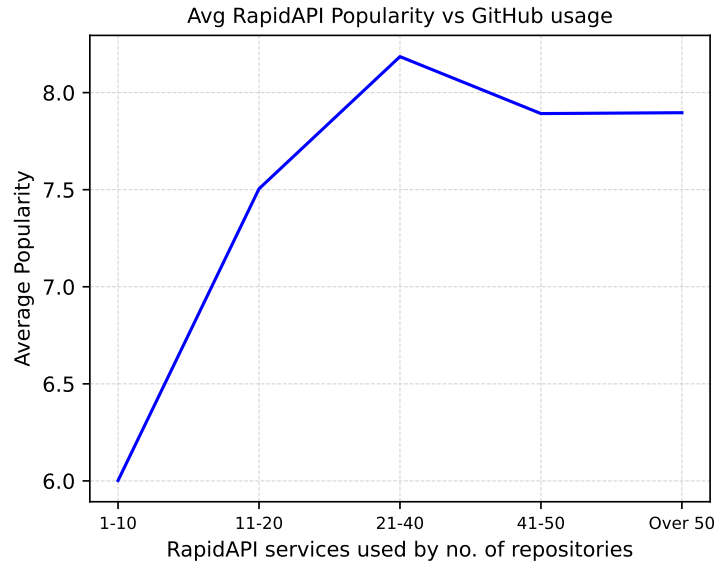


Figure 4.4: RapidAPI Service Usage on GitHub

in their average popularity, which may be caused by not so many end users using these repositories and sending fewer requests, as RapidAPI popularity is measured by both the number of subscribers and the invocation frequency.

Further in Fig. 4.3 We compared various RapidAPI service facets and analyzed their impact on service usage. As a general trend, we observe that higher service level (reliability), lower cost, better support from service providers and the community, and good- rated APIs attract more developers,

as indicated by Figs. 4.3b, 4.3c, 4.3d, 4.3e, and 4.3g which is not surprising.

Other observed insights are as follows:

- Latency (see Fig. 4.3a): 1) extremely low latencies of 1 to 99 *ms* may not increase the service's popularity, as confirmed by both RapidAPI and Github; 2) different impacts of latency on service usages are observed, i.e., Github developers favor services with less than 1s of latency, while RapidAPI indicates developers favor services with less than 10s of latency.
- Reliability (see Fig. 4.3b): as confirmed by both RapidAPI and Github, developers tend to prefer services that offer good reliability, typically in the range of 97% to 99%, rather than those with perfect reliability of 100%, which is against the common assumption that services need to guarantee the reliability of several nines. One possible reason causing this phenomenon could be that, as more developers use a service, sending their requests, the service's reliability decreases. Or we can say that instead of developers choosing services with lower reliability, developers' choices cause services' reliability to decrease.
- Support from Service Providers (see Fig. 4.3d): the correlations between support from service providers and service usages measured by GitHub and RapidAPI differ. Services with little or no documentation and no resource URLs may be more popular than services with either of these properties present, as observed on RapidAPI. Finding a plausible reason that explains this phenomenon remains an open research question.
- Followers (see Fig. 4.3f): the relationship between follower count and service usage, as assessed by GitHub and RapidAPI, displays variations. While there's a notable surge in GitHub repository counts for follower counts exceeding 50, there's a slight decline observed for follower counts ranging between 76-100. Surprisingly, the popularity decreases as follower count increases beyond 25, reaching its lowest point within the 76-100 range, before rising again for counts greater than 100. The explanation lies in there are only 0.6% of services with follower counts ranging from 76 to 100 and that too exhibiting low popularities.

4.2.2 Selecting Services from Similar Options

Here we delve into a comprehensive comparison between the services selected by developers through RapidAPI and potential alternatives that were not chosen. The aim is to examine the facets of RapidAPI that significantly influence developers' choices when choosing a service among alternatives with similar functionalities.

Three co-authors manually inspected how services are integrated into Github repositories. By calling RapidAPI-provided functions, they randomly selected 800 GitHub repositories, from which they identified approximately 382 unique RapidAPI services, all of which they inspected manually.

To identify alternative service options for services utilized within each GitHub repository, we adopted the following procedure: 1) we inspected the input/output parameters for each service in use; 2) to help with manually labeling alternative services, we filtered out candidate services that fall into the same collection with the service in use, and appended the candidate service set using keyword search on RapidAPI; 3) if at least two of the three manual inspectors agreed that a service in the candidate set can take the inputs of the service in use and generate outputs with similar physical meaning, we marked the service as an alternative for the current service in use. Out of 382 services analyzed, we identified 332 as having potential replacements.

We further analyzed each RapidAPI facet distribution for 332 services with potential replacements and compared them with the possible alternatives. The facets analysis is divided into 3 categories:

- Facets with no impact: Service level and pricing show similar trends for the selected and alternative services as in Fig. 4.5c 4.5d. For pricing, Adam Smith’s “invisible hand” [40] can influence the pricing strategies of competing services, as each market consistently provides more appealing pricing to attract customers. The pricing graph indicates a similar outcome, with the selected and alternative services reaching nearly identical pricing.
- Facets with minor impact: The popularity and latency as shown in Fig. 4.5a 4.5b may have a minor impact while selecting services from a set of similar options. Selected services exhibit comparatively higher popularity, alongside low-latency performance, specifically those with latencies below 1 second.
- Facets with significant impact: The support provided by service providers and the community significantly influences developers, as illustrated in Fig. 4.5e and 4.5f, where it is evident that most selected services enjoy better support from both service providers and the community compared to the alternative services.

4.2.3 How Developer Proficiency Impacts Service Integration

We analyzed the developers’ categories as per their proficiency levels in Sec. 3.2.1.5 labeled as “Skilled”, “Average”, and “Novice” developers. In particular, we studied how developer proficiency impacts two aspects of service integration: 1) service selection; and 2) error handling. We paid special attention to error handling, driven by the finding of many popular services being afflicted by high latency and low reliability.

