Tampere University

Tamara Meghla

TESTING SERVERLESS APPLICATIONS

A Systematic Review of Frameworks, Tools and Best Practices

Faculty of Information Technology and Communication Sciences
M. Sc. Thesis
Davide Taibi
Xiaozhou Li
June 2023

ABSTRACT

Tamara Meghla: Testing Serverless Applications: A systematic Review of Frameworks, Tools & Best Practices
Master's Thesis
Tampere University
Master's Degree Programme in Computing Sciences
June 2023

Testing serverless applications presents unique challenges due to the distributed and event-driven nature of these architectures. This thesis aims to address these challenges by conducting a systematic literature review to explore testing techniques, patterns, anti-patterns, and tools specific to serverless applications. The research questions focus on identifying the available testing techniques, evaluating testing tools, and understanding the challenges encountered in testing serverless applications.

The findings indicate that there are many different ways to test serverless applications. These include unit testing, integration testing, performance testing, and monitoring & debugging. The techniques provide comprehensive coverage, improve reliability and stability, enable faster feedback cycles, mitigate risks, and enhance system resilience. However, they come with certain restrictions, such as increased complexity, limited scope, test environment restrictions, and the need for specialized knowledge.

The evaluation of testing tools reveals a diverse landscape of options, including performance testing, infrastructure and deployment, monitoring and observability, distributed tracing and debugging, language-specific testing, test case generation, cloud service-specific testing, and specialized graph modeling tools. Each tool has advantages and limitations, and developers must evaluate their specific demands and limits while selecting the proper tools.

Based on the research, several best practices are synthesized for testing serverless applications. Emphasizing performance testing, automating testing processes, considering cross-platform compatibility, validating event flow and interactions, and adopting distributed tracing and debugging techniques are identified as crucial practices for ensuring the quality and reliability of serverless applications.

The thesis acknowledges its limitations, such as the exclusion of gray literature, the lack of practical implementation, and the potential oversight of relevant studies. Future research is recommended to incorporate gray literature, conduct practical experiments, and investigate specific application scenarios to enhance the understanding and applicability of testing techniques for serverless applications.

Overall, this thesis provides valuable insights into the landscape of testing techniques, tools, challenges, and best practices for serverless applications. The findings contribute to the existing body of knowledge and offer guidance to practitioners and researchers in improving testing practices and ensuring the quality and reliability of serverless applications.

Keywords: Serverless Function, Function-as-a-Service, Serverless Testing, Regression, Debugging, AWS Lambda, Google Cloud Function, Microsoft Azure Function, OpenWhisk, OpenFaaS, OpenLambda, Fn.

PREFACE

I want to thank my thesis supervisor, Davide Taibi, for his constant support and guidance. I am also grateful to my friends and family, especially Miikka Mänttäri, for their unwavering encouragement. When circumstances were challenging, your belief in me and your understanding kept me going.
Special thanks to Xiaozhou Li for his valuable guidance. Thank you to everyone who helped me during my thesis and academic journey.

Tampere, 02 June 2023

Tamara Meghla

CONTENTS

1. INTRODUCTION 1
1.1 Context and Description 1
1.2 Motivation 1
1.3 Research Questions and Objectives 2
1.4 Solution 3
1.5 Key Contributions 3
1.6 Structures 4
2. BACKGROUND: SERVERLESS ARCHITECTURE AND PLATFORMS 6
2.1 Serverless Computing 6
2.2 Serverless Architecture 7
2.3 Serverless Platforms 7
2.3.1 AWS Lambda 7
2.3.2 Google Cloud Function 9
2.3.3 Microsoft Azure Function 9
2.3.4 OpenWhisk 10
2.3.5 OpenLambda 11
2.3.6 OpenFaaS 12
2.3.7 Fn 12
2.3.8 Other Serverless Platforms 13
2.4 Common Use Cases for Serverless Architecture 13
2.4.1 Event-Driven Applications 13
2.4.2 Microservices 13
2.4.3 APIs 13
2.4.4 Data Processing 14
2.4.5 Scheduled Tasks 14
2.5 Benefits & Challenges of Serverless Architecture 14
2.5.1 Cost 14
2.5.2 Cold Start 15
2.5.3 Vendor Lock-in 19
2.5.4 Technology Limitations 20
2.5.5 Event-Based Programming Model 20
2.5.6 Monitoring & Observability 21
2.5.7 Performance-Related Issues 21
2.5.8 Technical Debt 22
2.6 Serverless Deployment Strategy 23
2.7 Serverless Application Pattern 23
2.7.1 Externalized State 23
2.7.2 Routing Function 24
2.7.3 Function Chain 24
2.7.4 Function Pinging 24
2.7.5 Oversized Function 24
3. TESTING FUNDAMENTALS 25
3.1 The Importance of Testing in Software Development 25

3.2 Testing Levels 25
3.2.1 Unit Testing 26
3.2.2 Integration Testing 26
3.2.3 System Testing 27
3.2.4 Acceptance Testing 28
3.3 Testing in Serverless 28
3.4 Challenges in Testing Serverless Applications 29
3.5 Testing and Debugging Approaches in Existing Literature 32
3.5.1 Debugging Approaches 35
3.5.2 Crash-Reproducing Test Cases 37
3.5.3 Other Best Practices Across Developers 40
3.5.4 Determining Adequacy: When is Testing Sufficient? 42
3.5.5 Approaches to Determine Coverage Criteria 44
3.5.6 Testing Tools 46
4. METHODOLOGY 52
4.1 Research Approach and Design 52
4.2 Search Strategy and Selection Criteria 53
4.2.1 Search Strategy 53
4.2.2 Selection Criteria 55
4.3 Data Extraction and Synthesis 56
4.3.1 Data Extraction 56
4.3.2 Data Synthesis 56
4.4 Quality Assessment and Bias Control 57
4.4.1 Quality Assessment 57
4.4.2 Bias Control 57
4.4.3 Assessment of Inter-Rater Agreement 58
4.5 Snowballing 59
4.6 Creation of a Final Pool of Sources 60
5. FINDINGS AND DISCUSSION 61
5.1 Testing Techniques, Patterns, and Anti-patterns in Serverless Applications (RQ1) 61
5.1.1 Common Testing Techniques in Serverless Applications 61
5.1.2 Testing Approaches in Serverless Applications 65
5.1.3 Testing Patterns in Serverless Applications 66
5.1.4 Anti-patterns in Serverless Testing 67
5.1.5 Unique Considerations in Serverless Testing 68
5.1.6 Discussion and Analysis of Testing Approaches in Serverless Applications 70
5.2 Tools for Testing Serverless Applications (RQ2) 71
5.2.1 Available Tools for Testing Serverless Applications 71
5.2.2 Comparison of Testing Tools 77
5.2.3 Discussion and Analysis of Testing Tools for Serverless Applications 79
5.3 Challenges in Testing Serverless Applications (RQ3) 83
5.3.1 Testing Issues in Serverless Applications 83
5.3.2 Discussion: Addressing the Challenges in Testing Serverless Applications (RQ3) 86

5.4 Evaluation and Comparison of Identified Solutions 87
5.4.1 Strengths and Weaknesses 87
5.4.2 Common Patterns and Trends 89
5.5 Evaluation of Testing Tools 89
5.6 Synthesis of Best Practices in Testing Serverless Applications 90
5.7 Limitations and Future Research 91
6. CONCLUSIONS 93
REFERENCES

1. INTRODUCTION

1.1 Context and Description

Serverless architecture, also known as Function as a Service (FaaS), has gained significant popularity in recent years. It offers a paradigm shift in application development, allowing developers to focus on writing code for individual functions without having to manage the underlying infrastructure. Applications built using a serverless architecture consist of stateless functions that scale automatically in response to demand.

The context of this thesis revolves around testing serverless applications. As serverless architecture becomes more prevalent, it is essential to ensure the quality and reliability of these applications. However, testing serverless applications presents unique challenges compared to traditional application testing approaches. The distributed and event-driven nature of serverless functions, the lack of control over the underlying infrastructure, and the complexities of integrating multiple functions and external services require specialized testing techniques and tools.

1.2 Motivation

The adoption of serverless architecture has grown rapidly in recent years. Many organizations are leveraging the benefits of serverless computing to build scalable and cost-effective applications. However, with the increasing complexity of serverless applications, testing them effectively has become a critical challenge. Traditional testing approaches and tools are often inadequate for serverless applications, which require new techniques and strategies to ensure their quality and reliability.

This research aims to address the testing challenges specific to serverless applications. We aim to provide insights and guidelines for practitioners and researchers in the field by investigating and evaluating the available testing techniques and tools. The goal is to enhance the quality and reliability of serverless applications by identifying effective testing practices.

In this thesis, we focus on conducting a systematic literature review to explore and assess the existing knowledge related to testing serverless applications. By addressing the research questions outlined in this study, we aim to contribute to understanding testing techniques, patterns, and tools in the context of serverless architecture. The findings

and insights from this research will assist practitioners in making informed decisions regarding testing strategies and tool selection for their serverless applications.

Overall, this thesis aims to bridge the gap between the increasing adoption of serverless architecture and the need for effective testing methodologies. By addressing the unique challenges of testing serverless applications, we strive to improve serverless deployments' quality, reliability, and overall success in real-world scenarios.

1.3 Research Questions and Objectives

In light of the challenges faced in testing serverless applications, this thesis aims to investigate and assess the available testing techniques and tools for serverless architecture. The research problem at the core of this thesis can be distilled into the following three questions:

RQ1: What testing techniques, patterns, and anti-patterns are available for serverless applications?

The first question aims to explore the landscape of testing techniques employed in serverless applications. This includes identifying the approaches, methodologies, and patterns proposed and utilized for testing serverless applications. This question also seeks to identify any anti-patterns or common pitfalls developers should avoid when testing serverless applications.

RQ2: What testing tools are available for serverless applications?

The second question investigates the range of testing tools designed and developed for serverless applications. This includes commercial and open-source tools that assist developers in testing various aspects of serverless applications, such as unit testing, integration testing, performance testing, and monitoring & debugging. The objective is to provide an overview of the available tools and their features, strengths, and limitations.

RQ3: What are the challenges in testing serverless applications?

The third research question focuses on the challenges encountered in testing serverless applications. Through the systematic literature review, the thesis aims to identify and analyze the common obstacles, limitations, and complexities associated with testing serverless architectures. By understanding these challenges, the thesis aims to provide insights into potential areas of improvement and future research directions to enhance testing practices for serverless applications.

The thesis aims to contribute to the existing body of knowledge on testing serverless applications by conducting a systematic literature review and addressing these research questions. The review findings will inform practitioners and researchers about the available testing techniques, tools, and challenges specific to the serverless architecture. This

comprehensive understanding will enable better decision-making and foster the development of more effective and efficient testing practices for serverless applications.

1.4 Solution

A systematic literature review will be conducted to address the research questions and objectives outlined in this thesis. The review will identify relevant research articles, conference papers, and other scholarly sources discussing testing techniques, patterns, and tools for serverless applications. The selected literature will be analyzed and synthesized to extract key findings, insights, and recommendations.

Based on the literature review findings, a comprehensive overview of testing techniques, patterns, and anti-patterns for serverless applications will be presented. This will include a discussion of different approaches and methodologies used in testing serverless functions and any common pitfalls and challenges to be aware of. The aim is to provide a comprehensive understanding of the available testing techniques and their applicability in serverless architectures.

Additionally, the thesis will investigate the range of testing tools available for serverless applications. Both commercial and open-source tools will be examined to identify their features, strengths, and limitations. This analysis will assist practitioners and researchers in selecting the most appropriate tools for their testing needs.

The literature review will identify and analyze the challenges in testing serverless applications. This will involve understanding the complexities and limitations associated with testing serverless architectures, such as the distributed nature of functions, the lack of control over the underlying infrastructure, and the difficulties in reproducing and debugging issues. By gaining insights into these challenges, the thesis aims to provide recommendations and potential areas of improvement for testing practices in serverless applications.

1.5 Key Contributions

The key contributions of this thesis will be as follows:

1. A comprehensive overview of testing techniques, patterns, and anti-patterns for serverless applications.
2. An analysis of the available testing tools specifically designed for serverless architectures, including their features, strengths, and limitations.
3. Identification and analysis of the challenges faced in testing serverless applications, along with recommendations for addressing these challenges.

4. Insights and recommendations for practitioners and researchers on effective testing practices in the serverless architecture.
5. Identification of potential areas for future research and development of testing methodologies, techniques, and tools for serverless applications.

By addressing these research questions and providing these key contributions, this thesis aims to advance the understanding and practice of testing in the rapidly evolving field of serverless computing.

1.6 Structures

The remainder of this thesis is structured as follows:

Chapter 2: Background: Serverless Architecture and Platforms

In this chapter, the background of serverless architecture and platforms is discussed in detail. It covers the concepts, characteristics, and benefits of serverless computing. The chapter also explores popular serverless platforms such as AWS Lambda, Google Cloud Functions, Microsoft Azure Functions and so on. Common use cases for serverless architecture are presented, highlighting the diverse applications of this technology.

Chapter 3: Testing Fundamentals

This chapter delves into the fundamentals of testing in software development. It emphasizes the importance of testing and discusses various testing levels, including unit testing, integration testing, system testing, and acceptance testing. The chapter also explores the challenges in testing serverless applications and examines different testing and debugging approaches mentioned in the existing literature.

Chapter 4: Methodology

This chapter presents the research methodology employed in this study. It explains the research approach and design, including the systematic literature review approach used to gather relevant research articles and publications. The chapter outlines the search strategy, selection criteria, data extraction process, and analysis methods utilized to synthesize the findings.

Chapter 5: Findings and Discussion

This chapter presents the findings obtained from the systematic literature review conducted in this research. It addresses the research questions and provides an analysis of the identified testing techniques, patterns, and anti-patterns in serverless applications. The chapter also discusses the available testing tools for serverless applications, compares their features, and explores the challenges encountered in testing serverless architectures. The evaluation and synthesis of best practices in testing serverless applications are also presented in this chapter.

Chapter 6: Conclusion

The final chapter concludes the study by summarizing the key findings and contributions. It emphasizes the implications and insights gained from the literature review and highlights the limitations of the study. The chapter also suggests future research directions to further enhance testing practices for serverless applications.

2. BACKGROUND: SERVERLESS ARCHITECTURE AND PLATFORMS

2.1 Serverless Computing

Serverless computing, commonly referred to as Function as a Service (FaaS) [1], is a cloud computing paradigm where stateless functions are deployed to a FaaS platform [2], enabling developers to build and deploy applications without managing or maintaining the underlying infrastructure [3], [4]. With conventional cloud computing models, developers must maintain servers, operating systems, and other infrastructure components, which may take time and effort. With serverless computing, however, developers are free to concentrate only on writing and deploying code [5].

While servers are still used in the process, the term "serverless" is a little misleading since server management is now the duty of the cloud provider [3]–[5]. Developers may concentrate on event-driven functionality rather than maintaining the infrastructure. These small functions perform a specific job and are performed when needed.

Compared to conventional cloud computing approaches, serverless computing offers several benefits. It eliminates the requirement for server administration, freeing developers to concentrate on code logic and functionality that address business issues [5]. The code developers write consists of distinct functions or services triggered by certain events. When a trigger event happens, the cloud provider automatically provisions the resources required to perform the function, execute the code, and then release the resources after the function is completed [5]. This concept of on-demand resource allocation allows developers to avoid paying for idle resources while lowering the cost of operating applications. Companies may reduce their IT budgets by purchasing the computing power they need [3], [4], [6]. Since functions may be scaled up or down automatically in response to demand, serverless computing is highly scalable and beneficial for applications requiring rapid scaling. Furthermore, serverless architectures increase application availability and resiliency since the cloud provider maintains the underlying infrastructure and transparently resolves faults.

In a nutshell, serverless computing is a game-changing concept that allows developers to build and deploy applications without managing the underlying infrastructure [7]. It is scalable, constantly accessible [5], and cost-effective, making it an excellent option for enterprises of all sizes.

2.2 Serverless Architecture

Serverless architecture is a software architectural pattern in which applications are divided into discrete functions which are deployed and executed on the cloud [5], [8]. These functions are triggered by certain events and execute on demand without the requirement for dedicated servers or infrastructure management. In other words, Serverless architecture refers to the design of an application that utilizes serverless computing as its primary computing model.

There is a slight distinction between "serverless architecture" and "serverless computing," which are frequently used interchangeably. Serverless architecture refers to the software architectural pattern that encompasses not only the cloud computing platform but also the design of the individual functions and their interconnections. In contrast, serverless computing refers to the unique cloud computing model that permits the execution of these separate functions.

Serverless architecture is a powerful way to build applications because it allows developers to make scalable, cost-effective applications that can adapt quickly to fluctuating demands [3], [4]. By designing an application as a collection of serverless functions, developers take advantage of automatic resource allocation and event-driven architecture provided by serverless computing.

2.3 Serverless Platforms

In recent years, serverless computing has become increasingly popular among developers. This paradigm enables developers to write event-driven functions without worrying about managing the underlying infrastructure [8], [9]. Instead, developers may focus on designing and implementing application logic while the cloud provider handles resource management.

Several serverless platforms are available in the market, including AWS Lambda, Google Cloud Functions, Azure Functions, and other providers [10]. This section will discuss some popular serverless platforms and their features.

2.3.1 AWS Lambda

AWS Lambda is a powerful and flexible serverless computing platform launched by Amazon in 2014 [11] that allows for the execution of functions in response to events or explicit API calls [12]. Key dimensions like cost, programming model, deployment, resource limits, security, and monitoring were defined by AWS [3]. AWS Lambda is an

event-driven [5] and highly scalable serverless computing platform that supports container-based execution. It allows for the execution of functions in response to events such as uploading a file to Amazon S3 or HTTP calls made to a predefined endpoint created with the AWS API Gateway service [1].

The platform's asynchronous invocation method [1] enables it to scale up almost simultaneously hundreds of functions, making it suitable for applications that have to deal with a high amount of short-lived tasks. Additionally, AWS Lambda's support for container-based execution allows users to run virtually any kind of application without having to introduce changes, providing flexibility in application development. The combination of AWS Lambda's fast invocation and execution time with the pay-per-use model offered by the platform makes it an attractive option for deploying applications that need to deal with a high amount of short-lived tasks, reducing costs and improving efficiency.

Despite its benefits, AWS Lambda does impose several limits on function execution, memory usage, local storage, and deployment package size. For instance, AWS Lambda has a time limit of 900 seconds for function execution, a memory limit of 128 MB to 3008 MB, and a local storage limit of 512 MB. The platform also has a deployment package limit of 250 MB, which restricts the size of the function's code and dependencies that can be deployed [13]. However, despite these limitations, AWS Lambda has tight integration with AWS S3 and allows for credential delegation, simplifying the deployment process and enabling cloud functions to hold delegated credentials required to access storage [13].

AWS Lambda also offers automatic resource management [1][6], which optimizes the use of computing resources and reduces costs. Pricing is based on a pay-per-usage model [1], where the cost of each function invocation depends on its memory configuration and execution time. Optimal memory allocation is important to balance the trade-off between cost and function execution time. In addition, AWS Lambda is widely used in developing and maintaining smart home applications for appliances such as smart TVs, smart ovens, smart refrigerators, smart air conditioners, and smart washing machines, demonstrating its flexibility and applicability to a range of use cases [6]. AWS Lambda can be integrated with other Amazon Web Services, such as API Gateway, S3, DynamoDB, and SNS, enabling the development of complex applications [5].

To help developers monitor the performance and behavior of their Lambda applications, AWS provides two performance monitoring services, CloudWatch and X-Ray [5]. CloudWatch collects data on AWS service and resource use and allows applications to write their own performance records. However, accessing CloudWatch via API is limited and can result in long delays between event execution and log availability. CloudWatch logging is available in all AWS regions but is local to a region and may not be distinct for

concurrent function invocations [11]. Developers must write complex applications to extract insights into Lambda performance and behavior, which can be time-consuming and error-prone. X-ray, on the other hand, links function activities and presents performance and dependency data to developers as logs and service graph summaries for each application, although it has its own limitations [5], [11].

AWS Lambda is also subject to several operational limits that restrict the amount of memory and CPU that can be allocated to a function, as well as the maximum duration of a function's execution. These limitations make AWS Lambda suitable for applications that require a small amount of storage and memory [12]. However, AWS Lambda is still a powerful and versatile platform that offers efficient resource management and pricing, making it a popular choice for a variety of applications [7], [14].

2.3.2 Google Cloud Function

Google Cloud Function is a serverless computing platform that was introduced in July 2018 and is now accessible in seven of Google's twenty regions.

Similar to AWS Lambda, Google Cloud Functions automatically scale in response to requests. It also supports event-driven architectures allowing developers to set up triggers to execute functions in response to events such as changes in data stored in a Google Cloud Storage bucket.

Google Cloud Functions offers a variety of CPU and memory combinations and supports three programming languages, namely Node.js, Python, and Go [15]. It allows an unlimited number of allocated instances per function but a maximum of 1,000 concurrently executing functions. The platform is user-friendly, with a straightforward web portal and a CLI utility that supports single-command operations. However, its monitoring measurements lag behind actual execution by several minutes, delaying interventions. In addition, it only supports a limited number of runtime systems, and deployers have limited configuration options, limiting clients' ability to perform fine-tuned operations. The initial start latency is greater than that of AWS and IBM but is still usable. Google Cloud Functions is the least efficient option for cost purposes [15].

