OS^3: A Step Forward Towards Enhancing Share and Reuse in Serverless Functions

by

Sarvesh Rakesh Bhatnagar

A thesis in partial fulfillment
of the requirements for the degree of
Master of Science
(Computer and Information Science)
in the University of Michigan–Dearborn
2023

Master's Thesis Committee:

Assistant Professor Zheng Song, Chair
Assistant Professor Probir Roy
Professor Eli Tilevich, Virginia Polytechnic Institute and State University

Sarvesh Rakesh Bhatnagar

sarveshb@umich.edu

ORCID iD: 0000-0003-2391-0341

© Sarvesh Rakesh Bhatnagar 2023

Dedication

This work is dedicated to all those who have supported me throughout my journey. My deepest gratitude goes to my parents, Rashmi Bhatnagar and Rakesh Bhatnagar, my elder brother Shalabh Bhatnagar, and my younger brother Charlie. I am also immensely thankful to my advisors and mentors, Dr. Zheng Song and Dr. Eli Tilevich, for their guidance and unwavering support. Thank you all for being a part of my journey.

Acknowledgements

I am deeply grateful to Dr. Zheng Song, my advisor, for his unwavering support and guidance throughout my Master's journey. His contributions have been invaluable, and this work would not have been possible without his guidance. I would also like to extend my heartfelt thanks to Dr. Eli Tilevich for his mentorship, support, and for always taking the time to listen and offer guidance. Their contributions have been greatly appreciated and have played a crucial role in my growth and success. Furthermore, I would also like to express my sincere gratitude to Dr. Probir Roy for giving me the opportunity to share my research with the committee and the wider community. Dr. Roy's feedback and insights have been invaluable in helping me to improve my work.

Preface

My passion for Community Detection and Recommendation Systems began before I even considered pursuing a Master's degree. I was drawn to the topic and spent countless hours reading papers and imagining what it would be like to conduct research in this field. During my Bachelor's studies, I was able to bring my ideas to life by publishing two papers on the subject. However, I felt there was still more to explore and wanted to delve deeper into the topic, which led me to pursue a Master's degree.

Having said that, I was lucky to have Dr. Zheng Song as my advisor and mentor, who recognized my potential and took a chance on exploring a new and relatively uncharted area of Serverless Computing. Despite the many challenges that came with this new domain, we were determined to make meaningful contributions to the field. Through hard work and perseverance, I hope this paper is able to close the gap and shed light on recommendation for serverless function. Needless to say, I am grateful for all that I have learned along the way and people who have taken part in my journey for the same.

Table of Contents

Dedication . . . **ii**

Acknowledgements . . . **iii**

Preface . . . **iv**

List of Figures . . . **vii**

List of Tables . . . **viii**

Abstract . . . **ix**

Chapter

1 Serverless Computing, An Overview . . . **1**

1.1 What is Serverless Computing . . . 1
1.2 Growth and Obstacles in Serverless Computing . . . 2
1.3 Open-Source and Serverless Computing . . . 3
1.4 Success Criteria for Open-Source Contributions . . . 5
1.5 Problem Domain and Contributions . . . 6

2 Search and Recommendation in Serverless Computing . . . **8**

2.1 Keyword Based Search and Recommendation Techniques . . . 9
2.1.1 Exact Match Keyword Based Search . . . 10
2.1.2 Word2Vec Embedding Distance Based Search . . . 11
2.2 Peculiarities in Serverless Computing . . . 12
2.3 Known Problems for Search and Recommendation . . . 13
2.4 Current State of Search and Recommendation for Serverless Functions . . . 14

3 Data Collection and Analysis . . . **16**

3.1 Data Collection . . . 16
3.2 Data Preprocessing . . . 18
3.3 Data Analysis . . . 21
3.3.1 Study 1: Usages of Externally Managed API . . . 21
3.3.2 Study 2: Equivalent APIs for Different Platforms . . . 22
3.3.3 Study 3: License Considerations . . . 23
3.3.4 Study 4: Cost Considerations . . . 25

4 OS^3: Design Considerations . . . **26**

4.1 Capability of Incorporating Existing Search Algorithms 26
4.2 Improving Search by Exploiting Serverless Peculiarities 27
4.3 Filtering and Reranking the Search Results 28

5 OS^3 Reference Implementation 30

5.1 Phase 1: Underlying Search Algorithm 30
5.2 Phase 2: Search Result Refining 31
5.3 Phase 3: Filtering and Reranking 31

6 Case Study 34

6.1 Query 1: Machine Analysis and Learning 34

7 Evaluation 36

7.1 Experimental Design 36
7.2 Experimental Results and Discussion 37
7.3 Interpreting Experimental Results to Address Research Questions 41

8 Related Work 43

8.1 Serverless Computing Dataset's 43
8.2 Serverless Function Sharing and Reusing 44
8.3 Code-Based Search and Recommendation 44

9 Conclusion and Future Directions 46

Appendix: Relevant Paper Publications 47

A.1 OS^3: The Art and the Practice of Searching for Open-Source Serverless Functions 47
A.2 Sharing and Reusing Serverless Functions: Software Licensing Constraints on My Mind 48

References 49

List of Figures

1.1 Unique characteristics of Serverless Functions 6

2.1 Information Retrieval System 9

3.1 Preprocessing and Data Cleaning for OS^3 Dataset 19
3.2 Validity Check Workflow for Serverless config files 20
3.3 License Distribution In Serverless Functions (log scale) 23

5.1 Case Study: Searching for "Machine Analysis and Learning" Serverless Functions . . 33

7.1 Comparison of False Positive Results w/ and w/o OS^3 for all Search Queries 39
7.2 False Positives For Keyword Matching with and without OS^3 40
7.3 False Positives For Word Vector Based Search with and without OS^3 41
7.4 Accuracy of the Repositories Appended by OS^3 for all Search Queries 42

List of Tables

1.1 Open-Source Tools for Serverless Computing . . . 4

2.1 Challenges and Solutions for Serverless Repository Search and Recommendation . . . 14

3.1 Platform Validation using Config files . . . 17
3.2 Equivalent Managed Services . . . 22

7.1 Repository Discovery and Filtering using OS^3 . . . 38

Abstract

Known for its execution flexibility and pricing elasticity, Serverless computing deploys the code of services on demand and charges the client for their actual execution. Serverless computing enables service developers to focus on creating useful services, without being concerned about how these services would be deployed and provisioned. These developments have fostered the rapid growth of a community of open-source developers creating and improving various serverless functionalities. Because of this variety, the ecosystem of serverless functionalities suffers from a high degree of duplication, with multiple serverless functionalities sharing similarities or being outright identical. In order to integrate a serverless function into a project, a developer first needs to be able to find a suitable implementation among multiple alternatives. Unfortunately, existing search technologies have not been designed to address the needs of searching for serverless functionalities. To address this problem, this paper presents a novel approach to search for serverless functions, called **O**pen-**S**ource **S**erverless **S**earch (OS^3) that maximizes the utility of the returned serverless functions by (1) basing the search process on both descriptive keywords and library usages, thus increasing the search results' precision and completeness; (2) filtering and ranking the search results on both the software license and execution cost parameters. Implemented in 3,000 lines of Python, with a search space of 5,981 serverless repositories from four major serverless platforms, OS^3 outperforms existing search approaches in terms of the suitability of the returned serverless functions, based on our evaluation with realistic use cases. Enhancing the search facilities for serverless functions with the insights presented herein can help fully fulfill the enormous promise of serverless computing.

CHAPTER 1

Serverless Computing, An Overview

1.1 What is Serverless Computing

Serverless Computing, also known as Functions-as-a-Service (FaaS), is a cutting-edge and rapidly expanding trend in cloud computing. It made its debut in 2014 at the AWS re:Invent event with the introduction of AWS Lambda offerings[1]. Since then, numerous cloud service providers have followed suit and added similar services to their cloud infrastructure offerings. Despite the name ”serverless”, this type of architecture does not mean the absence of servers, but rather that a third-party is responsible for managing them, effectively hiding them from the user.

In contrast to traditional cloud computing models where resources are allocated through the rental or provisioning of servers, serverless computing operates differently. The cloud provider dynamically allocates resources as needed to execute and scale an application’s code, enabling developers to upload their code in the form of small, single-purpose functions that are only triggered by events or API requests. This eliminates the need for allocating resources through the rental or provisioning of servers and simplifies the development process.

Serverless Computing holds enormous potential for drastically improving numerous important facets of developing modern enterprise applications. Serverless computing’s powerful abstractions of the underlying infrastructure management (e.g., load balancing, scaling-on-demand, etc.), rapid prototyping, and flexible pay-as-you-go model relieve application developers from the concerns about the low-level aspects of deploying and provisioning remote service code. Due to seemingly infinite cloud resources allocated at runtime, this model allows for the perceived infinite elasticity of serverless functions, so developers can focus on the high-level design aspects of service-oriented applications [2].

The applications developed using Serverless Functions are generally broken down into individual functions that can be triggered by different types of events such as manual invocation, scheduled invocation, HTTP requests or message queues. Each function operates independently from other with its own isolated environment and resources where the cloud provider automatically allocates them to handle the incoming requests. From the software engineering standpoint, serverless

promotes reusability, as service developers naturally produce modular service components, to be reused across multiple projects, rather than monolithic services, tailored for specific projects.

1.2 Growth and Obstacles in Serverless Computing

With an architecture that supports developers to concentrate on the core logic rather than the underlying infrastructure, the growth of serverless computing can be attributed to its attractive features such as pay as you go model, built-in scalability, and elimination of operational responsibilities. These features make it a desirable option for businesses and developers who aim to reduce costs, enhance efficiency, and focus on development instead of spending time and efforts in management of the underlying resources.

In a traditional server-based model, businesses must allocate and fund a predetermined amount of computing resources, regardless of their actual utilization. This is because maintaining the servers that host the compute logic necessary to fulfill customer requests is mandatory. In contrast, Serverless Computing enables customers to only pay for the exact amount of computing resources they consume, thus eliminating the requirement for costly upfront investments and reducing overall costs. Furthermore, its pay as you go model combined with inherent scalability enables developers to adopt a microservice architecture instead of developing monolithic applications. This simplifies the process of modifying individual modules and incorporating new features, providing greater flexibility to the overall architecture.

With the recent growth in internet-based services and tools, it has become crucial to be able to adapt to changing customer demands. The unpredictable nature of demand makes it essential to have the ability to seamlessly scale resources, avoiding both service interruptions and unnecessary over-allocation or under-allocation of resources. This is where serverless functions come in, offering the added benefit of eliminating the need for operational management. The cloud provider takes care of all the underlying infrastructure and maintenance, freeing businesses to focus solely on their core products and reducing the time and cost associated with managing their own infrastructure.

Given the above mentioned benefits of Serverless Computing, it has been adopted into many applications, both enterprise and academic in nature. One such example of use of Serverless Computing is Netflix which uses AWS Lambda for operations tasks such as video encoding, file backup, security audits of EC2 instances, and monitoring [3]. In the scientific community, it has also been used for scientific workloads such as SNP Genotyping[4], Seismic Imaging[5], Parallel Analytics[6], etc.

Serverless Computing offers several advantages in terms of scalability and cost-effectiveness, but it also comes with its limitations. One of the key limitations is the limited invocation time per instance, which is capped at 15 minutes (as of 10th Feb, 2023) on AWS Lambda, resulting

in a stateless environment where subsequent invocations are not guaranteed to run on the same machine. To overcome this, developers often use additional services provided by the cloud provider to enhance their application's workflow and make their functions stateful.

Another limitation of serverless functions is their cold start time, which refers to the delay that occurs when a function has not been executed in a while, and the cloud provider needs to allocate new resources. This delay can last from several hundred milliseconds to a few seconds[7] and is a significant drawback of serverless computing, particularly for time-sensitive applications where low latency is crucial. However, there are ongoing efforts to address this issue, with researchers and developers actively working on improving cold start duration. One such effort is Snapstart[8], recently introduced by AWS, which significantly improves the startup performance of AWS Lambda functions written in Java by over 10X. Snapstart works by using a Firecracker microVM snapshot of memory and disk state, which is cached for low latency access.

The choice of cloud provider also presents a challenge in serverless computing as it can result in vendor lock-in. Once a cloud provider is selected, it can be difficult to move to a different provider due to the tight coupling with a single provider. Furthermore, the limited control over the environment and infrastructure can make it challenging to incorporate dependencies and debug serverless functions, which are inherently ephemeral in nature.

1.3 Open-Source and Serverless Computing

The open-source community has made a significant impact in the development and maintenance of open-source tools for Serverless Computing. These tools are designed to help developers in their serverless development and deployment processes. To better understand the open-source tools in the realm of serverless computing, it is important to categorize them into two main categories: Platforming Tools and Deployment Tools.

Table 1.1 shows these two categories of open-source tools for Serverless Computing, along with examples of tools for each category. Each tool has its unique features and advantages that cater to different development requirements.

Platforming Tools are designed to help developers create a serverless platform that enables them to run serverless functions on their selected cloud infrastructure. These tools include OpenFaas, Apache OpenWhisk, and Kubeless, which provide developers with the necessary platform to execute and scale their serverless functions in a cloud environment. These tools have received wide-spread recognition and support from the open-source community and have seen a rapid growth in usage among developers.

Deployment Tools, on the other hand, are focused on helping developers push their code to the existing cloud infrastructure platforms such as AWS and Google Cloud. The Serverless Framework

Table 1.1: Open-Source Tools for Serverless Computing

Category	
Platforming Tools	OpenFaas, Apache OpenWhisk, Kubeless
Deployment Tools	Serverless Framework, AWS SAM, Gordon, Zappa

is a popular deployment tool that provides developers with a unified experience across cloud providers and offers a wide range of features for faster and easier deployment of serverless functions. Other tools in this category include AWS SAM, Gordon, and Zappa, which help developers to manage and deploy their serverless functions more effectively.

AWS SAM, Gordon, and Zappa are other popular deployment tools that serve a similar purpose to the Serverless Framework. These tools have their unique features and advantages that cater to different development requirements. For instance, AWS SAM provides a framework for defining serverless applications and generating CloudFormation templates, which are critical for managing resources in the AWS cloud platform. Gordon, on the other hand, offers fast and straightforward deployment for Python-based serverless applications by leveraging the power of AWS CloudFormation. Zappa is a serverless web framework for Python that makes it easy to develop and deploy serverless applications on AWS Lambda and API Gateway.

In recent years, there has been a growing emphasis on the sharing of serverless function code, which enables developers to take full advantage of the inherent modularity of serverless functions. This idea of code sharing is driving initiatives such as the Amazon AWS Serverless Application Repository (SAR), which enables developers to open source and share their code with others, making it easier to reuse core components across multiple projects.