1) **Service Selection:** Although services selected by skilled developers slightly outperformed as compared to those selected by average and novice developers in terms of service level, pricing,

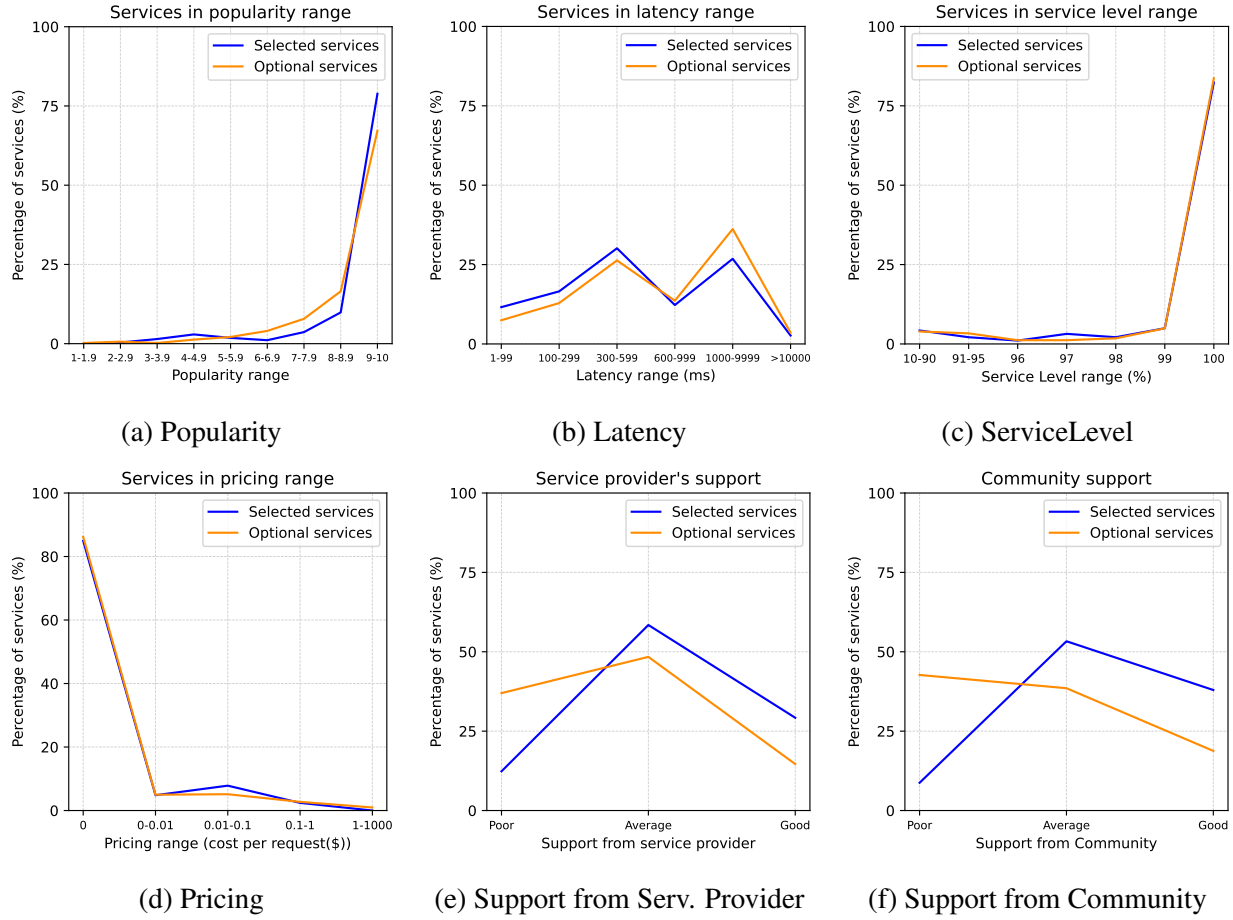


Figure 4.5: RapidAPI Facets Comparison for Selected Services and their Alternatives

and community support (Fig. 4.6c, 4.6d and 4.6f) the difference between them was not particularly evident.

As per Figure 4.6b, the latency aspect reveals that skilled developers prefer integrating services with lower latencies, typically ranging between 1 millisecond to 1 second, compared to average and novice developers. Moreover, beyond the 1s threshold, the usage of services by skilled developers declines, compared to their average and novice developers. This finding indicates that skilled developers recognize the importance of fast responsiveness when selecting services.

Fig. 4.6e illustrates both skilled and average developers consider service provider's support like good documentation details, resource links, tutorials, and similar resources when selecting services. It might seem surprising, but skilled developers highly value documentation quality. Experienced developers might view good documentation as a sign of solid software engineering practices. Essentially, well-made software often has detailed documentation. While experienced developers may not always need the documentation for integration, they like to see it as evidence of well-crafted services.

