# Integration Testing Criteria for Serverless Applications


Dissertation
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
der Fakultät
Wirtschaftsinformatik und Angewandte Informatik
der Otto-Friedrich-Universität Bamberg

vorgelegt von
Stefan Winzinger

Bamberg 2025

Diese Arbeit hat der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg als Dissertation vorgelegen.

1. Gutachter: Prof. Dr. Guido Wirtz
2. Gutachter: Prof. Dr. Andreas Henrich

Tag der mündlichen Prüfung: 20.12.2024




URN: urn:nbn:de:bvb:473-irb-1055875

DOI: https://doi.org/10.20378/irb-105587

# Kurzfassung

Die Entwicklung von Systemen, die nativ in der Cloud ausgeführt werden, nimmt stetig zu. Insbesondere wird hierbei häufig auf die Nutzung von Serverless Computing zurückgegriffen. Serverless functions ermöglichen hier durch ihre Zustandslosigkeit, das System effektiv zu skalieren. Die verwendeten Funktionen sind gut isoliert testbar. Allerdings wird durch die Integration der einzelnen Komponenten ein Problemraum aufgespannt, den es zu testen gilt und dessen Komplexität sich zudem durch ein stetig änderndes Umfeld schnell verändern kann.
Daher wird in dieser Arbeit untersucht, wie man das Integrationstesten eines solchen Systems unterstützen kann. Hierzu wird zuerst untersucht, wie man mithilfe eines Modells die relevanten Charakteristiken erfassen kann, um mit diesem das System zu analysieren und Testfälle zu erstellen. Des Weiteren werden mehrere potenzielle Überdeckungskriterien basierend auf dem Kontroll- und Datenfluss der Komponenten untereinander vorgestellt. Es wird gezeigt, wie diese Kriterien in serverlosen Anwendungen implementiert werden können und mithilfe von Modellen automatisiert Testfälle erzeugt werden können. Schließlich werden die Überdeckungskriterien unter Anwendung von Mutationstests anhand verschiedener Applikationen evaluiert.

# Abstract

The development of systems executed natively in the cloud is steadily increasing. In particular, serverless computing is used more often. Because of their statelessness, serverless functions enable an effective scaling of the system. The utilized functions are well-testable in isolation. However, integrating individual components creates a problem space that needs to be tested. Additionally, its complexity can rapidly change caused by a constantly evolving environment. Therefore, this work investigates how integration testing of such systems can be supported. Firstly, it is investigated how the relevant characteristics can be modeled to analyze the system and create test cases with them. Additionally, several potential coverage criteria based on the control and data flow between components are introduced. This work also demonstrates how these criteria can be implemented in serverless applications and how test cases can be automatically generated using models. Finally, the coverage criteria are evaluated using mutation testing with various applications.

# Contents

# List of Figures

# List of Tables

# List of Listings

# List of Definitions

# List of Abbreviations

**AWS** Amazon Web Services

**CRUD** create, read, update, and delete

**NIST** National Institute of Standards and Technology

**SaaS** Software as a Service

**PaaS** Platform as a Service

**IaaS** Infrastructure as a Service

**FaaS** Function as a Service

**BaaS** Backend as a Service

**MBaaS** Mobile Backend as a Service

**API** Application Programming Interface

# Part I.

# Background and Problem Identification

# 1. Introduction

This chapter provides an introduction to this work by motivating it and giving an outline. The outline contains both research questions tackled in this work and an overview of the contributions of this work. All links referenced in this work were last accessed on August 19th, 2024

## 1.1. Motivation

The concept of cloud computing where computing resources are shared over a network can be traced back to 1961 and evolved over the years by providing various service models like *Infrastructure as a Service (IaaS)*, *Platform as a Service* (PaaS), and *Software as a Service* (SaaS) [100]. Another evolvement was the introduction of Amazon's *AWS Lambda* in 2014, which was the start of the popularity of serverless computing. However, the support for cloud testing, in particular integration testing, was not good at this time [36], and it got not better with the introduction of serverless computing. Since serverless applications run mostly in a non-local environment, testing the correct integration of the components is more difficult [86]. In serverless computing, serverless functions which should be stateless to enable their scalability are connected with other serverless functions and resources provided by the cloud platform provider. Such a combination of components can become quite complex. While the complexity of testing single components in isolation can be reduced by using mocks and stubs, there is a lack of support for integration testing in serverless applications which should be improved by this work.