2.3.3 Microsoft Azure Function

Microsoft Azure's Azure Functions is a serverless computing platform that was released publicly in November 2016. Azure Functions supports five runtime systems and seven different programming languages such as C#, F#, Node.js, Java, PowerShell [15].

Azure Functions can dynamically scale in response to incoming requests, similar to Amazon Lambda and Google Cloud Functions. It also supports event-driven architectures, so developers can configure triggers to run functions in response to events such as changes to data stored in an Azure Storage account.

Azure Functions has the advantage of having no limit on the maximum number of concurrent runs. Additionally, the platform provides automatic or manual scaling depending on the chosen plan, which is a significant benefit for developers who require flexible scaling options [16].

Microsoft Azure Functions is a serverless computing platform that offers similar functionality as OpenWhisk. The platform provides an Azure Function App that enables developers to upload their function code bodies and interact with Azure services. The supported runtimes are C# and JavaScript, with experimental options for Python and other languages. Users can choose between Windows NT or Linux (preview) instances to host their function(s) when creating a Function App, and the source uses Windows NT for this work. Azure Functions can be triggered by updates to Azure Blob Storage, Cosmos DB, HTTP requests, and Queue Storage, among other services [17].

The platform offers three different hosting plans, including a consumption plan that adapts to the load and popularity of the deployed function, a premium plan with finer-grain control over the computing instance size and pre-warming support, and an app service plan customized for a given application's needs. Azure Functions can utilize up to 200 instances and 1.5 GB memory, and the platform can run on Windows or Linux hosts, with 28 out of 46 publicly accessible regions. The virtual machines of type Av2.1 execute Azure Functions, utilizing three different CPUs: Intel Xeon 8171M at 2.1 GHz, Intel Xeon E5-2673 v4 at 2.3 GHz, and Intel Xeon E5-2673 v3 at 2.4 GHz [15].

2.3.4 OpenWhisk

OpenWhisk is a serverless computing platform enabling developers to construct and deploy event-driven applications rapidly. IBM introduced it, and it is currently an open-source Apache Software Foundation initiative. It is available on GitHub under an Apache open-source license [18]. OpenWhisk supports many programming languages and connects with numerous cloud platforms, making it a versatile and potent tool for developing serverless applications. OpenWhisk's support for several programming languages is one of its primary advantages. Popular programming languages such as Java, Python, Node.js, and Swift, as well as arbitrary binaries embedded in a Docker container used by developers to create serverless applications [3]. In OpenWhisk, a trigger refers to an

event that initiates an action, such as a Github pull request trigger or a Slack new message trigger. The action is a function that receives JSON data as input and returns JSON data as output. A rule establishes the connection between a trigger and an action [19]. Users can connect multiple actions to build a sequence, where the sequence's children, which can be actions or sequences, are run consecutively with data transferred between them.

When an event, rule, action, or sequence executes on the OpenWhisk platform, a JSON log referred to as activation is created to record the execution details, including activation ID, start time, duration, success or failure status, developer log array, and output JSON data. By capturing the ID of the cause activation, developers can get the input data of activation from the cause activation's output data.

OpenWhisk offers an Action Sequence feature that enables users to chain multiple actions together in a single sequence. However, the platform currently only supports a single chain, which means that more complex composition patterns, such as branching and parallelization, are not currently possible. OpenWhisk's use of complex components, such as Kafka for message passing and CouchDB for function lifetime management, also introduces additional overheads. Furthermore, OpenWhisk expects application developers to pack their dependencies into the function package and does not handle dependencies on the platform side. Consequently, serverless application developers should exercise caution with regard to the size and dependencies of their functions to avoid potential issues with performance and scalability [20].

OpenWhisk's API enables programmers to dynamically invoke actions from code, allowing for the execution of different actions based on a variable value during runtime, in addition to building static sequences. However, it should be noted that activations that are dynamically invoked do not currently record their cause activation ID. This limitation impedes the ability of tools to establish causal relationships between dynamically invoked actions, potentially hindering debugging, and development efforts. As an open-source project, OpenWhisk has acknowledged this issue, and the community is actively working to address it [18].

2.3.5 OpenLambda

OpenLambda is a serverless computing platform that offers an open source under the Apache 2.0 license, a lightweight and flexible alternative to other serverless platforms.

OpenLambda uses Linux containers for isolation and orchestration [21]. A key feature of OpenLambda is its flexibility, which enables a high degree of control over lambda placement, administrative access, and IP addresses for hosting virtual machines (VMs).

This degree of control is not possible on serverless commercial platforms, such as AWS Lambda. By utilizing OpenLambda, users can attain the required level of control over the platform, making it an ideal choice for projects requiring precise control over the deployment environment. In addition, as an open-source platform, it allows for the customization and extension of its functionality to satisfy specific needs, making it a versatile and adaptable platform [10].

OpenLambda can interact with popular serverless frameworks, including Google Cloud Functions and Amazon Lambda. OpenLambda offers a fault-tolerant and scalable architecture that can manage a high volume of requests and dynamically scale resources in response to demand. Moreover, it facilitates the execution of multiple functions in a single process, which may enhance the platform's performance and efficiency.

2.3.6 OpenFaaS

OpenFaaS is an open-sourced, serverless computing framework that facilitates the scalability of CPU-bound computations implemented as functions. It offers auto-scaling by default based on the number of requests per second and supports both asynchronous invocations via message brokers and synchronous invocation via basic HTTP requests. OpenFaaS has a large community, supports a variety of programming languages, and provides numerous predefined and practical templates. The platform's functions are written in Python and are activated based on the capacity of each Redis queue. OpenFaaS's auto-scaling capabilities, by default, ensure that only the required resources are utilized by calculating the number of requests per second [22].

2.3.7 Fn

Fn is a serverless computing platform that provides software developers with the ability to write, deploy, and execute event-driven functions in the cloud. It is an open-source project and container-native serverless. Fn is one of the emerging serverless platforms that has gained attention in the industry due to its flexibility and ease of use. Fn provides data binding; grouping functions into apps, and variable configuration at three levels (application, function, and route) that make it easy to develop and deploy serverless functions. Fn provides support for multiple programming languages, event sources, and integrations with other cloud services. Overall, Fn is a powerful and flexible serverless computing platform that enables developers the freedom and scalability to develop complex applications [23].

2.3.8 Other Serverless Platforms

In recent years, numerous serverless platforms have emerged to cater to various use cases, transforming the serverless computing ecosystem. Some noteworthy Function-as-a-Service (FaaS) platforms include Compute Function by Alibaba Cloud, Cloudflare Workers by Cloudflare, Nuclio, Knative by Google, Qinling by OpenStack, Fission by Platform9, and Kubeless by Bitnami. These platforms offer several benefits, such as auto-scaling, reduced operating costs, and faster time-to-market, allowing developers to focus on building and deploying code efficiently. Despite the widespread adoption of serverless computing, limited academic research is available on these platforms. Therefore, this study will not discuss them in depth.

2.4 Common Use Cases for Serverless Architecture

In recent years, serverless architecture has gained popularity due to its adaptability, cost-effectiveness, and scalability [10]. There are a number of widespread use cases for serverless architecture, such as:

2.4.1 Event-Driven Applications

Serverless architecture is ideally suited for event-driven applications that demand real-time processing of incoming data streams [1]. IoT devices and sensors, for instance, create large amounts of data that must be processed and evaluated in real-time, making serverless the optimal solution [8].

2.4.2 Microservices

Serverless enables the creation of small, independent, and scalable services that may be independently developed, deployed, and maintained [24]. By adopting serverless functions for microservices, developers may create more efficient and scalable applications with improved fault tolerance and easier maintenance [24], [25].

2.4.3 APIs

Serverless platforms such as Amazon Lambda and Azure Functions enable developers to build and manage APIs for their applications without the need for dedicated servers. This allows them to design and scale APIs while reducing operational overhead efficiently. This technique enables more efficient management of APIs, enabling developers to focus on business logic [11].

2.4.4 Data Processing

Serverless is ideally suited for large-scale data processing, such as analyzing, filtering, or aggregating data from several sources [5]. This is especially advantageous for big data and analytics applications since serverless architecture can provide the necessary processing capacity on demand, hence maximizing resource use and lowering costs [1].

2.4.5 Scheduled Tasks

Serverless functions can be used to do scheduled operations [5], such as sending emails, creating reports, and performing routine maintenance. This enables developers to automate these activities without the requirement for specialized infrastructure, enhancing productivity and cost-effectiveness.

2.5 Benefits & Challenges of Serverless Architecture

In software development, serverless architecture has become an increasingly popular solution. By abstracting server management, developers are able to focus on developing code while cloud providers maintain the underlying infrastructure. In the scientific literature, the topics that received the most attention were cost, cold starts, vendor lock-in, application state, technology constraints, event-based programming model, monitoring, and observability. This section will discuss the advantages and disadvantages of serverless architecture.

2.5.1 Cost

Cost is a significant factor in a serverless architecture. The adoption of serverless computing platforms in next-generation cloud software has enabled the decomposition of software into independent components bundled and run using isolated containers or microVMs, resulting in scalable, manageable, and extensible systems with loose coupling. This approach enables application hosting using fine-grained cloud infrastructure and promises reduced hosting costs while achieving high availability, fault tolerance, and dynamic elasticity. However, these advantages are offset by pricing obfuscation, performance variance, multitenancy, and provisioning variation, which obscure the true cost of hosting applications with serverless platforms and make predicting hosting costs for serverless applications far more complex than traditional VM-based application deployments [26]. There is a formula for calculating the execution bill for a serverless function [20]:

C * Time * Resource

where:

- C is a platform-specific constant
- Time is the function execution time with a millisecond-level granularity.
- Resource typically represents the memory and CPU resources provisioned to the function.

However, this cannot always calculate the actual cost.

Serverless computing is a "pay-as-you-go" paradigm; therefore, optimizing code and service configurations reduces expenses. Without execution time and memory use, determining a function's cost is difficult. Inefficient code and configurations can result in higher costs, particularly if they maintain certain connections active all the time. It is important to optimize code and configurations to reduce costs. Therefore, when utilizing Function-as-a-Service, it is important to concentrate on the execution time of requests in order to minimize costs (FaaS). According to research conducted by Lin et al. (2018), external dependencies, such as databases and authentication services, can have a measurable impact on the cost, so external services must be carefully considered [27].

Due to multi-dimensional billing models and heterogeneous CPUs, hosting applications on Function-as-a-Service (FaaS) platforms is hard to compute. The lack of tool support to estimate hosting costs is a major limitation of FaaS platforms. Current cloud pricing calculators mainly provide cost estimates based on average performance for IaaS computing and storage performance, while FaaS calculators are limited to generating cost estimates based on average runtime and memory size. These calculators do not consider how FaaS function runtime scales relative to the memory reservation size, which is a feature coupled to CPU power on several FaaS platforms [26]. Many methods, such as cost forecasting and cost management via workload optimization, have been proposed by researchers as potential solutions to this problem [28].

A tool was developed by Cordingly et al. (2020) to predict how CPU metrics scale across different function deployments with varying CPUs, memory settings, and cloud providers [26].

2.5.2 Cold Start

In serverless computing, a cold start occurs when a function takes longer to execute than usual, typically after deployment or after a period of inactivity [16], [28]. Several steps contribute to the latency as well as the cold start effect, including allocating resources, configuring the runtime environment: code, packages, and extensions, loading the function in memory, and executing it [1], [10], [17], [20], [22].

In existing serverless platforms, a function instance can handle a request with either a cold start or a warm start, depending on whether there are available idle sandboxes. Cold start occurs when a function instance is executed for the first time and requires preparation, such as loading function codes and creating sandboxes, which can cause long latency. Warm start (Warm start is the reuse of an initialized container instance to handle subsequent requests for the same function, reducing the impact of cold start latency) occurs when the sandbox is paused for a specified period after execution to serve subsequent requests for the same function more quickly. Slow cold start not only affects startup latency but can also poison the execution performance in the nested function chain and lead to a drastic rise in serverless cost. However, execution time and overhead variation in one function do not affect the execution time of another in a sequence chain, which avoids the double-billing problem [20].

When a function is warm, it is ready in memory and can immediately serve requests. Cold starts are measurable and can have a degrading effect on latency. Cold start latency varies across different cloud providers and programming languages.

Cold start is a notable issue in serverless platforms, such as AWS Lambda, where the initial invocation of a function experiences high response times due to the time taken to prepare the environment and deploy the function [29]. A study was conducted to measure the cold start latency of major serverless platforms, wherein a function experiences increased execution time after being idle or following initial deployment [1], including AWS, Azure, Google, and IBM. The study used 512 MB of memory and avoided external packages to minimize loading time. Results showed that AWS had the lowest average cold start latency (335 ms) for Node.js, Python, and Go, with higher latency for .NET. Azure had cold start latencies ranging from 2 to 5 seconds, with .NET on Windows slightly faster, while IBM performed similarly to AWS. Google Cloud had higher cold start latencies ranging between 2-3 seconds. The recycling and re-insertion time of a computing instance into the available pool varied from 10 minutes to 10 hours depending on the platform, with AWS and IBM being around 10 minutes, Azure taking up to 20 minutes, and Google varying from 10 minutes to 10 hours [15]. Another study [16] has shown that languages like .Net and Java have longer cold-start times, while Python and Node.js have the shortest. McGrath et al. (2017) [30] and Wang et al. (2018) [31] have also conducted measurements on major serverless platforms. The results show that a cold start generally occurs five minutes after the last execution and can cause a significant impact on responsiveness, with some platforms having cold start latency more than ten times that of a warm start. This problem is crucial for serverless application development, where function interaction latencies are important. Wang's measurements [31] show that

the median warm start latency in major serverless platforms like AWS, Google, and Azure ranges from 25 to 320 ms, while the median cold start latency is much higher, ranging from 110.77 to 3640.02 ms, confirming that cold start can significantly affect the time it takes to handle a request compared to a warm start [32].

Cold start results in significantly higher minimum response times compared to microservices. The overhead of cold start can range from 300ms up to seconds for a single function, causing a direct impact on performance and cost [28]. However, once the environment is set up, the response times stabilize, and the duration of high request response time is usually shorter than in microservices. Numerous research efforts have aimed to reduce the cold start time in serverless platforms by using methods such as pre-warmed containers [29], periodic warming with dummy requests [29], and pausing containers [33]. DevOps professionals must consider these factors when deploying applications and determine if a serverless deployment strategy is suitable for their specific use case [29], [32].

In serverless computing, each service corresponds to a stateless function [34] executed in a dedicated container with limited resources. Unlike traditional Infrastructure-as-a-Service (IaaS), containers in serverless computing are launched only when the function is invoked and sleep immediately after processing the request [31]. However, if the function is not invoked for a while, the container will be deleted, and a cold start occurs when the function is invoked again, leading to prolonged latency that hinders the adoption of serverless computing [30]–[32]. Traditional strategies for reducing cold start latency come at the expense of resources, making it a challenging problem to simultaneously minimize cold start latency while reducing resource consumption in strategy implementation [32]. Cold start can negatively impact the performance of the application. Additionally, challenges in serverless architectures include the absence of point-to-point communication and the limited knowledge providers have regarding the content of lambda input data [10].

Cold start is a common phenomenon that occurs when a virtual machine or container is started up from a "cold" state, which means that it does not have any pre-existing resources allocated to it [35], [36]. To mitigate the issue of cold start in serverless computing, several researchers have proposed various approaches and techniques aimed at improving the performance and efficiency of serverless applications. For example, by analyzing metrics such as incoming request counts and execution time, it is possible to identify the impact of cold start latency on the overall performance of VMs and containers and then can be used to optimize resource allocation and reduce cold start overhead [35], [36].

Another solution is proposed by McGrath et al. (2017) [30] to optimize resource allocation and minimize the impact of a cold start on the system's overall performance by utilizing the cold queue and warm queue.

When a function is invoked, the system searches for an available container to execute the function. If no container is available, the system places the request in the "cold queue," which is used for auto-scaling. The cold queue reflects the available space across the platform, which helps optimize resource allocation and reduce cold start latency. By employing the cold queue and warm queue, the system can efficiently allocate resources and minimize the impact of a cold start on the system's overall performance [30].

Bardsley et al. (2018) [37] proposed a warming strategy that optimizes performance in AWS Lambda. Preheating the functions and enhancing their responsiveness for later invocations is another solution [1]. McGrath et al. (2017) [30] proposed a .Net-oriented framework on Microsoft Azure to increase the performance of FaaS platforms by evaluating function expiration times and cold start duration. Jackson and Clynch proposed [38] a testing framework to analyze the cost and performance of commercial FaaS platforms, showing that .Net and Python have the highest performance with low cold start times on Azure Functions and AWS Lambda.

OpenFaaS utilizes container reuse and default autoscaling to keep functions warm and avoid the cold start problem. Container reuse ensures that running containers are reused for subsequent invocations of the same function, while autoscaling considers the number of requests per second to ensure optimal resource allocation [22].

The serverless framework Nimbus [39] provides a solution to the cold start issue by allowing for changes to be made to individual functions without requiring the redeployment of all functions. This approach reduces deployment time and resources by only building and deploying the necessary JAR. This strategy can be especially useful for large projects with multiple functions with different dependencies, such as interactions with databases. By avoiding the need to redeploy unchanged functions, Nimbus minimizes the occurrence of cold starts and simplifies the development process [39].

Efforts have been made to optimize the cold start problem in serverless computing through different strategies, such as the Container Pool (CP) and Warm-Up (WU) strategies [30]–[32], but these strategies reduce the cold start latency at the expense of resources. WU involves regularly restarting function containers to warm them up and reduce cold start latency, but it results in resource waste. CP strategy utilizes a prelaunched container pool to save launching time and reduce cold start latency but also results in resource waste [32]. Other techniques, such as isolation or optimizing the func-

tion runtime environment package, have been explored to reduce cold start without additional resource consumption [40], [41]. However, the optimization of cold start remains a critical issue as it requires a balance between reducing latency and minimizing resource waste [32]. Xu et al. (2019) propose predicting function invoking time and dynamically adjusting the container pool's capacity to minimize cold start latency while consuming fewer resources. Additionally, the author highlights the need to reduce cold start latency in serverless computing to broaden its adoption and use in various applications [32].

2.5.3 Vendor Lock-in

Vendor lock-in is a critical challenge associated with serverless computing, affecting the long-term sustainability and flexibility of serverless applications. The tight coupling of serverless applications with other services in the cloud provider’s ecosystem makes considerable vendor lock-in unavoidable. Vendor lock-in can occur in serverless environments due to their reliance on additional services, such as databases, logging mechanisms, and API mappers, which are typically provided by the same platform. This dependency can limit the flexibility of the code and restrict the ability to migrate to alternative platforms [5]. Developers need to carefully assess the vendor lock-in that is prevalent in virtually all existing offerings as serverless applications are tightly coupled with other services in the cloud provider's ecosystem. AWS API Gateway, S3, Kinesis, SNS, DynamoDB, and StepFunctions are cloud provider-specific serverless solutions [42]. Switching providers would require a significant effort to re-implement the application with new services, which might lock developers who use these services in serverless applications onto a single provider. Similarly, other cloud providers offer platform-specific comparable services, which can contribute to platform lock-in when using their serverless offerings [39]. When selecting the concrete FaaS provider to use, developers should consider possibilities and required efforts for integrating with other cloud services, which is a common pattern for FaaS usage. If they already rely on many services from one provider, integrating FaaS may be easier, and familiar tools may be re-used. Developers must evaluate the risk of vendor lock-in against the advantages of utilizing a particular provider.

Serverless applications often lead to systems comprised of many separate functions, making it difficult to run the system without deploying it to the cloud. The cloud infrastructure becomes an integral part of the application rather than just a hosting platform. The tight integration of serverless applications with cloud providers' ecosystems poses a significant challenge for DevOps teams, as it results in vendor lock-in [42]. Moreover, the

implementation of the DevOps pipeline for serverless applications strongly affects the choice of tools, such as IaC solutions and mocking libraries [5]. The close connection of Serverless applications with cloud infrastructures usually requires deploying the application to the cloud platform and heavy use of the cloud platform services [5]. The deployment process for a Java REST API on AWS Lambda in an industrial project was found to be time-consuming, with multiple deployments being necessary due to configuration errors. Errors ranged from typos to incorrect permissions, and a limit on maximum file size caused the codebase to be restructured into separate modules. The difficulties associated with serverless development have the potential to increase the possibility of vendor lock-in, particularly in cases where deployment and infrastructure are tightly connected [39].

Research should be conducted to explore ways to mitigate the risks of vendor lock-in that are prevalent in Serverless architectures. As Serverless becomes more widely adopted, companies may find themselves locked in with a particular cloud provider due to the dependence on their proprietary tools and services. This phenomenon may lead to increased expenses and reduced adaptability in selecting the most suitable service provider to fulfil their requirements [5]. Consequently, it would be advantageous to explore strategies for enhancing interoperability among diverse Serverless providers, and devising methodologies and instruments that facilitate the seamless migration of Serverless applications across providers would be advantageous.

Overall, understanding the implications of vendor lock-in and developing strategies to mitigate its risks will be crucial for the continued growth and success of Serverless architectures.

2.5.4 Technology Limitations

Serverless computing also raises issues with technology limitations. Programming languages, software libraries, and development tools may include restrictions on developers. Although cloud service providers have increased the languages and libraries they offer, academics have suggested a number of strategies to get around these limitations, including the use of multi-cloud architectures [39].