By leveraging the modular nature of serverless functions and the potential for reuse, particularly for common utility functions such as video encoding, text-to-speech conversion, and image compression, serverless computing holds great promise for the future of application development. Developers can save significant time and effort by leveraging existing tools and functions developed by the open source community, allowing them to focus on the core logic of their applications and avoid duplicating code.

For instance, a developer working on an image classification machine learning model can reuse functions such as RGB to grayscale conversion and image resizing, freeing up valuable time and resources to focus on the essential aspects of the project. This can reduce development efforts and streamline the development process, providing a more immersive experience for developers.

1.4 Success Criteria for Open-Source Contributions

As serverless computing becomes increasingly widespread, the open-source community will continue to be a crucial player in its development and the future of cloud computing. The community has already made a significant impact by providing developers with the resources and tools needed to build and maintain serverless functions, but there is still room for improvement. Currently, there are gaps in the sharing and reuse of serverless functions, which could greatly benefit the community. The tools for researching and developing serverless functions to encourage sharing and reuse are limited and have not received much attention.

The very success of an ideal situation for a software project that utilizes serverless computing hinges on the ability of application developers to find functions that are *the most suitable for the task at hand*. For instance, in a chatbot project, functions like tokenization, lemmatization, and encoders may be deemed critical. As it turns out, this is a formidable task due to two main complications.

First, the high heterogeneity of vendors and platforms rules out a one-stop platform for service developers to share their serverless functions. The serverless ecosystem is highly fragmented across multiple infrastructure providers, including Microsoft Azure [9], IBM Cloud [10], AWS [11], Google Cloud [12], and open source serverless frameworks [13] that support multiple infrastructure providers. AWS [14] maintains its own platform for developers to share open source repositories(AWS SAR), Google Cloud maintains 51 sample functions [12], while developers of other serverless platforms open source their functions on Github.

Second, the existing search facilities for serverless functions rely on keyword-based searching heuristics, which alone are insufficient to find the functions based on their suitability for a given task. To be able to effectively search for serverless functions, a search facility must take into account their unique characteristics(see Fig. 1.1) that include

1. library usage, a critical issue for serverless as compared with traditional services
2. high user sensitivity for the runtime costs
3. software licensing, which imposes legal restrictions on how serverless functions can be used

The success of finding the most suitable serverless functions for a software project depends on the ability to take into account important characteristics. Current search facilities fall short in this regard and cannot accurately identify the best functions. To overcome this, it is crucial to consider a combination of performance, resource utilization, and legal criteria when searching for the right serverless functions.

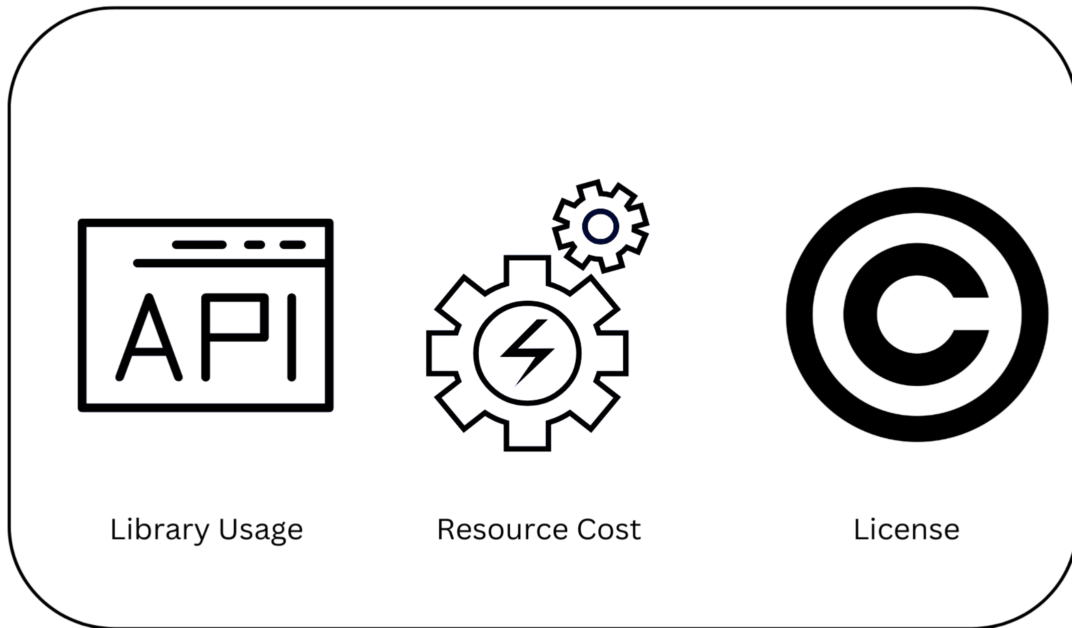

Figure 1.1: Unique characteristics of Serverless Functions

1.5 Problem Domain and Contributions

Driven by these insights, this thesis presents a novel approach to search for serverless functions. This paper presents a novel approach to searching for serverless functions. Dubbed OS^3, our approach maximizes the suitability of returned serverless functions by incorporating the unique characteristics above. For (1), it improves the precision and completeness of the search results by complementing keyword-based search results with library usage; for (2), it estimates the complexity of serverless repositories by analyzing their source code to rank the search results; for (3), it filters out license-problematic repositories based on the individual use case.

The reference implementation of OS^3 searches for open-source serverless functions across multiple vendor platforms. Its search space comprises 5,981 serverless repositories, filtered out from over 60K repositories, collected for four major vendors from GitHub. OS^3 not only improves the precision and completeness of keyword-based search, but also identifies repositories protected by appropriate software licenses, thus returning search results that are most suitable for the task at hand.

This thesis makes the following contributions:

1. We collect and filter over 60K Github repositories to curate a high-quality dataset of 5,981 serverless repositories. We carefully study the data and identify several unique aspects of searching for serverless functions.

2. We introduce a clustering-based algorithm to improve keyword-based search by considering library usage; it refines the search results of existing search algorithms by removing false positives and appending repositories originally missing.

3. We build the first search engine for cross-platform, open-source serverless functions. The search engine enables incremental development of serverless functions based on similar open-source versions, filling in the missing pieces of the serverless ecosystem.

CHAPTER 2

Search and Recommendation in Serverless Computing

When it comes to Search and Recommendation systems, it's important to understand the underlying technology powering them, which often involves Information Retrieval Systems. These systems are designed to quickly and efficiently retrieve relevant information, helping users find what they're looking for in a timely manner. Fig. 2.1 shows the modules involved when working with information systems.

To achieve this, Information Retrieval Systems use several different modules that work together to enable the smooth functioning of the system. The first module is the Query Normalizer, which is responsible for standardizing user queries so that they can be effectively processed by the system. This module helps to ensure that the system can handle a wide range of queries, no matter how they are phrased or spelled.

Next, the Preprocessing module is responsible for parsing and analyzing the documents or data that the system will be searching. This module is crucial for ensuring that the system can quickly and accurately retrieve relevant information. It also involves techniques like stemming and stop-word removal, which help to improve the precision and recall of the system.

The Search algorithm is the core of the Information Retrieval System, and it's responsible for matching user queries with relevant documents or data. This module employs techniques like keyword matching, fuzzy matching, WordVector based matching and so on to ensure that users receive the most relevant results possible.

After the Search algorithm has retrieved a set of documents or data, the Ranking module comes into play. This module is responsible for determining the relevance of each document or data point, and ranking them in order of relevance. This ensures that users see the most important results first, and helps them to quickly find what they're looking for.

Finally, the Filtering module is responsible for removing any irrelevant or low-quality results from the set of documents or data that have been retrieved. This module uses a variety of techniques, including machine learning algorithms, to help identify and filter out irrelevant results, improving the overall quality of the results provided to the user.

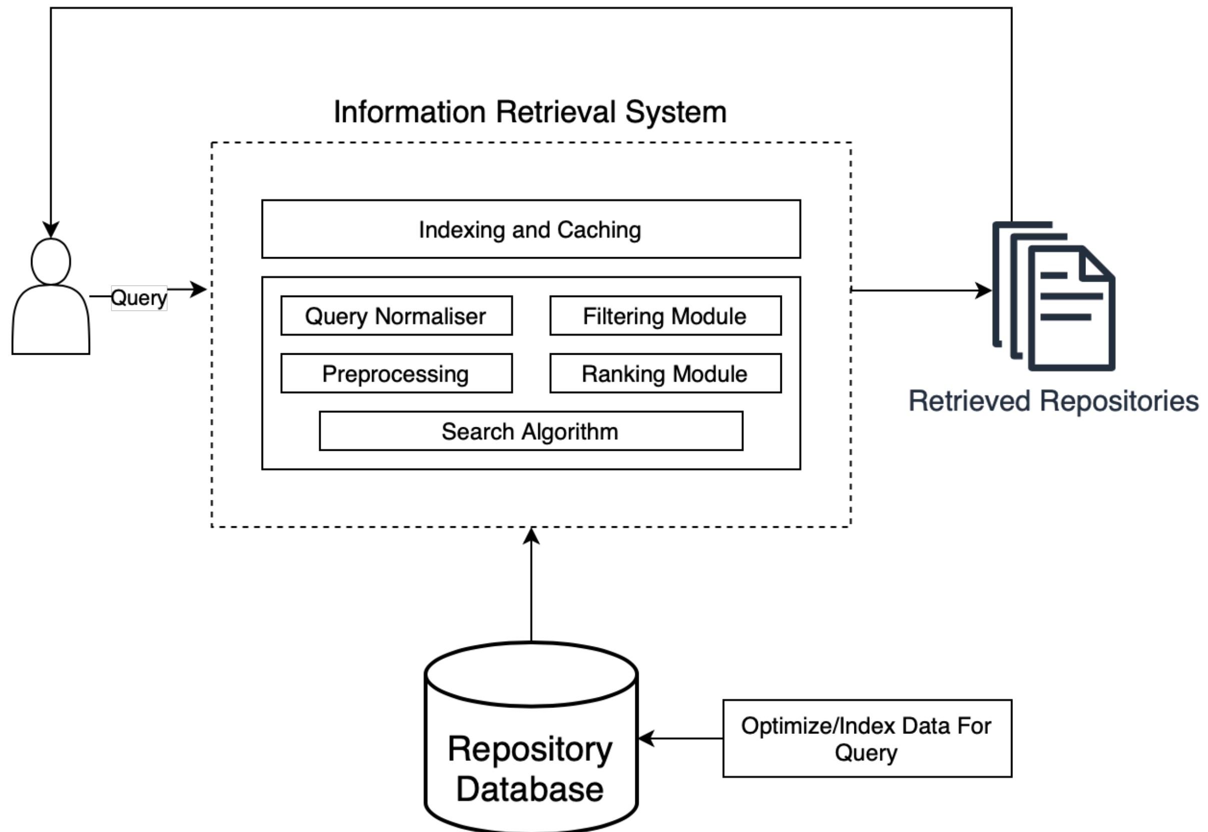

Figure 2.1: Information Retrieval System

2.1 Keyword Based Search and Recommendation Techniques

Keyword matching algorithms are an essential component in various applications like search engines and recommendation systems. These algorithms help to match keywords entered by users with relevant content stored in a database, thus ensuring an efficient and effective search or recommendation experience. The objective of keyword matching algorithms is to provide the most relevant results for a user's query.

There are several variations of keyword matching algorithms, including exact match, partial match, and fuzzy match. Exact match algorithms return results that exactly match the keyword entered by the user. For instance, if someone searches for ”serverless function for image encoding,” the algorithm will return search results that contain the words ”serverless”, ”function”, ”for”, ”image” and ”encoding”. On the other hand, partial match algorithms return results that contain the keyword entered by the user, even if it is not an exact match. For the same example, the algorithm will return results that contain words like ”server”, ”serverless”, ”functions”, ”encode”, ”image”, ”images” etc. Some Fuzzy match algorithms goes a step further to return results that are similar to the keyword entered by the user, even if they are not an exact match. For instance, the keyword

”serverless” might also be associated with the keywords ”cloud computing”, ”function”, ”lambda”, ”AWS”, ”Azure”, etc.

Keyword matching algorithms not only consider the type of match but also take into account various factors to rank the results and determine their order. These factors include keyword frequency, relevance, and user intent. For instance, a keyword that appears frequently in the content is more likely to be considered relevant than one that appears less often. To perform such filtering, various techniques like TF-IDF can be used[15].

Moreover, the relevance of a keyword depends on the context and user intent. For example, different users might search for different things using the same keyword. To provide accurate results, it is essential to have domain knowledge, which helps in recommending contextual-based results. Thus, understanding user intent and context is crucial for providing the most relevant results.

To improve the accuracy of keyword matching algorithms, machine learning and natural language processing techniques can also be used. These techniques enable the algorithm to better understand user queries, disambiguate similar keywords, and provide more precise results. By incorporating these advanced technologies, keyword matching algorithms can more effectively recognize and address user needs, leading to higher user satisfaction and engagement.

2.1.1 Exact Match Keyword Based Search

Exact match keyword-based search is a simple search algorithm that involves matching the query precisely with the keywords or phrases in the documents. The main goal of this type of search is to find documents that most match the query terms, making it ideal for straightforward search scenarios.

To perform an exact match keyword-based search, the query is typically broken down into individual keywords, a process known as tokenization. Then, the search system looks for documents that contain those keywords, taking into account lemmatization and considering the root forms of the tokens. To provide more relevant results, the search algorithm also penalizes common words such as ”is,” ”the,” and ”a” to ensure that only the most important and relevant words are used for ranking.

The search results are then ranked based on the number of matches and the context, where the documents that contain more matches appear higher in the rankings. Additionally, to improve the quality of the results, the system ensures that the words are important and relevant within the context of the query while giving less importance to the commonly occurring words as discussed.

One advantage of exact match keyword based search is that it is simple and straightforward to implement, and can be easily integrated into existing systems. Furthermore, it is also very fast and efficient, since it only requires a direct comparison of the query and document terms, without any additional processing.

Given the benefits of exact match keyword based search, it also has several limitations. It can only return results that match the exact query terms, and cannot handle synonymy, polysemy, or other semantic relationships between words. Additionally, it is sensitive to spelling errors and other forms of user error, and can only match exact terms, rather than semantically related terms.

2.1.2 Word2Vec Embedding Distance Based Search

The introduction of Word2Vec based word embeddings marked a major advancement in keyword based search. They are vectors of continuous values that represent high-dimensional data in low-dimensional vector space. These representations are commonly used in various natural language processing and machine learning applications, including sentiment analysis, document classification, and recommendation systems. In the realm of information retrieval, embeddings can be utilized to depict documents or query terms in a continuous vector space that can then be compared through a similarity measure such as cosine similarity.