Lenarduzzi et al. interviewed experts in the domain of serverless computing in [59]. They identified integration testing as a crucial problem in serverless computing. According to these experts, integration tests are often created in a way that does not test the integration completely or are not written at all because they are hard to accomplish. In particular, missing tool support for integration testing is identified as a reason for missing integration tests by the experts. In an interview among experts on serverless computing in [58], they identified *measuring coverage* in serverless applications as a challenge, and the need for coverage criteria for serverless applications. These experts expect that coverage criteria on the integration level will be defined and applied in the future.

Kritikos and Skrzypek reviewed different serverless frameworks in [52]. Support for unit testing for these frameworks could be identified but no support for integration testing at all.

Therefore, this work tries to reduce the lack of support for integration testing in serverless applications. This is done by providing a model for the representation of serverless applications that helps to create integration test cases based on coverage criteria focusing on the integration of serverless applications. Furthermore, tool support is given for the measurement of the coverage and the automatic creation of test cases.

## 1.2. Outline

This section provides an outline of the structure of the thesis. The thesis is divided into four parts. The first part motivates the work done in this thesis, gives an outline of the thesis, and also provides some general background knowledge for the context of the thesis.

The second part is the main part of the thesis which is mainly based on papers published by the author of this work. The research questions addressed in the second part are introduced in Section 1.2.1 where also the research methodology is described. The contributions of the work including the tools developed and the corresponding chapters are described in Section 1.2.2.

The third part provides an outlook on future work and summarizes the work. Finally, in the fourth part, the models of applications that were used are attached.

### 1.2.1. Research Questions

Each chapter of the second part addresses at least one research question. The later chapters depend on the previous ones. The relationship of the chapters to the previous one is described in this section and shown in Figure 1.1, as well as the methodology used.

**Modeling** In order to have a general abstraction of the system which represents the relevant aspects, the first step of the thesis consists of the representation of serverless applications. Since there were no models available yet that capture the characteristics of serverless applications, the first research question of Chapter 3 is as follows:

> **Research Question 1.1:** How can a serverless application be modeled to capture the relevant aspects needed for integration testing?

Several factors are described which can be represented by the model and the purpose of the modeled factors are evaluated. Since a model is an abstraction