2.5.5 Event-Based Programming Model

The event-based programming paradigm is a distinctive feature of serverless architecture. This architecture enables developers to create applications that respond to

events, such as database updates or the arrival of a new message in a queue. Complex event-driven systems may be tough to develop and test, despite the model's many advantages. While there are many advantages to using this architecture, it may be difficult to build and test event-driven applications [25].

2.5.6 Monitoring & Observability

Monitoring and observability are fundamental aspects of serverless architecture. It might be difficult for developers to monitor and debug their programs in real time in a distributed, event-driven environment. Researchers have suggested a variety of solutions to this problem, including distributed tracing and log aggregation [28].

2.5.7 Performance-Related Issues

Serverless computing, while providing benefits such as scalability and reduced infrastructure costs, is subject to several challenges that can impact both its performance and cost-effectiveness, including issues with cold start times, function call duration limitations, CPU selection, and state management. These challenges include limited runtime resources, the need for optimized code, concurrent request limits, and the availability of memory and CPU for function invocation. Other key requirements for performance are optimized storage, low latency, and high throughput. Furthermore, increasing the number of resources allocated for caching in a Function-as-a-Service (FaaS) environment enhances performance, but increasing the amount of resources allocated for caching at the application level may not lead to a linear gain in performance. Function invocation, as well as intra-communication, can also result in infrastructure overhead. Scalability issues can arise when microservice applications simultaneously upload and download data, and slow network data transfer can cause latency flexibility issues. Slow network data transport may result in latency problems.

The performance of serverless computing can differ depending on the cloud service provider; portability is essential for evaluating performance. Four distinct phases of serverless infrastructure - provider, virtual machine (VM), cold container, and warm container - demonstrate that microservice output can vary by a factor of up to 15 based on these conditions. Memory, execution time, bandwidth, and CPU usage are enforceable resource limits for serverless functions. Proactive resource provisioning is necessary for scalability, and provisioning and prediction are essential for optimal solutions. Better configuration can improve performance, and an application needs enough resources for both peak and spike situations.

Cold start in FaaS affects function performance when an application is used occasionally, and memory and resource allocation have a direct impact on function performance; however, resource allocation may not always improve application performance. The warm queue is a problematic FIFO queue, and the relationship between code quality and cost in pay-per-execution time models is direct [4]. Lastly, performance analysis is required for estimating data indices under diverse execution conditions [28].

2.5.8 Technical Debt

Serverless computing can be affected by various types of technical debt (TD), including architectural debt, code debt, and testing debt. The continuous change in serverless-based applications increases the risk of quickly degrading the architecture, while the use of multipurpose functions and boilerplate code can significantly increase the likelihood of code debt. Testing multiple serverless functions can also become very complex very quickly, causing testing activity to be postponed, leading to high risk with respect to quality and reliability. As compared to microservice-based systems, serverless functions face a higher risk of accumulating higher TD due to the serverless programming model's experimentation flexibility, leading to misunderstandings among teams that work in a single serverless function and create distributed monolith systems composed of too many serverless functions. Further work is needed to manage the risks associated with serverless functions to mitigate the technical debt they may accrue [25].

The adoption of serverless functions can lead to higher levels of TD due to several conceptual antipatterns, particularly related to test debt. Unit testing in serverless systems is difficult as identifying atomic business logic units for isolation testing is a challenge [20], [43], leaving business-critical behaviors uncovered. Integration testing is also incomplete due to a lack of automated tools. Continuous Integration/Continuous Delivery (CI/CD) and Infrastructure as Code (IaC) practices are limited by low testing coverage and incomplete IaC setups. Inadequate debugging and monitoring mechanisms further limit the observability and testability of systems built on serverless platforms, with contemporary logging, tracing, security practices, and technologies not yet fully adapted to the serverless context. Improving serviceability while maintaining system maintainability and testability is a challenge in serverless computing that requires innovative solutions and practices [25].

In conclusion, serverless architecture provides several advantages but also brings significant challenges. Organizations that are thinking about using serverless solutions should carefully weigh the pros and cons. The pros include cost savings, scalability,

faster development and deployment, and flexibility. The cons include lock-in to a single vendor, cold starts, security concerns, and limited customization. Organizations may leverage the potential of serverless architecture to develop new, efficient, and scalable applications in the rapidly changing world of cloud computing by understanding the trade-offs and addressing the challenges ahead of time.

2.6 Serverless Deployment Strategy

The deployment of a serverless application differs from traditional methods of deploying applications that require managing physical servers and containers.

In serverless architecture, developers do not have to configure physical servers, or containers, or stress about scalability, as the serverless platform handles all of these tasks. Instead, developers provide the application's source code alongside a deployment specification that explains the required functions, APIs, permissions, configurations, and events for the serverless application.

The serverless framework builds and deploys the application on the serverless platform using the deployment specification. Cloud service providers such as AWS offer cloud-based IDEs and plugins that make serverless application development and deployment simple. AWS also offers the Serverless Application Model (SAM), an open-source framework that facilitates the creation of serverless applications on AWS [29].

In addition, developers can test and debug their serverless applications locally by utilizing the AWS-provided local environment. Developers can also use function execution logs and request tracing tools, such as AWS X-Ray, to identify and debug potential serverless application issues. In general, the serverless architecture provides a more streamlined and effective method for deploying applications [29].

2.7 Serverless Application Pattern

A scientific investigation conducted by Leitner et al. (2019) presents important findings about serverless application patterns. The study identifies five recurring patterns to address the challenges and limitations of serverless computing [42]. The following subsections provide a comprehensive understanding of the application patterns.

2.7.1 Externalized State

This pattern addresses the problem of state management in serverless functions by allowing developers to externalize function states in a key/value data storage like Redis.

The advantages of this pattern include reliable persistence of function state between calls, but the disadvantages include latency overhead and additional programming effort.

2.7.2 Routing Function

This pattern addresses the problem of API gateway and route configuration by using a central routing function to simplify the configuration process. The advantages of this pattern include simplified API configuration and reduced workload for developers, but the disadvantages include hidden routing information and potentially increased complexity in the function implementation.

2.7.3 Function Chain

This pattern addresses the problem of function call duration limitations imposed by serverless platforms by splitting a function into multiple parts that can be chained together to prolong the allowed call duration. The advantages of this pattern include the ability to circumvent platform timeouts, but the disadvantages include creating two deployment units for one logical service and introducing strong coupling between the chained functions.

2.7.4 Function Pinging

This pattern addresses the problem of container cold start times and high latency for some requests by periodically triggering a function to keep containers warm and avoid timeouts. The advantages of this pattern include avoiding timeouts, but the disadvantages include periodic pings inducing unnecessary costs and the need for additional code to manage to ping.

2.7.5 Oversized Function

This pattern addresses the problem of CPU selection in serverless platforms by increasing the memory requirements for a function to deploy it to a stronger physical machine with a faster CPU. The advantages of this pattern include improved performance, but the disadvantages include being billed significantly more for the higher memory allowance without using it.

These patterns demonstrate the challenges and limitations of serverless computing and provide solutions to mitigate these problems. Before implementing a pattern in an application, developers should thoroughly evaluate its benefits and drawbacks.

3. TESTING FUNDAMENTALS

Testing is an integral part of software development, as it ensures the reliability, correctness, and robustness of software applications. This chapter will explore the significance of testing in software development, followed by an in-depth analysis of the various testing stages, including unit, integration, system, and acceptance testing in serverless applications. It also discusses the challenges with serverless testing and different testing & debugging approaches found in the literature. Additionally, the chapter provides insights into determining the adequacy of testing, coverage criteria, and available testing tools.

3.1 The Importance of Testing in Software Development

Testing is essential to software development because it ensures the delivery of high-quality, maintainable, and reliable software products [24]. Testing may significantly decrease the number of faults, enhance system quality, and lower the risk of failure during production, according to the findings of several scientific studies [44]. Moreover, good testing may lead to lower maintenance costs, higher user satisfaction, and improved software development efficiency [45], [46].

Software testing has several significant advantages. Testing helps in detecting and resolving issues and errors, ensuring that the application performs as planned. Secondly, it confirms that the program satisfies the defined criteria and fulfils its intended function [44]. Rigorous testing increases software dependability by verifying that the program functions as intended under varied scenarios. Overall, proper testing may reveal usability issues and enhance user experience [44].

In conclusion, testing guarantees project success by delivering high-quality, maintainable, and reliable software applications through defect identification and resolution, software verification and validation, increased dependability, and better user experience.

3.2 Testing Levels

Software testing is typically conducted at various levels, each focusing on different aspects of the system. This section will discuss four common testing levels: unit testing, integration testing, system testing, and acceptance testing.

3.2.1 Unit Testing

Unit testing is an essential part of software development that entails evaluating individual components or units of code in isolation to verify that they function properly. These components are often small, such as functions, classes, and methods. These tests use mock objects to separate the unit under test from any external dependencies. Mock objects are simulated objects that mimic the behavior of other software components or services [47]. Practitioners may mock any dependencies, but cloud services and domain objects (application modules and components) are often mocked. Unit testing is white-box testing, which requires testers to be familiar with the application's source code to implement the tests efficiently.

The primary objective of unit testing is to investigate all possible executions of the unit, including valid and invalid function calls, and to guarantee that the unit can handle different scenarios to evaluate the quality of the code and discover any errors or inconsistencies in the logic.

Developers can find errors and correct them before integrating them with the rest of the software by testing each unit separately. This helps prevent issues from growing and leading to worse problems in the future. Furthermore, it helps developers debug and maintain code by allowing them to isolate and detect issues easily.

Unit tests are at the very bottom of the V-model and serve as the foundation for more advanced testing. These are the fastest to run since they test fewer complex components of the software in isolation.

3.2.2 Integration Testing

Integration testing tests the interaction between multiple components or units to ensure they function correctly. Integration testing aims to identify issues in the interfaces and interactions between distinct units, such as data exchange or function calls. Integration testing is typically conducted after unit testing and involves testing larger application pieces [46].

Integration testing typically follows unit testing and includes testing larger program components. Depending on the objective of the test, external integrations may or might not be mocked. Both white-box and black-box testing techniques can be employed to perform integration tests [45]. An integration test for an HTTP API application may, for instance, call an endpoint with inputs to verify that the returned response matches the expected output values.

Big-bang and incremental testing are the primary approaches used in integration testing. In the big-bang method, all functionality is implemented before testing, and all units

are tested simultaneously. In contrast, the incremental method, which includes both the top-down and bottom-up techniques, only needs some modules (or units) to be developed for testing, and any missing functionality is mocked.

The bottom-up technique tests the lowest components first and integrates them level-by-level, mocking higher-level implementations using driver implementations. The top-down strategy tests lower-level modules from the highest level, mocking any missing unit implementations. The top-down approach prioritizes high-level planning and decision-making, whereas the bottom-up approach prioritizes task execution and detailed knowledge.

Incorporating tests while development is still in progress is a key benefit of incremental testing. This approach helps developers identify and address issues before they become more complicated and expensive to solve. Overall, integration testing is essential for guaranteeing that the software program and its many components perform as intended, resulting in a high-quality product that satisfies the user's expectations.

3.2.3 System Testing

System-level testing is a comprehensive and crucial approach to software development that evaluates the complete system behavior under various conditions, including all of its components and applications. System testing is performed after integration testing, where different modules of the system are combined and tested as a whole. This testing method ensures that the software satisfies the business and user needs by providing a complete picture of its performance. The system's functionality, performance, security, and usability are tested at this level.

End-to-end (E2E) testing is a typical system testing technique. It entails testing the complete system, including both front-end and back-end applications, by manually exploring the Interface and analyzing the results of each operation. With modern full-stack systems, for instance, tests are typically created using a black-box technique, in which the front-end application is tested by clicking buttons, filling out forms, and performing other activities to invoke the back-end application. This procedure verifies that the entire system functions as intended, including both front-end and back-end code [45].

System-level testing and functional testing cover non-functional testing methods such as load and performance testing. Load testing ensures that the system can manage a large number of users and data, whereas performance testing evaluates the software's response time and speed under different conditions.

Additionally, system testing can uncover issues such as performance bottlenecks, security vulnerabilities, and usability problems that might not have been detected during

unit or integration testing [46]. By identifying such issues, system testing helps ensure that the software meets the highest standards of quality, reliability, and user satisfaction.

In addition, system testing might reveal problems like performance bottlenecks, security vulnerabilities, and usability difficulties that may not have been discovered during unit or integration testing. By discovering such issues, system testing helps guarantee that the program meets the highest quality, dependability, and user satisfaction criteria.

3.2.4 Acceptance Testing

The highest and final phase of testing, known as acceptance testing, involves the application's users as well as other key stakeholders to determine whether it meets the end user's requirements and expectations. It includes multiple testing kinds, including operational, contract, regulatory, and user acceptance testing.

User acceptance testing, for instance, can be undertaken via alpha and beta releases to collect user feedback prior to the launch of the final product. Acceptance testing ensures that the software meets business and user criteria and is ready for deployment [45].

Acceptance testing is a crucial aspect of the software development life cycle, and teams should prioritize it to satisfy consumers' needs. Developers may improve the software by incorporating end-user and stakeholder feedback, creating a high-quality solution that fulfils business and user needs.

Testing is an essential component of software development that ensures the dependability, accuracy, and robustness of software systems. By testing at different levels, including unit, integration, system, and acceptance, developers can successfully find and address errors, verify compliance with requirements, and confirm that the software achieves its intended purpose.

3.3 Testing in Serverless

Serverless computing, also called Function as a Service (FaaS), is a way to build cloud-based applications that allow developers to build and deploy applications without having to manage the infrastructure under it. Serverless applications use many complexes, stateless, transient short-lived, and concurrent functions that need special development and testing tools [7], [20]. In serverless architectures, events trigger functions and scale in response to demand. While there are many benefits to serverless computing, there are also some unique challenges when it comes to testing serverless applications [4][48]. According to the study, testing is essential in serverless environments, and

developers must adapt testing levels to serverless architectures to assure software quality.

This section aims to explore the topic of testing in the context of serverless computing. It will discuss the significance of testing in serverless environments and explore how testing can be adapted to address the unique challenges posed by serverless architectures.

Testing serverless applications is crucial to ensure that functions execute correctly in response to specific events and adhere to performance, security, and reliability requirements. The testing process for serverless applications should follow the established levels of testing discussed in the previous section (unit, integration, system, and acceptance testing). However, specific considerations must be taken into account due to the unique characteristics of serverless environments, such as event-driven architectures, stateless functions, and auto-scaling [3].

3.4 Challenges in Testing Serverless Applications

Due to the relative immaturity of serverless technology and the dynamic and complex nature of serverless computing (highly modular and loosely coupled nature of serverless functions), testing serverless applications presents distinct challenges compared to testing traditional applications [24], [25].

In serverless computing, one major challenge is testing functions, particularly when it comes to integrating multiple functions or external services [7], [42]. Therefore, it is difficult to test the system as a whole is difficult because code is submitted as small, self-contained services that are deployed in containers triggered by an event [20]. Unit testing is relatively straightforward, but testing the integration of multiple functions or external services is more difficult, as local replication of the entire system is often not possible. In addition, serverless applications cannot be deployed locally, and developers cannot anticipate which container will be used during deployment. This makes system-level testing on local machines unfeasible [24].

It is difficult to measure code coverage in serverless applications due to a large number of interacting lambda functions (in AWS). While unit testing is still possible, it is difficult to assure that a lambda function has been tested for all possible events it may receive from other functions [24].

One solution is to test functions directly in production or have a second account with the provider dedicated to the development and testing environment, but both approaches have the disadvantage of requiring developers to pay for test invocations [24]. Addition-

ally, testing in actual production environments can have negative side effects on production systems. To address this issue, developers can perform canary releases or A/B testing to assess possible side effects for a small number of requests [24], [42].

The testing of an event-driven system's entire event flow is difficult. The developers must verify that the entire sequence of events triggered by a single API call is functioning properly. This involves testing multiple services that interact via HTTP requests or direct invocations. In addition, dealing with asynchronous events and multiple messages published to the same event bus can also be challenging [24]. Besides, applications built on microservices can be dynamic and complex to test. Managing versions of each microservice and exchanging data between multiple microservices further complicates the testing scenarios [35]. Testing serverless-based systems require extensive usage of mocks, and developers need to change their approach to unit and integration testing. Developers also need to simulate the entire application in local machines to test the system comprehensively [24].

The challenges faced by developers while testing and analyzing serverless applications due to the complexity of the system and the potential parallel executions involved. A model is required to assist developers in visualizing and analyzing a serverless application at runtime, concentrating on its characteristics, such as the statelessness of functions and the interdependencies between functions, data storage, and other resources [49]. Existing cloud modeling languages, such as AWS SAM, focus mainly on the deployment and do not consider runtime aspects or parallelism [7]. Graph-based analyses are employed for microservices and event-driven apps but not serverless applications. Winzinger and Wirtz proposed an approach to creating a dependency graph from a serverless application as the basis for an advanced tool that helps developers to analyze, visualize, and track the dependencies between serverless functions, data storages, and other resources deployed in several stages of its life cycle [7]. A dependency graph, along with relevant characteristics added to it, can help developers detect errors during integration testing [50].

Performance testing is essential in ensuring a better user experience, but it is a complex activity in serverless computing with multiple components, interfaces, and technologies. It is essential to validate performance at different test levels, such as:

- Unit-level testing focuses on verifying the functionality of individual code units and functions.
- Integration-level testing focuses on the integration points and external interfaces between various microservices or components to ensure they function as intended.

- System-level testing focuses on the system as a whole to ensure that it satisfies user requirements and operates as anticipated.

Additionally, simulating the workload in a test environment similar to the production environment and validating application response time and system behavior regarding resource utilization is also important for performance testing. Monitoring performance is required to capture multiple matrices and guarantee the system's overall performance [35]. A study by Leitner et al. (2019) highlights a lack of tooling and insufficient documentation as major challenges for developers in serverless testing [42].

Unit tests or the system's deployment to the cloud in a development environment can be used to test serverless computing systems, which can be slow and costly. However, Nimbus offers a solution by allowing the entire application to be deployed locally for local integration testing and then to the cloud without further changes. This local deployment method is more like traditional web development. In contrast to other tools and frameworks, Nimbus allows for local replicating of the whole environment, including data stores, queues, static web pages, and Websockets. It lets developers get feedback faster and supports automated tests on a developer's local computer or as part of a build pipeline. Unfortunately, Nimbus is only for Java-based applications and is currently limited to AWS Lambda deployment [39].

In conclusion, testing challenges for serverless applications summarize as follows:

- Difficulty testing the entire application due to the distributed nature of serverless computing.
- Limited control over the underlying infrastructure and runtime environment.
- Difficulty reproducing and debugging issues due to the ephemeral nature of serverless functions.
- Lack of standardization and portability across different serverless platforms.
- Need for specialized testing tools and frameworks that can handle the unique characteristics of serverless applications.

To overcome these challenges, developers need specialized testing techniques, tools, and frameworks that can accommodate the unique characteristics of serverless applications. Researchers have come up with different ways to help developers test and analyze serverless applications, such as model-based analysis, dependency graphs, and local deployment methods. However, further research is required to address the challenges and develop effective testing approaches for serverless computing.

3.5 Testing and Debugging Approaches in Existing Literature

It is clear that testing is a critical issue in serverless computing, and it is often a complex and challenging activity. Testing a serverless application at different levels is important, especially for integration and regression testing, which can be challenging for large systems with complex configurations. Each microservice in microservices applications can be regarded as a small monolithic system and tested using established unit-testing strategies. However, assessing integration between different microservices and considering scalability across different configurations can be difficult [24]. Separation of code into reusable pieces of business logic is often an afterthought, making it difficult to write unit tests.

Serverless-based applications present further challenges, as each function should be tested through unit tests, but functional testing and debugging involve many functions and external services. Due to the unavailability of environmental dependencies in runtime, testing in a local environment may be problematic. Testing directly in production or a specific cloud-development environment incurs additional costs [24]. Developers adopt good practices such as running unit tests locally for each function by mocking environments and conducting integration testing through canary releases or A/B tests [24], [42].

Many companies struggle with writing integration tests since there are not many platforms available for emulation, and there is always the risk of unforeseen consequences when testing in production, which is necessary to verify the entire system's operability [25]. Instead, they may rely on testing the integrated architecture in a staging environment or may postpone testing altogether. However, this approach can lead to accumulating invisible technical debt that might be more dangerous compared to visible load/stress tests [25]. Load and stress testing is important to check the application behavior and determine the stability and robustness of the system; load/stress tests can also be problematic if they result in oversizing the system, which can increase costs or hit maximum execution times or memory. Postponing testing or not testing a serverless application completely, especially at the integration level, can result in the system not correctly responding or complying with requirements [25]. To address these issues, developers can use patterns and techniques to support a higher load and ensure that the system can react under stress while also avoiding oversizing [25]. Debugging in cloud-based systems is also complex and requires tracking and observing requests among various services, databases, caches, and external API calls. To address unpredictable faults, such as database overloads and communication failures, developers can use fault injection to find and fix these issues [24]. Fault injection is used to imitate real-world

circumstances and test how a system handles failures. This technique can be used to find and fix unpredictable faults, such as database overloads, and communication failures among different microservices and other components. Fault injection helps developers identify potential weaknesses in a system and improve its reliability and resilience. The different programming languages used to implement cloud-based systems can also pose a challenge, which can be addressed through testing services using external containers that host different test designs and implementations [24].