Word2Vec is a popular method for learning word embeddings, where each word is represented as a vector of real numbers. The goal of Word2Vec is to preserve the semantic relationships between words in the embedding space. For example, words that are semantically similar should have similar embeddings, and words that are semantically dissimilar should have dissimilar embeddings. Word2Vec learns these embeddings by predicting the surrounding words of a given word, using either a continuous bag-of-words (CBOW) or skip-gram model.

Given a sequence of words $(w_1, w_2, \ldots, w_T)$, the CBOW model maximizes the average log probability (Equation 2.1) where w_t is the target word, and $w_{t-k}, \ldots, w_{t+k}$ are the surrounding context words. For which the probability $p(w_t|w_{t-k}, \ldots, w_{t+k})$ is computed using the softmax function as defined in Equation 2.2 where $\mathbf{v}_w$ is the vector representation (embedding) of word w, and W is the total number of words in the vocabulary.

$$\frac{1}{T}\sum_{t=1}^{T}\log p(w_t|w_{t-k}, \ldots, w_{t+k}) \tag{2.1}$$

$$p(w_t|w_{t-k}, \ldots, w_{t+k}) = \frac{\exp(\mathbf{v}w_t \cdot \frac{1}{2k}\sum j = 1, j \neq t - k^{t+k}\mathbf{v}w_j)}{\sum w = 1^W \exp(\mathbf{v}w \cdot \frac{1}{2k}\sum j = 1, j \neq t - k^{t+k}\mathbf{v}_{w_j})} \tag{2.2}$$

The skip-gram model in Word2Vec is similar, but instead of predicting the target word given the context words, it predicts the context words given the target word. This form of embeddings allows us to compare sentences based on their semantic meaning. As an example for our case, the words ”serverless”, ”cloud computing”, ”AWS Lambda”, etc will be close to each other in the vector space while the words such as ”Server” or ”Virtual Machines” should be placed away from each other.

To check how similar or dissimilar the query is, we generally use Word2Vec and find the cosine

distance between query vector and document vector. The goal here is to find documents that are semantically related to the query, rather than just matching the exact terms in the query. Hence, improving and expanding the search results.

Equation 2.3 is used for determining the cosine distance between two word vectors where $\vec{q}$ represents the query vector and $\vec{d}$ represents the document vector. While $||\vec{q}||$ and $||\vec{d}||$ are the magnitude of the query vector and word vector respectively.

$$\text{cosine distance} = \frac{\vec{q} \cdot \vec{d}}{||\vec{q}|| \times ||\vec{d}||} \tag{2.3}$$

One advantage of Word2Vec based search is that it can handle queries with multiple terms and handle synonymy, polysemy, and other semantic relationships between words. This is because the embeddings capture the semantic meaning of words, rather than just matching exact terms. Additionally, Word2Vec based search can handle misspellings and other forms of user errors, since the embeddings capture the semantic relationships between words, regardless of the exact spelling.

Having said that to train an accurate Word2Vec Embedding model we require a large, high quality training set. If the training set is biased or limited, the embedding may not accurately represent the semantic relationships between words, leading to inaccurate search results. Additionally, Word2Vec based search may struggle with rare or domain-specific words that may not appear frequently enough in the training data to generate accurate embeddings. This can result in poor search performance for specialized domains or niche topics.

2.2 Peculiarities in Serverless Computing

As discussed, Serverless computing is a model of cloud computing that operates on a pay-per-use basis. When a user requests a service, the cloud provider allocates and creates the necessary resources on a virtual machine. This process is designed to be fast and efficient, but it can sometimes suffer from what is known as the ”cold start problem”. This occurs when the required resources are not readily available and must be newly instantiated, leading to slow response times.

To address this challenge, serverless computing platforms have imposed certain restrictions on the amount of compute time and resources that can be utilized for each instance. This is done to ensure that the process remains efficient and reliable, even when the demands on the system are high. As a result of these restrictions, developers must write functions that are concise, well-defined, and focused on a specific task.

This approach to writing code encourages developers to create simple, efficient functions that are optimized for use in a serverless environment. It also requires developers to make strategic decisions, particularly when importing libraries as the size of external libraries can significantly impact the performance of the function.

Also, when suggesting serverless functions for reuse, it is crucial to take into account their accompanying licenses, which dictate the permitted usage and deployment of the serverless function, without the need to open-source the entire application. Different types of licenses are available and are typically used to release code as open source. Each license possesses distinct characteristics and fulfills a specific purpose, and identifying them based on usage can result in beneficial outcomes.

Additionally, it's crucial to ensure that the recommended functions belong to the same platform as the one the application is being developed upon. This is because of the vendor lock-in problem, which does not allow the customers to change the platforms with ease. When implementing a search and recommendation system, it is necessary to take all of these factors into consideration in order to recommend a useful and effective serverless function to developers.

2.3 Known Problems for Search and Recommendation

When it comes to working with search and recommendation systems, the quality of the data is paramount to the success of the system. However, despite the best efforts to ensure high-quality data, the data that we have are often not up to the desired standards. Many of times, we even forget to incorporate something important as the licenses which can lead the found repository moot.

One of the major challenges faced when we are trying to improve search and recommendation is the presence of missing, vague, or incorrect descriptions of the of the source code. To make matters worse, it can be even more challenging to relate the descriptions provided with the actual functions defined within the source code. This makes it harder for the recommendation systems to relate with the functionalities within the code. This can lead to a situation where the system is unable to provide accurate recommendations or search results for the same.

Furthermore, the repositories found also have numerous example repositories, repositories for the purposes of learning, teaching and so on. Such repositories are not generally suitable for production as they are not optimized for maintainability as opposed to a production ready repositories. Also, a lot of times we see repositories that are exactly the same with little to no difference. This creates multiple recommendation results for the same repository. Thus, when recommending repositories it is of high importance to return a diverse suggestion maintaining a balance between relevance and diversity of the results.

Table 2.1 provides a summary of the challenges we explored regarding their impact on Search and Recommendation for Serverless functions, along with potential solutions to mitigate these impacts.

Table 2.1: Challenges and Solutions for Serverless Repository Search and Recommendation

Challenge	Impact	Solution
Redundant Data	Repeated Suggestions and decreased utility	Shallow compare and remove same repositories from the dataset
Toy Repositories	Poor recommendation quality	Remove repositories with keywords like example, toy, test, etc.
Bad Descriptions	Inaccurate Results or Missed Results	Ensure higher cleaning and preprocessing procedures, Rely on diverse set of features instead of descriptions only.
Ignoring Licenses	Unusable Repositories	Usage based recommendation that inherently considers licenses

2.4 Current State of Search and Recommendation for Serverless Functions

As previously discussed, there are initiatives in place to promote collaboration among individuals and facilitate the sharing and reuse of serverless functions. However, these efforts are limited to the development of tools like the Serverless Framework and repositories like AWS SAR. While the Serverless Framework automates and simplifies the deployment process, allowing users to share code on platforms such as Github, SAR promotes sharing and reuse within the vendor platform.

Despite these initiatives, the growth of AWS SAR lags behind the repositories found on Github, and the sharing and reuse of serverless functions has yet to reach its full potential. Developers continue to implement similar functions repeatedly, rather than leveraging the existing functions already developed by members of the open-source community.

The number of serverless functions available through public repositories is growing, which can make it challenging for developers to find the right function for their needs. Furthermore, multiple functions may perform the same task, and it may not be immediately clear which one is the best fit. Additionally, serverless functions may have different performance characteristics, dependencies, and licensing requirements, all of which need to be considered when making a recommendation.

The search and recommendation of serverless functions is essential to promote better sharing and reuse, yet there have been few efforts to research and improve this process, specifically tailored

to serverless functions. Keyword-based matching can be problematic due to incomplete or missing information in Readme files. Finding specific functions based on an exact query can also be challenging when working with code examples that require more context than what can be included in a search query. Therefore, it is crucial to continue exploring ways to improve the search and recommendation of serverless functions to better promote sharing and reuse in the open-source community.

CHAPTER 3

Data Collection and Analysis

In this chapter, we will discuss the main considerations we took into account when collecting our dataset of serverless repositories. Currently, there is limited research on serverless functions and the datasets available are not comprehensive enough[3, 16]. Some studies have attempted to analyze serverless with custom datasets but the collected data is insufficient in both quality and quantity. For example, the largest dataset available is Wonderless, which contains only 1877 serverless repositories, and another high-quality dataset[3] contains 89 repositories.

To enable effective search and recommendation, a much larger and high-quality database is required. To address this gap, we collected a dataset of over 60K serverless repositories from GitHub and analyzed it to inform the design of our search engine. We then narrowed it down to 5,981 repositories after cleaning and preprocessing the data. In the following sections we discuss our methodology for the same.

3.1 Data Collection

During our research on datasets related to serverless repositories, we analyzed multiple sources and found the recently published Wonderless dataset [16] to be the most comprehensive among its peers, with a collection of 1,877 serverless repositories. While the size of this dataset is impressive, we found that it has some limitations in terms of its collection procedure and criteria for cleaning data.

One of the major drawbacks of the Wonderless dataset is that it relies solely on the Serverless framework [13] as the source of repositories. This could potentially limit the dataset's relevance for search and recommendation purposes, as it may not capture the full range of serverless repositories available on various platforms.

Moreover, the criteria used to clean the data in Wonderless are overly strict and may remove repositories that could have had significant recommendation value. In particular, we found that Wonderless removes multiple repositories that could be useful for developers to consider for reuse purposes.

Another important factor to consider when working with source code deployed on the internet is the presence of licenses for repositories. It is often a topic that goes unattended but is critical when search and recommendation is concerned. We found that Wonderless did not take this into account during its collection procedure. As a result, many of the repositories in the dataset are moot and cannot be used for any reuse purposes.

On that note, after accounting for the presence of licenses, we found that the number of repositories that could be used from the Wonderless dataset reduced to 819 repositories from 1,877 repositories. This is a massive drop for usable repositories accounting to only 44% of the dataset. The remaining 1058 repositories did not contain licenses making them unusable.

Recognizing these limitations of the current dataset, we have concluded that a more comprehensive and diverse dataset is necessary to facilitate effective search and recommendation of serverless repositories. Accordingly, we opted to gather data from multiple sources instead of relying solely on the Serverless framework, which is the case with the Wonderless Dataset.

We are confident that our dataset, which is larger and more diverse, will enable us to overcome some of the shortcomings of the existing dataset and better meet the needs of developers searching for reusable serverless functions. By having access to a more extensive and varied dataset, we can provide more comprehensive and relevant search and recommendation outcomes. Additionally, our dataset will prove to be an invaluable resource for conducting further research and analysis on serverless functions.

To make such an approach possible we consider different kinds of configuration files and their peculiarities to search for repositories on GitHub for different platforms. We identified configuration files such as "`serverless.yml`" for repositories that use the Serverless framework, "`template.yml`" for repositories that use AWS, "`function.json`" for repositories that use Azure, and "`manifest.yml`" for repositories that use IBM functions.

By following this process, we collected a total of 67,744 repositories from GitHub, which comprises 29,995 for the Serverless framework, 14,164 for AWS, 21,523 for Azure, and 2,062 for IBM. This diverse and expansive dataset would provide us with a more comprehensive and relevant set of repositories, enabling us to conduct further analysis and research related to serverless functions.

Table 3.1: Platform Validation using Config files

Platform	**Configuration File**	**Validation Check**
AWS	template.yml	AWSTemplateFormatVersion
Azure	function.json	direction
IBM	manifest.yml	actions

3.2 Data Preprocessing

As we previously mentioned, after completing our Data Collection process, we have acquired a substantial amount of data, with approximately 67,744 repositories. However, it is important to note that these repositories are not all appropriate for search and recommendation purposes. The reason being is that the collected repositories may contain irrelevant repositories, as previously discussed, due to common names such as "`function.json`" and "`manifest.yml`". Moreover, the repositories may also consist of examples that are used to setup basic structure of serverless functions, repositories that serve the purpose of training and teaching, and repositories that are very similar to one another. As a result, these factors could lead to the difficulty of distinguishing between the relevant and irrelevant repositories during the recommendation process.

In order to enhance the quality of the collected dataset, we have implemented several preprocessing steps. The primary objective of these preprocessing steps is to ensure that the repositories obtained are of top-notch quality and are relevant for the purpose of searching and recommending serverless functions. The overall workflow for preprocessing can be observed in Fig. 3.1. In this section, we will elaborate on the preprocessing steps that we have taken to achieve this goal.

The initial step that we took in the preprocessing of our dataset was to eliminate empty repositories and toy repositories that consist of specific keyword phrases such as `example`, `demo`, and `test` in their Readme files. This removal process is carried out as these repositories often do not accurately represent genuine serverless functions. By removing these repositories, we can enhance the precision and relevance of our search and recommendation results.

Subsequently, we proceeded with the filtering of all repositories by removing unlicensed repositories, a step that was not performed during the creation of the Wonderless dataset. The removal of unlicensed repositories is deemed essential for the effective recommendation of serverless functions as it ensures that only repositories for which the owner has authorized reuse and modification can be considered. It is imperative to note that the owner alone has all rights reserved for an unlicensed repository, and its use without the explicit permission of the owner may result in legal consequences. Additionally, we carried out the elimination of all duplicate repositories to ensure that the resulting dataset contains only unique and distinct repositories.

During the data collection process, we also took into account the potential removal of repositories with no stars or those that were inactive, a step that was implemented in the Wonderless dataset. However, we decided to keep such repositories as they could still be useful for the purpose of searching and recommending serverless functions.