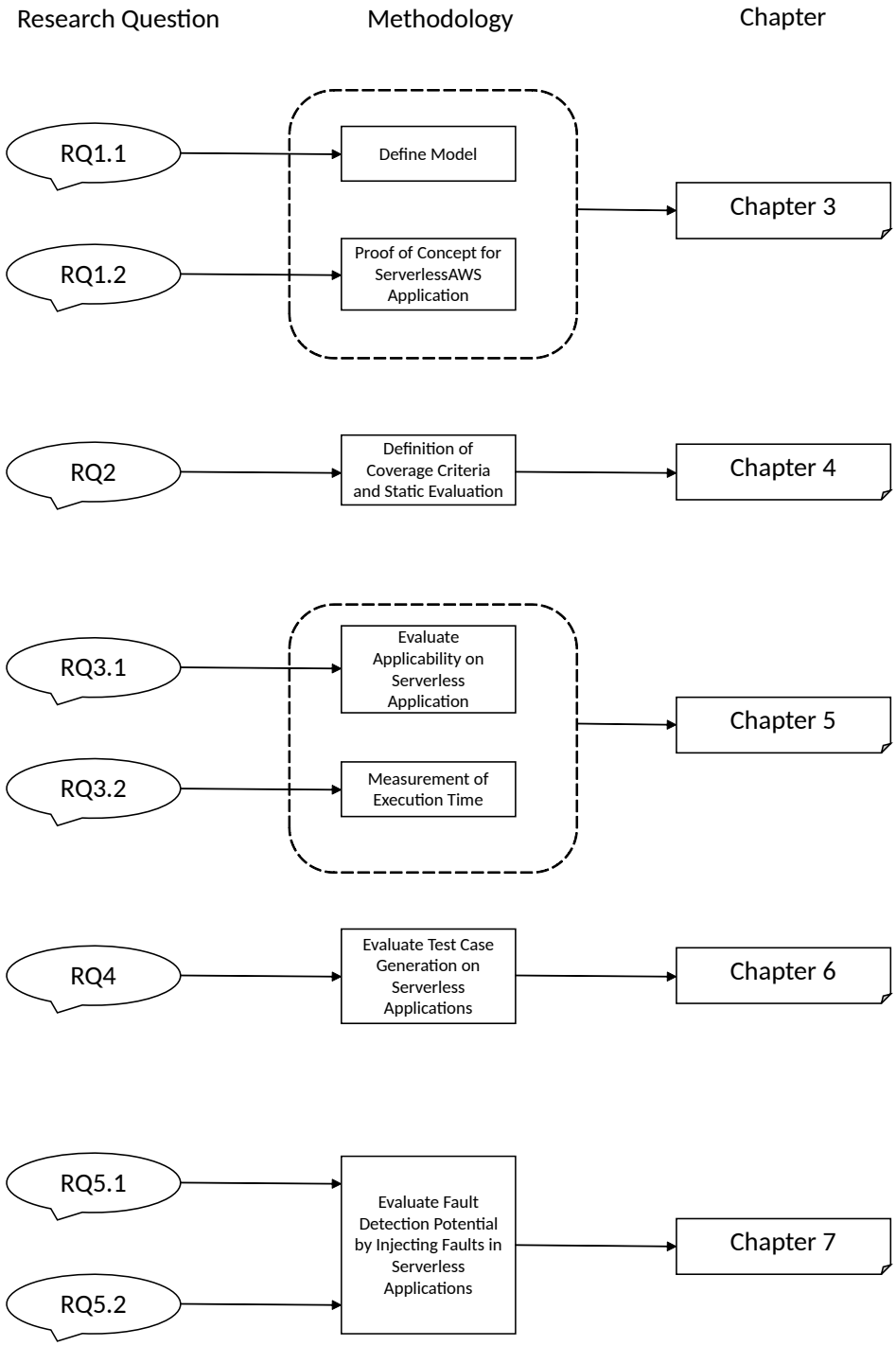



Figure 1.1.: Research questions and methodology

of the actual application, only the relevant factors should be in the model that are relevant for supporting integration testing.

Furthermore, an automatic generation of a compact model of a serverless application can support the acceptance of models. Therefore, the following question is asked:

**Research Question 1.2:** How can a model representing a serverless application be created automatically based on existing code?

By elaborating how the components of a serverless application can be modeled, a tool is created that implements the approach to show its applicability.

**Coverage Criteria** Coverage criteria help to evaluate the quality of test suites, e.g., to estimate if a test suite has enough test cases or if additional test cases are needed to trigger other scenarios where potential errors could be provoked. Therefore, based on the model, coverage criteria are defined in Chapter 4 to answer the following research question:

**Research Question 2:** What are potential coverage criteria for integration testing that can capture the interaction between the components of serverless applications?

Different coverage criteria are defined considering different aspects of a serverless application. By selecting several applications from *GitHub*, the potential of the criteria is statically analyzed.

**Measurement of Coverage Criteria** Since coverage criteria have to be measurable if they are applied to applications, in Chapter 5, the following question is answered first:

**Research Question 3.1:** How can the coverage of testing targets based on coverage criteria be measured in serverless applications?

A general approach for the measurement of the criteria is shown which is finally implemented for some scenarios on *Amazon Web Services (AWS)* to answer the following research question:

**Research Question 3.2:** How does the measurement influence the performance?

**Automatic Test Case Generation** The measurement of the coverage of the coverage criteria used is needed for the creation of automatic test cases. Otherwise, it could not be measured if a test case covers a testing target. The possibility to create test cases automatically is helpful if many test cases are needed like in load testing. In particular, for regression testing where a variety of scenarios has to be tested to rule out that side effects were introduced by a code change, automatically generated test cases based on coverage criteria are helpful. Therefore, the following research question is answered in Chapter 6:

**Research Question 4:** How can test cases be automatically generated to cover testing targets based on coverage criteria?