A survey was conducted by Lenarduzzi and Panichella (2021) with two tool vendors and one practitioner to identify good and bad practices for testing and debugging serverless-based applications [24]. The survey findings are summarized in Table 1, which presents various testing approaches for serverless applications along with their key features.

In their research, Lenarduzzi and Panichella (2021) [24] outlined the steps proposed by Cui to address the challenges of testing serverless-based systems. These steps aim to enhance the testing process and ensure the robustness of serverless applications. The outlined steps provide practical guidance for improving the testing process.

Test the entire flow: Develop a comprehensive testing plan to test the entire flow of the event-driven system. This involves triggering an API call that sets off a chain of events across multiple lambdas. Verify that all components are functioning properly by evaluating each component separately and then integrating them.

Use simulation tools: Since serverless applications are deployed in a cloud environment, testing the system in a local environment is challenging. Use simulation tools to replicate the cloud environment and test the system locally.

Test asynchronous events: Develop tests that check the order of messages received in the last N seconds by subscribing and capturing messages at the moment they were sent. When a lambda function publishes a message on the event bus, verify that the request is handled accurately (e.g., the database is successfully updated).

Determine which messages have been published to the event bus: Use appropriate techniques to manage the complexity of event-driven systems when testing multiple messages published to the same event bus. Determine which messages have been published and handle possible side effects.

Overall, testing is a complex activity, and developers must prioritize testing and use specialized testing tools and techniques to address the unique characteristics of serverless computing.

Table 1: *Testing Approaches for Serverless Applications*

Approaches	Key Features
Design test cases to cover possible events and data values each function may receive from the outside. Test for possible values that lambda functions may return. Test lambda functions against failure scenarios and how the functions recover/handle in the case of a failure. Write unit and integration tests for the applications and the local machine/environment. Rely on mocking frameworks. (Epsagon) [24]	Test for non-functional problems like robustness and resilience. Use checklists and experience.
Employ a variety of testing levels, including unit and component-level testing. Test components as a black box where data is shipped into the entry points and verify whether the expected value is obtained at the end state of the API using assertions. Write unit tests locally for the components under maintenance. Test microservices' interactions and mock data from other microservices that interplay to implement a given functionality. (Thundra) [24]	Given the dynamic nature of serverless applications, focus on testing the single microservices' interactions. Monitor serverless applications in real-time to detect issues.
Reverse the testing pyramid and focus testing efforts toward integration points. Use end-to-end testing based on usage scenarios that involve multiple (many) services. Execute end-to-end tests against the product as the system is being deployed. Be cautious with mocks and test the actual services, not just mocks. (Yan Cui) [24]	Integration points are the most probable location of bugs or issues. Be cautious with mocks as they may hit mocks rather than actual services. It involves utilizing multiple testing methods, including unit, integration, and end-to-end testing.

3.5.1 Debugging Approaches

A survey was conducted by Lenarduzzi and Panichella(2021) with two tool vendors and one practitioner to identify good and bad practices for monitoring and debugging serverless-based applications [24]. The survey findings are summarized in Table 2, which presents various monitoring and debugging approaches employed by the participants.

Various companies have employed different approaches to monitoring and debugging their applications. Still, they prioritize gaining visibility into system behavior and identifying and resolving issues through instrumentation, log analysis, and distributed tracing. Epsagon, Thundra, and Cui all have unique approaches to monitoring and debugging, with Epsagon using post-execution (offline) debugging with line-level code instrumentation, Thundra utilizing distributed tracing to analyze how a call propagates through different services and resources, and Cui relying heavily on instrumentation with structured logs and log metrics to understand performance at other integration endpoints. Despite the challenges of reproducing specific issues locally and analyzing execution logs stored in chunks or pieces, all participants prioritize monitoring and debugging to ensure the reliability and stability of their applications [24].

Table 2: *Monitoring and Debugging Approach for Serverless Applications*

Participant	Monitoring and Debugging Approach	Challenges and Solutions
Epsagon	Post-execution (offline) debugging with line-level code instrumentation, an inspection of variable changes during execution	Logging can be complicated and often needs to be updated after a failure has occurred.
Thundra	Distributed tracing to analyze how a call propagates through different services and resources, use of various logs (lambda function, API gateway, message queue, etc.)	Complex monitoring and debugging challenges due to the length of the chain of services and resources involved in an application and the need to analyze execution logs stored in chunks or pieces.
Yan Cui	Heavy reliance on instrumentation (structured logs, centralized logging framework), and use of log metrics to understand Performance at different integration endpoints	The difficulty of reproducing certain issues locally, as many issues can only be found in production under specific conditions.

3.5.2 Crash-Reproducing Test Cases

In their research on testing and debugging serverless-based applications, Lenarduzzi and Panichella (2021) [24] conducted a survey involving multiple participants to identify effective approaches for reproducing crashes and writing test cases. This investigation aimed to improve the understanding of how crashes occur in serverless environments and to develop robust test cases to prevent regression issues. The findings of this survey are summarized in Table 3, which outlines the approaches employed by different participants when it comes to reproducing crashes and creating test cases.

To improve the testing and debugging process for serverless-based applications, there are several best practices that tool vendors can implement. These practices have been identified by experts in the field, such as Ribenzaft and S¸amdan, and have been widely recognized as effective ways to enhance the overall testing and debugging efficiency and effectiveness of serverless-based applications. By implementing these best practices, vendors can improve the overall testing and debugging efficiency and effectiveness for serverless-based applications [24]. Table 4 presents a set of recommendations for tool vendors to adopt as best practices.

Table 3: Approaches to Reproducing Crashes and Writing Test Cases

Participant	Approach to Reproducing Crashes	Approach to Writing Test Cases
Ribenzaft [24]	Reproduce failure by writing a test case or scenario using instrumentation and distributed tracing data	Write test case for future regression testing
S¸amdan [24]	Proceed with fixes for minor crashes, then write postfix tests for regression purposes. For more complicated crashes, retrieve data and write tests upfront for replication purposes	Write test cases for regression purposes
Thundra [24]	Retrieve data and write tests upfront for replication purposes for complicated crashes	Write test cases for regression purposes
Yan Cui [24]	Write crash-reproducing test cases as much as possible, particularly for unusual or unknown failure scenarios. If developers know the most likely failing scenario, write test cases after the fix.	Write test cases for regression purposes, but this may not always be possible due to rare conditions under which the failure occurs.

Table 4: *Recommendations for Tool Vendors - Best Practices*

Best Practice	Recommendations for Tool Vendors
Efficient Replication of Application States	Develop tools that enable efficient replication of application states by recording the states of variables from different lambda functions and microservices.
Automated Test Case Generation and Design	Create automated test case generation and design tools that generate tests from high-level specifications and exercise microservices starting from their main APIs.
Integration Test Generators	Develop integration test generators that can automatically mock out external services, reducing the effort needed to focus on tiny details.
Library Upgrade Decision Tools	Provide tools that help developers decide whether to upgrade a library or not.
Automated Code Review Tools	Create automated code review tools that can determine whether changes hinder the security, scalability, and interactions of the modified services with the unchanged part of the application.

3.5.3 Other Best Practices Across Developers

It is important to learn and compare the best practices used by various developers to improve the testing and debugging of serverless-based applications. In this subsection, we compare the best practices of Thundra, Epsagon, and Cui across various areas, including code review, distributed tracing, third-party library management, instrumentation, and root cause analysis [24]. By analyzing and comparing these best practices, we can identify commonalities and areas for improvement, ultimately leading to more effective testing and debugging approaches for serverless-based applications.

The following table, Table 5, compares different best practices across developers: Thundra, Epsagon, and Cui. While all three developers follow code review practices, Thundra is the only one that mentions careful management of third-party libraries. Distributed tracing is used by all developers, with Thundra using it to provide performance information and Epsagon using it with caution. Instrumentation is used by all three developers, with each taking a slightly different approach. Finally, root cause analysis is approached differently by each developer, with Thundra using distributed tracing to visualize the traces and identify the root cause, Epsagon using distributed tracing and manual inspection, and Cui using log metrics to identify the specific transaction and dig deeper.

Table 5: *Recommendations for Tool Vendors - Best Practices and Tool Features*

Best Practice	Thundra	Epsagon	Cui
Code review	code review before merging changes to the main branch	Not explicitly mentioned	Not explicitly mentioned
Distributed tracing	Analyzes and monitors distributed logs	Uses distributed tracing with caution	Uses log metrics to identify specific transactions
Third-party library management	Locks library versions and performs manual testing when updating	Not explicitly mentioned	Not explicitly mentioned
Instrumentation	Uses instrumentation with caution	Uses instrumentation with caution	Deploys target applications with low debug-level logs
Root cause analysis	Uses distributed tracing tools to visualize traces and identify the root cause. OpenTelemetry used.	Uses distributed tracing and manual inspection	Uses log metrics to identify specific transactions and digs deeper

3.5.4 Determining Adequacy: When is Testing Sufficient?

Testing plays a crucial role in software development as it ensures the quality and reliability of the final product. However, deciding when testing is sufficient can be a complex task, particularly in the case of serverless-based applications. In this section, we explore the insights of three experts in the field - Ribenzaft, Thundra, and Yan Cui - on determining the adequacy of testing for serverless-based applications. Each participant offers unique perspectives on prioritized forms of testing, code coverage, and the importance of edge cases and fault-oriented testing. By understanding their perspectives, we can better understand how to test serverless applications and ensure their reliability and efficacy. Table 6 presents an overview of testing practices, prioritized forms of testing, and approaches to code coverage as shared by each participant.

Table 6: *Testing Practices, Prioritized Forms of Testing, and Code Coverage Approaches*

Participant	Testing Practices	Prioritized Forms of Testing	Approach to Code Coverage
Ribenzaft [24]	Different levels of testing, dynamic testing, strict code reviews	Performance testing, code coverage	Considers memory usage and running time, gain expertise through hands-on training
Thundra [24]	Unit testing, functional and API testing, performance, and load testing	Unit testing for stable services, maintain and update tests for unstable services	Verify changes do not break APIs, test scenarios designed upfront
Yan Cui [24]	Target edge cases on top of code coverage, fault-oriented testing	Edge cases, fault-oriented testing	Not a firm believer in coverage metrics, advocates for test adequacy criteria specific to serverless-based apps

In conclusion, the approaches taken by Ribenzaft's team, Thundra, and Cui provide valuable insights into the different levels and types of testing that can be used to ensure the quality and stability of complex systems. Ribenzaft's team focuses on performance testing and hands-on training to gain expertise, while Thundra heavily relies on unit testing and designing test scenarios upfront. Cui targets obvious edge cases and advocates for test adequacy criteria specific to serverless-based applications, prioritizing edge cases over coverage metrics. These varied approaches highlight the importance of tailoring testing strategies to a given system's specific characteristics and needs of a given system, and of constantly updating and improving test cases to ensure ongoing quality and stability.

3.5.5 Approaches to Determine Coverage Criteria

To address the challenges of testing complex serverless applications composed of stateless, short-running functions that are automatically scaled by cloud providers based on workload, the proposed coverage criteria can be used to ensure effective integration testing. Winzinger (2019) proposed the following coverage criteria to address the unique characteristics of serverless applications [43]. Table 7 provides an overview of these coverage criteria.

Table 7: *Coverage Concepts and Their Explanations*

Characteristic	Coverage Concept	Explanation
Services	Percentage of called resources of different services	Indicates the confidence that the resources of the services are deployed correctly and work properly
Connections	Coverage of all connections between resources	Ensures that the communication between the resources works correctly
Parallelism	Coverage criteria checking for race conditions	Evaluates if hot spots of conflicts are called in certain situations by the test case sets. Both parallel calls of the same workflow and different workflows have to be checked.
Execution time	Testing with different cold start delays and defining patterns using time	Reveals different kinds of errors caused by different execution times. It can influence workflow and provoke race conditions. Errors caused by timeouts can also occur.
Access rights	Test cases using each of the assigned access rights	Indicate the usage of the rights. If no test cases can be created for the relevant rights assigned, the necessity of the assigned access right must be reconsidered.
Dataflow	Coverage of certain code between the definition and the use of a variable	The functions are stateless, so their states have to be persisted somewhere else. Test case quality could be improved by applying well-known coverage criteria to serverless applications.
Workflow	Coverage criteria requiring all workflows to be covered	Identifies application workflows and can be used to ensure that the test case set covers all of them.
Errors	Test cases covering errors and their recovery scenarios	Specific errors for serverless applications include execution errors caused by limited resources like memory and CPU power assigned to a function. Test cases should cover these errors and their recovery scenarios.

Overall, the approach of creating coverage criteria for a system by focusing on its specific characteristics and applying known concepts for those systems can be applied to various types of complex systems. Different criteria can be used to reveal and prevent different types of errors through testing. Automating the testing process and improving tool support for tracking coverage criteria is crucial for ensuring the effectiveness of this approach and improving the quality of complex systems.

3.5.6 Testing Tools

In the dynamic and ever-evolving landscape of cloud-based software systems, testing serverless applications is paramount to guarantee their quality and reliability. As serverless architectures gain popularity, the need for robust testing tools becomes increasingly crucial.

Serverless applications are composed of individual functions that can be invoked and scaled independently, presenting unique challenges for testing and debugging. This section will examine the varied landscape of serverless testing tools and their capabilities, features, and benefits. These tools aim to accelerate the testing process, expand test coverage, and ultimately elevate the quality of serverless applications.

Winzinger and Wirtz (2019) [7] introduced a novel approach for modeling serverless applications using a specialized graph. They proposed using AWS CloudFormation as the infrastructure tool to create the infrastructure model, describing the application's infrastructure and deployed resources. The static graph is created by analyzing the infrastructure file and source code files of serverless functions. In cases where the infrastructure file or source code is unavailable, manual modeling based on worst-case assumptions is possible. The generated graph captures important features of the serverless application, enabling behavior analysis, test case generation, and system monitoring throughout the application's life cycle. The proposed modeling approach applies to AWS Lambda and other platform-specific services, showcasing its broad compatibility.

Apache JMeter is highlighted as a versatile testing tool for evaluating web application performance in multiple studies. Villamizar et al. [51] and Khatri et al. [35] both utilize Apache JMeter for performance testing of web applications. Villamizar et al. compare different architectural approaches and assess the cost reduction potential of microservices in AWS Lambda. Khatri et al. employs Apache JMeter for performance testing strategies, including baseline testing, load and stress testing, and threshold and breaking point testing.

Cicconetti et al. [52] describe an emulator that incorporates Mininet6 and user-space processes for emulating edge computers and routers. This emulator enables testing and

analysis of lambda function performance, including factors like throughput, response time, resource utilization, load balancing, and latency management within a simulated edge/fog computing environment. Table 8 provides a concise summary of the different tests performed within the emulator.

Ivanov and Smolander [5] focus on implementing a DevOps pipeline specifically for Serverless applications. They highlight the importance of monitoring and introduce CloudWatch, Alarms, Metrics, and X-Ray as tools for implementing an effective DevOps pipeline. CloudWatch is used for collecting and tracking metrics, logs, and events related to the application, while Alarms are used to set thresholds for specific metrics to enable proactive monitoring. Metrics collected in CloudWatch provide quantitative data on the application's performance, and X-Ray offers tracing and debugging capabilities.

Pérez, Moltó, Caballer, and Calatrava [1] present a programming model and middleware for high throughput serverless computing applications. While CloudWatch is not explicitly discussed, it is implied that CloudWatch is utilized as a monitoring and observability service offered by AWS. CloudWatch is used to store logs generated by lambda function invocations, facilitating debugging, troubleshooting, and performance monitoring.

Fan, Jindal, and Gerndt [29] compare the performance of microservices and serverless architectures in a cloud-native web application. They utilize k6 as a load and performance testing tool, Grafana for performance data visualization, and InfluxDB as the backend storage for the collected data. These tools enable comprehensive performance testing, monitoring, and analysis of microservices and serverless architectures in a cloud-native web application.

Khatri, Khatri, and Mishra [28] highlight the importance of selecting appropriate performance testing tools for serverless computing. They categorize the tools into performance testing tools (JMeter, Gatling, OpenSTA, Micro Focus LoadRunner, HP Mercury LoadRunner, IBM Performance Tester, BlazeMeter, Neoload), performance monitoring tools (Splunk, Nagios, AppDynamics, Dynatrace), and metrics and monitoring services (AWS CloudWatch, CPU, and memory usage metrics). The selection of performance testing tools should consider factors like performance requirements, critical business transactions, workload modeling, and infrastructure for high load.

Table 8: *Emulator Testing Summary*

Test Type	Description
Throughput Testing	Measures the rate at which lambda functions are processed and executed, providing insights into the system's capacity to handle a high volume of requests.
Response Time Testing	Measures the time taken for a lambda function to be executed and respond to a request, providing information about the system's responsiveness and efficiency.
Resource Utilization	Monitors the utilization of system resources such as processing units (PUs) and memory during the execution of lambda functions, assessing the efficiency and optimization of resource allocation.
Load Balancing Testing	Tests the distribution of lambda requests across multiple edge computers, evaluating the effectiveness of the load balancing algorithm and the system's ability to handle varying workloads.
Latency Management	Facilitates testing and optimization of the routing of lambda requests to minimize latency, evaluating the effectiveness of different lambda forwarding policies in reducing response time and improving overall system performance.

Koch and Hao [53] evaluate AWS services and tools in the context of edge computing. They explore the performance and cost-effectiveness of services such as CloudFront, CloudTrail, CloudWatch, S3, Local Zones, Apache ab, and Athena. These tools facilitate content serving, monitoring, performance analysis, and benchmarking in edge computing scenarios.

GammaRay is introduced as a tracing and debugging tool for AWS Lambda applications in the papers by Ivanov and Smolander [5] and Lin et al. [11]. GammaRay captures important performance and dependency data through logs and service graph summaries, enabling comprehensive analysis and optimization of serverless applications.

Lowgo, introduced by Lin, Krintz, and Wolski [17], provides a comprehensive tracing solution for serverless functions. It records crucial information such as function entry and exit, SDK calls, and causal dependencies within serverless functions. Lowgo adopts a distributed architecture, co-locating multiple instances with serverless functions across different cloud platforms, ensuring record replication, and creating a consistent and distributed shared log. Integration of Lowgo into serverless applications is facilitated through the use of the gRPC framework and HTTP POST for record transmission, with client APIs available for Python, Node.js, and Java. The paper briefly mentions other tracing tools like Google Megastore, Chariots, X-trace, Kronos, LogBase, and Corfu but focuses primarily on highlighting the capabilities and advantages of Lowgo.

The gg tool, implemented using gg's C++ SDK, enables parallel execution of unit tests written with Google Test, as mentioned by Fouladi et al. [54]. Although not directly focused on serverless testing, gg's parallel execution capabilities have broader applicability in various domains, including serverless applications.

Kaplunovich et al. [8] present toLambda, a tool designed to convert Java monolith application code into AWS Lambda Node.js microservices. They emphasize the crucial role of testing and highlight AWS SAM (Serverless Application Model) as a valuable tool for simplifying the deployment, testing, and event generation of Lambda functions.

Carvalho and Araújo [55] leverage a remote procedure call (RPC) approach in conjunction with Function as a Service (FaaS) services to test serverless functions. They utilize the Node2FaaS Framework and Terraform to convert and deploy Node.js applications as serverless functions on AWS Lambda, Google Cloud Functions, and Azure Functions.

Nimbus, introduced by Chatley et al. [39], simplifies the testing process for serverless applications. It offers a local testing environment that allows developers to run and test their serverless application locally before deploying it to the cloud. Nimbus further facilitates unit testing and integration testing of serverless functions through annotation-based configuration, streamlining the testing of individual functions in isolation.

Martins et al. [54] introduced a benchmarking test suite and software tool that automates the execution of tests to evaluate the performance of serverless cloud platforms. The test suite covers aspects such as scalability, memory allocation, CPU-bound scenarios, payload size impact, programming language influence, resource management, and overall platform overhead.

In the troubleshooting phase of Function as a Service (FaaS), researchers have identified a lack of tooling for monitoring, logging, and debugging cloud functions [2]. The monitoring and logging services provided by FaaS providers cover layers like facility, network, hardware, and OS. Middleware and application layers are partly providers and partly user-monitored, with the middleware containing monitoring and logging services. Tools like LogEnhancer and event generation based on system log analysis aid in log quality improvement and test generation. The troubleshooting process involves instrumenting cloud functions using tools like SeMoDe, monitoring with custom implementations or integrated backend services, and fault management through error detection and visualization tools.

Researchers utilize various tools for load testing, metric collection, visualization, and statistical analysis to evaluate the performance of serverless architectures. JMeter is used for load testing, simulating different workload patterns. Prometheus is integrated with the OpenFaaS framework to collect and store cloud-native metrics. Grafana provides customizable dashboards for visualizing the collected metrics. Statistical tests allow for comparisons and insights into the system behavior, assessing scalability, fault tolerance, and high availability.

Maissen et al. [15] introduce FaaSdom, a benchmark suite for serverless computing that enables test deployment, result collection, and price estimations across multiple serverless platforms. The prototype system effectively manages test execution, leveraging InfluxDB as a time series database for storing and retrieving test results. Grafana is integrated for data visualization. The FaaSdom prototype supports multiple cloud provider CLIs, customizable runtime options, and workload injectors for performance evaluation. Baseline tests and benchmarking are conducted using tools like wrk2, evaluating the performance and scalability of different serverless platforms and programming languages.