The decision was based on the fact that serverless functions are typically small modules that perform a specific task. Therefore, if a function is not actively updated, it is highly likely that it already performs its intended functionality and is considered complete. Moreover, not all

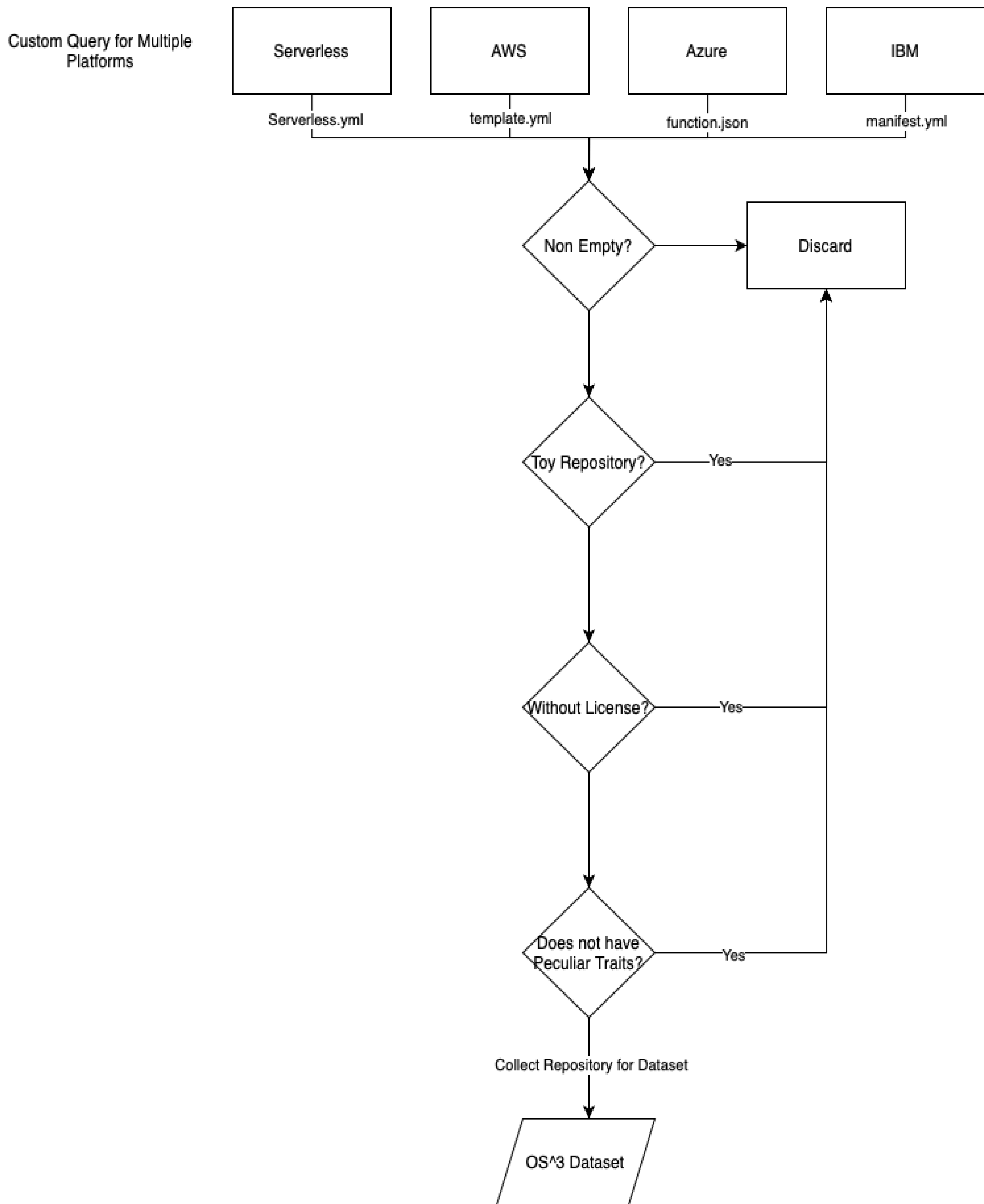

Figure 3.1: Preprocessing and Data Cleaning for OS^3 Dataset

functions can be easily discovered, and a repository with no stars might indicate that it was merely undiscovered, and not necessarily imply that it should be excluded from recommendation. In light

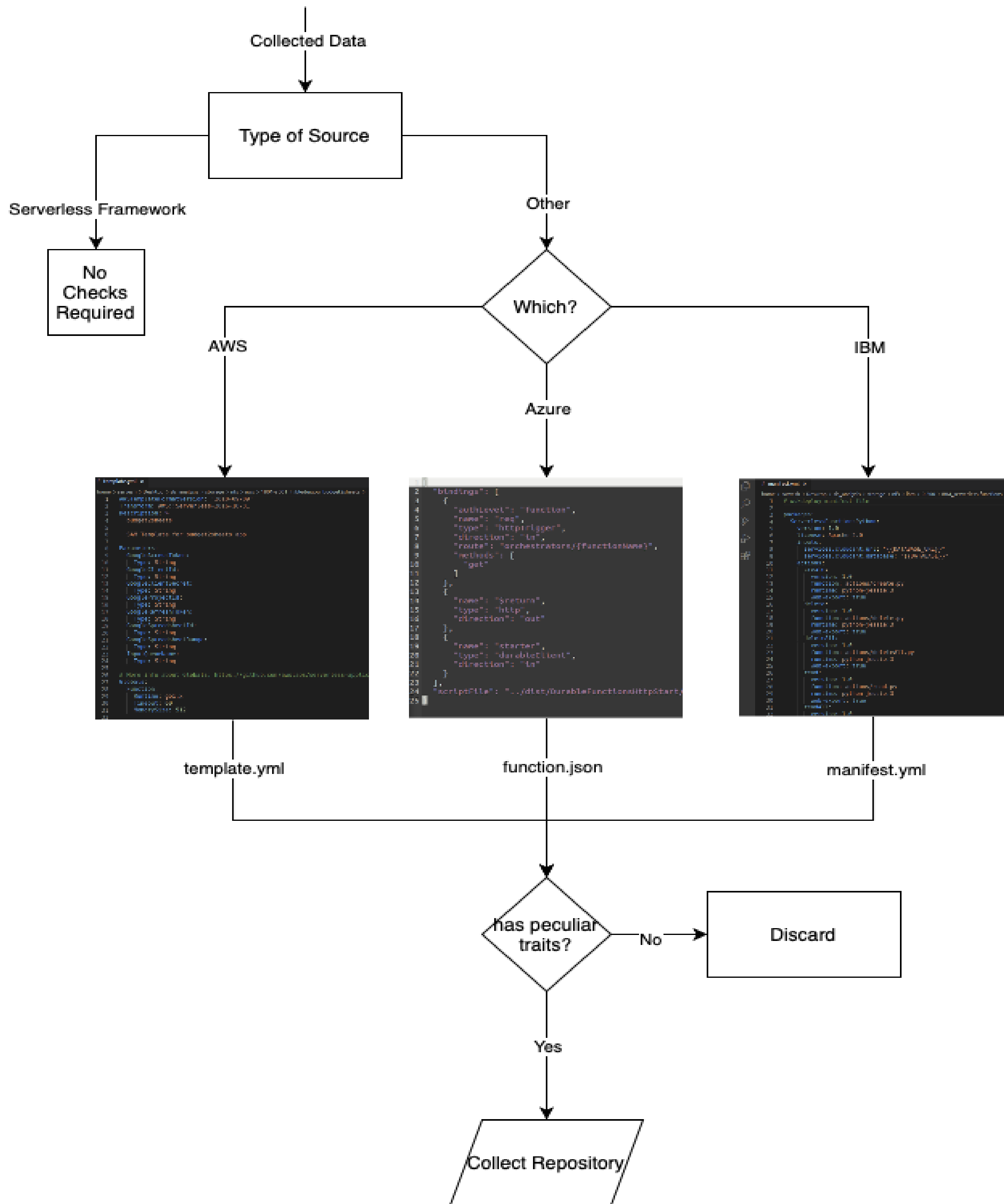

Figure 3.2: Validity Check Workflow for Serverless config files

of these considerations, repositories with no stars or those that are inactive are still deemed useful, and their removal could prove counterproductive for searching and recommendation tasks.

Furthermore, as noted above, some configuration files such as "`function.json`" and "`manifest.yml`", are commonly found in software repositories that are not related to serverless functions. To avoid having irrelevant repositories in our dataset, we use configuration peculiarities to validate whether a repository is genuinely used for implementing a serverless function. These peculiarities are based on the documentation for their respective platforms, which are listed in Table 3.1. This table is used in workflow as defined in Fig. 3.2 which is the part of Fig. 3.1.

Following the execution of all the cleaning and filtering steps mentioned above, we were able to considerably reduce our dataset to a total of 5,981 repositories. Among these, 5,220 repositories were developed using the Serverless framework, 498 were created for AWS, 241 for Azure, and 22 for IBM. From these results, we can infer that the most commonly used deployment method is via the Serverless framework, followed by AWS, Azure, and IBM, respectively. It is noteworthy that we observed a relatively low level of interest in IBM, while there was a heavy interest in AWS. Additionally, we would like to highlight that our repository numbers for different platforms align with the popularity of these platforms, as demonstrated in [17].

3.3 Data Analysis

The purpose of this section is to conduct an in-depth analysis of the repositories we have collected, with a specific focus on identifying the peculiarities that are inherent to serverless functions. Our goal is to gain a better understanding of these peculiarities, as such insights will be valuable when designing and implementing a search and recommendation system for serverless functions.

3.3.1 Study 1: Usages of Externally Managed API

Our analysis of the collected repositories has revealed that a significant majority (88%) of serverless functions rely on externally managed APIs, such as S3 [18], DynamoDB [19], Lex [20], Polly [21], and SQS [22], among others.

In addition to these APIs, we have also noted that developers often utilize cloud services like AWS Textract to implement the desired functionality quickly. This trend is a positive direction for serverless functions, as it enables developers to rapidly implement their desired functionality by providing pre-built serverless functions for their use. Our goal of providing a search and recommendation system for serverless functions is aligned with this trend of facilitating faster development by leveraging existing services and functions.

Additionally we observe that developers of serverless functions are more likely to avoid importing unnecessary libraries as compared with other software developers. On average for serverless functions, we find 2.8 library imports per serverless function, whereas for standard software, the

number is much higher with an average of 11.6 library imports per project [23]. This difference is mainly due to the following reasons: 1) serverless functions are usually developed following the microservice software architecture, in which monolithic software is partitioned into smaller components, so each component is encapsulated as a serverless function. One serverless function is concerned with completing a single task; 2) the pay-as-you-go pricing model of serverless requires that serverless developers be very cautious about the size of their serverless containers, as a larger container takes longer to ship and more memory to deploy, thus increasing its execution costs. These reasons force developers to make their code efficient and only include those libraries that are truly used in the functionality of a serverless function.

We also noticed another notable characteristic that stands out is their heavy reliance on externally managed libraries. These libraries are utilized to perform various tasks and therefore should tend to be more strongly correlated with the specific functionalities of the serverless function in question compared to other types of software. This observation is significant because it highlights the importance of considering the library usages of serverless repositories when developing a system like OS^3. With this in mind, it becomes clear that understanding the nuances of library usage in the context of serverless functions is essential for creating an effective and relevant search and recommendation system for serverless function repository.

3.3.2 Study 2: Equivalent APIs for Different Platforms

When we take a look at the Services and APIs provided by different cloud providers, we find similarities. Most of the aforementioned externally managed APIs have their corresponding versions on other platforms as well. We have compiled representative sets of equivalent APIs for different platforms in Table 3.2.

Table 3.2: Equivalent Managed Services

	AWS	**Azure**	**IBM**
1	S3	Blob Storage	COS
2	DynamoDB	Cosmos DB	DB2
3	SNS	Event Grid	Cloud Event Notification Service
4	SQS	Storage Queues	MQ
5	SES	SendGrid	APP Connect
6	Kinesis	Event Hubs	Streams
7	Lex	Bot Service	Watson
8	Polly	Text to Speech	Watson Text to Speech

The wide existence of such equivalent APIs is caused by Vendor Lock-In [24], as developers are likely to choose externally managed APIs provided by the same platform. This strategy greatly

reduces the execution latency and resource consumption of their serverless functions. Furthermore, it is easier to use such services when they belong to the same vendor due to inherent integration.

The availability of equivalent APIs can facilitate the identification of comparable serverless functions that have been developed for different cloud platforms. To illustrate this, let's consider two serverless functions - one that employs `S3`, `SNS`, and `Polly`, and another that utilizes `COS`, `Cloud Event notification Service`, and `Watson Text to Speech`.

It can be inferred that these two functions have been developed for AWS and IBM respectively, and offer similar functionalities. This is mainly due to similar functionalities provided by the APIs being used. We can see many such examples in the collected repositories. This observation further serves as a motivating factor for us to incorporate equivalent APIs into OS^3, as it can help enhance the platform's versatility and compatibility with other cloud providers. By leveraging equivalent APIs, OS^3 can enable users to seamlessly integrate their existing serverless functions and find functions for different cloud vendor regardless of the cloud platform they were originally developed for.

3.3.3 Study 3: License Considerations

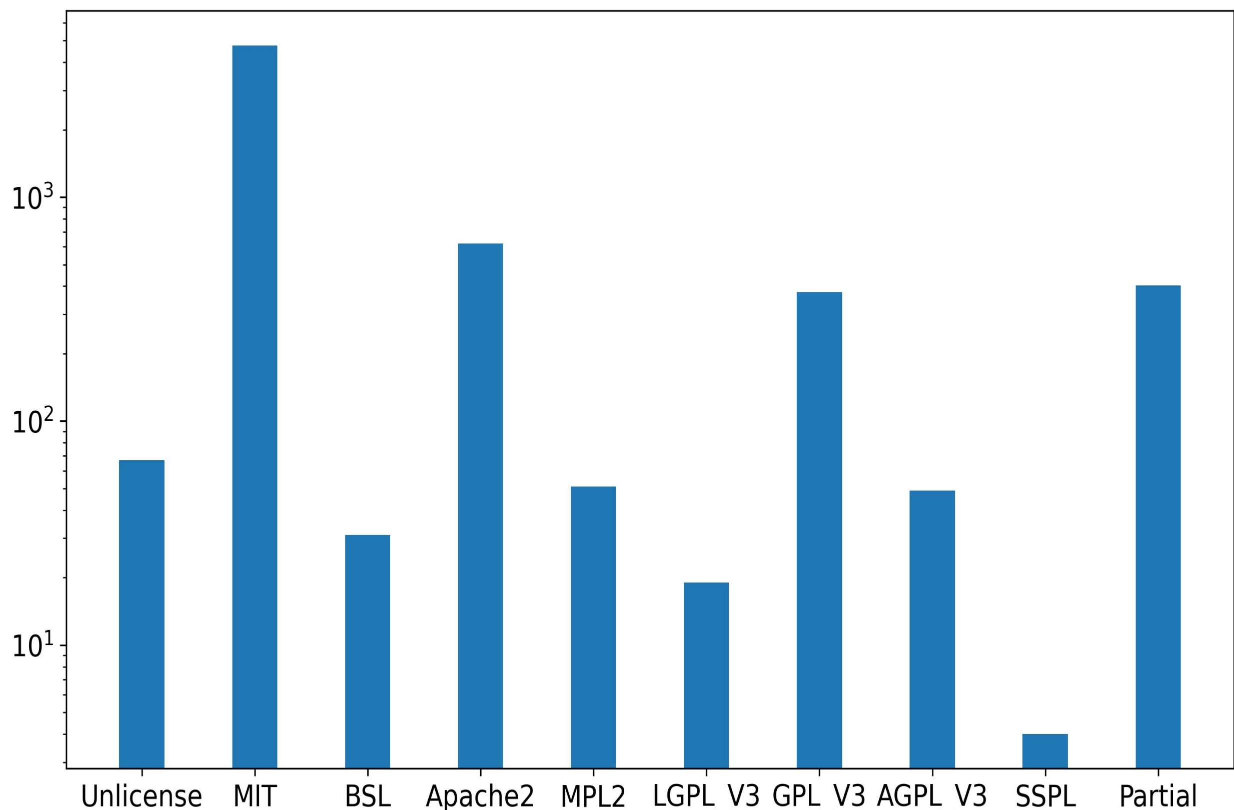

Figure 3.3: License Distribution In Serverless Functions (log scale)

Open source licenses are legal instruments that govern the use, distribution, and modification of open source software. These licenses grant users the right to access, use, and modify the source code of the software, and distribute it to others. There are several types of open source licenses, each with its own set of terms and conditions. There are many different licenses and these licenses differ in terms of their restrictions, obligations, and compatibility with other licenses.