An approach for the automatic generation of test cases is introduced which is finally implemented and evaluated on two practical serverless applications.

**Evaluation of Criteria** Even if the defined coverage criteria sound reasonable, it is not clear if they detect faults at all. Therefore, test cases based on these criteria are compared to randomly generated test cases. Based on the previous chapters which introduced means to measure the coverage and create test cases automatically, a comparison can be made in Chapter 7 which answers the following research question:

**Research Question 5.1:** Do test suites based on the coverage criteria detect more injected faults than test suites that were randomly built?

By applying mutation operators on three applications, faults are injected into the applications. Test suites built based on coverage criteria and randomly created test suites are run against these mutants to evaluate which test suites perform better. Furthermore, the test suites are compared with each other, answering the following research question:

**Research Question 5.2:** Which coverage criterion reveals the most injected faults?

### 1.2.2. Contributions

The contribution of this thesis is mainly based on the following papers:

- [107] S. Winzinger, "Towards coverage criteria for serverless applications," in *11th Central European Workshop on Services and their Composition (ZEUS)*, 2019
- [108] 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, 2019
- [109] S. Winzinger and G. Wirtz, "Coverage criteria for integration testing of serverless applications," in *13th Symposium and Summer School On Service-Oriented Computing*, 2019
- [110] S. Winzinger and G. Wirtz, "Applicability of coverage criteria for serverless applications," in *2020 IEEE International Conference on Service Oriented Systems Engineering (SOSE)*. IEEE, 2020

- [111] S. Winzinger and G. Wirtz, "Data flow testing of serverless functions," in *Proceedings of the 11th International Conference on Cloud Computing and Services Science- CLOSER*, INSTICC. SciTePress, 2021, pp. 5664
- [112] S. Winzinger and G. Wirtz, "Automatic test case generation for serverless applications," in *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2022
- [113] S. Winzinger and G. Wirtz, "Comparison of integration coverage criteria for serverless applications," in *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2023

The author is the first author of all of these papers and the main contributor. Furthermore, in the context of a master thesis, the following paper was published which is also used in Chapter 5:

- [27] C. Eder, S. Winzinger, and R. Lichtenthäler, "A comparison of distributed tracing tools in serverless applications," in *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2023

The main contribution of the work are coverage criteria for integration testing of serverless applications whose efficiency was shown. Besides that, a model for the visualization of the serverless application was developed, and a model representing the influential factors of a serverless function.

Different tools were developed in the context of this work. For the search of applications, a search tool was developed that searches *GitHub* for serverless applications. The first version of this tool[1] was applied in Chapter 4. Another refactored and updated version of this tool[2], which also collects the number of serverless functions and the programming language used in the serverless applications identified, is utilized in Chapter 5 and Chapter 6.

A tool[3] for the visualization and creation of a model based on an existing serverless application was built in the context of Chapter 3. For the measurement of the coverage criteria, this tool was extended in Chapter 5. Finally, the tooling[4] of Chapter 6 and Chapter 7 uses and adapts the components of the previous chapters. Each tool needs components of the tooling which was developed earlier. The components developed and their relationships are visualized in Figure 1.2. In Chapter 3, components are developed for the visualization of the model and the supported creation of a model by using an existing application. These components are used in Chapter 5 where in particular the model is needed to assign the corresponding instrumentations for the measurement of the coverage to the source code. Furthermore, the measurement of the

[1] `https://github.com/snwinz/ServerlessApplicationIdentification`
[2] `https://github.com/snwinz/ServerlessApplicationSearcher`
[3] `https://github.com/snwinz/ServerlessApplicationTool`
[4] `https://github.com/snwinz/ServerlessCoverageTesting`