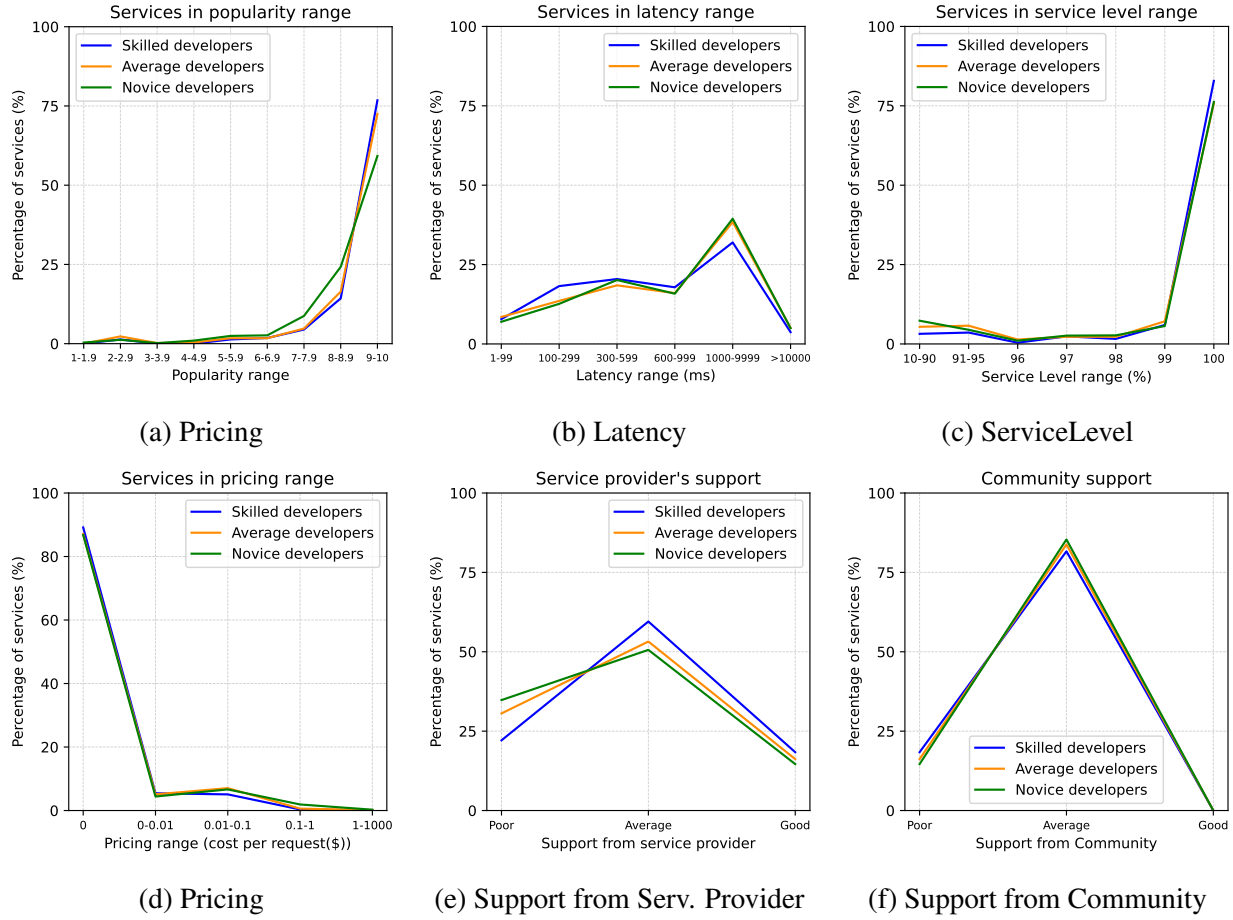


Figure 4.6: RapidAPI Facets as per Developers' Proficiency

Skilled and average developers also prefer the popularity facet distribution, as shown in Fig. 4.6a. One explanation for this finding is that services gain popularity as skilled and average developers select them rather than some services possessing high popularity upfront.

2) Error handling: Our analysis, as depicted in Fig. 4.6b and 4.6c, indicates that many services are impacted by high latency and low reliability. Without proper error handling, an App may be frozen, waiting for services' responses or even crash if the request is not successful. Many HTTP libraries have a default timeout (e.g., 5 minutes for Python *requests*). The default timeout may be inappropriate for services, as some of which may take an unusually long time to execute, while others need to be retried as soon as their execution fails to meet a shorter latency. Hence, developers must realize the importance of error handling in service invocations and implement proper strategies such as retries after failures and customized timeout.

To understand the error handling patterns followed by the developers, we manually inspected the call site of the 800 repositories in GitHub, which we selected randomly to study developers' service selection preferences. We observed the following distribution in error handling, as shown in

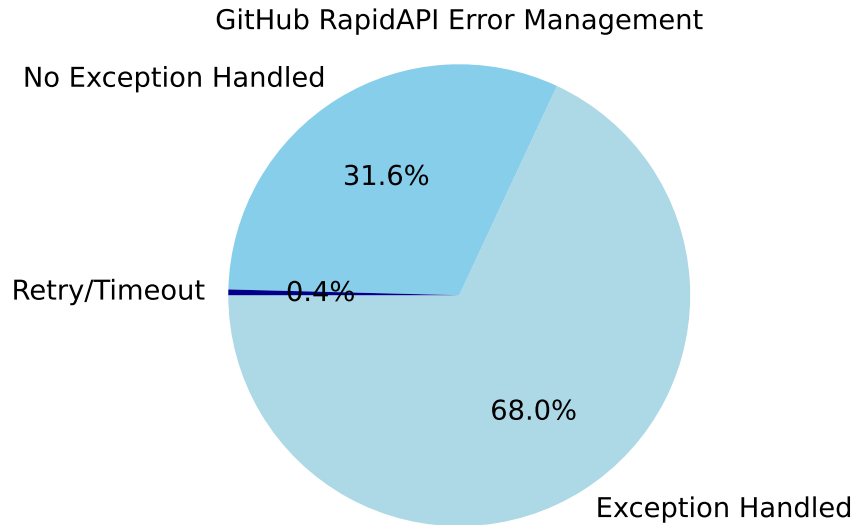


Figure 4.7: Error Handling Code Patterns

Fig. 4.7:

- No exception handling, 31.6% of repositories.
- Basic exception handling, such as usage of try-catch blocks or providing error details by logging HTTP status codes (e.g., 404, 500) or other customized messages, which comprises 68% of repositories.
- Proper exception handling, such as timeout and retries, which comprises only 0.4% repositories.

3) Key Leakages:

Leaving API keys in code poses significant security risks, allowing unauthorized access to sensitive resources and potentially leading to data breaches or financial loss [41]. During a manual inspection of the call site in GitHub, we discovered that numerous developers have inadvertently left their RapidAPI keys exposed in their code. To address this issue, we developed a codebase to identify key leakages across all collected GitHub repository URLs. Additionally, we conducted further investigations by examining GitHub commit logs and Q&A discussions to confirm instances of key exposure. Our analysis revealed that approximately 3.74% of GitHub repositories were impacted by key exposure, affecting around 11% of RapidAPI services.

Fig. 4.8 depicts the key leakages identified from GitHub repositories, commit logs, and Q&A. It is observed the majority of contributions are from the repository code base where developers commonly hard-code the key. This is followed by contributions from Q&A, where developers post queries and commit logs, which document the history of committed versions of the code-base.

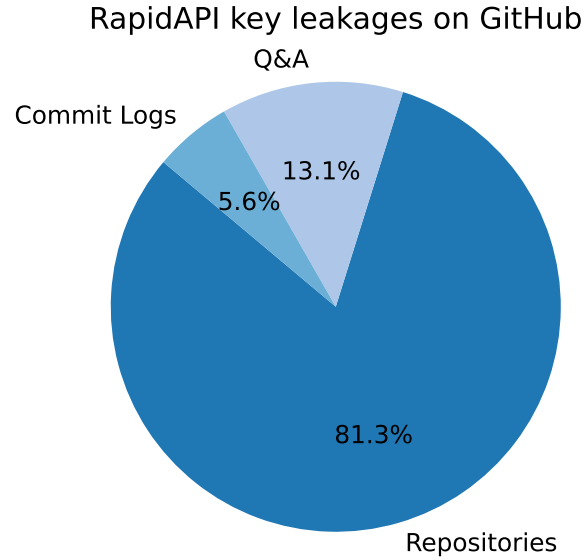


Figure 4.8: GitHub Key Leakages

```

1 headers = {
  "X-RapidAPI-Key": "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
  "X-RapidAPI-Host": "workout-planner1.p.rapidapi.com",
}

2 response = requests.get(url, headers=headers, params=querystring)
  data = response.json()

  print(data)

if __name__ == "__main__":
  main()

```

Figure 4.9: Error Handling and Key Leakage Example

Figure 4.9 shows a code snippet extracted from a GitHub repository utilizing the RapidAPI service. In the first highlighted part, the developer has hard-coded the API key into the code, this practice is observed in 3.74% of repositories where keys are exposed. The second highlighted part developer does not employ any exception handling or code retry/timeout mechanisms in the API call, this practice is observed in 32% of the repositories.