EvoSuite, Botsing, and EvoMaster are test case generation tools that automate the generation of test suites and aid in system-level testing and coverage. EvoSuite generates test suites for Java programs, Botsing reproduces crashes and exceptions in Java applications, and EvoMaster focuses on generating test cases for RESTful APIs. Additionally, the Python framework Locust is an open-source tool suitable for load testing and

performance testing of serverless applications, particularly for highly concurrent workloads and distributed load testing with virtual users.

Overall, these tools and approaches contribute to the diverse landscape of serverless testing, offering capabilities in benchmarking, monitoring, logging, load testing, metric collection, visualization, test case generation, and debugging. By utilizing these tools, developers can enhance the quality, reliability, and performance of serverless applications.

4. METHODOLOGY

This chapter presents an overview of the research methodology employed in the present study. A Systematic Literature Review (SLR) was employed due to the subject matter being currently under active investigation and the abundance of valuable online content that poses challenges in terms of verification. The remainder of this section provides an overview of the overall process, including the Search Strategy and selection criteria, Data Extraction and Synthesis, and Quality Assessment and Bias Control.

4.1 Research Approach and Design

A Systematic Literature Review (SLR) is an ideal choice for conducting a study on testing serverless applications, especially given the topic's lack of maturity. The Systematic Literature Review (SLR) is a valuable methodology, offering a comprehensive, structured, and transparent overview of the existing research. This approach assists in identifying research gaps and establishing a robust foundation for the study. By reviewing and analyzing methodologies used in the literature, an SLR assists in refining the research methods for the study, ensuring they are effective and suitable. Moreover, due to the rapid evolution of the serverless applications field, conducting an SLR ensures that the research remains current with the most recent technologies and practices, thereby enhancing the overall quality and relevance of the study.

The SLR procedure includes the subsequent steps:

Phase 1: Preparation
- Define Research Question
- Plan Search Strategy
    - Determine Databases
    - Define Search Terms
    - Establish Inclusion/Exclusion Criteria
    - Consult with Subject Experts

Phase 2: Literature Search
- Conduct Literature Search
    - Execute Search Strategy
    - Retrieve Articles
    - Record Search Results

Phase 3: Study Selection
- Screen Titles and Abstracts
    - Review Titles and Abstracts
    - Apply Inclusion/Exclusion Criteria
        - Relevant Studies Found?
            - Yes: Proceed to Screen Full Texts
            - No: End (No studies selected)

- Screen Full Texts
    - Obtain Full Texts
    - Assess Full Texts
    - Apply Inclusion/Exclusion Criteria
        - Selected Studies for Analysis?
            - Yes: Proceed to Phase 4: Data Extraction
            - No: End (No studies selected)

Phase 4: Data Extraction

- Develop Data Extraction Form
- Extract Relevant Data from Studies
- Complete Data Extraction

Phase 5: Quality Assessment

- Assess Study Quality
    - Select Assessment Tools/Criteria
    - Evaluate Study Quality
    - Complete Quality Assessment

Phase 6: Data Analysis and Synthesis

- Analyze and Synthesize Results
    - Perform Qualitative/Quantitative Synthesis
    - Identify Patterns and Trends
    - Analyze Relationships Among Studies

Phase 7: Snowballing

- Identify Relevant Studies from Reference Lists of Selected Articles
- Apply Inclusion/Exclusion Criteria to Snowballing Results
    - Relevant Studies Found?
        - Yes: Proceed to Data Extraction
        - No: Proceed to Interpretation and Reporting

Phase 8: Findings Interpretation

- Interpret and Discuss Findings
    - Analyze and Interpret Synthesized Results
    - Discuss Implications and Limitations
    - Identify Areas for Further Research

Phase 9: Reporting

- Report and Document
    - Prepare Comprehensive Report
    - Follow Publication/Organization Guidelines

End

4.2 Search Strategy and Selection Criteria

To identify relevant literature on serverless testing, a systematic search was employed. The search aimed to include a comprehensive range of research papers from reputable sources. This section presents the search strategy and selection criteria used in the study.

4.2.1 Search Strategy

Identify relevant databases: To initiate the process, begin by selecting databases that comprehensively cover the field of software engineering, cloud computing, and serverless computing in the context of academic papers. Notable databases such as Google

Scholar, ACM Digital Library, SCOPUS, and IEEEXplore Digital Library were chosen for their extensive collection of scientific research papers. These databases are widely recognized for their reputable and scholarly content, ensuring that the selected papers meet the academic standards required for a thorough review.

Define search terms: Developing a robust search strategy is essential for conducting a comprehensive literature review. To ensure the inclusiveness of our study on testing serverless applications, we carefully constructed a set of relevant search terms. The chosen search terms comprised a range of aspects related to serverless computing and testing, incorporating keywords such as "serverless," "function-as-a-service (FaaS)," "AWS Lambda," "Cloud Function," "Azure Function," "Compute Function," "openwhisk," "Cloudflare," "openlambda," "openfaas," "testing," "debugging," and "regression." These keywords were chosen to cover different aspects of testing serverless applications, testing methodologies, and related tools. By employing a combination of these keywords, we aimed to capture a broad range of literature relevant to our research topic.

Construct search queries: To establish a comprehensive search strategy, we applied Boolean operators (AND, OR) to create search queries that effectively incorporate the key concepts of our study. By employing iterative refinement techniques, we meticulously experimented with various combinations of terms to optimize the search queries. This rigorous approach ensured that our exploration spanned a wide spectrum of pertinent literature, facilitating a thorough examination of the current state-of-the-art in testing serverless applications. Through this meticulous process, we aimed to guarantee the inclusion of the most relevant and up-to-date studies, enhancing the rigor and validity of our literature review.

The following search string was adopted to facilitate an extensive and focused exploration of the literature:

- **TITLE-ABS-KEY ((serverless OR faas OR "AWS Lambda" OR "Cloud Function" OR "Azure Function" OR "Compute Function" OR openwhisk OR cloudflare OR openlambda OR openfaas OR Nuclio OR Knative OR Qinling OR Fission OR Kubeless) AND (test* OR debug* OR regression)) AND (LIMIT-TO (SUBJAREA , "COMP"))**

The search string was designed to capture a comprehensive range of literature related to serverless computing and testing methodologies. By combining relevant keywords and concepts, we aimed to retrieve studies that explore various aspects of testing serverless applications, including challenges, strategies, best practices, debugging techniques, performance testing, and so on.

Execute the search: After formulating the search queries, we proceeded to execute the search across the chosen databases. The defined search queries were employed to retrieve a diverse range of scholarly resources, including relevant journals and conference papers. This approach allows us to gather a broader range of scholarly resources and capture the latest research findings and advancements in the field of testing serverless applications. By incorporating both journals and conference papers, we aim to access a diverse set of publications that represent both the academic rigor of journal articles and the timely and innovative contributions often presented at conferences. By leveraging these search queries, we aimed to gather a comprehensive collection of literature pertaining to our research topic. The retrieved materials serve as valuable sources of information for our systematic literature review, providing insights and evidence to support our investigation of testing approaches for serverless applications.

4.2.2 Selection Criteria

To ensure a rigorous and focused selection process for our systematic literature review, we established clear inclusion and exclusion criteria. These criteria serve as guidelines for identifying relevant studies on testing serverless applications. The following are example criteria that were employed:

Inclusion Criteria:

1. Papers that explicitly mention serverless, Function-as-a-Service (FaaS), serverless testing, testing in serverless, and test criteria for serverless applications. These criteria ensure that the selected papers directly address the testing aspects specific to serverless computing.
2. Papers that discuss current test techniques and tools specifically designed for serverless applications. This criterion allows us to explore the existing methodologies and tools employed in testing serverless applications.
3. Papers that describe the implementation of tests for serverless applications. This criterion enables us to examine practical examples of test implementations in the context of serverless computing.
4. Papers that highlight the testing challenges specific to serverless applications. This criterion enables us to understand the unique challenges and considerations associated with testing in the serverless paradigm.

Exclusion Criteria:

1. Papers that are not written in English. This criterion ensures that the selected papers are comprehensible for analysis and synthesis.

2. Papers that solely describe the concepts of serverless or Function-as-a-Service without providing relevant information regarding our research questions (**RQs**). This criterion excludes papers that do not directly address the testing aspects of serverless applications.
3. Duplicated results: Multiple publications reporting the same research or results will be excluded to prevent repetition and redundancy.
4. Dissertations, Thesis, Books, and Technical Reports are excluded to maintain a focus on peer-reviewed research publications.

4.3 Data Extraction and Synthesis

4.3.1 Data Extraction

Data extraction involved systematically extracting relevant information from the selected studies. We developed a structured data extraction form to capture key details from each paper, including authors, publication year, research objectives, research methodology, testing techniques or approaches, Pros/Cons of the testing technique/pattern adopted, tools used, Platform, Platform/Language specific, and major findings related to testing serverless applications. The data extraction form served as a standardized template to ensure consistency in data collection across the selected studies.

Two independent reviewers initially performed the data extraction process, with each reviewer independently extracting data from a subset of the selected papers. In the event of any discrepancies or differences in the extracted data, a third reviewer was consulted to provide an unbiased perspective. Through thorough discussion and consensus among the reviewers, any disagreements were resolved, ensuring the reliability and accuracy of the extracted information. This approach, involving a third reviewer for resolution, further strengthened the validity and consistency of the data extraction process.

4.3.2 Data Synthesis

The data synthesis phase involved analyzing and synthesizing the extracted data from the selected studies. We employed a thematic analysis approach to identify common themes, patterns, and trends related to testing serverless applications. Through a systematic examination of the extracted data, we identified recurring topics, challenges, and emerging practices in the field of serverless testing.

To ensure rigor in the data synthesis process, we used a qualitative approach to interpret the findings and provide insightful discussions. We employed a rigorous process

of comparing the findings across the selected studies, identifying relationships, and drawing meaningful conclusions.

Throughout the data extraction and synthesis phases, we maintained transparency and rigor by maintaining an audit trail of decisions, documenting the analytical process, and ensuring that our interpretations and conclusions were well-supported by the extracted data. This methodology allowed us to provide a comprehensive and evidence-based analysis of the current state of testing serverless applications, as reported in the selected studies.

4.4 Quality Assessment and Bias Control

4.4.1 Quality Assessment

The quality assessment phase involved evaluating the selected studies to assess their methodological rigor, validity, and reliability. We utilized established assessment tools and criteria specific to serverless computing, testing, and tools. These methods helped us objectively assess each study's quality and findings.

Two independent reviewers conducted the quality assessment process. They independently assessed the selected studies based on the predetermined criteria. Any discrepancies or differences in quality assessments were discussed with a third reviewer and resolved through discussion and consensus. This approach helped to ensure the objectivity and consistency of the quality assessment process.

4.4.2 Bias Control

Bias control was a crucial aspect of our systematic literature review. We implemented various measures to minimize potential biases and increase the objectivity of our review.

Firstly, we conducted an extensive search across multiple databases and sources, ensuring a comprehensive coverage of the available literature. By including both published journal articles and conference papers, we aimed to reduce publication bias and capture a broader range of research findings. To conduct a comprehensive search, we employed specific search strings for the ACM Digital Library, SCOPUS, and IEEEXplore Digital Library databases. These search strings were tailored to the search syntax and conventions of each database, optimizing the retrieval of relevant literature. In Google Scholar, a slightly different approach was employed due to its search capabilities. To ensure a comprehensive search, separate searches were conducted for each serverless platform, utilizing specific search strings tailored to the respective platform. For example, “Serverless and (Test or Debug or Regression or Testing or Debugging)”,

"AWS Lambda" and (Test or Debug or Regression or Testing or Debugging)" and so on.

Secondly, the selection process involved independent reviewers who applied predefined inclusion and exclusion criteria. The involvement of multiple reviewers helped mitigate individual biases and increased the objectivity of study selection.

Furthermore, comprehensive documentation of our search methodology, selection criteria, and data extraction procedures was undertaken, ensuring transparency and reliability. By maintaining an audit trail of decisions and discussions, we minimized potential bias in the interpretation and synthesis of the findings.

Lastly, snowballing, both backward and forward citation searches, was conducted to further minimize potential bias. This approach helped identify additional relevant studies that might have been missed in the initial search, reducing the risk of overlooking important contributions.

4.4.3 Assessment of Inter-Rater Agreement

To ensure the reliability and consistency of the data obtained from multiple raters, the assessment of inter-rater agreement was conducted using Cohen's kappa coefficient. Cohen's kappa is a widely recognized measure for evaluating the degree of agreement beyond chance [57].

In this study, the agreement between Rater 1 and Rater 2 was assessed based on the coding of 10 papers and the abstract/title. For the coding of the 10 papers, it was found that both raters agreed by indicating "ACCEPT" in 80 instances, while there were 10 instances where Rater 2 said "NOT ACCEPT," but Rater 1 said "ACCEPT," indicating a disagreement. Additionally, there were no instances where Rater 1 said "NOT ACCEPT," but Rater 2 said "ACCEPT," resulting in a disagreement count of 0. Furthermore, both raters agreed by indicating "ACCEPT" in 20 instances.

Cohen's kappa coefficient was calculated using the equation:

K = (1 - P(e)) / (P(a) - P(e))

Where:

K represents Cohen's kappa coefficient.

P(a) denotes the observed proportion of agreement.

P(e) represents the proportion of agreement expected by chance.

By applying the equation to the obtained data, the computed Cohen's kappa coefficient for the coding of the 10 papers was found to be 0.74. This value indicates a substantial level of agreement in accordance with the guidelines proposed by Landis and Koch [58].

For the assessment of the abstract/title, it was observed that both raters agreed by indicating "ACCEPT" in 131 instances. There were 29 instances where Rater 2 said "NOT ACCEPT," but Rater 1 said "ACCEPT," and 64 instances where Rater 1 said "NOT ACCEPT," but Rater 2 said "ACCEPT," resulting in disagreements. Moreover, both raters agreed by indicating "ACCEPT" in 152 instances.

By applying the same equation to the obtained data for the abstract/title, the computed Cohen's kappa coefficient was found to be 0.51. This value indicates a moderate level of agreement.

These findings demonstrate the degree of agreement between the raters for both the coding of the 10 papers and the assessment of the abstract/title. The substantial level of agreement for the coding of the 10 papers and the moderate level of agreement for the abstract/title provide evidence of reliable and consistent data interpretation. The inclusion of the assessment of inter-rater agreement using Cohen's kappa coefficient in the study exemplifies a commitment to methodological rigor and data quality. By employing a widely recognized statistical measure, the study enhances the validity and reliability of the research findings, ensuring a robust evaluation of agreement between raters.

Overall, through the rigorous quality assessment and bias control measures implemented, we aimed to ensure a systematic and unbiased evaluation of the literature, enhancing the credibility and reliability of our systematic literature review on testing serverless applications.

4.5 Snowballing

In this systematic literature review, we employed snowballing as an additional search strategy to augment the initial literature search. Through forward and backward snowballing, we identified and included 19 additional papers that were not captured in the initial search. These papers were thoroughly evaluated using the same inclusion/exclusion criteria as in the earlier screening phases. The inclusion of these papers enriched our analysis, allowing for a more comprehensive exploration of the research topic. The snowballing process proved valuable in ensuring that relevant studies, including potentially older or less well-known ones, were included in our review, thereby enhancing the breadth and depth of our findings.

4.6 Creation of a Final Pool of Sources

In the process of conducting our systematic literature review on testing serverless applications, an initial pool of 972 sources was identified through the search strategy. Following a rigorous screening and selection process, we have refined the pool to a final selection of 39 research papers, comprising 34 conference papers and 5 journal articles.

The initial screening involved applying predefined inclusion and exclusion criteria to the titles, abstracts, and full texts of the identified sources. This screening process aimed to filter out irrelevant or duplicate studies that did not meet the specific focus of our review. Through this meticulous screening process, we identified the most relevant research papers for our systematic literature review.

Furthermore, we employed snowballing, both backward and forward citation searches, to expand our pool of sources. Through this iterative process, an additional 19 research papers were retrieved, enhancing the comprehensiveness of our review.

The final pool of sources, consisting of 39 research papers from the initial screening and 19 from snowballing, represents a refined selection that aligns with our research objectives. These papers encompass a diverse range of studies, including conference papers and journal articles, providing a well-rounded perspective on testing approaches for serverless applications.

The final pool of sources serves as the foundation for our data extraction, analysis, and synthesis, allowing us to derive meaningful insights and draw well-informed conclusions in our systematic literature review on testing serverless applications.

5. FINDINGS AND DISCUSSION

5.1 Testing Techniques, Patterns, and Anti-patterns in Serverless Applications (RQ1)

This section presents the findings related to testing techniques, patterns, and anti-patterns in serverless applications. It explores their effectiveness, applicability, and unique considerations specific to serverless architectures.

5.1.1 Common Testing Techniques in Serverless Applications

This section provides a comprehensive overview of commonly used testing techniques in serverless applications. Table 9 showcases the specific testing techniques, associated platforms, and corresponding references. The listed techniques include unit testing, integration testing, performance testing, debugging, and various specialized tests such as overhead testing, concurrent load testing, and container reutilization testing. The mentioned platforms include AWS Lambda, Google Cloud Function, Apache OpenWhisk, Microsoft Azure Functions, and IBM Cloud Functions. The provided references offer further insights and details into each testing technique and its application in serverless environments. This section serves as a valuable resource for understanding the diverse testing approaches available for ensuring the quality and reliability of serverless applications.

Unit Testing: In serverless computing, unit testing involves testing individual functions or components of a serverless application in isolation. Each microservice in a microservices application can be treated as a small monolithic system and tested using established unit-testing strategies. However, assessing integration between different microservices and considering scalability across different configurations can be challenging. It is important to design test cases that cover possible events and data values each function may receive from the outside. Unit tests should also check for possible values that lambda functions may return and test the functions against failure scenarios and how they recover or handle failures. Unit testing can be done locally for each function by mocking environments and relying on mocking frameworks [25] [24] [42] [54].

Integration Testing: Integration testing is vital for serverless applications as it focuses on testing the interaction and integration between different microservices, functions, and external services. It verifies that the application's components function correctly and produces the expected results. Data flow, communication between micro-

services, and the functionality of APIs, databases, and other external services are examined during integration testing. Developers employ a variety of testing levels, including unit and component-level testing. They test components as black boxes, shipping data into the entry points and verifying whether the expected value is obtained at the end state of the API using assertions. Additionally, integration testing includes testing microservices' interactions and mocking data from other microservices to implement a given functionality [24], [25], [42], [43], [46], [50].

Performance Testing: Performance testing is essential in serverless computing to evaluate the performance characteristics of the application. It involves measuring and assessing factors such as response times, resource utilization, scalability, and throughput under different workloads. Performance testing identifies performance constraints, scalability problems, and possibilities for optimization. Load and stress testing are important aspects of performance testing to check application behavior, stability, and robustness under high traffic or heavy workloads. However, load and stress tests can also be problematic if they result in oversizing the system, which can increase costs or hit maximum execution times or memory. Patterns and techniques can be applied to support a higher load and ensure that the system can react under stress while avoiding oversizing [15], [24], [28], [29], [35], [51].

Load Testing: Load testing is a subset of performance testing that specifically focuses on assessing the behavior and performance of a serverless application under expected and peak load conditions. It involves subjecting the application to a significant volume of concurrent requests or transactions to evaluate its stability, responsiveness, and ability to handle high loads. Load testing helps identify performance limitations, optimize resource allocation, and ensure the application can handle the expected user demand [22], [29].

Local Testing: Local testing involves running and testing a serverless application in a local development environment or on a developer's machine. It allows developers to test the application's functionality, behavior, and integration without the need for cloud deployments. Local testing involves setting up a local serverless runtime or emulator that replicates the behavior and environment of the production environment. It helps developers catch issues early, debug effectively, and save costs by avoiding unnecessary deployments to cloud environments [39], [55].

Debugging: Lenarduzzi et al. (2021) identified several approaches to monitoring and debugging in serverless applications [24]. These approaches include:

1. **Instrumentation:** Instrumentation plays a crucial role in gaining visibility into the behavior of serverless applications. It involves the insertion of code hooks or

markers to collect data and capture relevant information during execution. This data can include variable changes, performance metrics, and execution traces.

2. **Log Analysis:** Logs generated by serverless applications offer valuable information for monitoring and debugging purposes. Analyzing logs helps identify errors, exceptions, and performance bottlenecks. Structured logs and log metrics provide insights into application behavior and enable effective troubleshooting.
3. **Distributed Tracing:** Distributed tracing techniques are used to track the flow of requests and call across different services and resources within a serverless application. By analyzing the propagation of calls, developers can identify performance issues, latency bottlenecks, and dependencies between components.
4. **Post-execution Debugging:** Post-execution debugging involves analyzing the behavior of a serverless application after it has been executed. It includes examining variables, execution paths, and error conditions to identify and resolve issues. This approach can be particularly useful for offline debugging and in-depth analysis.
5. **Reproduction of Issues:** Reproducing issues in a controlled environment is crucial for effective debugging. However, it can be challenging in the serverless architecture due to the distributed nature of applications and dependencies on external services. Special considerations and techniques may be required to simulate and replicate specific conditions for issue reproduction.

Table 9 presents the common testing techniques used in serverless applications, along with the corresponding platforms and references. The table provides an overview of the testing techniques and their association with specific platforms.