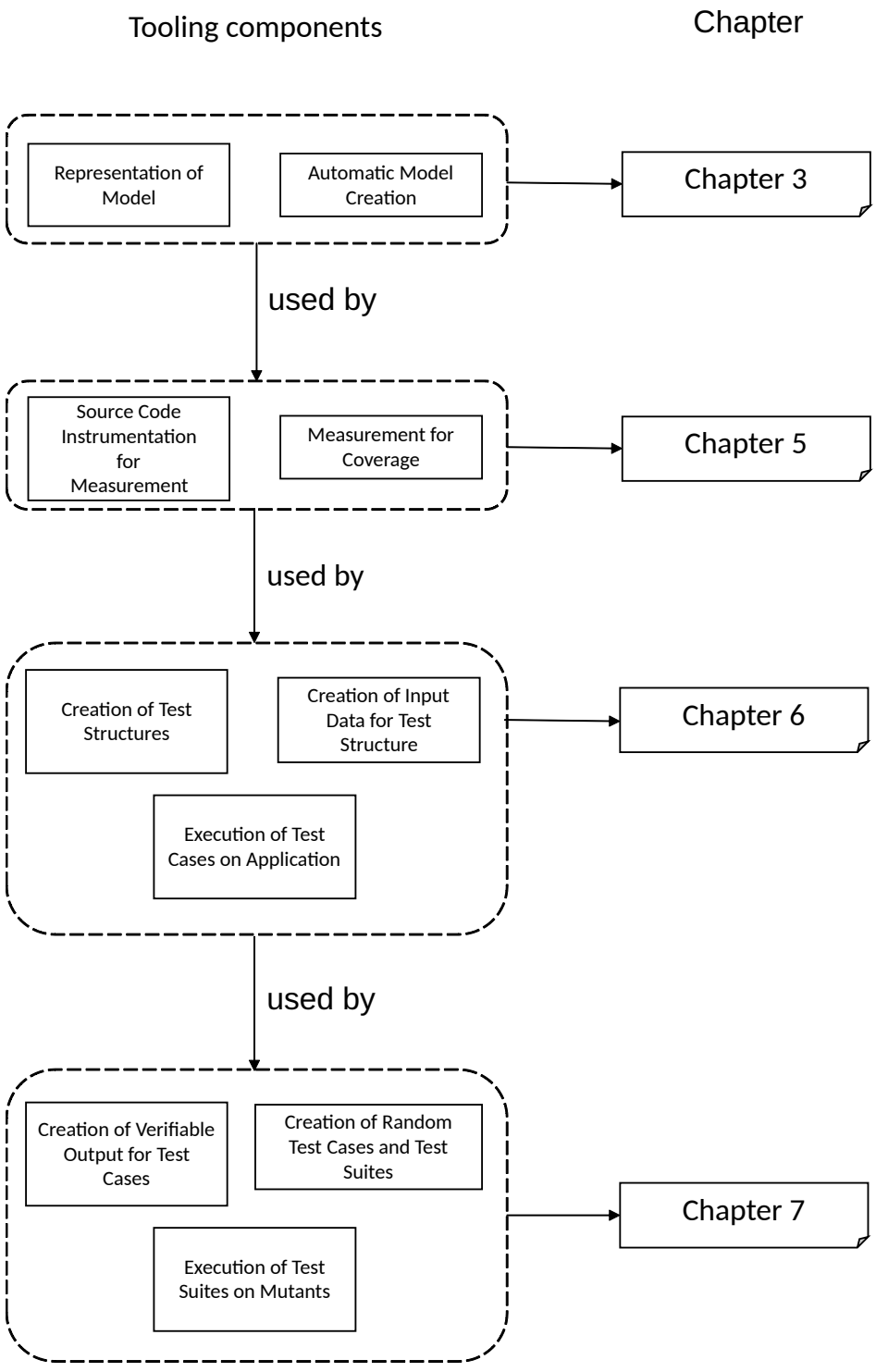



Figure 1.2.: Tooling components and their dependencies

coverage of the coverage criteria is implemented which is in particular needed for the following Chapter 6 where test cases are created automatically. Here, additional components are added that create test case structures based on the model and create input data for these test case structures. Additionally, the test cases are applied to serverless applications which are instrumented to measure if the test case fulfills its intended testing target.