4.2.4 Service Facets Vs Usages by Eliminating Prototype Repositories

We filtered the prototype repositories by considering commits and contributors parameters of GitHub as discussed in Sec. 3.2.1.7. We eliminated 14.69% of prototype repositories, which were created merely for experimental or practice purposes. This section expands upon the discussion

initiated for RQ3 in Sec. 4.2.1, examining the correlation between different RapidAPI facets and their respective usages. With a slight modification involving the elimination of prototype repositories, the aim is to ensure that insights are derived from relevant and matured data.



Figure 4.10: Service Facets Vs GitHub Service Usages (All Repositories and Only Matured Repositories)

Figure 4.10 illustrates service usages based on RapidAPI facets for all repositories and for matured repositories i.e. excluding prototype repositories. A consistent pattern of developer preferences is noted for reliability, pricing, support from service providers, and community support, as observed in Figures 4.10b, 4.10c, 4.10d, and 4.10e for both all repositories and matured repositories. Other observed insights are as follows:

- Latency (see Fig. 4.10a): The average number of repositories remains relatively consistent across most latency ranges, except for the range of 100-299 ms which shows the lowest value. The repository count is lower for such a low latency value because 84.18% of them are updated repositories that were last updated more than six months ago.
- Reliability (see Fig. 4.10b): While there is a certain level of preference for low-reliability services ranging from 10% to 97%, the majority of preference is for high-reliability services of 98% and above.
- Followers (see Fig. 4.10f): The trend in the number of repositories, with an increase in followers, exhibits a similar pattern for both categories: an initial increase, followed by a

decrease, and then another increase. This suggests that the follower count facet might not be significantly impactful.

- Aggregated ratings (see Fig. 4.10g): The number of repositories is higher for those that are rated compared to those that are not. The decline observed for aggregated ratings greater than 50 can be attributed to the fact that while the ratings are high, the number of votes is relatively low. This suggests that the ratings and votes facet of RapidAPI may have some impact on service usage.

4.3 Addressing Developers' Concerns: Insights, Evaluation and Solution

This section discusses actionable suggestions to web service stakeholders, including service providers, App developers, and marketplaces. Below we first briefly summarize the answers to the RQs.

- RQ7: The taxonomy is quite different from that of a static registry, with more issues related to usability, security, documentation, and performance.
- RQ8: A LLM-based model fine-tuned with the labeled data can achieve 85% accuracy for answering developers' questions, 88% accuracy for identifying service names, and 77% accuracy for detecting key leakages.

4.3.1 Developers' Questions Taxonomy

The outcome of our effort is a taxonomy of developers' concerns (Fig.4.11), as well as a few observations that could benefit stakeholders including the marketplace platform, service developers, and App developers.

Table 4.1 describes the issue count from platforms like GitHub, SO, and G2.com. We observed Documentation or general help/support, as well as performance-related inquiries, are predominantly found on GitHub. On the other hand, usability concerns and key leakages in the queries posted by users tend to be noticed on SO. Next we introduce these categories in detail.

Platforms	General Queries	Usability	Performance	Security
GitHub	29	12	15	3
SO	-	20	8	12
G2.com	-	-	4	-

Table 4.1: Issue distribution in GitHub, SO, and G2.com

1) GENERAL QUERIES (Help/Support):

This category encompasses issues regarding assistance related to services on the marketplace.

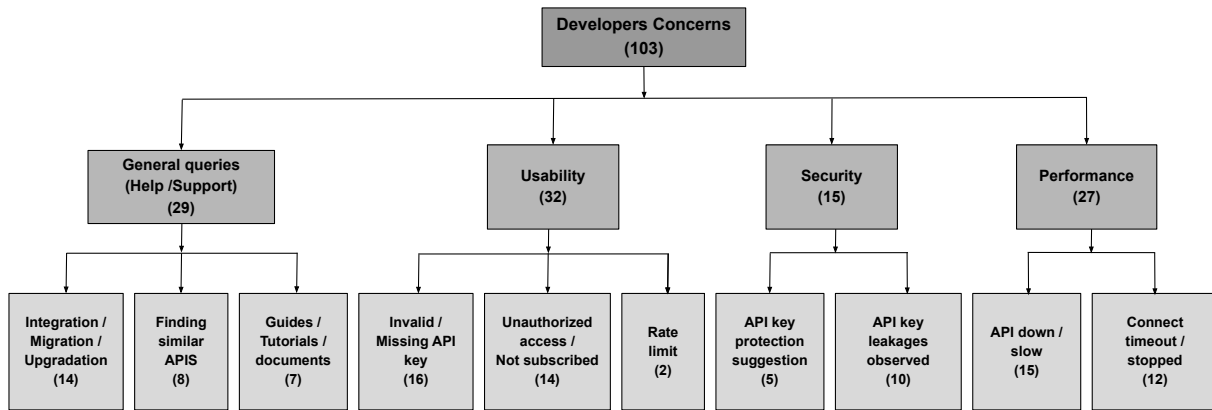


Figure 4.11: API marketplaces user concerns

- *Integration/Migration/Upgradation:* This subcategory involves incorporating the API into an existing system or application, transferring the API from one environment to another, and updating the API to a newer version.
- *Finding Similar APIs:* In this subcategory, developers want to locate other APIs within the platform that offer comparable functionality, services, or features to a specific API. This allows users to explore alternative options or identify additional resources that may better suit their needs.
- *Guides/ Tutorials/ Documents:* This subcategory involves users seeking help with some uncertainties, enhancing understanding, and acquiring practical knowledge on utilizing APIs effectively.

An example of such issue is “(where can I find) Documentation for API TMDB?”

As indicated in the table 3.1, while being able to offer customized documentation in the service marketplace, many service providers still fail to provide adequate documentation concerning endpoints, integration methods, and other details. Consequently, App developers express concerns about the need for API documentation.

Actionable Suggestions: According to our analysis of RapidAPI data, only 56% of RapidAPI services include listed resource links containing information about the API product, while 22% of services provide documentation details outlining service endpoint information, integration steps, and more. The platform should monitor APIs lacking sufficient documentation support by offering resource links, documentation, and tutorials for the service. It should also notify API providers to improve their documentation by incorporating detailed information on service endpoints, API integration methods, and more.

2) USABILITY:

Usability issues are associated with problems or difficulties that users encounter when interacting with a product, system, or interface.