Table 9: *Testing Techniques and Platforms in Serverless Applications*

Testing Technique	Platform(s)	Reference(s)
Unit Testing	AWS Lambda	[5], [2], [39], [24], [54]
Integration Testing	AWS Lambda	[24], [39]
Performance Testing	AWS Lambda	[1], [13], [58],[11], [24], [12], [2], [14], [20], [26], [10], [54], [55], [15], [30], [29], [17], [51], [53], [56], [6]
Debugging	AWS Lambda	[2]
Unit Testing	Google Cloud Function	[54]
Performance Testing	Google Cloud Function	[26], [54], [55], [15], [30], [56]
Performance Testing	Apache OpenWhisk	[18], [20]
Performance Testing	Microsoft Azure Functions	[17], [26], [55] [15], [30], [56]
Unit testing	IBM Cloud Functions	[55]
Performance Testing	IBM Cloud Functions	[26], [54], [15], [30], [56]
Performance Testing	OpenFaaS	[22], [16]
Performance Testing	Fn	[20]
Performance Testing	OpenLambda	[10]
Overhead Test, Concurrent Load Test, Container Reutilization Test, Payload Size Test, Programming Language Test, Memory Size Test, Intensive Computation Test	AWS Lambda, Google Cloud Function, IBM Cloud Functions, Microsoft Azure Functions	[56]
Concurrency Test, Backoff Test	AWS Lambda, Google Cloud Function, IBM Cloud Functions, Microsoft Azure Functions	[30]
Local Testing	AWS Lambda, Google Cloud Function, Microsoft Azure Functions	[39], [55]

5.1.2 Testing Approaches in Serverless Applications

This section focuses on exploring various testing approaches employed in serverless applications. These approaches cover different aspects such as benchmarking, monitoring, and other specialized methods to evaluate the performance, efficiency, and behavior of serverless applications. The section provides insights into the diverse strategies adopted by researchers and practitioners to ensure the quality and reliability of serverless deployments.

Benchmarking Approache(s)

Cordingly et al. [26]: propose a benchmarking approach that utilizes Linux CPU time and multiple regression techniques for deployments across different CPUs, memory sizes, and platforms. They collect metrics from the /proc file system and append them to the JSON payload returned by the function invocation.

Anand et al. [58]: evaluate retrieval latency and cost in serverless applications. They measure component latency and end-to-end latency, considering specific serverless platform parameters such as latency, cost, and error logs. Their approach combines benchmarking and cost analysis.

Mahgoub et al. [10]: propose a benchmarking approach for performance assessment in serverless applications. Their goal is to minimize end-to-end execution time by analyzing and optimizing different stages of the application.

Barcelona-Pons et al. [12]: perform performance testing and comparison of serverless applications with other systems, such as Spark. This allows for a comprehensive evaluation of performance and identification of potential optimizations.

Balis et al. [14]: present a benchmarking approach using the HyperFlow engine for executing workflows in serverless applications. This approach allows for evaluating the performance and efficiency of workflow execution in a serverless environment.

Maissen et al. [15]: propose a benchmarking approach that includes latency tests, CPU tests, filesystem tests, and custom tests implemented through templates. This comprehensive approach enables the evaluation of various performance aspects of serverless applications.

Carvalho et al. [55]: perform CPU, I/O, and memory tests in their benchmarking approach for serverless applications. These tests help in assessing the performance and resource utilization of the application.

Andersen et al. [22]: conduct scalability tests by varying the workload through HTTP requests. They evaluate the performance of serverless applications under different load scenarios, including fixed increasing/decreasing load and varying load.

Monitoring Approache(s)

Lin et al. [11]: introduce a monitoring approach that intercepts Lambda function entry points to record events and generate offline graphs. This approach provides insights into the behavior and performance of serverless applications.

Manner et al. [2]: propose a monitoring approach that involves cloud function instrumentation and the generation of tests based on log data from faulty cloud functions. This approach helps in identifying and resolving issues related to the behavior and performance of serverless applications.

Other Approache(s)

Sedefoğlu et al. [6]: present a forecasting approach for estimating the performance impact of memory settings in serverless applications. They use regression analysis to model the relationship between memory settings and performance metrics.

5.1.3 Testing Patterns in Serverless Applications

The subsection explores important testing patterns for serverless applications. These patterns address specific challenges in testing serverless applications and offer unique benefits such as improved test repeatability, incremental deployment, data-driven decision-making, and enhanced system resilience. These patterns play a crucial role in ensuring the reliability and stability of serverless functions and overall application performance.

Mocking and Stubbing: Mocking and stubbing are techniques used in serverless application testing to simulate external dependencies or services. They allow developers to isolate components during testing by creating fake implementations or predefined responses. This enables unit testing without relying on actual services, improves test repeatability, and facilitates faster feedback cycles. However, it's important to ensure that mocks and stubs accurately represent the behavior of real dependencies. These techniques contribute to the reliability and stability of serverless functions while reducing reliance on external services during testing [5], [24], [42].

Canary Release: Canary release is a testing technique used in serverless applications to minimize the impact of potential issues during deployment. It involves gradually

rolling out new features or updates to a subset of users while monitoring their behavior and performance. By releasing changes to a small group of users, developers can gather real-time feedback and detect any issues before a full deployment. This technique allows for incremental testing and mitigates risks associated with deploying new versions of serverless functions. However, it requires careful monitoring and observability to quickly identify and address any adverse effects on the system [24], [42].

A/B Testing: A/B testing refers to a testing technique used to compare and evaluate different versions or variations of serverless functions or applications. It is a method to determine the most optimal solution or configuration by running experiments with different versions and measuring their performance and behavior [24], [42].

Fault Injection: Fault injection is a technique used to imitate real-world circumstances and test how a system handles failures. In serverless applications, fault injection can be used to identify and fix unpredictable faults, such as database overloads and communication failures among different microservices and other components. By intentionally introducing faults or failures into the system, developers can evaluate the system's resilience and reliability. Fault injection helps identify potential weaknesses in a system and improve its robustness. It is particularly useful for testing error-handling mechanisms, failure recovery strategies, and system stability under adverse conditions [24].

Crash-Reproducing Test: Execute specific test cases that are designed to replicate crashes or failures in a software system. In serverless-based applications, these tests aim to reproduce failures within serverless functions or the overall application. Developers use various approaches, such as using instrumentation and tracing data, fixing minor crashes, writing regression tests, retrieving data and writing tests upfront, and creating tests for unusual or unknown failure scenarios. By reproducing failures in controlled environments, developers gain insights into the root causes and weaknesses of the system, enabling them to improve its stability and resilience. Implementing crash-reproducing tests enhances the overall quality and reliability of serverless applications [24].

5.1.4 Anti-patterns in Serverless Testing

This section discusses common anti-patterns that can hinder the effective testing of serverless applications. These anti-patterns represent common pitfalls and practices to avoid ensuring the reliability, performance, and quality of serverless applications. The anti-patterns discussed include postponing testing, insufficient test coverage, reliance solely on testing in production, oversizing in load and stress testing, lack of environment simulation, and insufficient documentation. By understanding and addressing these anti-

patterns, developers can improve their testing strategies and overcome potential challenges in serverless application development. It is essential to identify and mitigate these anti-patterns to establish robust testing practices and enhance the overall quality and reliability of serverless applications.

Postponing Testing: Postponing or overlooking testing, particularly at the integration level, can lead to non-compliance with requirements and reduced system reliability. It is important to prioritize testing throughout the development lifecycle [25].

Insufficient Test Coverage: Not adequately covering different aspects and functionalities of the application in testing can leave potential vulnerabilities or bugs undetected. It is important to have comprehensive test coverage to ensure a thorough examination of the system under different scenarios and conditions [24].

Testing in Production: Relying solely on testing in the production environment can incur additional costs and risks. It is important to have separate testing environments to ensure proper testing and prevent impact on live systems [25].

Oversizing in Load and Stress Testing: Oversizing the system in load and stress testing can result in increased costs or exceeding execution limits. It is important to carefully plan and execute load and stress tests to accurately simulate real-world scenarios [25].

Lack of Environment Simulation: Facing challenges in simulating the cloud environment locally for testing due to the unavailability of environmental dependencies can impede thorough testing. It is important to leverage appropriate tools and techniques to simulate the required environments effectively [24], [25].

Insufficient Documentation: Inadequate documentation of serverless applications hinders the testing process by limiting understanding of the application's architecture, functionality, and dependencies. This leads to errors, delays, and inefficiencies in testing. Prioritizing thorough documentation improves testing efficiency, collaboration, and the overall quality of serverless applications [42].

5.1.5 Unique Considerations in Serverless Testing

This section discusses several important factors to consider when testing serverless applications. These considerations include addressing cold starts, handling environmental dependencies, managing costs and scalability, implementing monitoring and observability practices, and mitigating vendor lock-in risks. Developers can ensure the dependability, efficacy, and adaptability of serverless applications by understanding and addressing these unique testing considerations.

Cold Starts: Cold starts refer to the delayed execution of a serverless function when it is invoked for the first time or after a period of inactivity. Testing in serverless applications should consider the impact of cold starts on performance and latency. Strategies for testing should account for variations in cold start times across different cloud providers and programming languages. Techniques such as pre-warming containers, periodic warming with dummy requests, and pausing containers can help mitigate the effects of cold starts and improve overall application responsiveness [1], [16], [29], [32].

Environmental Dependencies: Testing serverless applications in a local environment can be challenging due to the unavailability of certain environmental dependencies during runtime. Developers must find ways to simulate or mock these dependencies effectively. Special attention should be given to testing interactions with external services, databases, and APIs to ensure thorough testing and replicate real-world scenarios accurately [24], [25], [30].

Cost and Scalability: Testing serverless applications should carefully consider the cost and scalability aspects. Proper resource management and optimization are crucial to avoid overspending on testing environments and exceeding execution limits. Strategies for load and stress testing should be designed to accurately simulate real-world scenarios while optimizing resource usage and cost efficiency [24], [26], [27].

Monitoring and Observability: Serverless applications require effective monitoring and observability mechanisms to track their behavior and performance. Real-time monitoring helps detect and address issues promptly, ensuring the reliability and stability of the application. Testing strategies should incorporate monitoring and observability practices to gain insights into the performance of serverless functions and overall application health [5], [7], [24], [25], [28].

Vendor Lock-in: Vendor lock-in is a critical challenge in serverless computing, limiting the flexibility and portability of applications. The tight coupling with cloud providers' ecosystems makes it difficult to migrate to alternative platforms. Mitigating vendor lock-in risks and enhancing interoperability is important for the long-term success of serverless architectures [5], [39], [42].

5.1.6 Discussion and Analysis of Testing Approaches in Serverless Applications

RQ1, which examines testing techniques, patterns, and anti-patterns in serverless applications, has been addressed in Section 5.1. The section provides a comprehensive overview of common testing techniques, approaches, and patterns used in serverless applications, covering various aspects such as unit testing, integration testing, performance testing, debugging, and specialized tests. It also discusses important considerations specific to serverless architectures, including monitoring, vendor lock-in, environmental dependencies, and more.

The discussion on testing techniques offers valuable insights into different strategies that developers can employ. It highlights the significance of unit testing for individual functions or components and integration testing for verifying interactions between microservices and external services. Performance testing techniques provide ways to assess scalability, response times, and resource utilization in serverless applications. The section also addresses debugging approaches like instrumentation, log analysis, distributed tracing, and post-execution debugging.

The presentation of testing patterns, such as mocking and stubbing, canary release, A/B testing, fault injection, and crash-reproducing tests, offers practical solutions to common challenges in serverless applications. These patterns contribute to improving reliability, resilience, and overall application quality.

However, there are critical points that need to be considered for further improvement:

1. **Lack of empirical evidence:** The section lacks empirical evidence or quantitative data to support the effectiveness and applicability of the discussed testing techniques in real-world scenarios. While the references provide insights, additional empirical studies and case studies are necessary to validate and compare these approaches across diverse serverless applications and use cases.
2. **Inadequate consideration of security testing:** Security testing is a crucial aspect of application development, including serverless applications. However, the section does not extensively cover security testing techniques or address specific security challenges associated with serverless architectures. Future research should explore comprehensive security testing practices and provide guidelines for ensuring robust security in serverless applications.
3. **Limited discussion on test automation and CI/CD integration:** The section does not explicitly address test automation or the integration of testing into the CI/CD pipeline for serverless applications. Test automation and CI/CD integra-

tion are vital for achieving efficient and reliable testing processes. Future research should delve into best practices and tooling for automating tests and seamlessly integrating them into the CI/CD pipeline specific to serverless architectures.

4. **Lack of performance benchmarking standards**: Although benchmarking approaches are mentioned, there is no discussion on standardized performance benchmarking methodologies or frameworks for serverless applications. Establishing industry-wide standards and guidelines for performance benchmarking would enable more accurate comparisons and evaluations of serverless platforms.

Addressing these critical points would enhance the understanding and application of testing techniques, patterns, and anti-patterns in serverless applications. Future research efforts should focus on conducting empirical studies, exploring comprehensive security testing approaches, emphasizing test automation and CI/CD integration, and establishing performance benchmarking standards. By addressing these issues, serverless application quality, dependability, and security can be strengthened further.

5.2 Tools for Testing Serverless Applications (RQ2)

This section focuses on Research Question 2 (**RQ2**) and aims to identify and evaluate tools specifically designed for testing serverless applications. The objective is to explore the available tools and provide insights into their features, functionalities, and suitability for different testing requirements.

5.2.1 Available Tools for Testing Serverless Applications

1. General-Purpose Testing Tools:

Apache JMeter: Apache JMeter is a versatile testing tool commonly used for evaluating web application performance. It can be utilized for load testing, stress testing, and performance testing of serverless applications. JMeter allows developers to simulate realistic user workloads, generate load, and measure performance metrics such as response time and throughput. It is easy to use and provides comprehensive reports and analysis. JMeter is platform-independent and can be used with various serverless platforms like AWS Lambda, Google Cloud Functions, and Azure Functions.

2. Infrastructure and Deployment Tools:

AWS CloudFormation: AWS CloudFormation is an infrastructure tool provided by Amazon Web Services (AWS) that allows developers to create and manage the infrastructure resources required for serverless applications. It can be used to define and

provision the resources needed for serverless functions, such as AWS Lambda functions, API Gateway, and DynamoDB tables. CloudFormation provides a declarative way to model the infrastructure, making it easy to manage and reproduce the environment for testing. It is compatible with AWS Lambda and other platform-specific services.

3. **Monitoring and Observability Tools:**

AWS CloudWatch: CloudWatch is a monitoring service provided by AWS that collects and tracks metrics, logs, and events related to serverless applications. It can be used to monitor various resources within the serverless architecture, including AWS Lambda functions, DynamoDB tables, and S3 buckets. CloudWatch provides a centralized platform for visualizing and analyzing performance data, allowing developers to monitor resource utilization, detect performance issues, and troubleshoot serverless applications. It can be integrated with other AWS services like AWS X-Ray for advanced tracing and debugging capabilities.

Grafana: Grafana is a powerful data visualization and analytics platform that is often used in conjunction with monitoring services like AWS CloudWatch or Prometheus. It allows developers to create customized dashboards and visualizations to monitor and analyze metrics collected from serverless applications and infrastructure. Grafana helps provide insights into system performance, resource utilization, and other key metrics, aiding in the monitoring and observability of serverless architectures.

Prometheus: Prometheus is an open-source monitoring and alerting toolkit used to collect, store, and analyze metrics from various sources, including serverless applications. Prometheus enables developers to monitor and visualize key performance indicators, troubleshoot issues, and optimize the performance of serverless applications.

4. **Distributed Tracing and Debugging Tools:**

AWS X-Ray: X-Ray, also provided by AWS, is a tracing and debugging tool specifically designed for AWS Lambda applications. It offers performance and dependency data through logs and service graph summaries. X-Ray provides a comprehensive view of the request flow within the serverless architecture, enabling developers to identify bottlenecks, latency issues, and areas for performance optimization. It facilitates detailed insights into the execution of Lambda functions and the interaction between different application components.

5. **Emulation and Simulation Tools:**

Mininet and EmuFog: Mininet and EmuFog are tools that facilitate the testing and analysis of serverless applications within an edge/fog computing environment. These tools enable the emulation of edge computers and routers, allowing for the performance

evaluation of lambda functions. Mininet provides capabilities for testing throughput, response time, and resource utilization, while EmuFog supports load balancing and latency management for lambda request routing.

6. **Performance Testing and Benchmarking Tools:**

GammaRay: GammaRay is a cloud service and tracing tool developed for AWS Lambda applications. It captures causal dependencies and provides a holistic view of application behavior and performance. GammaRay offers comprehensive tracing capabilities, including event profiling and performance data collection. It eliminates the need for developers to build their own log parsing and aggregation tools, simplifying the testing and analysis process. GammaRay is particularly effective for concurrent, multi-function Lambda applications and supports different implementation alternatives to minimize execution time and memory overhead.

7. **Language-Specific Testing Tools:**

Locust: Locust is an open-source tool for load testing and performance testing. It allows developers to write test scenarios using Python code and is suitable for testing highly concurrent workloads and performing distributed load testing with virtual users. Locust can be used to test the scalability and performance of serverless applications, including those built on platforms like OpenFaaS.

Google Test Library: Google Test is a testing framework for C++ that provides a comprehensive set of tools and utilities for writing unit tests. It can be used to test the functionality and correctness of serverless applications implemented in C++.

8. **Test Case Generation Tools:**

EvoSuite: A test suite generation tool for Java programs at the unit level.

Botsing: A tool focused on reproducing crashes and exceptions in Java applications.

EvoMaster: A tool for generating test cases for RESTful APIs.

9. **Performance Testing and Benchmarking Tools:**

Apache ab: Apache ab is a benchmarking tool used to simulate stress loads on serverless applications and measure performance metrics such as throughput and response time. It helps evaluate the scalability and performance of the system under various load conditions.

10. **Serverless Testing Frameworks:**

ServerlessBench: ServerlessBench is a testing framework specifically designed for serverless architectures. It offers a set of performance tests and benchmarks to evaluate the performance and scalability of serverless applications. It helps identify performance bottlenecks and optimize resource utilization.

11. Automated Test Case Generation Tools:

Custom Shell Script: Custom shell scripts can be developed to automate the generation of test cases for serverless applications. These scripts can be tailored to the specific requirements and characteristics of the application, enabling efficient and systematic test case generation.

12. Monitoring and Alerting Tools:

Splunk, Nagios, AppDynamics, Dynatrace: These are monitoring and alerting tools commonly used in serverless environments. They provide real-time monitoring, performance analysis, and issue detection capabilities to ensure the smooth operation of serverless applications.

13. Cloud Service-Specific Testing Tools:

AWS SAM and AWS Cloud9: AWS SAM (Serverless Application Model) is a framework that simplifies the development, testing, and deployment of serverless applications on AWS. It provides a local testing environment and integration with AWS CloudFormation for infrastructure management. AWS Cloud9 is an integrated development environment (IDE) that supports the editing, deployment, and debugging of serverless functions.

14. Debugging and Troubleshooting Tools:

X-Ray, CloudTrail, and LogEnhancer: X-Ray, CloudTrail, and LogEnhancer are tools that facilitate the tracing, debugging, and troubleshooting of serverless applications. X-Ray provides insights into the execution flow and performance of Lambda functions, while CloudTrail logs API calls for auditing and analysis. LogEnhancer enhances log quality and supports test generation based on system log analysis.

15. Other Testing Tools:

Locust: An open-source tool for load testing and performance testing.

Spring: A framework providing monitoring, logging, and debugging capabilities for cloud functions.

SeMoDe: Tools for error detection and visualization in serverless functions.

FaaSDOM: A benchmark suite and software tool for evaluating the performance of serverless cloud platforms.

Node2FaaS Framework and Terraform: Tools for converting and deploying Node.js applications as serverless functions on multiple cloud platforms.

Nimbus: Nimbus is a testing tool specifically designed for serverless applications. It provides a local testing environment and supports annotation-based configuration for serverless functions. These features make it easier for developers to test and verify the behavior of their serverless functions before deploying them to the cloud.

The following table (Table 10) presents a comparison of testing tools and techniques used for various serverless platforms. The listed tools cater to different testing requirements, such as unit testing, integration testing, performance testing, monitoring, benchmarking, and troubleshooting. Each serverless platform has its own set of tools and techniques available for testing purposes.