Finally, since the last chapter evaluates the potential of the testing targets, test cases produced automatically in the previous chapter are used for comparison to randomly created test cases and test suites. Therefore, the tool supports the generation of random test cases and test suites. Since the test cases need to be checked for their fault detection potential, their correct execution has to be checked. Therefore, a component was added that identifies the intended output of the serverless functions invoked within a test case by removing the dynamic parts. Additionally, a component was required that executes all test cases on different mutants where faults were injected in order to check their fault detection potential.

So, the tools of the later chapters require the toolset of the previous chapters. Furthermore, the toolset shows how the development process can be supported by modeling the applications first. Based on these models, testing targets can be created. The tooling supports the creation of test cases focusing on these testing targets by providing serverless functions with input data and specifying the data that needs to be verified after their execution.

An overview of the workflow for the tooling of the chapters is visualized in Figure 1.3.

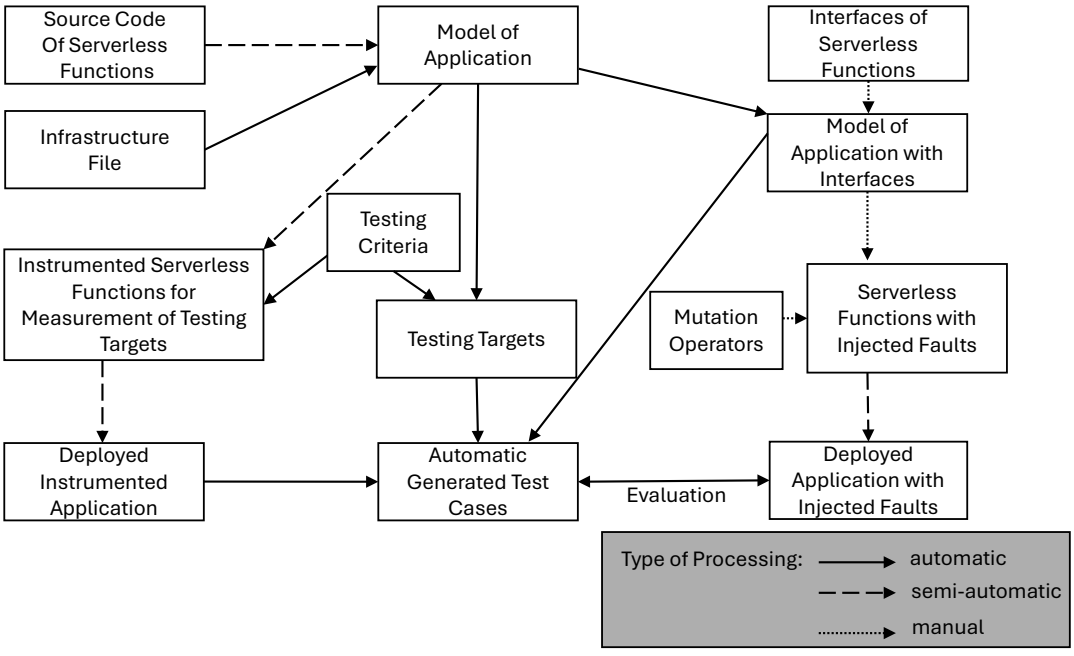



Figure 1.3.: Workflow of tooling

By using the source code of the serverless functions and an infrastructure file which is used for the deployment of the serverless application, a model of the serverless application is created automatically. If a dependency cannot be parsed from the source code, the dependency must be added manually to the model. Based on this model, all testing targets that have to be covered to fulfill the coverage criteria can be created automatically. Additionally, instrumented versions of the serverless functions can be created with tool support which

measure the coverage of the serverless application when deployed. The model needs to be extended with the interfaces of the serverless functions to enable the automatic creation of test cases by using also a deployed serverless application that can measure the coverage (a more concrete visualization is illustrated in Chapter 6 in Figure 6.1). Furthermore, by using mutation operators, the serverless functions must be mutated to inject faults. Based on these faulty serverless functions, serverless applications are created that can be used to evaluate the test cases created automatically.