The choice of license can have significant implications for the development and distribution of open source software, as it can affect the ability of users to freely modify and redistribute the software. As such, it is important for developers and organizations to carefully consider the implications of different open source licenses before choosing one for their project.

In our collected repositories the license distribution can be seen in Fig. 3.3, where the licenses are arranged in the order of restriction, with the left-most licenses being the least restrictive and rightmost being the most restrictive. One should note that, in the list of licenses mentioned, the "Unlicense" license is actually a license that poses the least restrictions on users. Although a large part of serverless functions uses one of the least restrictive licenses, most are either not licensed or are toy repositories.

To top that, of the remaining repositories we found that about 9.5% of the repositories use licenses towards the restrictive end (from MPL2 to SSPL) whereas about 7.7% are only partially licensed. Partially licensed repositories have some components that can be reused while some are not usable. This is because many popular software licenses grant permissions for developers to use and modify open-source software under the terms of certain agreements.

As an example, some licenses allow for commercial use of open source software along with any software built upon it [25, 26]. Some licenses require an application that is built upon the software they protect to open source its code as well if the application is distributed to end users [27]. Some licenses require an application to open source its entire code if any part of the application is built upon open source software protected by the licenses [28, 29].

When it comes to reusing serverless repositories, the software license considerations share some similarities but also have noticeable differences with cloud services. Different from traditional software, cloud-based services require special considerations for software licenses for two reasons: 1) as services are deployed in the cloud, they may not be considered as "being distributed to the end users." Therefore, a cloud service can choose not to open source its code even if it is developed upon open source software protected by a restrictive license; 2) if a monolithic service is partitioned into microservices, developers of a cloud service only need to open source the part of the code, or more precisely, the microservice that uses the open source repository protected by the restrictive license, rather than open sourcing all microservices.

Similar to cloud services, using a serverless function can avoid the need to open source the entire application while deploying a serverless function protected by a restrictive license in the cloud does

not require to open source its code. However, serverless functions may be deployed on edge servers, owned by individuals, so the deployment becomes subject to the “being distributed to the end users” clause. Hence, we argue that a search engine for serverless functions should consider the planned application deployment scenarios. If a developer has no plans to open source their modifications on a serverless repository with a license starting from MPL2 to Partial, the resulting code cannot be deployed in edge environments. Such nuances in license usage for serverless functions makes it even more important to have it considered when recommending serverless functions to a user for reuse.

3.3.4 Study 4: Cost Considerations

Serverless functions offer a significant advantage over traditional server-based computing as they allow for resource utilization on an as-needed basis. Unlike server-full computing, where a fixed amount of resources must be paid for, regardless of whether they are utilized, serverless computing enables developers to scale resources up or down in response to application demand. This is particularly beneficial for applications with variable workloads that may experience fluctuations in traffic.

Another benefit of serverless computing is its ability to minimize costs. By only charging for resources consumed, serverless functions offer a more cost-effective solution than traditional server-based computing. This can be especially valuable for implementing expensive functionalities that would be cost-prohibitive in a server-full environment. By using a serverless approach, developers can ensure that their applications are highly efficient, scalable, and affordable.

There are two key factors to consider when evaluating serverless functions: the amount of resources utilized and the associated cost. Compared to traditional software, serverless functions are typically smaller in size with most being in the range of 10 MB per repository[3]. Additionally, developers can utilize the cyclomatic complexity of functions to estimate their processing time as well.

While cyclomatic complexity can be a useful metric for estimating function time, it is important to note that a lower complexity does not always equate to a faster function, nor does higher complexity always indicate a slower function. However, when comparing functions with similar functionality, lower cyclomatic complexity can be advantageous in terms of reducing processing time and optimizing performance. By considering both resource consumption and function complexity, developers can create highly efficient and cost-effective serverless applications.

CHAPTER 4

OS^3: Design Considerations

The design of our OS^3 system is informed by both the characteristics of serverless functions and insights gleaned from analyzing collected data. OS^3 is specifically designed to leverage the unique capabilities of serverless functions in order to achieve three primary goals: (1) prevent incorrect results, (2) identify relevant but previously overlooked results, and (3) prioritize highly-usable serverless functions with higher rankings. By utilizing the features of serverless functions, OS^3 aims to provide a more accurate and efficient search experience for users.

In addition to the aforementioned priorities, an important consideration for the OS^3 system is the ability to utilize the existing underlying search infrastructure. Rather than tightly coupling the search system with OS^3, the design of the system should allow for seamless integration with existing search tools and frameworks.

This will enable the OS^3 system to leverage the benefits of existing repositories while also taking advantage of the existing search infrastructure, resulting in a more efficient and streamlined search process. By maintaining flexibility in the system design, OS^3 can adapt to changes in the underlying search technology landscape, ensuring long-term viability and scalability.

4.1 Capability of Incorporating Existing Search Algorithms

Developers currently apply multiple keyword based mature techniques to search for and recommend open source functions. Some of these techniques includes algorithms such as keyword matching [30], topic modeling-based search[31, 32, 33], and Word2Vec-based search[34]. When we look into these algorithms, the keyword matching search generally matches keywords within Readme files or their descriptions. When considering topic-modeling based search, it finds a set of topics to determine to which category a repository belongs. Finally, The Word2Vec-based search widens the results by using word vectors to find similar repositories.

Although existing serverless repositories, including Github and AWS Serverless Application Repository(SAR)[14], apply these techniques to search for and recommend serverless functions, these techniques tend to overgeneralize their search results, thus causing high false positive rates[35].

For instance, keyword matching might lead to many more matches than required, thereby causing many false positives[36].

Similarly, topic modeling approaches, such as Latent Dirichlet Allocation(LDA)[37], overgeneralize their results by limiting the search categories and thus are vastly inaccurate. The word vector-based approaches also face similar problems due to them being extremely model reliant and prone to overgeneralization due to vector distance-based search and influence of outliers [38]. Therefore, these techniques may not be the best solution for finding accurate and relevan results, more so if we desire to find serverless functions to integrate with the application under development.

After careful consideration of the aforementioned details, instead of creating a new algorithm, our solution, OS^3, focuses on enhancing existing algorithms to improve the accuracy of results for serverless functions. Our goal is to offer a solution that can be integrated into current search engines without requiring a complete overhaul of the underlying algorithm. This approach enables us to capitalize on the strengths of existing algorithms while addressing the limitations they face regarding serverless-specific features such as configuration files, platforms, and peculiarities related to the use of libraries in serverless functions.

By keeping the underlying search algorithm intact and improving the results, OS^3 can extend the existing search engines rather than create a new one. This design feature makes it feasible to customize the existing search engines to use OS^3 without restructuring the underlying search algorithm. Therefore, customization can focus on serverless-specific features, including configuration files, platforms, and peculiarities related to the use of libraries in serverless functions.

4.2 Improving Search by Exploiting Serverless Peculiarities

By utilizing the unique features of serverless functions, our objective is to enhance the precision of search and recommendation algorithms. Currently, these algorithms oversimplify and fail to deliver a comprehensive range of relevant results. Additionally, they do not consider cost and resource consumption as a primary factor for finding the best match. To address these limitations, we aim to identify relevant repositories and incorporate them into the search results to ensure diversity and relevance.

OS^3 takes advantage of serverless-specific peculiarities to improve the search results. The desirable properties of serverless functions that led to their unprecedented growth (i.e., infinite perceived elasticity, reduced DevOps, and the pay-as-you-go model) also impose certain restrictions on the developer. As revealed in our study, the usages of externally managed libraries are more correlated with the functionality of a serverless function. We use this peculiarity of serverless functions to find more comprehensive results and filter out inaccurate results.

In particular, we take as input the search results of an existing search and a ranking algorithm,

and then refine the given search results by following these steps:

1. Cluster the repositories in the search results by their library usages. To explain this, consider a simplified example, where the serverless functions in the search results are $\{a, b, c, d\}$, with a being the most recommended and d being the least recommended. Applying the clustering algorithm forms two clusters, $\{a, c\}$ and $\{b, d\}$. If we look closer into the serverless functions in the clusters, we may find a, c both uses libraries l_1 and l_2, while b and d both use libraries l_3 and l_4. b may also use l_2, but as it has a stronger connection to d, b thereby falling into the same cluster.

2. Find the cluster $\mathcal{C}$ with the highest energy. We define the energy of a cluster of serverless functions $f \in \mathcal{C}$ as $e = \frac{\sum_{f \in \mathcal{C}} R_f}{|\mathcal{C}|}$, where R_f denotes the ranking of function f in the search result, with a higher R_f indicating the function is more recommended. The energy of a cluster indicates the likelihood of the most accurate serverless functions being included in the cluster. To continue with the example, serverless functions a and c are the two most highly ranked results. Hence, we recognize the cluster of $\{a, c\}$ as with the highest energy.

3. Extract the most identical libraries from the selected cluster and expand the equivalent libraries. By extracting the libraries from the selected cluster, we identify what libraries are used in the cluster of serverless functions that are most likely to fit the user's search intent. We then expand the libraries with equivalent libraries from other vendors (see Table 3.2). To continue with the example, as $\{a, c\}$ are identified as the cluster with the highest energy, l_1 and l_2 are recognized as the most identical libraries. If l_1 is "S_3" from AWS, the "Blob Storage" from Azure and the "COS" from IBM should be expanded to the set of the most identical libraries, because they provide the same functionalities.

4. Filter out the serverless functions that use identical libraries insufficiently. In our case, we will filter out d, as it excludes libraries l_1 and l_2. At this time, the recommended serverless functions are $\{a, b, c\}$.

5. Append the serverless functions excluded in the original result if they contain libraries in the most identical set. To continue with the example, if one serverless function i contains l_2 and "Blob Storage" but is not included, we will append i to the search results. The refined search results are $\{a, b, c, i\}$, with a being the most recommended and i being the least recommended.

4.3 Filtering and Reranking the Search Results

So far, the underlying search algorithms still rank the serverless functions in the search results. As discussed in the study, when developers search for serverless functions, they are concerned about

whether they can reuse the functions at will and whether the serverless function they choose to use or build their own function on is the most cost-efficient.

To offer a more effective solution, OS^3 introduces two additional ranking strategies: (1) based on licenses and (2) based on cost estimation. The license-based reranking strategy reorders the recommendation list by placing serverless functions that are not suitable for reuse at the bottom. On the other hand, the cost-based reranking strategy prioritizes the most cost-efficient serverless functions, taking into account how the function will be used and deployed. This approach provides a more organic way of identifying repositories that can be repurposed for development, particularly in the context of serverless functions.

When estimating the cost of a serverless function, we consider several factors, including the cost of invoking externally managed services, the cost of consuming resources on the hosting platform needed to execute the function, and, the cost of shipping and installing the function executable.

The cost of invoking externally managed services is determined through the use of configuration files and API invocations. In cases where a configuration file like serverless.yml is available, we can use it to estimate the cost. If not, we can make use of the API invocations to provide an approximate cost for the use of the repository. Additionally, the complexity of the code is taken into consideration to calculate the cost of executing a serverless function. Furthermore, the shipping and installation cost can be inferred by examining the size of the serverless function executable. By considering all of these factors, we can offer a more precise assessment of the overall cost of a serverless function.

CHAPTER 5

OS^3 Reference Implementation

Our reference implementation of OS^3 comprises about 3,000 lines of Python code. OS^3 operates in three phases: the first phase performs the preliminary search of repositories; the second phase applies the search result refining algorithm;and the third phase enables an engine user to filter and rerank the search results based on their costs and licenses. In the following subsections, we discuss the implementation of OS^3 in detail.

When working with serverless function recommendation, we note the difficulty in pinpointing specific function present in the repository based on its Readme file. Additionally, this problem becomes more pronounced due to the fact that a single code file might be an entry point for multiple serverless function where each function might further invoke another corresponding function. In our reference implementation, we only recommend a repository instead of a single serverless function, and along with that we also estimate the execution cost of a serverless function by measuring the code complexity of the entire repository to aid users in selecting optimal repository.

5.1 Phase 1: Underlying Search Algorithm

The first phase of the OS^3 implementation involves the utilization of the existing search algorithm. It is important to note that the OS^3 system is designed to be impartial to the underlying search algorithms. The way we achieve this abstraction is by working on the results of the underlying search algorithm instead of working on the search algorithms itself. The reference implementation, however, provides two search algorithms based on keywords: keyword-matching and word-vector based.

In the keyword-matching search algorithm, the Readme files are preprocessed by removing stopwords and performing stemming. The stem of the keywords is then matched with the files, and the raw scores are calculated to determine the similarity. A preference is given to the highest match to provide the most relevant search results.

In the second search algorithm, word vectors are used to find the relevant files. The algorithm is based on the Word2Vec[34] algorithm, which is a widely used and highly effective algorithm for

natural language processing tasks. This algorithm compares word vectors between the user's search query and the corresponding Readme files of serverless repositories.

First, The preprocessed words in the Readme files are converted to vectors and added together, taking the average at the end to determine the approximate location of the Readme file in the vector space. Similarly, the word vectors for the search query are also aggregated and averaged to obtain the query vector. The similarity between the query vector and the Readme file vectors is determined by calculating the cosine distance between them. The search results are then ranked based on the similarity scores, with the most similar results appearing at the top of the list.

Both of these search algorithms are designed to provide efficient and accurate search results for the users of the OS^3 system. The use of both algorithms provides greater flexibility and options to users to find relevant files in serverless repositories. Additionally, the underlying search algorithms can easily be switched and replaced making OS^3 easily adaptable for other use cases.

5.2 Phase 2: Search Result Refining

Once the underlying search algorithm has provided initial recommendations, we use a clustering algorithm to refine the results and eliminate false positives, as well as add any repositories that were initially missed. To accomplish this, we apply the Louvain algorithm for clustering [39], which finds the cluster with the highest energy and identifies the APIs that are most representative of that cluster. If necessary, the clustering algorithm can be changed to meet specific requirements e.g. Label Propagation Algorithm.

Using the APIs identified by the clustering algorithm, we filter out repositories that do not have any APIs in common that are representative of the search query. This helps us to solve the problem of over-generalization in the underlying search algorithm, as it eliminates repositories that are not relevant to the search query. Furthermore, to discover new repositories, we conduct an API-based search using the same set of APIs. This allows us to identify repositories that may have problems such as missing Readme files or language differences in the Readme.