Table 10: *Comparison of Testing Tools for Serverless Platforms*

Serverless Platform	Testing Tools	Testing Techniques
AWS Lambda	AWS CloudFormation [7], Apache JMeter [51], Mininet and EmuFog [52], AWS CloudWatch [53] [1], Alarms [5], Metrics and X-Ray [5], GammaRay [17] [11], Dapper, Google Megastore, Chariots, X-trace, Kronos, LogBase, and Corfu [17], CloudTrail, Apache ab and Athena [53], AWS SAM and AWS Cloud 9 [8], Google Test Library [54] LogEnhancer, Spring and SeMoDE [2], FaaSDOM [15], Custom shell script [55] Benchmarking Tool [56], Nimbus [39], Custom Java program [58], k6 [29], ServerlessBench [20]	Unit Testing, Integration Testing, Performance Testing, Monitoring, Benchmarking, Troubleshooting
Google Cloud Function	JMeter [56], Benchmarking Tool [56], FaaSDOM [15], Custom Shell Script [55]	Performance testing, Load Testing, Benchmarking
Apache OpenWhisk	Witt [18], Developed tool [30], ServerlessBench [20]	Performance Testing, Monitoring, Benchmarking
Microsoft Azure Function	JMeter [56], X-Ray, CloudWatch, CloudTrail, Benchmarking Tool [56], FaaSDOM [15], Custom Shell Script [55]	Performance Testing, Monitoring, Benchmarking
OpenFaaS	Apache JMeter, Prometheus and Grafana [16], Locust [22]	Performance Testing, Monitoring
IBM Cloud Function	Apache JMeter [56], Custom Java program, Benchmarking Tool [56], FaaSDOM [15]	Performance Testing, Benchmarking
Fn	ServerlessBench [20]	Performance Testing, Benchmarking
OpenLambda	SONIC [10]	Performance Testing

5.2.2 Comparison of Testing Tools

Testing serverless applications is crucial in ensuring their quality and reliability within the dynamic and evolving landscape of cloud-based software systems. As serverless architectures gain popularity, the demand for robust testing tools becomes increasingly significant. However, due to the unique nature of serverless applications, testing and debugging present specific challenges that require specialized tools. In this section, we compare various testing tools for serverless applications, considering their capabilities, features, and benefits to aid developers in making informed choices for their testing needs.

One category of testing tools for serverless applications is performance testing tools. Apache JMeter, a versatile testing tool, is frequently mentioned in the literature. It offers comprehensive options for evaluating web application performance, making it suitable for load testing, stress testing, and performance testing of serverless applications across different platforms. Apache JMeter enables developers to simulate various workload patterns, measure response times, and assess throughput and scalability. Its versatility and wide range of performance testing capabilities make it a popular choice for evaluating the performance of serverless applications [35], [51].

Infrastructure and deployment tools are other essential categories for testing serverless applications. AWS CloudFormation, a widely used infrastructure tool, simplifies the management of infrastructure resources required for serverless applications. It allows developers to describe the application's infrastructure and deploy resources using templates. AWS CloudFormation streamlines the provisioning and configuration of serverless functions and their associated resources, enabling the efficient and reliable deployment of serverless applications [7].

Monitoring and observability tools play a critical role in monitoring the performance and resource utilization of serverless applications. Tools such as AWS CloudWatch, Grafana, and Prometheus enable developers to collect, track, and analyze metrics, facilitating effective monitoring and optimization of serverless applications. AWS CloudWatch provides a centralized platform for visualizing and analyzing metrics, logs, and events related to the application. It monitors various resources and services within the serverless architecture, including Lambda functions, DynamoDB tables, and S3 buckets. Grafana offers customizable dashboards for visualizing performance metrics, while Prometheus provides a robust monitoring and alerting framework. These tools, in combination, provide developers with comprehensive monitoring capabilities to ensure the optimal performance of serverless applications [7], [29].

Distributed tracing and debugging tools are essential for gaining insights into the request flow and dependencies within serverless architectures. Tools like AWS X-Ray enable developers to trace and debug serverless applications and identify performance bottlenecks, latency issues, and areas for optimization. AWS X-Ray provides a comprehensive view of the request flow, allowing developers to understand the interactions between different application components and pinpoint areas that require optimization. By utilizing distributed tracing and debugging tools, developers can enhance the performance and reliability of their serverless applications [5], [7].

Language-specific testing tools are designed to cater to developers working with specific programming languages, ensuring the functionality and correctness of serverless applications. For example, Locust and Google Test Library offer comprehensive frameworks for writing unit tests and validating serverless code. These tools enable developers to test their serverless functions in isolation and verify their behavior under different scenarios, ensuring the reliability of the application [22].

Test case generation tools automate the process of generating test cases for serverless applications, ensuring comprehensive test coverage. EvoSuite, Botsing, and EvoMaster are examples of such tools. EvoSuite generates test suites for Java programs at the unit level, while Botsing is designed to reproduce crashes and exceptions in Java applications. EvoMaster focuses on generating test cases for RESTful APIs, enhancing system-level testing and coverage. These tools assist developers in systematically testing their serverless applications and identifying potential issues [24].

Cloud service-specific testing tools, such as AWS SAM (Serverless Application Model) and AWS Cloud9, provide developers with platform-specific tools to simplify the development, testing, and deployment of serverless applications on specific cloud platforms. AWS SAM offers a framework for defining serverless applications and automating their deployment and management. AWS Cloud9 is a cloud-based integrated development environment (IDE) that supports the editing, deployment, and debugging of serverless applications. These tools enhance the productivity and efficiency of developers working on serverless applications within specific cloud environments [8].

In conclusion, testing tools for serverless applications encompass various categories, each addressing specific testing requirements. Apache JMeter and language-specific testing tools offer comprehensive options for performance testing and validating serverless code, respectively. Infrastructure and deployment tools like AWS CloudFormation streamline the provisioning and configuration of serverless applications. Monitoring and observability tools, including AWS CloudWatch, Grafana, and Prometheus, enable comprehensive monitoring and analysis of serverless applications. Distributed tracing and

debugging tools such as AWS X-Ray provide insights into the request flow and dependencies within serverless architectures. Test case generation tools automate the generation of test cases for thorough testing. Cloud service-specific testing tools simplify the development, testing, and deployment of serverless applications on specific cloud platforms. By leveraging these testing tools, developers can effectively test, optimize, and ensure the robustness of their serverless applications, leading to enhanced quality and reliability [7], [29], [51].

The following table (Table 11) presents a comprehensive comparison of various testing tools for serverless applications. This comparative analysis aims to showcase the key features, advantages, and limitations of each tool, empowering developers to make informed decisions when selecting the appropriate testing tools for their serverless applications. By carefully examining and contrasting these tools, valuable insights can be gained regarding their suitability for fulfilling different testing requirements within the serverless architecture.

5.2.3 Discussion and Analysis of Testing Tools for Serverless Applications

Research Question 2 (**RQ2**) focuses on identifying and evaluating tools specifically designed for testing serverless applications. The provided information in Section 5.2.1 outlines various testing tools categorized into different types based on their functionality and purpose.

The comparison of these tools in Section 5.2.2 helps to evaluate their features, functionalities, and suitability for different testing requirements. By examining the capabilities of each tool, developers can make informed decisions regarding the selection of appropriate testing tools for their serverless applications.

The discussion addresses RQ2 by highlighting the availability and diversity of testing tools specifically designed for serverless applications. It emphasizes the importance of using specialized tools to address the unique challenges and requirements of serverless architectures.

General-purpose testing tools, such as Apache JMeter, provide developers with versatile options for evaluating web application performance. These tools can be used for load testing, stress testing, and performance testing of serverless applications across different platforms.

Table 11: Comparison of Testing Tools

Tool	Key Features	Suitable For	Advantages	Limitations
Specialized Graph Modeling	Modeling serverless applications using a specialized graph	Understanding application behavior	Offers a systematic approach for modeling serverless applications. Enables behavior analysis and test case generation, and monitoring	Manual modeling is required if infrastructure files, or source code are unavailable. Limited to specific platforms
Apache JMeter	Load testing, stress testing, and performance testing of web applications	Performance testing and cost analysis	A versatile tool for evaluating web application performance. Enables load testing and comprehensive cost evaluation	Primarily focused on load and performance testing. Additional tools may be needed for detailed analysis.
AWS CloudWatch	Collecting and tracking metrics, logs, and events	Monitoring and optimization of applications	Provides comprehensive monitoring and troubleshooting capabilities. Centralized platform for visualization and analysis.	It may require additional tools for in-depth tracing and debugging.
Prometheus	Monitoring and Alerting	Monitoring and performance analysis	Offer powerful monitoring and alerting capabilities. Integrates well with various systems and provides rich visualization options.	Requires additional setup and configuration. Limited support for complex distributed tracing scenarios.
AWS X-Ray	Distributed Tracing and Debugging	Tracing and debugging of serverless apps	Provide insights into request flow and dependencies. Helps identify performance bottlenecks and optimize serverless applications.	Limited to AWS Lambda and may require additional configuration for complex architectures.
Locust	Load testing and performance testing using Python	Load testing of serverless applications	An open-source framework for highly concurrent and distributed load testing. Supports Python scripting for flexible test scenarios	Requires Python programming knowledge and configuration.
Google Test Library	A comprehensive framework for writing unit	Unit testing of serverless code	Robust testing framework for validating C++ serverless code. Offers various	Limited to C++ applications and may not cover other

Tool	Key Features	Suitable For	Advantages	Limitations
	tests for C++ applications		testing features and extensive community support	programming languages
Test Case Generation Tools	Automated generation of test cases for serverless applications	Comprehensive testing coverage	Assists in systematically testing serverless applications and identifying potential issues	It may require customization or adaptation for specific use cases and scenarios.
AWS SAM	Simplified development, testing, and deployment of serverless applications on AWS	Development and deployment of AWS serverless	Provides a streamlined workflow for building, testing, and deploying serverless applications on AWS	Limited to the AWS platform
AWS Cloud9	Integrated development environment (IDE) for serverless application development on AWS	Serverless application development	Offers a feature-rich IDE with built-in collaboration and deployment capabilities for AWS serverless development	Limited to the AWS platform
LogEnhancer	Enhancing log quality and generating test cases based on log analysis	Log analysis and test case generation	Improves log quality for better debugging and troubleshooting. Assists in generating test cases from log data.	Primarily focused on log analysis and may require additional tools for comprehensive testing

Infrastructure and deployment tools like AWS CloudFormation simplify the management of infrastructure resources required for serverless applications. They ensure the efficient provisioning and configuration of serverless functions and their associated resources.

Monitoring and observability tools play a critical role in monitoring the performance and resource utilization of serverless applications. Tools like AWS CloudWatch, Grafana, and Prometheus enable developers to collect, track, and analyze metrics, allowing them to monitor and optimize the performance of their applications.

Distributed tracing and debugging tools, such as AWS X-Ray, provide insights into the request flow and dependencies within serverless architectures. They help developers identify performance bottlenecks, latency issues, and areas for optimization.

Language-specific testing tools cater to developers working with specific programming languages, ensuring the functionality and correctness of serverless applications. Tools like Locust and Google Test Library offer comprehensive frameworks for writing unit tests and validating serverless code.

Test case generation tools automate the process of generating test cases for serverless applications, ensuring comprehensive test coverage. They assist developers in systematically testing their applications and identifying potential issues.

The availability of cloud service-specific testing tools, such as AWS SAM and AWS Cloud9, enables developers to simplify the development, testing, and deployment of serverless applications on specific cloud platforms.

Debugging and troubleshooting tools, including X-Ray, CloudTrail, and LogEnhancer, aid in tracing, debugging, and troubleshooting serverless applications. They provide detailed insights into application execution, log analysis, and system behavior, facilitating effective issue resolution.

In conclusion, the discussion addresses Research Question 2 by providing a comprehensive overview of the available testing tools for serverless applications. It highlights the suitability of each tool for different testing requirements and emphasizes the importance of selecting the appropriate tools based on specific needs and constraints. By leveraging these specialized tools, developers can effectively test, optimize, and ensure the robustness of their serverless applications.

5.3 Challenges in Testing Serverless Applications (RQ3)

5.3.1 Testing Issues in Serverless Applications

Testing serverless applications poses several unique challenges due to the dynamic and distributed nature of serverless computing. In this section, we will discuss the key challenges faced in testing serverless applications and explore their implications for ensuring software quality and reliability.

Difficulty Testing the Entire Application

1.1 Distributed Nature of Serverless Computing

- Replicating the entire system locally is challenging due to the independent deployment of functions triggered by events [24]
- Testing integration of multiple functions or external services becomes complex without local replication [7], [42].

1.2 Validating Event Flow and Interactions

- Comprehensive testing of event-driven architectures requires validating the event flow and interactions between services [24]
- Asynchronous events and multiple messages published to the same event bus add to the testing complexity [24]

Limited Control over the Underlying Infrastructure and Runtime Environment

2.1 Reproducing and Debugging Issues

- Lack of control over the underlying infrastructure makes it challenging to reproduce and debug issues effectively [23][48].
- The ephemeral nature of serverless functions adds difficulty in reproducing and analyzing issues during execution [25].

Lack of Standardization and Portability

3.1 Heterogeneity Across Serverless Platforms

- Variations among serverless platforms result in a lack of standardization and portability [42].
- Developing testing frameworks and tools that work seamlessly across multiple platforms becomes challenging.

3.2 Absence of Standard Testing Practices and Guidelines

- Lack of standardized testing practices and guidelines specific to serverless architectures [20].
- The difficulty for developers in adopting consistent and efficient testing approaches.

Need for Specialized Testing Tools and Frameworks

4.1 Handling Unique Characteristics of Serverless Applications

- Distributed, event-driven, and stateless nature of serverless functions requires specialized testing tools and frameworks [42].
- Existing cloud modeling languages and tools may not adequately address runtime characteristics or parallelism in serverless applications [7].

4.2 Performance Testing in Serverless Architectures

- Performance testing tools and methodologies tailored to serverless architectures are needed [35].
- Validating performance at different levels and simulating production workloads is essential.

To address these challenges and ensure the quality and reliability of serverless applications, further research and development are needed. This includes:

- Advancing specialized testing techniques, frameworks, and tools for serverless computing.
- Exploring model-based analysis and dependency graphs to visualize and analyze serverless applications [7], [50].
- Promoting standardization and portability efforts across serverless platforms [42].
- Investing in performance testing tools and methodologies specific to serverless architectures [35].
- Encouraging collaboration between researchers, developers, and cloud providers to establish best practices.

The following table (Table 12) summarizes the key challenges encountered in testing serverless applications. These challenges are categorized into distinct areas, including performance, cold start, testing methodology, resource management, testing environment, and workload distribution. Each challenge is supported by relevant references, providing additional resources for further investigation and analysis.

Table 12: *Key Challenges in Testing Serverless Testing*

Challenge(s)	Category	Reference(s)
Prevalence of heterogeneous CPUs found on AWS Lambda and IBM Cloud Functions, resulting in disparate performances	Performance	[26]
Lack of GPU support in AWS Lambda to accelerate neural inference	Performance	[58]
Evaluating Sensitivity to the cold-start Problem	Cold Start	[10]
Impact of startup overhead on short-running applications, necessitating payload size minimization, and startup process optimization	Cold Start	[11]
Annotating test cases requiring file system access	Testing Methodology	[54]
Identification of FaaS cold start at the beginning	Cold Start	[12]
Execution of short-lived tasks in AWS Lambda and longer tasks in Amazon ECS	Resource Management	[14]
Disparate cold start times across providers and call latency	Cold Start	[15]
Replicating the real environment in local machines, inability to perform system testing in a local environment, measuring test coverage	Testing Environment &Coverage	[24]
Imbalanced distribution across OpenFaaS pods during rapid workload increases	Workload Distribution	[22]

To address the challenges that arise in the testing of serverless applications and confirm their reliability and quality, it is important to concentrate on additional research and advancement. This involves advancing specialized testing techniques that are tailored to the unique characteristics of serverless architectures. Additionally, efforts should be made to promote standardization and portability across different serverless platforms, enabling consistent testing practices and tooling. Investing in performance testing tools and methodologies specific to serverless environments is crucial for evaluating scalability, responsiveness, and resource utilization. Moreover, fostering collaboration between researchers, developers, and cloud providers can facilitate knowledge sharing and the development of best practices in serverless testing.

5.3.2 Discussion: Addressing the Challenges in Testing Serverless Applications (RQ3)

Section 5.3.1 of the study sheds light on the intricate nature of serverless computing and its profound implications on the testing phase. By delving into Research Question 3 (**RQ3**), which centers on the obstacles encountered in testing serverless applications, a comprehensive understanding of the vital challenges that must be addressed to guarantee the excellence and dependability of such applications is presented.

Research Question 3 (**RQ3**) focused on the challenges in testing serverless applications. Through our analysis, we have identified several key challenges and explored the implications they have on ensuring software quality and reliability in serverless computing.

Testing serverless applications presents unique challenges due to their distributed nature and limited control over the underlying infrastructure. Unlike traditional monolithic applications, serverless applications consist of independently deployed functions triggered by events, making it challenging to test the entire system and integrate multiple functions or services. Additionally, the lack of standardization and portability across serverless platforms further complicates testing practices. Developers have limited control over the infrastructure and face difficulties in reproducing and debugging issues effectively. The ephemeral nature of serverless functions adds complexity to issue analysis during execution. These challenges highlight the need for specialized testing techniques and tools, as well as standardization efforts and collaboration among researchers, developers, and cloud providers to address the unique testing requirements of serverless applications.

Addressing these challenges requires the development of specialized testing techniques, frameworks, and tools tailored to the unique characteristics of serverless applications. Additionally, exploring model-based analysis and dependency graphs can aid in visualizing and analyzing serverless applications, assisting in error detection and understanding runtime characteristics. Promoting standardization and portability efforts across serverless platforms is crucial for enabling consistent testing practices and tooling.

Furthermore, investing in performance testing tools and methodologies specific to serverless architectures is necessary to validate performance at different levels and simulate real-world workloads. By evaluating scalability, responsiveness, and resource utilization, developers can ensure optimal performance in serverless applications.

To address **RQ3**, our research highlights the importance of advancing specialized testing techniques, exploring model-based analysis, promoting standardization and portability, and investing in performance testing tools and methodologies. Additionally, fostering collaboration between researchers, developers, and cloud providers is essential in establishing best practices and sharing knowledge in the field of serverless testing.

In conclusion, our analysis of the challenges in testing serverless applications has provided insights into the unique testing requirements in serverless computing. By addressing these challenges and implementing the recommended strategies, researchers, developers, and practitioners can enhance their understanding and improve the quality and reliability of serverless applications.

5.4 Evaluation and Comparison of Identified Solutions

5.4.1 Strengths and Weaknesses

The findings and discussions regarding testing techniques for serverless applications have revealed several advantages and disadvantages:

Pros:

1. **Comprehensive coverage:** The identified techniques address various aspects of testing, including unit testing, integration testing, performance testing, and monitoring. This comprehensive approach ensures thorough evaluation and validation of serverless applications.
2. **Improved reliability and stability:** By isolating dependencies, simulating failures, and reproducing crashes, these techniques help identify and fix potential issues in serverless applications. This improves the reliability and stability of the applications under different conditions.

3. **Faster feedback cycles:** Techniques such as mocking and stubbing enable faster feedback cycles by eliminating the need for external services and dependencies during testing. This speed up the testing process and allows for quicker iteration and bug resolution.
4. **Risk mitigation:** Canary release and A/B testing techniques provide risk mitigation strategies by gradually deploying changes and monitoring their impact. This minimizes the potential negative effects on users and allows for timely issue detection and resolution.
5. **Enhancing system resilience:** Fault injection and crash-reproducing tests help identify weaknesses and enhance the resilience of serverless applications. By intentionally introducing faults and simulating failures, developers can improve error-handling mechanisms and recovery strategies.

Cons:

1. **Complexity and additional overhead:** Some testing techniques, such as canary release and A/B testing, add complexity to the deployment process. They require careful monitoring, observability, and management of multiple versions, which can increase overhead and operational complexity.
2. **Limited scope:** While the identified techniques cover a wide range of testing aspects, there may still be specific scenarios or niche requirements that are not adequately addressed. Developers need to consider the specific needs of their applications and potentially combine multiple techniques for comprehensive coverage.
3. **Test environment limitations:** Simulating the cloud environment locally for testing can be challenging due to the unavailability of certain dependencies. Developers need to find effective ways to simulate or mock these dependencies to ensure thorough testing and replicate real-world scenarios accurately.
4. **Learning curve and expertise:** Some techniques, such as fault injection and crash-reproducing tests, require a certain level of expertise and knowledge. Implementing these techniques effectively may require additional learning and skill development.

Overall, the identified testing techniques provide valuable approaches to tackle the challenges of testing serverless applications. While they offer numerous benefits in terms of coverage, reliability, and feedback cycles, developers need to carefully consider the trade-offs and select the most appropriate techniques based on their specific application requirements and constraints.

5.4.2 Common Patterns and Trends

A common trend among the identified solutions is the recognition of the event-driven nature of serverless computing. Solutions prioritize validating event flow and interactions, indicating the importance of addressing event-driven behavior in testing serverless applications.

Another common pattern is the emphasis on performance testing. The solutions provide features and methodologies to evaluate performance at different levels, including function response time, scalability, and resource utilization. This reflects the significance of assessing and optimizing performance in serverless applications.

These commonalities suggest that testing event-driven behavior and evaluating performance are critical aspects of serverless application testing.

5.5 Evaluation of Testing Tools

The evaluation of testing tools for serverless applications reveals a range of options catering to different testing requirements. These tools can be categorized into performance testing, infrastructure and deployment, monitoring and observability, distributed tracing and debugging, language-specific testing, test case generation, cloud service-specific testing, and specialized graph modeling.

Performance testing tools like Apache JMeter offer extensive capabilities for load testing, stress testing, and performance testing, allowing developers to assess the scalability and performance of serverless applications. Infrastructure and deployment tools such as AWS CloudFormation simplify the provisioning and management of serverless resources, enabling efficient deployment and configuration.

Monitoring and observability tools like AWS CloudWatch, Grafana, and Prometheus facilitate the collection, tracking, and analysis of metrics, empowering developers to monitor and optimize serverless applications effectively. Distributed tracing and debugging tools like AWS X-Ray provide valuable insights into the request flow and dependencies within serverless architectures, helping identify performance bottlenecks and optimize application performance.