# 2. Background

This chapter introduces some basic concepts which are relevant to this thesis. First, the concept of *Serverless Computing* is introduced where the terms *serverless functions* and *serverless applications* are described. *Serverless Computing* is compared to traditional cloud computing models and the advantages and drawbacks of *Serverless Computing* are listed.

Furthermore, since this work focuses on integration testing, the phases of testing are described.

## 2.1. Serverless Computing

The term *Serverless Computing* is a computing concept where no precise definition exists yet [69]. It became popular with the introduction of Amazon's *AWS Lambda* in 2014 making the term *Serverless Computing* more popular [9]. Serverless computing can be seen as a misnomer since in contrast to the name, there are still servers used [11]. However, the name should intend that the user writes the code of the functions but leaves the operational tasks to the server [48]. *Function as a Service* (FaaS) is often used as a synonym for *Serverless Computing* [69] which encompasses cloud offerings enabling the execution of custom functions. It can be defined as follows according to [104, 105]:

"**Function-as-a-Service** is a form of serverless computing where the cloud provider manages the resources, lifecycle, and event-driven execution of user-provided functions."

Another service model that is often mentioned in this context is *Backend as a Service* (BaaS) which offers other services that can be combined with FaaS offerings. Typical services provided by *BaaS* are data storage, user management, file storage, or push notifications that are provided on an infrastructure that is automatically scaled and optimized [25]. So,*Serverless Computing* can be seen as a combination of *FaaS* and *BaaS* [48, 94] where *FaaS* offers the possibility to implement logic within user-provided functions and *BaaS* offers the possibility to use other services in its combination, for example, to persist state. According to [48], such a combination can be seen as serverless if it scales automatically without explicit need for provisioning and is billed on its usage. Such combinations can also be named *Serverless Applications* which "combine managed stateless ephemeral compute solutions such as AWS Lambda, Azure

Functions, or Google Cloud Functions (Function-as-a-Service, FaaS), and fully provider-managed services for messaging, file storage, databases, streaming, or authentication (Backend-as-a-Service, BaaS)" [29]. The term *Serverless Application* is used in this work for the naming of such applications whereas the functions that are deployed and called via a *FaaS* offering of a cloud platform provider are named *Serverless Functions* which is a popular naming used in other papers [10, 29, 44, 68, 103].

In [87], Roberts and Chapin suggest five key characteristics for serverless services which are described and summarized in the following list:

- **No need for a long-lived host or application instance:** Other services require that an instance is deployed, run, and monitored which is available for more than one request. In a serverless service, this is not required. For example, a serverless function only requires that its code is deployed, but the management of the execution environment is done by the cloud platform
- **Load-dependend auto-scaling and auto-provisioning:** If the load increases, the serverless service scales automatically without any adjustment. So, when for a serverless function more requests have to be handled than the actual function instances can handle, the platform provider can deploy new function instances without any interaction of the developer. Similarly, if a data storage service needs more capacity, this is provided automatically by the cloud platform.
- **Precise usage-based costs:** The costs of the application are correlated with its usage. This means that a service is paid for only when it is used. If it is not used, it has not to be paid. Similarly, the costs of a data storage service should be closely tied to its usage and not to its capacity.
- **Definition of performance capabilities in terms other than host size/-count:** The performance capabilities that can be defined should be abstracted away from the underlying instance or host type. So, it is enough to specify the capacity a serverless function needs, e.g., RAM, but if the function instances executing the serverless functions are deployed on machines that are either more powerful or weaker than defined, the cloud platform must decide what must happen with the underlying instances.
- **Implicit high availability:** The cloud platform provider should try to continue to process a request even if the underlying component fails. For example, if the execution of a serverless function fails, the cloud platform provider should try to reexecute this function.

The cloud platform provider can scale serverless functions since it can be assumed that they are stateless making it easier to scale the application [2]. The statelessness of the serverless function is an important characteristic of the

serverless functions making them often fine-granular and require the usage of other cloud platform services.

### 2.1.1. Architecture of Serverless Functions

The architecture of a serverless function running on a cloud platform is depicted in Figure 2.1 which is based on [9].