- *Invalid / Missing API key:* An Invalid API key error occurs when the API key provided by the user is either incorrect or expired and a "Missing API key" error occurs when the user fails to include the required API key in the request.
- *Unauthorized access / Not Subscribed:* The usability issue "Unauthorized access" or "Not Subscribed" typically occurs when users attempt to access certain features or resources within an API without proper authorization or subscription status.
- *Rate limit:* Rate limiting is a mechanism APIs use to control the number of requests made by a client within a specific period. Within Rapid API, every API service is bundled with pricing packages, each of which provides a specific allocation of requests either on a daily or monthly basis.

1. 403 error code representing Forbidden

I am currently making an app that fetches the data from rapid API. but I am getting the error :

```
retrofit2.HttpException: HTTP 403 Forbidden
```

I don't know what's the problem, if anyone know the solution please help me.

This is the get request I am making :

```
interface ApiService {  
  
    @GET("product/search?")  
    suspend fun getResult(@Query("keyword")keyword: String, @Query("country")country: St  
    @Header("x-rapidapi-key")apiKey: String, @Header("x-rapidapi-host")apiHost: String):  
  
}
```

2. 403 error code representing Not subscribed to API.

I am using vite, react, netlify and rapidapi for my project. All requests and functionalities were working perfectly when i was working on localhost and vscode. But when I deployed to the netlify, I ran into error which states e.slice is not a function. While debugging, i found out that in network tab in developer tools, the response from the rapidapi was code 403, and the message that you are not subscribed to the api.

Figure 4.12: HTTP Error code ambiguity

Fig. 4.12 shows two example questions falling into this category. These examples show App developers are often confused by the resulting return codes, wondering whether the problem lies with the HTTP library usage, the platform itself, or the service provider. For instance, a 403 error code may be caused by errors in crafting the request URL sent to the marketplace, key authentication failures on the marketplace, or the service provider's server going down.

3. Service invocation mechanism in service marketplace.



Figure 4.13: Service Invocation mechanism

Fig. 4.13 shows an example question related to the new subscription model. As explained in Sec. 4.1.1.1, static registry redirects App developers to service providers' platforms, where developers subscribe to the service and get invocation keys. In the marketplace, developers are granted an invocation key by default. However, to invoke services, they need to further subscribe to these services on the marketplace. Many of them are confused by such a transition, as they think having a key is enough for invoking services. However, when we inspected the detail page of web services managed by RapidAPI, which is usually the landing page when App developers search for keywords like "free face recognition web services," we discovered no hints suggesting that developers must subscribe to the services before invoking them. Furthermore, the executable code, tailored with the developer's unique key, is readily supplied.

Actionable Suggestions: To help App developers debug the problem, we suggest the marketplaces define standard and fine-grained error codes for all the services they manage. Instead of relying on the default HTTP response codes, Baidu API already allows service providers to specify their fine-grained error codes in documentation. We believe defining uniform error codes can help automate the debugging procedure. Besides, many of the questions related to the new subscription model could be easily answered by providing proper guidance information to developers.

3) SECURITY:

Security issues in API marketplaces involve vulnerabilities such as inadequate authentication, data exposure, and insufficient encryption, potentially leading to unauthorized access and data breaches.

- *API key protection suggestion:* Some questions seek help in safeguarding API keys. Many answers suggest securely storing them in environment variables or utilizing a dedicated secrets management service. Some answers also recommend avoiding hard-coded keys directly into source code or public repositories to reduce the risk of exposure.
- *API key leakages observed:* Within this specific issue subcategory, developers have acciden-

tally exposed API keys when posting their questions.

Figure. 4.14 gives a code snippet that has the key leakage issue. When invoking a service, App developers need to specify its host and a developer-specific API invocation key. Although static registries and the marketplace both face such issues, their impact on the marketplace is more severe. This is because one developer may subscribe to multiple services, with some being free services and others requiring payments, using the same key. We call such a mechanism “one-key-for-all-service.” Even if a developer’s question is about invoking a free service, a malicious third party could use the key to access all paid services the developer has subscribed to, with some containing sensitive customer data. Besides, as RapidAPI now plays an important role in service invocations, if attackers use the captured keys to perform a DDoS attack and cause RapidAPI delegation servers to fail, all services managed by RapidAPI will be impacted.

```
headers: {  
  'X-RapidAPI-Host': 'coinranking1.p.rapidapi.com',  
  'X-RapidAPI-Key': '2575-4ab-42m3na6000e-1a1b-021eop100740jdm2e3d4f2b0000',  
},
```

Figure 4.14: Key Leakage code snippet example

Actionable Suggestions: We believe it’s the platform’s responsibility to prevent exposure of the invocation key. The platform can prevent key leakages by 1) Marketplaces should find safer alternatives to the ‘one-key-for-all-service’ model; 2) being able to identify instances where keys are exposed and prompt users to securely hide them, which is one of the reasons why we developed the LLM-based tool; and 3) educating application developers.

4) PERFORMANCE:

Performance is the efficiency and responsiveness of a system or application. Within the Performance issue category, developers express concerns regarding slow response times or connection timeouts.

- *API down /slow:* The developers may face this issue when the API is either experiencing downtime or operating at a reduced speed.
- *Connect timeout/ stopped:* This typically happens when there’s a delay in the connection process, or when the connection attempt is terminated due to certain conditions such as network issues, server unavailability, or misconfiguration. In some instances, the service may have completely ceased to function on the side of the API provider.

A specific example is “(why) *Ecoindex.fr Backend API was down?*”

The services provided by the API providers are poorly coded and inadequately tested by the platform before being added to the platform. Despite marketplaces like as shown in Fig. 4.15

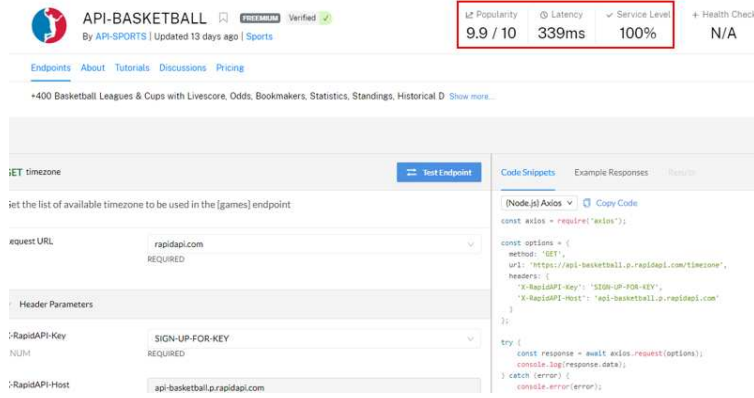


Figure 4.15: RapidAPI QoS

RapidAPI being designed to aid developers in monitoring Quality of Service (QoS), they mainly focus on monitoring the long-term QoS and respond to short-term QoS issues not swiftly enough. Consequently, we observe some developers inquiring whether the service is temporarily down or why latency is high, due to the platform’s slow updates of these parameters.