Language-specific testing tools, such as Locust and Google Test Library, offer comprehensive frameworks for writing unit tests and validating the functionality of serverless code. Test case generation tools automate the generation of test cases, ensuring comprehensive test coverage and assisting developers in systematically testing serverless applications.

Cloud service-specific testing tools like AWS SAM and AWS Cloud9 provide platform-specific tools for simplified development, testing, and deployment of serverless applications on specific cloud platforms. Specialized graph modeling tools aid in understanding application behavior and generating test cases based on specialized graphs.

The evaluation highlights the strengths and limitations of each testing tool. While some tools offer broad functionality and versatility, others are more focused on specific aspects of testing. Developers need to consider their specific testing requirements and choose the appropriate tools that align with their needs, development environment, and preferred programming languages. By leveraging the right combination of testing tools, developers can effectively test, monitor, optimize, and ensure the quality and reliability of their serverless applications, ultimately leading to enhanced user experience and system performance.

5.6 Synthesis of Best Practices in Testing Serverless Applications

Based on the research conducted, several best practices have been identified for testing serverless applications:

1. **Emphasize Performance Testing:** Performance testing is crucial to assess the performance characteristics, scalability, and resource utilization of serverless applications. It helps identify performance limitations, optimize resource allocation, and ensure the application can handle expected workloads effectively.
2. **Automate Testing Processes:** Automation plays a vital role in streamlining testing activities. Leveraging custom scripting, testing frameworks, and infrastructure tools can automate test case generation, deployment, and infrastructure provisioning, reducing manual effort and ensuring consistency.
3. **Consider Cross-Platform Compatibility:** Serverless applications are deployed across various cloud platforms. Therefore, it is essential to consider cross-platform compatibility when selecting testing frameworks and tools. Supporting multiple serverless platforms ensures broader applicability and flexibility.
4. **Validate Event Flow and Interactions:** Serverless architectures often rely on event-driven interactions. Validating event flow, handling asynchronous events, and verifying interactions between components are critical for ensuring the correctness and reliability of serverless applications.
5. **Adopt Distributed Tracing and Debugging Techniques:** Distributed tracing techniques and debugging tools provide visibility into the behavior and execu-

tion flow of serverless applications. Analyzing logs, tracing requests, and identifying performance bottlenecks aid in efficient debugging, troubleshooting, and performance optimization.

5.7 Limitations and Future Research

Despite conducting an extensive review and analysis of testing techniques, patterns, anti-patterns, and tools for serverless applications, this study has limitations that must be acknowledged. One significant limitation is the exclusion of gray literature from the research scope. By focusing solely on peer-reviewed academic literature, valuable insights from industry reports and technical white papers might have been overlooked. These sources often provide practical case studies and important technological developments that could have enhanced the study's findings.

Additionally, the implementation of the identified testing methodologies and tools was not feasible within the time constraints of this research. While the theoretical investigation offers a comprehensive examination of the existing literature, empirical data from hands-on experimentation is needed to expand the practical implications that can be drawn from the findings. It would have been beneficial to validate the theoretical concepts through practical application and experimentation.

Moreover, the limited duration of the research suggests that the exhaustive nature of the literature review might have been compromised, potentially leading to the oversight of relevant studies. Moreover, the rapidly evolving landscape of serverless technologies implies that newer developments may have emerged during the study period, which should have been considered.

To address these limitations, future research endeavors could encompass a comprehensive review of gray literature, incorporating insights from industry reports and technical white papers. Additionally, conducting practical experiments to validate the theoretical findings would provide empirical data and enhance the practical implications of the study. This approach would allow for a more thorough comprehension of the current state of the art in testing techniques and tools for serverless applications.

In addition, it would be beneficial to investigate specific cases or application scenarios to get a more in-depth comprehension of the practical challenges and evaluate the efficacy of presented solutions in a real-world setting. As the serverless landscape evolves rapidly, ongoing research is crucial to remaining at the forefront of emerging tools, techniques, and challenges in serverless application testing.

In conclusion, by acknowledging these limitations and proposing avenues for future research, we can elevate the depth and applicability of knowledge in the realm of testing

serverless applications. Expanding the scope, incorporating practical experimentation, and keeping up with the latest developments will contribute to a more comprehensive understanding of this evolving field.

6. CONCLUSIONS

In conclusion, this thesis has provided valuable insights into testing techniques and challenges specific to serverless applications. The research findings shed light on the importance of comprehensive testing practices and address the unique challenges faced in testing serverless architectures.

The evaluation of testing techniques has highlighted the significance of various approaches. Unit and integration testing have emerged as crucial techniques for ensuring the functionality and correctness of serverless functions and validating interactions with external services. Performance and load-testing techniques are essential for assessing scalability, responsiveness, and resource utilization in serverless applications. These techniques enable developers to optimize their applications for varying workloads and ensure efficient resource allocation. Additionally, techniques such as canary release and A/B testing provide risk mitigation strategies by gradually deploying changes and monitoring their impact, minimizing negative effects on users.

Testing serverless applications presents unique challenges. The distributed nature of serverless computing makes it difficult to replicate the entire application locally and validate event flow and interactions between services. The lack of control over the underlying infrastructure and runtime environment further complicates the reproduction and debugging of issues. Ensuring data consistency in a stateless environment, managing dependencies between serverless functions, handling complex event flows, and addressing security concerns are additional challenges that require specialized approaches and techniques.

The evaluation of testing tools has revealed a range of options catering to different testing requirements. Apache JMeter stands out as a versatile tool for performance testing, offering extensive capabilities for load testing, stress testing, and performance evaluation. Infrastructure and deployment tools such as AWS CloudFormation simplify the management of serverless resources, while monitoring and observability tools like AWS CloudWatch, Grafana, and Prometheus facilitate the collection, tracking, and analysis of metrics. Distributed tracing and debugging tools, such as AWS X-Ray, provide insights into the request flow and dependencies within serverless architectures. Language-specific testing tools, test case generation tools, and cloud service-specific testing tools offer further support for testing and validation in specific contexts.

The identified testing techniques and tools contribute to the overall goal of ensuring the quality and reliability of serverless applications. By adopting a comprehensive testing

approach, leveraging appropriate tools, and addressing the unique challenges, developers can enhance the performance, scalability, and functionality of their serverless deployments.

However, it is important to acknowledge the limitations of the research. The availability of literature on certain serverless platforms, such as OpenLambda, OpenFaaS, and Fn, was limited due to their relative novelty. Future research should explore these platforms and their specific testing practices to enrich the understanding of testing in serverless computing.

In conclusion, this thesis has provided valuable insights into testing techniques, challenges, and tools for serverless applications. By adopting appropriate testing techniques, leveraging tools specific to the serverless environment, and addressing the unique challenges, developers can ensure the quality, reliability, and performance of their serverless deployments. Future research should continue to explore advanced testing techniques, tools, and frameworks and foster collaboration between researchers, developers, and cloud providers to establish best practices and standards in the field of serverless testing. With ongoing research and advancements, serverless computing can realize its full potential as a robust and scalable paradigm for cloud-based applications.

REFERENCES

[1] A. Pérez, G. Moltó, M. Caballer, and A. Calatrava, “A programming model and middleware for high throughput serverless computing applications,” ACM, Apr. 2019, pp. 106–113. [Online]. Available: https://dl.acm.org/doi/10.1145/3297280.3297292

[2] J. Manner, S. Kolb, and G. Wirtz, “Troubleshooting Serverless functions: a combined monitoring and debugging approach,” *SICS Softw.-Intensive Cyber-Phys. Syst.*, no. 2–3, pp. 99–104, Jun. 2019.

[3] I. Baldini *et al.*, “Serverless Computing: Current Trends and Open Problems.” arXiv, Jun. 10, 2017. [Online]. Available: https://doi.org/10.1007/978-981-10-5026-8_1

[4] G. Adzic and R. Chatley, “Serverless computing: economic and architectural impact,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ACM, Aug. 2017, pp. 884–889. [Online]. Available: https://dl.acm.org/doi/10.1145/3106237.3117767

[5] V. Ivanov and K. Smolander, “Implementation of a DevOps Pipeline for Serverless Applications,” M. Kuhrmann, K. Schneider, D. Pfahl, S. Amasaki, M. Ciolkowski, R. Hebig, P. Tell, J. Klünder, and S. Küpper, Eds., Springer International Publishing, 2018, pp. 48–64. [Online]. Available: http://link.springer.com/10.1007/978-3-030-03673-7_4

[6] Ö. Sedefoğlu and H. Sözer, “Cost Minimization for Deploying Serverless Functions,” in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, Association for Computing Machinery, 2021, pp. 83–85. [Online]. Available: https://doi.org/10.1145/3412841.3442069

[7] S. Winzinger and G. Wirtz, “Model-Based Analysis of Serverless Applications,” in *2019 IEEE/ACM 11th International Workshop on Modelling in Software Engineering (MiSE)*, IEEE, May 2019, pp. 82–88. [Online]. Available: https://ieeexplore.ieee.org/document/8877078/

[8] A. Kaplunovich, “ToLambda--Automatic Path to Serverless Architectures,” in *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, IEEE, May 2019, pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/document/8844428/

[9] A. Byrne, S. Nadgowda, and A. K. Coskun, “ACE: Just-in-time Serverless Software Component Discovery Through Approximate Concrete Execution,” ACM, Dec. 2020, pp. 37–42. [Online]. Available: https://dl.acm.org/doi/10.1145/3429880.3430098

[10] A. Mahgoub, K. Shankar, S. Mitra, and A. Klimovic, “SONIC: Application-aware Data Passing for Chained Serverless Applications,” *USENIX Assoc.*, pp. 285--301, 2021.

[11] W.-T. Lin *et al.*, “Tracking Causal Order in AWS Lambda Applications,” in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, IEEE, Apr. 2018, pp. 50–60. [Online]. Available: https://ieeexplore.ieee.org/document/8360312/

[12] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, “On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures,” in *Proceedings of the 20th International Middleware Conference*, ACM, Dec. 2019, pp. 41–54. [Online]. Available: https://dl.acm.org/doi/10.1145/3361525.3361535

[13] M. Pawlik, P. Banach, and M. Malawski, “Adaptation of Workflow Application Scheduling Algorithm to Serverless Infrastructure,” U. Schwardmann, C. Boehme, D. B. Heras, V. Cardellini, E. Jeannot, A. Salis, C. Schifanella, R. R. Manumachu, D. Schwamborn, L. Ricci, O. Sangyoon, T. Gruber, L. Antonelli, and S. L. Scott, Eds., in Lecture Notes in Computer Science. Springer International Publishing, 2020, pp. 345–356. doi: 10.1007/978-3-030-48340-1_27.

[14] B. Balis, M. Orzechowski, K. Pawlik, M. Pawlik, and M. Malawski, “Cloud Infrastructure Automation for Scientific Workflows,” R. Wyrzykowski, E. Deelman, J. Dongarra, and K. Karczewski, Eds., Springer International Publishing, 2020, pp. 287–297. [Online]. Available: http://link.springer.com/10.1007/978-3-030-43229-4_25

[15] P. Maissen, P. Felber, P. Kropf, and V. Schiavoni, “FaaSdom: a benchmark suite for serverless computing,” in *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*, ACM, Jul. 2020, pp. 73–84. [Online]. Available: https://dl.acm.org/doi/10.1145/3401025.3401738

[16] H. Govind and H. GonzaleznVelez, “Benchmarking Serverless Workloads on Kubernetes,” IEEE, May 2021, pp. 704–712. [Online]. Available: https://ieeexplore.ieee.org/document/9499690/

[17] W.-T. Lin, C. Krintz, and R. Wolski, “Tracing Function Dependencies across Clouds,” in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, IEEE, Jul. 2018, pp. 253–260. [Online]. Available: https://ieeexplore.ieee.org/document/8457807/

[18] K. S.-P. Chang and S. J. Fink, “Visualizing serverless cloud application logs for program understanding,” in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE, Oct. 2017, pp. 261–265. [Online]. Available: http://ieeexplore.ieee.org/document/8103476/

[19] I. Baldini *et al.*, “Cloud-native, event-based programming for mobile applications,” in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, ACM, May 2016, pp. 287–288. [Online]. Available: https://dl.acm.org/doi/10.1145/2897073.2897713

[20] T. Yu *et al.*, “Characterizing serverless platforms with serverlessbench,” in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ACM, Oct. 2020, pp. 30–44. [Online]. Available: https://dl.acm.org/doi/10.1145/3419111.3421280

[21] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Serverless Computation with OpenLambda,” *USENIX Assoc.*, pp. 33–39, 2016.

[22] N. S. Andersen, M. Chiarandini, and J. Mauro, “Wandering and getting lost: the architecture of an app activating local communities on dementia issues,” in *2021 IEEE/ACM 3rd International Workshop on Software Engineering for Healthcare (SEH)*, IEEE, Jun. 2021, pp. 36–43. [Online]. Available: https://ieeexplore.ieee.org/document/9470894/

[23] K. Kritikos and P. Skrzypek, “A Review of Serverless Frameworks,” IEEE, Dec. 2018, pp. 161–168. [Online]. Available: https://ieeexplore.ieee.org/document/8605774/

[24] V. Lenarduzzi and A. Panichella, “Serverless Testing: Tool Vendors’ and Experts’ Points of View,” *IEEE Softw.*, vol. 38, pp. 54–60, Jan. 2021.

[25] V. Lenarduzzi, J. Daly, A. Martini, S. Panichella, and D. A. Tamburri, “Toward a Technical Debt Conceptualization for Serverless Computing,” *IEEE Softw.*, pp. 40–47, Jan. 2021.

[26] R. Cordingly, W. Shu, and W. J. Lloyd, “Predicting Performance and Cost of Serverless Computing Functions with SAAF,” in *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, IEEE, Aug. 2020, pp. 640–649. [Online]. Available: https://ieeexplore.ieee.org/document/9251165/

[27] L. F. A. Jr and F. S. Ferraz, “Function-as-a-Service X Platform-as-a-Service: Towards a Comparative Study on FaaS and PaaS,” *ICSEA 2017 Twelfth Int. Conf. Softw. Eng. Adv.*, 2017.

[28] D. Khatri, S. K. Khatri, and D. Mishra, “Potential Bottleneck and Measuring Performance of Serverless Computing: A Literature Study,” in *2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, IEEE, Jun. 2020, pp. 161–164. [Online]. Available: https://ieeexplore.ieee.org/document/9197837/

[29] C.-F. Fan, A. Jindal, and M. Gerndt, “Microservices vs Serverless: A Performance Comparison on a Cloud-native Web Application:,” in *Proceedings of the 10th International Conference on Cloud Computing and Services Science*, SCITEPRESS - Science and Technology Publications, 2020, pp. 204–215. [Online]. Available: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0009792702040215

[30] G. McGrath and P. R. Brenner, “Serverless Computing: Design, Implementation, and Performance,” in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, IEEE, Jun. 2017, pp. 405–410. [Online]. Available: http://ieeexplore.ieee.org/document/7979855/

[31] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, “Peeking Behind the Curtains of Serverless Platforms,” *USENIX Assoc.*, pp. 133–145, 2018.

[32] Z. Xu, H. Zhang, X. Geng, Q. Wu, and H. Ma, “Adaptive Function Launching Acceleration in Serverless Computing Platforms,” IEEE, Dec. 2019, pp. 9–16. [Online]. Available: https://ieeexplore.ieee.org/document/8975850/

[33] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, V. Sukhomlinov, and N. Nayak, “Agile Cold Starts for Scalable Serverless,” *USENIX Assoc.*, p. 21, 2019.

[34] T. Asghar, S. Rasool, M. Iqbal, Z. U. Qayyum, A. N. Mian, and G. Ubakanma, “Feasibility of Serverless Cloud Services for Disaster Management Information Systems,” in *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, IEEE, Jun. 2018, pp. 1054–1057. [Online]. Available: https://ieeexplore.ieee.org/document/8622913/

[35] D. Khatri, S. K. Khatri, and D. Mishra, “Performace Testing Approach for Enterprise Application comprising Serverless Component,” in *2021 International Conference on Intelligent*

Technologies (CONIT), Hubli, India: IEEE, Jun. 2021, pp. 1–4. doi: 10.1109/CONIT51480.2021.9498446.

[36] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, “Serverless Computing: An Investigation of Factors Influencing Microservice Performance,” in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, IEEE, Apr. 2018, pp. 159–169. [Online]. Available: https://ieeexplore.ieee.org/document/8360324/

[37] D. Bardsley, L. Ryan, and J. Howard, “Serverless Performance and Optimization Strategies,” in *2018 IEEE International Conference on Smart Cloud (SmartCloud)*, IEEE, Sep. 2018, pp. 19–26. [Online]. Available: https://ieeexplore.ieee.org/document/8513710/

[38] D. Jackson and G. Clynch, “An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions,” IEEE, Dec. 2018, pp. 154–160. [Online]. Available: https://ieeexplore.ieee.org/document/8605773/

[39] 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*, ACM, Jun. 2020, pp. 85–88. [Online]. Available: https://dl.acm.org/doi/10.1145/3377812.3382135

[40] I. E. Akkus *et al.*, “SAND: Towards High-Performance Serverless Computing,” *USENIX Assoc.*, pp. 923–935, 2018.

[41] E. Oakes *et al.*, “SOCK: Rapid Task Provisioning with Serverless-Optimized Containers,” *USENIX Assoc.*, pp. 57–69, 2018.

[42] P. Leitner, E. Wittern, J. Spillner, and W. Hummer, “A mixed-method empirical study of Function-as-a-Service software development in industrial practice,” *J. Syst. Softw.*, vol. 149, pp. 340–359, Mar. 2019.

[43] S. Winzinger, “Towards coverage criteria for serverless applications,” 2019.

[44] A. H. A. Kamal, C. C. Y. Yen, and G. J. Hui, “Risk Assessment, Threat Modeling and Security Testing in SDLC,” 2020, [Online]. Available: https://doi.org/10.48550/arXiv.2012.07226

[45] A. Verma, Department of Computer Science, Amity University, Gurgaon, India, A. Khatana, Department of Computer Science, Amity University, Gurgaon, India, S. Chaudhary, and Department of Computer Science, Amity University, Gurgaon, India, “A Comparative Study of Black Box Testing and White Box Testing,” *Int. J. Comput. Sci. Eng.*, vol. 5, no. 12, pp. 301–304, Dec. 2017.

[46] M. Polo, P. Reales, M. Piattini, and C. Ebert, “Test Automation,” *IEEE Softw.*, pp. 84–89, Jan. 2013.

[47] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli, “To Mock or Not to Mock? An Empirical Study on Mocking Practices,” in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, IEEE, May 2017, pp. 402–412. [Online]. Available: http://ieeexplore.ieee.org/document/7962389/

[48] S. Winzinger and G. Wirtz, “Applicability of Coverage Criteria for Serverless Applications,” IEEE, Aug. 2020, pp. 49–56. [Online]. Available: https://ieeexplore.ieee.org/document/9183589/

[49] A. Bergmayr, M. Wimmer, G. Kappel, and M. Grossniklaus, “Cloud Modeling Languages by Example,” in *2014 IEEE 7th International Conference on Service-Oriented Computing and Applications*, IEEE, Nov. 2014, pp. 137–146. [Online]. Available: http://ieeexplore.ieee.org/document/6978602/

[50] S.-P. Ma, C.-Y. Fan, Y. Chuang, W.-T. Lee, S.-J. Lee, and N.-L. Hsueh, “Using Service Dependency Graph to Analyze and Test Microservices,” in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, IEEE, Jul. 2018, pp. 81–86. [Online]. Available: https://ieeexplore.ieee.org/document/8377834/

[51] M. Villamizar *et al.*, “Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures,” in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Cartagena, Colombia: IEEE, May 2016, pp. 179–182. [Online]. Available: http://ieeexplore.ieee.org/document/7515686/

[52] C. Cicconetti, M. Conti, and A. Passarella, “An Architectural Framework for Serverless Edge Computing: Design and Emulation Tools,” IEEE, 2018, pp. 48–55. [Online]. Available: https://ieeexplore.ieee.org/document/8590993/

[53] J. Koch and W. Hao, “An Empirical Study in Edge Computing Using AWS,” IEEE, Jan. 2021, pp. 0542–0549. [Online]. Available: https://ieeexplore.ieee.org/document/9376039/

[54] S. Fouladi, F. Romero, D. Iter, Q. Li, and S. Chatterjee, “From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers,” *USENIX Assoc.*, pp. 475–488, 2019.
[55] L. Rebouças De Carvalho and A. F. De Araujo, “Remote Procedure Call Approach using the Node2FaaS Framework with Terraform for Function as a Service:,” in *Proceedings of the 10th International Conference on Cloud Computing and Services Science*, SCITEPRESS - Science and Technology Publications, 2020, pp. 312–319. [Online]. Available: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0009381503120319
[56] H. Martins, F. Araujo, and P. R. Da Cunha, “Benchmarking Serverless Computing Platforms,” *J. Grid Comput.*, vol. 18, no. 4, pp. 691–709, Dec. 2020, doi: 10.1007/s10723-020-09523-1.
[57] J. Cohen, “A Coefficient of Agreement for Nominal Scales,” *Educ. Psychol. Meas.*, vol. 20, no. 1, pp. 37–46, Apr. 1960.
[58] M. Anand, J. Zhang, S. Ding, J. Xin, and J. Lin, “Serverless BM25 Search and BERT Reranking,” *CEUR-WSorg*, pp. 3--9, 2021.