By filtering out irrelevant repositories and expanding the search to include additional repositories, we are able to provide more comprehensive and accurate search results to users. Our approach helps to ensure that the search results are highly relevant to the user's query, while also discovering new repositories that may not have been found through the initial search.

5.3 Phase 3: Filtering and Reranking

As previously mentioned, calculating the cost of a serverless function requires taking into account multiple factors such as the cost of invoking externally managed libraries, consuming resources on the hosting platform, and the cost of shipping and deploying the serverless function.

To estimate the resource consumption, we rely on the code complexity of the serverless function, which we measure using the widely-used cyclomatic complexity [40] metric. For this, we use the open-source implementation Mccabe [41] , which supports multiple programming languages such as Go, Python, and JavaScript.

To measure the deployment overhead, we consider the total size of the serverless function repository. To achieve this, we traverse all the files in the repository folder and accumulate their file sizes together as the measurement of deployment overhead.

As for the cost, we store all the costs of tested serverless functions from their official website into one JSON file. For each serverless repository, we search the code files for the function names provided by the JSON file and match its cost from the JSON file according to the searched name. We use this aggregated cost metric for each repository and rerank the results based on the low-cost repository first in our search results.

In addition to the cost-based approach, we also consider the usage scenario for filtering and reranking. To achieve this, we use licenses to determine useful repositories and filter out non-useful repositories. We identify three use cases: Repository for edge distribution, Repository at cloud, and Repository for open source development. In the case of edge distribution, we cannot have any licenses in use from MPL2 to Partial, as this requires the developer to open source their complete code. Similarly, when cloud distribution is considered, and the recommended repository is being provided to the user as a service only, we filter repositories from SSPL to Partial. Finally, in the default case where open source contribution by the developer is considered as well, we rerank the repositories such that repositories with the least restrictive license are recommended with a higher priority, while the most restrictive repositories are placed at the end (see Fig. 3.3). While reranking, we maintain the relative ranking by our search algorithm.

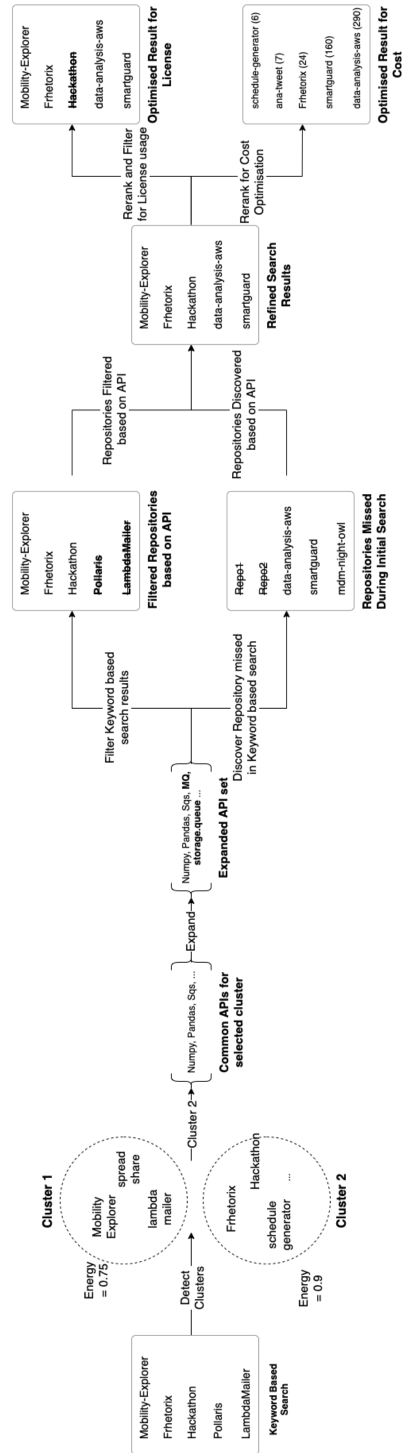

Figure 5.1: Case Study: Searching for "Machine Analysis and Learning" Serverless Functions

CHAPTER 6

Case Study

In this chapter we take a look at how OS[3] operates with real world data. We go deeper and analyze how the system interacts with the search results returned by the underlying search algorithm.

6.1 Query 1: Machine Analysis and Learning

Let us take a closer look at Fig. 5.1 that works with the search query "`machine analysis and learning`" and describes how OS^3 works internally with the results. We received 129 search results using our underlying Word2Vec based search algorithm. We apply the Louvain algorithm on the obtained results to identify the underlying clusters. Note that there were 8 clusters detected and each clusters had more repositories than depicted in the figure. We apply Step 2 in the search result refining algorithm to detect cluster with the highest energy.

In our example, we see that cluster 2 had the most energy. Using the detected cluster we find the API's with most count. In this case we were able to find some of the commonly used libraries such as *numpy*, *pandas* where analysis was involved. We also find *SQS* being used which we expand with *MQ* and *Azure Storage Queue*.

After finding the API's we use them to filter out the search results where we remove *Pollaris* and *LambdaMailer* because they didn't had any common API's. Furthermore, we also use this API to discover new repositories from our global repository database by selecting API's that had most number of API's in common with our selected API's.

In this case we were able to detect 3 repositories (*data-analysis-aws*, *smartguard*, *mdm-night-owl*) of which each had *pandas* and *numpy* in common. We then combine these filtered repositories and new recommendations together while keeping the existing recommendation intact and filling the filtered repositories (*Pollaris*, *LambdaMailer*). In instances where the number of repositories is insufficient to occupy the vacant positions, we employ an alternative strategy. Specifically, we rely on the recommendations preceding the original search result to fill the void. This approach helps ensure that all the slots are appropriately occupied and that the resulting recommendations remain comprehensive and informative.

Finally, we exemplify how the results are processed and makes changes to the recommendation based on how the user wants to use the repositories. Currently as mentioned, we define 3 common usage scenarios, Repository for edge distribution, Repository at cloud, and Repository for open source development. Based on the usage we filter the repositories that can be used for the defined usage scenario. We choose open source deployment hence, no repositories get eliminated since all licenses that we are considering allow for open source deployment.

Furthermore, we also perform reranking based on license if required where the repositories with least restrictions are ranked higher and repositories with more restrictions are placed lower in the ranking. In our case, we notice that the result *Hackathon* had a license that didn't suit our usage for the repository and was eliminated (shifted to the end of the results).

Similarly, for cost-based reranking, we apply our cost-finding algorithm to determine the relative invocation cost for these repositories for the ranking. Note that the numbers were changed for ease of explanation, but the ranking was kept intact. Furthermore, we also keep the relative license ranking intact as well. This means that license with least restrictions will be preferred compared to license with greater restrictions regardless of cost. To achieve this, an algorithm similar to bucket sort is applied where the repositories found are placed in buckets based on their licenses and then sorted within them.

CHAPTER 7

Evaluation

To evaluate the effectiveness of OS^3, we seek answers to the following research questions:
Research Question 1: Compared with basic keyword-based search approaches, how much can OS^3 improve precision?
Research Question 2: Can OS^3 accurately discover new repositories that are missing by the basic keyword-based search approaches?
Research Question 3: Can the license-based filtering improve the suitability of the search results?

7.1 Experimental Design

To generate a set of search queries that we will employ to measure the effectiveness of **keyword-based search** in comparison with **keyword-based + OS^3**, we conduct an extensive review of various datasets and research papers[42, 43, 44, 45], accumulating a total of 18 queries. These queries are formulated based on either using popular repositories for serverless or actively researched repositories.

We execute the underlying search algorithm separately, both with and without OS^3, using these queries to obtain suggestions. From these suggestions, we choose the top 5 results, disregarding the remaining ones. This provision ensures that we have enough results to make comparisions.

Subsequently, these outcomes are manually verified against their corresponding searched queries to determine the number of false positives within the top five recommendations. Two reviewers assess each search result manually. After thoroughly perusing the description file and source code of the repository, each reviewer independently determines whether the repository aligns with the search intent. If the two reviewers have differing opinions regarding a particular repository, a third reviewer is consulted to help reach the final decision.

Furthermore, to ensure the credibility of our results, we use the Cohen's Kappa inter-rater agreement coefficient to measure the level of agreement between our reviewers. The Cohen's Kappa score ranges from 0 to 1, with scores closer to 1 indicating a higher level of agreement between reviewers. We set the threshold for a satisfactory level of agreement at 0.6. If the Cohen's Kappa

score falls below this threshold, we review the recommendations again and discuss our discrepancies with more people until we reach a consensus. This approach helps with the validity and reliability of our research findings.

We also tap into the OS^3 algorithm for Keyword Matching and check how many new repositories were discovered and added based on API alone. Please note that we don't consider all the discovered repositories, but only the added repositories, as it is only the added repositories that contribute to the recommendation. The number of repositories that are added is mainly dependent on how many repositories were filtered based on missing common APIs.

7.2 Experimental Results and Discussion

Table 7.1 demonstrates 6 different search queries that we performed and how the OS^3 changes the recommended repositories. We do note that these queries are generic in nature, this is to give credibility to the search queries as they are in some form or the other directly related to repositories that are currently under research or for repositories that are used for various studies.

Furthermore, the drawbacks of overgeneralization becomes apparent when using basic search algorithms, particularly when we observe the repositories that are eliminated by the OS^3. For example, a search for "`text recognition`" generates numerous suggested repositories that are unrelated to text recognition or analysis, such as *text-my-wife*, *barrimage*, *text-adventure-api*, and *khanoi-serverless*. These repositories were suggested primarily because their Readme contained the keywords "text" (including alt text) or "recognition" in the case of barrimage.

During the initial recommendation process, the repository called *fasttext-serverless* was not given due consideration, but it was later uncovered by OS^3. Upon further exploration, it was found that this repository plays a crucial role in recommending relevant hashtags for Twitter posts by analyzing the text used in the post. This analysis is based on a popular text classification algorithm called "fastText" that can process vast amounts of text data efficiently.

The recommendation system's oversight of this repository could have led to a significant gap in its results, as it failed to consider the valuable repository. Instead of suggesting the repository, it went ahead with more explicit matches and used those repositories. This gives us more of a reason to put more focus on other features instead of just relying on the descriptions for the repositories.

The results of our study are depicted in Fig. 7.1. This figure illustrates the number of false positives returned by each search query. In order to compare the performance of different search methods, we conducted an analysis of false positives for both Keyword Matching and Keyword Matching + OS^3, which is presented in Fig. 7.2. Additionally, we also analyzed the performance of Word Vector based Search and Word Vector + OS^3 and the results are shown in Fig. 7.3.

Our analysis revealed that incorporating OS^3 with Keyword matching leads to a significant

Table 7.1: Repository Discovery and Filtering using OS^3

Search Query	**New Repositories Discovered**	**Removed Repositories**
text recognition	fasttext-serverless	text-my-wife, barrimage, text-adventure-api, khanoi-serverless
image resize	N/A	bom-radar, base64-image-receiver
machine analysis and learning	Data-Analysis-AWS, schedule-generator, mdm-nightowl, smart-guard	Pollaris, LambdaMailer
sentiment analysis	twitter-sentiment-analyzer, serverless-cognito, python-lambda-monorepo, aws-twitter-translate-bot	slackbot-reginald
crypto trading	trading-signal-processing, lambda-requestbin	serverless-github-webhook, serverless-tracking-pixel
notes or todo	serverless-notes, serverless-api, reloby	notes-js-api, ubi-be

improvement in precision by 8.89%. Specifically, the precision of Keyword matching increases from 65.55% to 74.44% when OS^3 is used.

Similarly, when we are working with Word Vector-based search, we increase the precision by 8.23% from 55.29% to 63.52%. In both cases, we can see the increased right skew in the graph. This right skew in the graph signifies the reduction in false positives during recommendation.

This finding highlights the effectiveness of OS^3 in enhancing the performance of Keyword matching. The results of our study provide valuable insights into the performance of different search methods with and without OS^3 and how it helps with recommendation.

In Fig. 7.4, we compiled the newly discovered repositories for each query. Our analysis indicates that the majority of the added repositories are aligned with the intended direction, with an accuracy

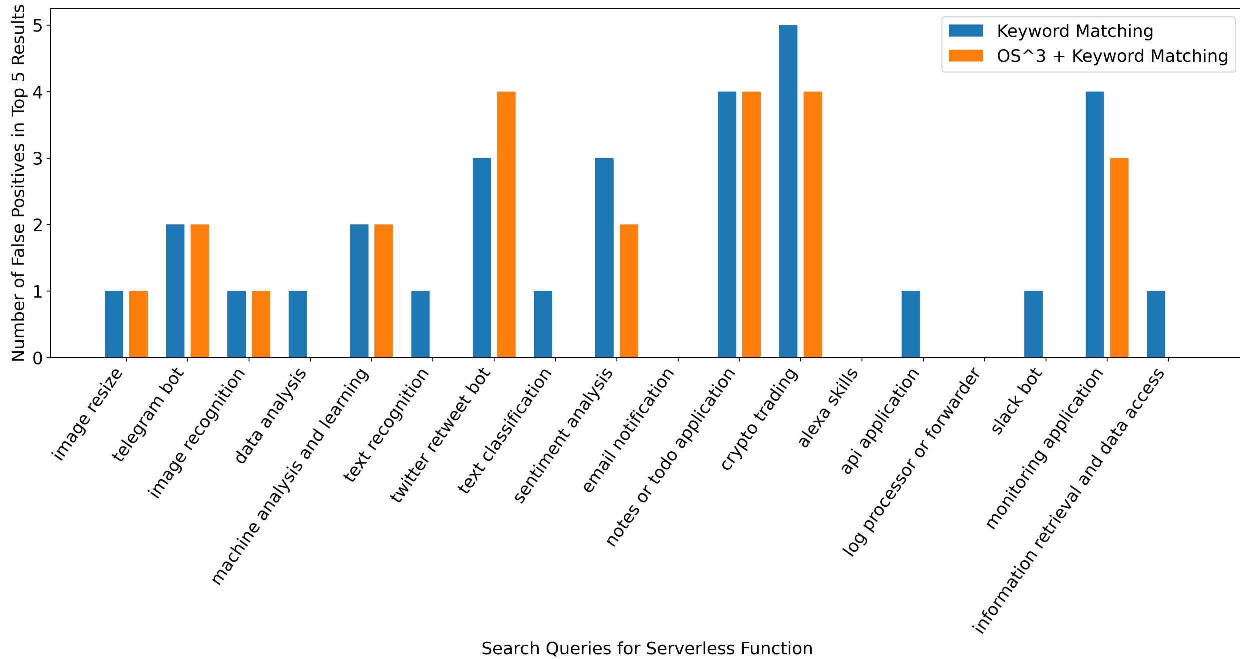

Figure 7.1: Comparison of False Positive Results w/ and w/o OS^3 for all Search Queries

rate of 70.58%. However, there are instances where we may have mistakenly suggested inaccurate recommendations. Sometimes we might also have removed the correct ones. Such situations cannot be entirely avoided.