Actionable Suggestions: API marketplaces should identify these services and create Service Level Agreements (SLAs) with service providers. The API marketplace must communicate to service providers the necessity of maintaining well-coded and thoroughly tested APIs. Our LLM tool aids application developers in understanding the cause of service downtime.

4.3.2 Evaluation of Fine-tuned Automated Tool

We evaluated our fine-tuned model to ascertain its proficiency in accurately identifying issue categories, impacted APIs, and key leakages. By comparing the model’s classification results with the manual labels, Table 4.2 lists the number of records correctly identified in each category. Interestingly, the model successfully categorized issues related to Usability, Performance, and General Queries. However, it erroneously classified 15 coding issues into different categories and failed to identify 11 API names and 6 instances of key leakages. Overall we achieved an accuracy of 85% for question classification, 88% for identifying service names, and 77% for identifying key leakages. We noticed that approximately 30-40% of API services were affected by key leakages.

	Dimension	Correctly classified(CoT)	Correctly classified(w/o CoT)	Total Issues
Classification	General Queries Issues	6	4	6
	Usability Issues	28	23	28
	Performance Issues	9	6	9
	Coding Issues	42	42	57
	Total Test data	85	75	100
Information Extraction	Service Detection from Total Test data	84	37	95
	Key Leakage Identified from Total Test Data	20	7	26

Table 4.2: Fine-tuned Model Accuracy for Test Dataset

Performance Issues	Key Leakages
calorieninjas.p.rapidapi.com	euromillions.p.rapidapi.com
api-football-v1.p.rapidapi.com	yummly2.p.rapidapi.com
omgvamp-hearthstone-v1.p.rapidapi.com	lambda-face-recognition.p.rapidapi.com
elenasport-io1.p.rapidapi.com	omgvamp-hearthstone-v1.p.rapidapi.com
bayut.p.rapidapi.com	api-football-v1.p.rapidapi.com
coinranking1.p.rapidapi.com	community-open-weather-map.p.rapidapi.com
matchilling-chuck-norris-jokes-v1.p.rapidapi.com	booking-com.p.rapidapi.com
textanalysis-keyword-extraction-v1.p.rapidapi.com	webknox-trivia-knowledge-facts-v1.p.rapidapi.com
	corona-virus-world-and-india-data.p.rapidapi.com
	google-translate1.p.rapidapi.com
	devru-times-of-india.p.rapidapi.com
	alpha-vantage.p.rapidapi.com
	indeed-indeed.p.rapidapi.com
	v1-sneakers.p.rapidapi.com
	coinranking1.p.rapidapi.com
	covid-19-data.p.rapidapi.com
	apidojo-yahoo-finance-v1.p.rapidapi.com
	faceplusplus-faceplusplus.p.rapidapi.com
	cryptocurrency.p.rapidapi.com
	dev132-cricket-live-scores-v1.p.rapidapi.com

Table 4.3: Impacted APIs Identified from Test Data

We further evaluate the impact of CoT in fine-tuning. The accuracy achieved by the training model without CoT was 75% for classification and around 40% for identifying service names and key leakages. It shows that CoT improves the accuracy of fine-tuning, especially for identifying service names and key leakages which requires understanding the logic behind the reasoning.

Table 4.3 lists all the impacted APIs that we’ve manually confirmed from the questions in the test dataset, with the help of our tool. It demonstrates that the tool can be used to detect performance outages and key leakages and give warnings to service providers and marketplaces.

CHAPTER 5

Implications

Our empirical analysis identified key differences between traditional service registries and service marketplaces and also revealed a taxonomy of common migration challenges that developers often face. Additionally, we developed a proof of concept demonstrating how the LLM model can assist developers. Furthermore, the study we have carried out discovered 1) the characteristics of RapidAPI-managed services by examining their various facets, 2) how these services are used on GitHub concerning their various facets, 3) how developers choose services over alternative options; and 4) how developers' proficiency levels impact their choices. Our findings can have useful implications for different stakeholders:

- **For SOA Stakeholders** Our analysis identified key differences between traditional service registries and Service Marketplaces. Such findings could be beneficial for developers to make decisions on migrating from traditional service registries and Service Marketplaces. Moreover, the taxonomy can guide developers on common issues of Service Marketplaces migration, their root causes, and the solution.
- **For Tool Builders** The evaluation showed that our proposed approach of utilizing an LLM-based issue categorization technique can effectively categorize migration issues. The tool builders can take further initiative to integrate that technique with IDEs and development tools so that developers can instantly get feedback on the issues they face during migration.
- **For Integration Developers:** Our insights provide integration developers with a deeper understanding of the diverse characteristics of services available on RapidAPI. Developers can make more informed decisions during the selection and integration process by systematically evaluating service facets such as latency, reliability, pricing, followers, aggregated ratings, and service provider and community support. Additionally, emphasizing the importance of implementing robust error-handling strategies and effective RapidAPI key management practices ensures smooth integration processes, minimizing disruptions and enhancing the overall reliability and security of their applications.

- **For Service Providers:** These insights can also inform service providers about developer expectations, guiding efforts in improving service usability.
- **For Marketplace Platforms:** Understanding developer proficiency and preferences enables platforms to tailor recommendations, improving satisfaction and platform engagement across varying expertise levels.
- **For Researchers:** The taxonomy of Service Marketplaces migration could help researchers to develop advanced research techniques such as recommendation systems [42, 43] and automated program repair [44, 45] techniques to assist developers in fixing migration issues more efficiently and in a timely manner. Additionally, this study identified the need for novel approaches for locating services with subpar QoS or documentation, as a way to enhance service provider feedback and platform trustworthiness.

CHAPTER 6

Future Work and Directions

The main contributions of this article lie in 1) Examining the distribution of services across various service facets, assessing their impact on service popularity and usage patterns, understanding their role in selecting services among similar options, exploring how facets impact different categories of developers, and evaluating their influence on service popularity and usage patterns after excluding prototype or practice repositories. 2) Understanding how the service registry transition has confused the developers using the new generation service marketplace. The research presented herein provides numerous opportunities and several potential avenues for future work, including:

- Developing a recommendation system that utilizes user inputs to generate a list of similar services, analyzing user preferences to suggest services of similar types, or assist developers in to fix migration issues instantly, would aid developers in selecting services from various alternatives more effectively and handle migration issues instantly. This facility would streamline the developer's experience, enabling comparisons to be made more efficiently.
- The service facets can be examined more comprehensively by exploring various marketplace platforms other than RapidAPI such as Apify, Baidu, APIGuru, etc., and comparing them. Similarly, different open-source coding platforms where studies regarding service invocations are conducted can be considered, in addition to GitHub. A promising opportunity to broaden the findings of this research would be to conduct an empirical study of real-time Android marketplaces.
- As discussed in RQ5 concerning error handling and key leakage Sec.4.2.3, there is an observed lack of awareness about coding practices and API key management. Developing an automated tool or LLM (Language Model-based tool) would be beneficial in helping developers detect exceptions, timeouts, and key leakages, thereby enhancing reliability and security.
- As observed in RQ2, the Quality of Service (QoS) changes over time3.2.1.4; hence, the development community would benefit from having an automated tool that detects poor QoS and informs the developers of any QoS updates.

CHAPTER 7

Conclusion

In conclusion, this article presents several key contributions. First, the demonstration of the shift in workflow is a primary source of confusion for developers.

Second, the conduct of a large-scale empirical study on how integration developers select web services. It is found that many services are with poor quality of service (QoS) and suffer from inadequate support from both service providers and developer communities. Key factors like service level, cost, aggregated ratings, and provider/community support significantly influence the selection and utilization of RapidAPI services, with support emerging as the most critical factor. Among similar options, developers prioritize provider and community support, with popularity and latency being the second most important factors. Moreover, skilled developers tend to focus on factors like popularity, latency, and provider support. Improving error-handling strategies and raising awareness about API key management would greatly benefit the service integration process.

Lastly, the taxonomy established of developers' concerns, details how they originate from the transition to service marketplaces. As a solution to deal with these concerns, the article introduces an automated tool that provides 3-dimensional assistance to app developers, service providers, and platforms, helping to identify and address these concerns. Our evaluation confirmed the tool's accuracy and the broader implications of this research.

CHAPTER 8

Publications

This research is successfully published in the below conferences/journal:

- **International Conference on Web Services (ICWS)**(Core Ranking A)

Title - “How Can I Be of Service?”—A Comprehensive Analysis of Web Service Integration Practices (<https://ieeexplore.ieee.org/abstract/document/10707446>)

Authors - Siddhi Baravkar, Pratiksha Gaikwad, Eli Tilevich, Long Cheng, and Zheng Song

- **International Conference on Software Services Engineering (SSE)** (Core Ranking B)

Title - Decoding and Answering Developers’ Questions about Web Services Managed by Marketplaces (<https://ieeexplore.ieee.org/abstract/document/10664239>)

Authors - Siddhi Baravkar, Chi Zhang, Foyzul Hassan, Long Cheng, and Zheng Song

- In process submission to journal **Springer Empirical Software Engineering**.

Title - Towards a Comprehensive Understanding of Web Service Integration: A Large-Scale Empirical Study from the Developers’ Perspective

Authors - Siddhi Baravkar, Pratiksha Gaikwad, Zhengquan Li, Eli Tilevich, Long Cheng, and Zheng Song

References

- [1] K. Kotsopoulos, P. Lei, and Y. F. Hu, “Managing ngns using the soa philosophy,” in *2008 First ITU-T Kaleidoscope Academic Conference - Innovations in NGN: Future Network and Services*, 2008, pp. 47–54.
- [2] N. Ibrahim and B. Bench, “Service-oriented architecture for the internet of things,” in *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2017, pp. 1004–1009.
- [3] M. H. Valipour, B. Amirzafari, K. N. Maleki, and N. Daneshpour, “A brief survey of software architecture concepts and service oriented architecture,” in *2009 2nd IEEE International Conference on Computer Science and Information Technology*, 2009, pp. 34–38.
- [4] A. Tewari, P. Nagdev, and A. Sahitya, “Sky computing: the future of cloud computing,” *IJCSIT) International Journal of Computer Science and Information Technologies*, vol. 6, no. 4, pp. 3861–3864, 2015.
- [5] N. T. Yazdi and C. H. Yong, “A potential way for efficient information sharing based on mobile text messaging,” in *2012 International Conference on Green and Ubiquitous Technology*, 2012, pp. 172–175.
- [6] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio, “Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services,” *IEEE Transactions on Services Computing*, vol. 3, no. 3, pp. 223–235, 2010.
- [7] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding, ““{MLaaS} in the wild: Workload analysis and scheduling in {Large-Scale} heterogeneous {GPU} clusters,” in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 945–960.
- [8] “Global machine learning as a service (mlaas) industry research report, competitive landscape, market size, regional status and prospect,” shorturl.at/mKMY3, accessed: 2023-12-30.
- [9] E. Al-Masri and Q. H. Mahmoud, “Investigating web services on the world wide web,” in *WWW’08*, 2008, pp. 795–804.
- [10] D. Pudasaini and C. Ding, “Service selection in a cloud marketplace: a multi-perspective solution,” in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*. IEEE, 2017, pp. 576–583.