Such inaccuracies can arise when the keyword match is weak and the learned APIs are not indicative of the search query. A case in point can be observed in Table. 7.1 for the query "`sentiment analysis`", where unsuitable repositories, such as *serverless-cognito*, *python-lambda-monorepo*, and *aws-twitter-translate-bot*, were included as recommendations. This occurred because the suggested libraries were more relevant to Twitter rather than sentiment analysis.

We see a similar example in query for "`crypto trading`" where *lambda-requestbin* was added to set of api based suggested repositories. Note that we do not use all the suggestions but only use them on need basis. i.e. when a repository is removed from recommended results due to missing common API.

Additionally, on a side note, we also see an incorrect removal when querying for "`notes or todo`" where *notes-js-api* was removed incorrectly. This is due to a low assigned score by the search algorithm and a library mismatch between the selected libraries and the libraries in the repository.

When we examined the licenses used by the recommended repositories, we discovered that about 11% repositories are unusable in search results. For the usability criteria, we applied the strictest criteria to find an upper bound by only considering least restrictive licenses, i.e., licenses from Unlicense to Apache2 in Fig. 3.3. This filtering process enables us to take into account the

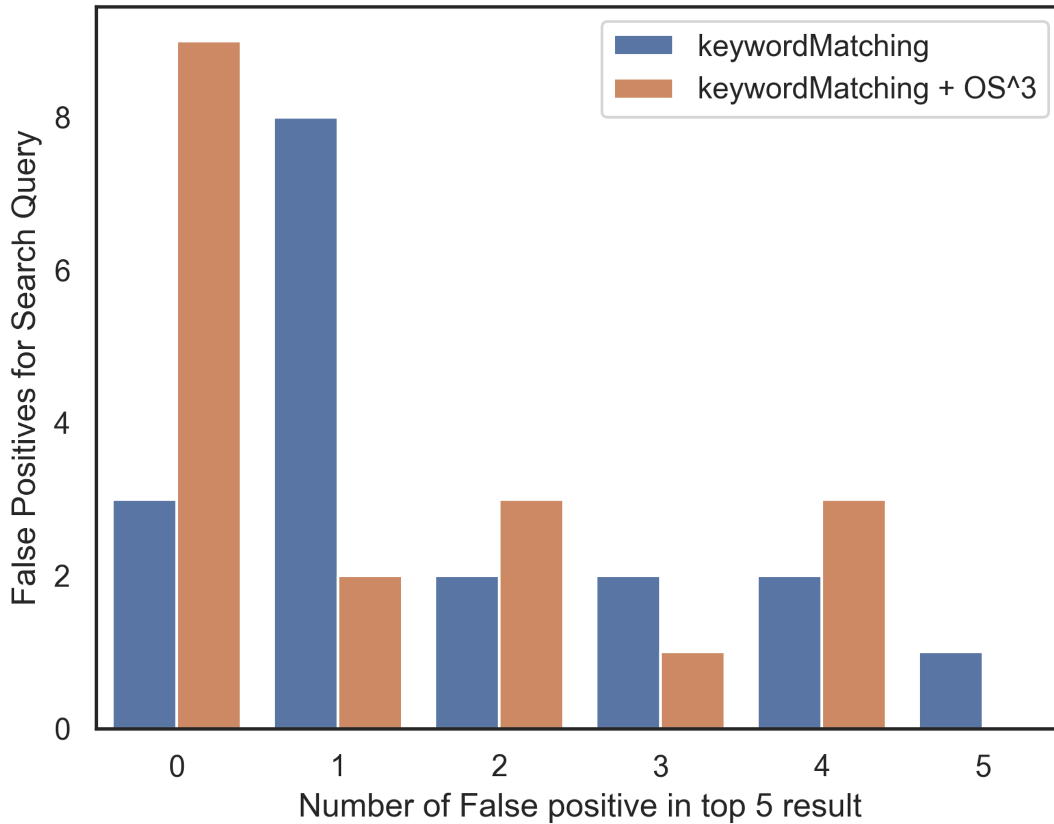

Figure 7.2: False Positives For Keyword Matching with and without OS^3

general use case and facilitates the assessment of repositories relevant to each scenario.

Specifically, we observed that out of 90 repositories, approximately 10 would have been unsuitable for use in 18 queries, thereby warranting attention. It is worth noting that the percentage of unusable repositories among the top 5 search results for a query, i.e., 11%, is relatively high compared to the overall proportion of such repositories, which is only 17.2% as depicted in Fig. 3.3.

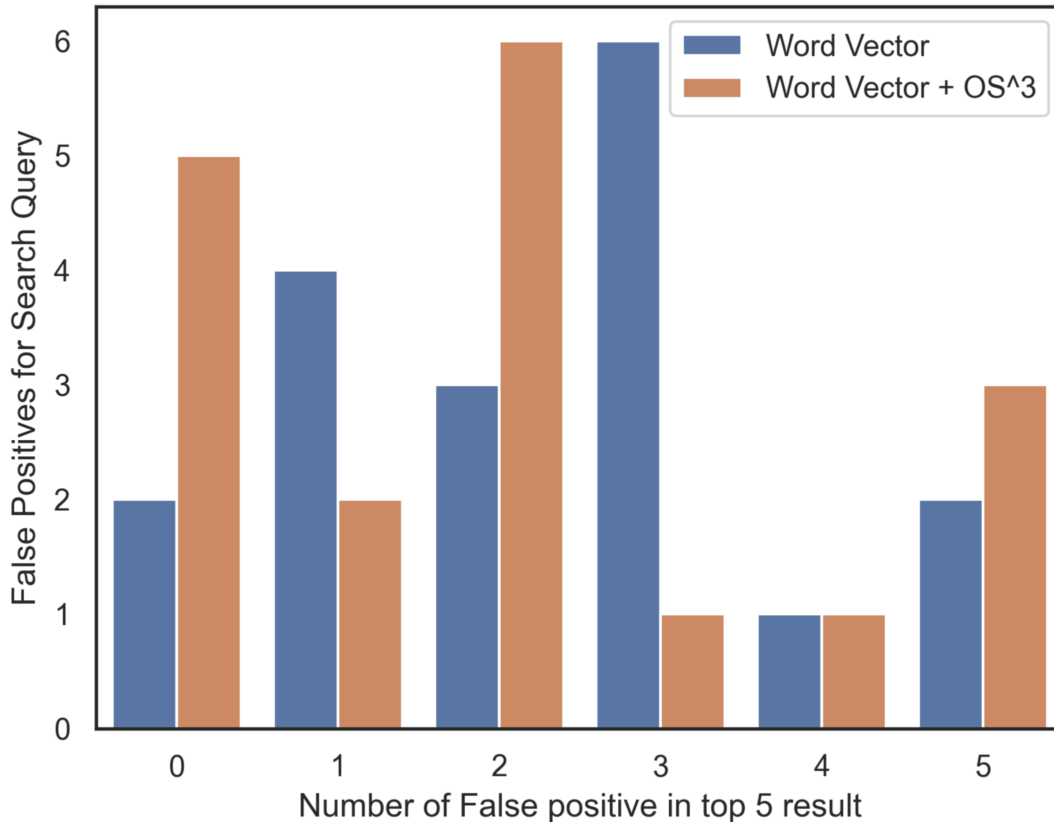

Figure 7.3: False Positives For Word Vector Based Search with and without OS^3

7.3 Interpreting Experimental Results to Address Research Questions

As discussed earlier, we are mainly interested in answering the below three research questions. Based on our evaluation results, we answer the research questions above as follows:

RQ1: Compared with basic keyword-based search approaches, how much can OS^3 improve precision?

Using OS^3 over the underlying search algorithms does increase the precision by a decent amount; in our evaluations, we find that to be by about 8.89% for keyword matching algorithm and 8.23% for word vector based approach.

RQ2: Can OS^3 accurately discover new repositories that are missing by the basic keyword-based search approaches?

Our evaluation shows that OS^3 is capable of uncovering new repositories that were not detected by the underlying search algorithm, as depicted in Fig. 7.4, with an accuracy rate of 70.58

RQ3: Can the license-based filtering improve the suitability of the search results?

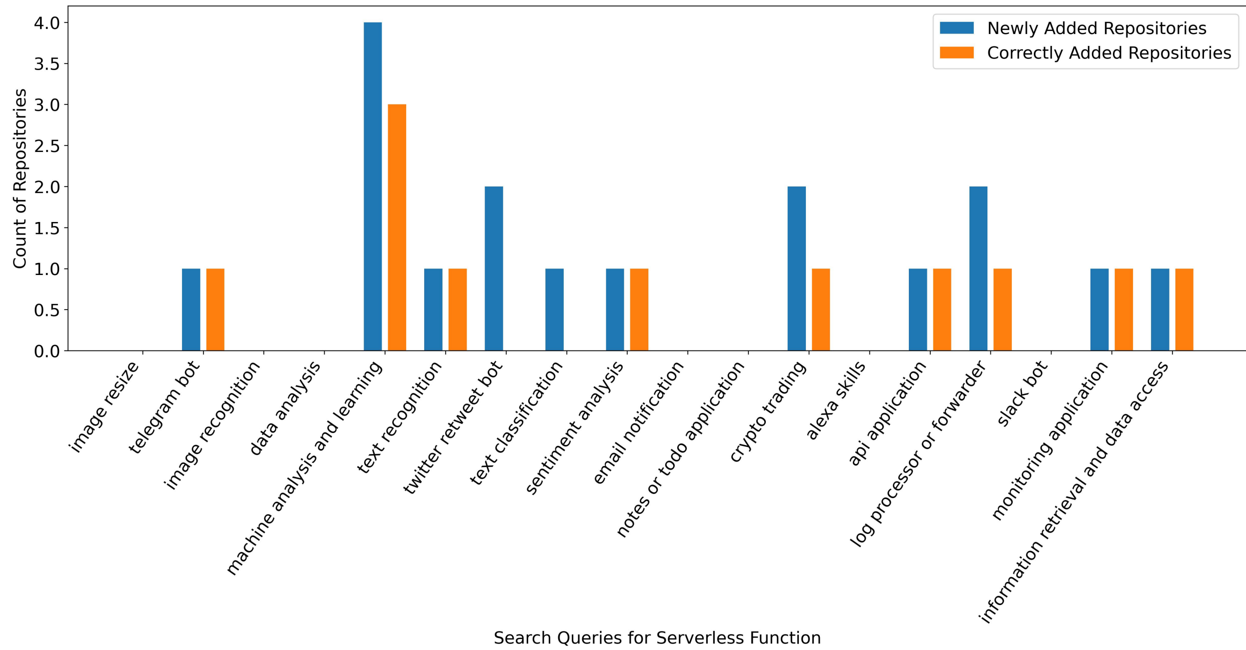

Figure 7.4: Accuracy of the Repositories Appended by OS^3 for all Search Queries

Our assessment reveals that approximately 11% of the recommended repositories, among the top-ranked ones, may be unsuitable for real-life applications. Moreover, the sample set for nearly unusable licenses accounts for about 17.2%, thereby underscoring the significance of taking into account the license usage when making recommendations.

CHAPTER 8

Related Work

In this section, we introduce the current status of serverless function sharing, and code-based search and recommendation.

8.1 Serverless Computing Dataset's

The field of serverless computing is still in its nascent stages, and as such, there is a paucity of datasets available for researchers and practitioners to use. While there have been some attempts at collecting data for serverless computing, the datasets that have been gathered so far have been either too small or limited to a single source. This makes it difficult for researchers to draw meaningful conclusions about the performance and efficacy of serverless computing as a whole.

One of the primary challenges in gathering data for serverless computing is the issue of licensing. Many repositories that contain code related to serverless computing may not have proper licensing, which means that those repositories becomes unusable as per the copyright laws. This can make it difficult for researchers to reproduce, distribute, or create derivative works from the code. This is particularly problematic when trying to find functions, as it is already challenging to find suitable functions for serverless computing.

One attempt at gathering data for serverless computing is the Wonderless project. The authors of Wonderless collect data by scraping the serverless framework's specification file, serverless.yml, from GitHub. They then preprocess the data and only keep repositories that are recently active and popular, as measured by recent commits and the number of stars for that repository. While this approach has yielded a dataset of 1,877 repositories, it poses certain problems for researchers. For example, it excludes many repositories that are not popular or recently updated, even if they could be useful for recommendation purposes. Additionally, Wonderless only considers repositories that use the serverless framework, which limits the dataset to a single source. Furthermore they also fail to consider licenses making 56% repositories moot reducing the number of usable repositories to 819 repositories.

To address these challenges, we have collected the largest multi-source serverless dataset to

date. By using this dataset, we can overcome the limitations of the Wonderless dataset and provide more comprehensive and diverse data for researchers and practitioners in the field of serverless computing. By having access to a larger and more diverse dataset, we can better understand the performance and effectiveness of serverless computing and make more informed recommendations for its use.

8.2 Serverless Function Sharing and Reusing

The modular design of serverless functions inherently makes them an attractive target for sharing and reusing. One obstacle in serverless development is getting used to platform-specific APIs. Finding open source implementations of serverless functions with similar functionalities enables developers to implement their business logic incrementally, thereby saving their time and effort.

To aid developers in this effort, Many serverless vendors (e.g., Google Cloud Functions) and the Serverless framework create and share boilerplate templates for implementing basic functionalities. There have also been some attempts at enabling developers to share their own serverless functions (e.g., AWS Serverless Application Repository). However, the attempts for encouraging serverless code sharing have not been successful as of yet, with only 1,438 repositories being shared on AWS SAR. Many developers still use GitHub to develop and share their serverless functions.

However, it is a challenging task for serverless developers to search for the required functionalities. The current serverless repositories are chained to certain platforms, while searching for serverless functions on Github fails to take serverless specific features, such as platform, plugins, and events into consideration. The absence of a search engine specifically tailored for serverless functions reduces the accuracy of search results and hinders the sharing and reusing of serverless functions as a whole.

To address this challenge, we suggest improvements in search and recommendation domain specifically for serverless functions. Serverless functions present many peculiarities that can be utilized for helping developers focus on their core logic and develop with the aid of some helpful recommendations. As far as we know, our approach is the first in implementing a search engine that considers the unique features of serverless functions and is tailored for serverless. By improving the accuracy and efficiency of serverless function search, our approach has the potential to significantly increase the sharing and reusing of serverless functions among developers, thereby accelerating the development of serverless applications.