- [11] A. A. Olu, “Modelling the performance of web services in cloud e-marketplaces based on consumer waiting time and provider cost,” Ph.D. dissertation, University of Zululand, 2016.
- [12] G. Gu and F. Zhu, “Trust and disintermediation: Evidence from an online freelance marketplace,” *Management Science*, vol. 67, no. 2, pp. 794–807, 2021.
- [13] L. Dahlander, D. M. Gann, and M. W. Wallin, “How open is innovation? a retrospective and ideas forward,” *Research Policy*, vol. 50, no. 4, p. 104218, 2021.
- [14] N. Zhang, Y. Zou, X. Xia, Q. Huang, D. Lo, and S. Li, “Web apis: Features, issues, and expectations—a large-scale empirical study of web apis from two publicly accessible registries using stack overflow and a user survey,” *IEEE Transactions on Software Engineering*, vol. 49, no. 2, pp. 498–528, 2022.
- [15] K. Mahmood, G. Rasool, F. Sabir, and A. Athar, “An empirical study of web services topics in web developer discussions on stack overflow,” *IEEE Access*, vol. 11, pp. 9627–9655, 2023.
- [16] R. Akolkar, T. Chefalas, J. Laredo, C.-S. Perng, A. Sailer, F. Schaffa, I. Silva-Lepe, and T. Tao, “The future of service marketplaces in the cloud,” in *2012 IEEE Eighth World Congress on Services*, 2012, pp. 262–269.
- [17] A. Menychtas, S. G. Gomez, A. Giessmann, A. Gatzoura, K. Stanoevska, J. Vogel, and V. Moulos, “A marketplace framework for trading cloud-based services,” in *Economics of Grids, Clouds, Systems, and Services: 8th International Workshop, GECON 2011, Paphos, Cyprus, December 5, 2011, Revised Selected Papers 8*. Springer, 2012, pp. 76–89.
- [18] E. Zeydan, L. Blanco, S. Barrachina-Muñoz, F. Rezazadeh, L. Vettori, and J. Mangues, “A marketplace solution for distributed network management and orchestration of slices,” in *2023 19th International Conference on Network and Service Management (CNSM)*, 2023, pp. 1–6.
- [19] S.-Y. Hwang, C.-C. Hsu, and C.-H. Lee, “Service selection for web services with probabilistic qos,” *IEEE transactions on services computing*, vol. 8, no. 3, pp. 467–480, 2014.
- [20] Y. Ma, S. Wang, F. Yang, and R. N. Chang, “Predicting qos values via multi-dimensional qos data for web service recommendations,” in *2015 IEEE International Conference on Web Services*, 2015, pp. 249–256.
- [21] X. Chen, Z. Zheng, Q. Yu, and M. R. Lyu, “Web service recommendation via exploiting location and qos information,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 7, pp. 1913–1924, 2014.
- [22] D. Pudasaini and C. Ding, “Service selection in a cloud marketplace: A multi-perspective solution,” in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, 2017, pp. 576–583.
- [23] R. Ramacher and L. Mönch, “Cost-minimizing service selection in the presence of end-to-end qos constraints and complex charging models,” in *2012 IEEE Ninth International Conference on Services Computing*. IEEE, 2012, pp. 154–161.

- [24] Y. Wang, J. Zhang, and J. Vassileva, “Effective web service selection via communities formed by super-agents,” in *2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, vol. 1. IEEE, 2010, pp. 549–556.
- [25] F. Binzagr, H. Labbaci, and B. Medjahed, “Fame: An influencer model for service-oriented environments,” in *Service-Oriented Computing: 17th International Conference, ICSOC 2019, Toulouse, France, October 28–31, 2019, Proceedings 17*. Springer, 2019, pp. 216–230.
- [26] A. Owraq, A. Namoun, and N. Mehandjiev, “Quality evaluation within service-oriented software: a multi-perspective approach,” in *2012 IEEE Ninth International Conference on Services Computing*. IEEE, 2012, pp. 594–601.
- [27] Z. Song, O. Rowader, Z. Li, M. Tello, and E. Tilevich, “Quality of information matters: Recommending web services for performance and utility,” in *2022 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2022, pp. 41–48.
- [28] N. C. Mendonca, J. A. F. Silva, and R. O. Anido, “Client-side selection of replicated web services: An empirical assessment,” *Journal of Systems and Software*, vol. 81, no. 8, pp. 1346–1363, 2008.
- [29] E. Jawabreh and A. Taweel, “Time-aware qos web service selection using collaborative filtering: A literature review,” in *European Conference on Service-Oriented and Cloud Computing*. Springer, 2023, pp. 55–69.
- [30] M. Daaji, A. Ouni, M. M. Gammoudi, S. Bouktif, and M. W. Mkaouer, “Multi-criteria web services selection: Balancing the quality of design and quality of service,” *ACM Transactions on Internet Technology (TOIT)*, vol. 22, no. 1, pp. 1–31, 2021.
- [31] M. Bano and D. Zowghi, “Users’ voice and service selection: An empirical study,” in *2014 IEEE 4th International Workshop on Empirical Requirements Engineering (EmpiRE)*. IEEE, 2014, pp. 76–79.
- [32] P. K. Venkatesh, S. Wang, F. Zhang, Y. Zou, and A. E. Hassan, “What do client developers concern when using web apis? an empirical study on developer forums and stack overflow,” in *ICWS’16*. IEEE, 2016, pp. 131–138.
- [33] G. Ajam, C. Rodríguez, and B. Benatallah, “Api topics issues in stack overflow q&as posts: An empirical study,” in *2020 XLVI Latin American Computing Conference (CLEI)*, 2020, pp. 147–155.
- [34] H. Zhang, S. Wang, T.-H. Chen, Y. Zou, and A. E. Hassan, “An empirical study of obsolete answers on stack overflow,” *IEEE Transactions on Software Engineering*, vol. 47, no. 4, pp. 850–862, 2021.
- [35] B. Xu, T.-D. Nguyen, T. Le-Cong, T. Hoang, J. Liu, K. Kim, C. Gong, C. Niu, C. Wang, B. Le *et al.*, “Are we ready to embrace generative ai for software q&a?” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1713–1717.

- [36] Y. Feng, S. Vanam, M. Cherukupally, W. Zheng, M. Qiu, and H. Chen, “Investigating code generation performance of chat-gpt with crowdsourcing social data,” in *Proceedings of the 47th IEEE Computer Software and Applications Conference*, 2023, pp. 1–10.
- [37] R. Khankhoje, “Web page element identification using selenium and cnn: A novel approach,” *Journal of Software*, vol. 1, no. 1, 2023.
- [38] H. Li, F. Khomh, M. Openja *et al.*, “Understanding quantum software engineering challenges an empirical study on stack exchange forums and github issues,” in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 343–354.
- [39] F. Corno, L. De Russis, and J. P. Sáenz, “On the challenges novice programmers experience in developing iot systems: A survey,” *Journal of Systems and Software*, vol. 157, p. 110389, 2019.
- [40] J. C. Panzar, *Competition and Efficiency*. London: Palgrave Macmillan UK, 2016, pp. 1–4. [Online]. Available: https://doi.org/10.1057/978-1-349-95121-5_669-1
- [41] S. A. Kingaby and S. A. Kingaby, “The real estate api,” *Data-Driven Alexa Skills: Voice Access to Rich Data Sources for Enterprise Applications*, pp. 365–385, 2022.
- [42] D. Kim, Y. Tao, S. Kim, and A. Zeller, “Where should we fix this bug? a two-phase recommendation model,” *IEEE transactions on software Engineering*, vol. 39, no. 11, pp. 1597–1610, 2013.
- [43] X. Xia, D. Lo, X. Wang, and B. Zhou, “Accurate developer recommendation for bug resolution,” in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 72–81.
- [44] C. L. Goues, M. Pradel, and A. Roychoudhury, “Automated program repair,” *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.
- [45] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, “The strength of random search on automated program repair,” in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 254–265.