8.3 Code-Based Search and Recommendation

There have been many approaches for code-based search and recommendation that take into account the semantics or structure of code snippets. One such recent attempt is Aroma [46], which

focuses on finding contextual code samples dissimilar from each other within the structural similarity scope, and thereby helps in finding different ways a piece of code has been written by different developers. Another approach is to use code to code pattern matching that could suggest similar code using code snippet as a query. Researchers have also tried using AST trees to enable code vector embedding that can be used for code recommendation [47]. However, such semantics-based or structure-based approaches are language dependent and require dedicated parsers for different languages. Compared to the code-based search approach, our underlying keyword-based search is naturally more suitable for the language agnostic feature of serverless functions.

CHAPTER 9

Conclusion and Future Directions

To address the emerging need for an efficient search and recommendation system for serverless functions this paper introduces OS^3, a custom-tailored search engine for serverless functions. Furthermore, we also collect and open source the largest dataset for serverless functions repositories that comprise over 5,981 licensed serverless repositories, collected from multiple sources belonging to three major platforms, AWS, Azure, and IBM.

OS^3 is optimized to search for serverless functions by focusing on serverless specific features. It does that by taking licenses, unique properties such as invocation cost, and higher library correlation into consideration. Our evaluation demonstrates that OS^3 can become a valuable tool in developing serverless applications, helping developers to efficiently search and find the most suitable serverless functions for their projects.

To our knowledge, our work is the first to shed light on share and reuse of serverless functions providing a high quality dataset and pushing towards research in search and recommmendation for serverless functions. By taking into account the specific traits of serverless functions, including externally managed library usage, cost, and license, a search engine can significantly increase the accuracy and comprehensibility of the search results. Such improvements would better promote reuse of existing serverless functions and motivate the open source community to share their work as well.

Moving forward, we plan to extend this work in several directions. Firstly, we plan to use program analysis techniques to accurately locate the entries of serverless functions as well as their control flow paths within a serverless function repository. Secondly, we aim to work on an estimator that would help us estimate the execution costs based on the code provided. Thirdly, we would like to categorize the data to implement more advanced models for recommendation.

While current research and development efforts are primarily focused on infrastructure improvements (addressing issues such as the cold start problem and statelessness of serverless functions), there is ample scope for research in code recommendation systems that aid developers in discovering the most relevant serverless functions. This would allow us to utilize the inherent features of serverless functions and enable developers to code faster and obtain relevant information.

APPENDIX

Relevant Paper Publications

A.1 OS^3: The Art and the Practice of Searching for Open-Source Serverless Functions

Citation:

S. Bhatnagar, Z. Li, Z. Song, and E. Tilevich, “Os 3: The art and the practice of searching for open-source serverless functions,” in 2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops).

Abstract:

Serverless computing enables service developers to focus on creating useful services, without being concerned about how these services would be deployed and provisioned. Many developers reuse existing open-source serverless functions to create their own functions. However, existing technologies for searching open-source software repositories have not taken into consideration the unique features of serverless functions. This paper presents a novel approach to searching for serverless functions, called **O**pen-**S**ource **S**erverless **S**earch (OS^3) that maximizes the utility of the returned serverless functions by (1) basing the search process on both descriptive keywords and library usages, thus increasing the search results' precision and completeness; (2) filtering and ranking the search results based on the software license, to accommodate the unique requirements of deploying serverless functions on dissimilar platforms, including cloud and edge computing. Implemented in 3K lines of Python, with a search space of 5,981 serverless repositories from four major serverless platforms, OS^3 outperforms existing search approaches in terms of the suitability of the search results, based on our evaluation with realistic use cases.

Commentary:

This paper discusses and proposes the OS^3 model that we use to improve the search and recommendation for serverless functions.

A.2 Sharing and Reusing Serverless Functions: Software Licensing Constraints on My Mind

Citation:

S. Bhatnagar, Z. Li, Z. Song, and E. Tilevich, "Sharing and Reusing Serverless Functions: Software Licensing Constraints on My Mind" in 2023 IEEE Communications Magazine. (Under Review as of date 1st April, 2023)

Abstract:

Serverless computing has become an important model for developers aspiring to build scalable, efficient, and responsive applications in both cloud and edge environments. Serverless enables service developers to focus on creating useful services, rather than on deploying and provisioning these services. In this article, we report on the results of a study for which we collected and analyzed open-source serverless repositories from various sources. We observe a rapid increase in the amount and quality of open-source serverless repositories, a trend that reflects that the open-source community has broadly embraced the serverless paradigm. By further studying the dataset, we discuss two concerns related to the sharing and reusing of serverless functions: unique software licensing issues and the lack of a customized search engine. We discuss these concerns and their potential solutions and also describe our proposed approach to searching for serverless functions, **O**pen-**S**ource **S**erverless **S**earch (OS^3), which considers both descriptive keywords and library usages in the search process. We also propose several potential solutions for finding serverless functions suitable for the usage scenarios and integrate some of them into OS^3.

Commentary:

This article is a revised and extended version of a workshop paper, presented at the STARLESS Workshop 2023. This modified focus of our article is reflective of the discussion that followed the presentation of our paper at the workshop. Specifically, the revised manuscript puts a special emphasis on discussing the software licensing issues and proposing potential solutions. Besides, this article provides new statistical insights derived from: 1) analyzing the statistics of the open source dataset we collected; 2) comparing our dataset with the "Wonderless" dataset collected two years ago following similar methodologies. Quantitatively, we thoroughly edited 2 pages of the original manuscript and added 2 pages of brand new content. We also open-sourced our dataset and search engine at Github.

References

[1] "Aws re:invent 2014 recap," https://aws.amazon.com/blogs/developer/aws-reinvent-2014-recap-2/, accessed: 2023-2-17.

[2] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: A survey of opportunities, challenges, and applications," *ACM Computing Surveys (CSUR)*, 2019.

[3] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "Serverless applications: Why, when, and how?" *IEEE Software*, vol. 38, no. 1, pp. 32–39, 2020.

[4] R. Crespo-Cepeda, G. Agapito, J. L. Vazquez-Poletti, and M. Cannataro, "Challenges and opportunities of amazon serverless lambda services in bioinformatics," in *Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, 2019, pp. 663–668.

[5] P. A. Witte, M. Louboutin, H. Modzelewski, C. Jones, J. Selvage, and F. J. Herrmann, "An event-driven approach to serverless seismic imaging in the cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 9, pp. 2032–2049, 2020.

[6] V. Giménez-Alventosa, G. Moltó, and M. Caballer, "A framework and a performance assessment for serverless mapreduce on aws lambda," *Future Generation Computer Systems*, vol. 97, pp. 259–274, 2019.

[7] P. Vahidinia, B. Farahani, and F. S. Aliee, "Cold start in serverless computing: Current trends and mitigation strategies," in *2020 International Conference on Omni-layer Intelligent Systems (COINS)*. IEEE, 2020, pp. 1–7.

[8] "Improving startup performance with lambda snapstart," https://docs.aws.amazon.com/lambda/latest/dg/snapstart.html, accessed: 2023-2-10.

[9] "Azure functions overview," https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview, accessed: 2022-8-17.

[10] "Ibm cloud functions," https://developer.ibm.com/components/ibm-cloud-functions/tutorials/, accessed: 2022-8-17.

[11] "Aws lambda sample applications," https://docs.aws.amazon.com/lambda/latest/dg/lambda-samples.html, accessed: 2022-8-17.

[12] "All cloud functions code samples," https://cloud.google.com/functions/docs/samples/, accessed: 2022-5-10.

[13] "Serverless framework," https://github.com/serverless/serverless/, accessed: 2022-5-10.

[14] "Aws serverless application repository," https://serverlessrepo.aws.amazon.com/applications, accessed: 2022-5-10.

[15] L.-P. Jing, H.-K. Huang, and H.-B. Shi, "Improved feature selection approach tfidf in text mining," in *Proceedings. International Conference on Machine Learning and Cybernetics*, vol. 2. IEEE, 2002, pp. 944–946.

[16] N. Eskandani and G. Salvaneschi, "The wonderless dataset for serverless computing," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 565–569.

[17] I. Pavlov, S. Ali, and T. Mahmud, "Serverless development trends in open source: a mixed-research study," 2019.

[18] "Cloud object storage," https://aws.amazon.com/s3/, accessed: 2022-5-10.

[19] "Fast nosql key-value database," https://aws.amazon.com/dynamodb/, accessed: 2022-5-10.

[20] "Conversational ai and chatbots," https://aws.amazon.com/lex/, accessed: 2022-5-10.

[21] "Amazon polly," https://aws.amazon.com/polly/, accessed: 2022-5-10.

[22] "Amazon sqs — message queuing service," https://aws.amazon.com/sqs/, accessed: 2022-5-10.

[23] "60678 libraries on github," https://www.overops.com/blog/we-analyzed-60678-libraries-on-github-here-are-the-top-100/, accessed: 2022-5-10.

[24] D. Taibi, J. Spillner, and K. Wawruch, "Serverless computing-where are we now, and where are we heading?" *IEEE Software*, vol. 38, no. 1, pp. 25–31, 2020.

[25] "Mit license," https://choosealicense.com/licenses/mit/, accessed: 2022-5-11.

[26] "Apache license 2.0," https://choosealicense.com/licenses/apache-2.0/, accessed: 2022-5-11.

[27] "Gnu lesser general public license v3.0," https://choosealicense.com/licenses/lgpl-3.0/, accessed: 2022-5-11.

[28] "Gnu general public license v3.0," https://choosealicense.com/licenses/gpl-3.0/, accessed: 2022-5-11.

[29] "Gnu affero general public license v3.0," https://choosealicense.com/licenses/agpl-3.0/, accessed: 2022-5-11.

[30] P. Chaudhari and M. L. Das, "Keysea: Keyword-based search with receiver anonymity in attribute-based searchable encryption," *IEEE Transactions on Services Computing*, 2020.

[31] J. Zhao, J. X. Huang, H. Deng, Y. Chang, and L. Xia, "Are topics interesting or not? an lda-based topic-graph probabilistic model for web search personalization," *ACM Trans. Inf. Syst.*, vol. 40, no. 3, dec 2022. [Online]. Available: https://doi.org/10.1145/3476106

[32] Z. Gao, Y. Fan, C. Wu, W. Tan, J. Zhang, Y. Ni, B. Bai, and S. Chen, "Seco-lda: Mining service co-occurrence topics for composition recommendation," *IEEE Transactions on Services Computing*, vol. 12, no. 3, pp. 446–459, 2019.

[33] B. A. Yilma, N. Aghenda, M. Romero, Y. Naudet, and H. Panetto, "Personalised visual art recommendation by learning latent semantic representations," in *2020 15th International Workshop on Semantic and Social Media Adaptation and Personalization (SMA*, 2020, pp. 1–6.

[34] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[35] T. Montecchi, D. Russo, and Y. Liu, "Searching in cooperative patent classification: Comparison between keyword and concept-based search," *Advanced Engineering Informatics*, vol. 27, no. 3, pp. 335–345, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1474034613000219

[36] G. Liu, G. Yang, S. Bai, H. Wang, and Y. Xiang, "Fase: A fast and accurate privacy-preserving multi-keyword top-k retrieval scheme over encrypted cloud data," *IEEE Transactions on Services Computing*, vol. 15, no. 4, pp. 1855–1867, 2022.

[37] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[38] L. Jiang, Y. Kalantidis, L. Cao, S. Farfade, J. Tang, and A. G. Hauptmann, "Delving deep into personal photo and video search," in *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, 2017, pp. 801–810.

[39] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, p. P10008, 2008.

[40] C. Ebert, J. Cain, G. Antoniol, S. Counsell, and P. Laplante, "Cyclomatic complexity," *IEEE software*, vol. 33, no. 6, pp. 27–29, 2016.

[41] "Mccabe - code complexity checker," https://here-be-pythons.readthedocs.io/en/latest/python/mccabe.html, accessed: 2022-5-10.

[42] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation*, 2019, pp. 225–236.

[43] B. Carver, J. Zhang, A. Wang, and Y. Cheng, "In search of a fast and efficient serverless dag engine," in *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*. IEEE, 2019, pp. 1–10.

[44] J. Spillner, "Quantitative analysis of cloud function evolution in the aws serverless application repository," *arXiv preprint arXiv:1905.04800*, 2019.

[45] M. Obetz, S. Patterson, and A. Milanova, "Static call graph construction in {AWS} lambda serverless applications," in *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[46] S. Luan, D. Yang, C. Barnaby, K. Sen, and S. Chandra, "Aroma: Code recommendation via structural code search," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–28, 2019.

[47] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[48] D. E. Knuth, *The TeXbook*. Addison-Wesley Professional, 1986.

[49] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," *arXiv preprint arXiv:1812.03651*, 2018.

[50] J. Nupponen and D. Taibi, "Serverless: What it is, what to do and what not to do," in *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2020, pp. 49–50.

[51] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. Abad, and A. Iosup, "The state of serverless applications: Collection, characterization, and community consensus," *IEEE Transactions on Software Engineering*, 2021.

[52] S. Bhatnagar, Z. Li, Z. Song, and E. Tilevich, "Os 3: The art and the practice of searching for open-source serverless functions," in *2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*.

[53] "Open source tool for cyclomatic complexity," https://github.com/long2ice/mccabe, accessed: 2022-8-31.

[54] "App manifest attribute reference," https://docs.cloudfoundry.org/devguide/deploy-apps/manifest-attributes.html, accessed: 2022-8-17.

[55] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking functions to warm the serverless cold start," in *Proceedings of the 21st International Middleware Conference*, 2020, pp. 1–13.

[56] S. Lee, D. Yoon, S. Yeo, and S. Oh, "Mitigating cold start problem in serverless computing with function fusion," *Sensors*, vol. 21, no. 24, p. 8416, 2021.

[57] R. Chatley and T. Allerton, "Nimbus: Improving the developer experience for serverless applications," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, 2020, pp. 85–88.

[58] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, “Edge computing: Vision and challenges,” *IEEE internet of things journal*, vol. 3, no. 5, pp. 637–646, 2016.

[59] L. Baresi and D. F. Mendonça, “Towards a serverless platform for edge computing,” in *2019 IEEE International Conference on Fog Computing (ICFC)*. IEEE, 2019, pp. 1–10.

[60] A. Glikson, S. Nastic, and S. Dustdar, “Deviceless edge computing: extending serverless computing to the edge of the network,” in *Proceedings of the 10th ACM International Systems and Storage Conference*, 2017, pp. 1–1.

[61] M. S. Aslanpour, A. N. Toosi, C. Cicconetti, B. Javadi, P. Sbarski, D. Taibi, M. Assuncao, S. S. Gill, R. Gaire, and S. Dustdar, “Serverless edge computing: vision and challenges,” in *2021 Australasian Computer Science Week Multiconference*, 2021, pp. 1–10.

[62] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, “The rise of serverless computing,” *Communications of the ACM*, vol. 62, no. 12, pp. 44–54, 2019.