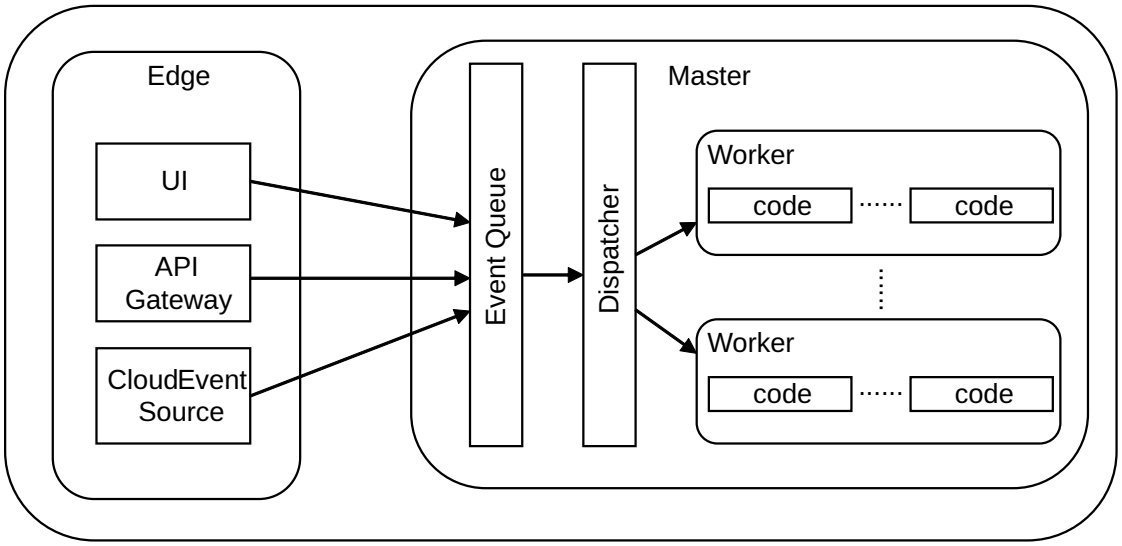



Figure 2.1.: Architecture of serverless functions based on [9]

Events are sent to the master component where they are stored in a queue. These events can be created in different ways. They can be created via a user interface provided by the cloud platform which is usually used during the creation of the application and is not the typical use case. Most serverless functions provide an API Gateway which is an interface that can be invoked, for example via HTTP, and sends finally an event to the platform. This API Gateway can also be configured to do some filtering tasks or mapping tasks of the input data before sending an event. Another event source are other components within the same platform, for example, some *BaaS* offerings, like data storages, where data are persisted. Such components can also create events that have to be processed.

The events received must be dispatched to existing instances of the function or a new instance must be created [9]. Furthermore, the platform must wait for a response and make it available to the user, gather execution logs and stop the function instance when it is not needed anymore [9]. The target of the platform is to start a function and process its input quickly and efficiently while also considering aspects like cost, scalability, and fault tolerance [9]. So, it cannot be foreseen by the user if a function instance is reused when an invocation to a serverless function is made. If the function instance is reused, the state of a previous invocation can be used which is, for example, saved in a global variable. However, since the developer cannot rely on a reuse of a function

instance, the developer has to assume that the serverless function is stateless and uses such things only for caching.

### 2.1.2. Comparison to Traditional Cloud Computing

This section compares *Serverless Computing* with other cloud computing models. The *National Institute of Standards and Technology (NIST)* defines "Cloud Computing" in [74] as follows: "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

Furthermore, the model defined in [74] is composed of five essential characteristics (*on-demand self-service*, *broad network access*, *resource pooling*, *rapid elasticity*, and *measured service*) and four deployment models representing how and where the cloud infrastructure is provisioned (*private*, *community*, *public*, or hybrid cloud).

Additionally, the cloud model in [74] is composed of three service models, namely *Software as a Service*, *Platform as a Service*, and *Infrastructure as a Service*. They describe which capabilities are provided by the cloud. Their definitions are as follows according to [74]:

- *Software as a Service (SaaS)*: "The capability provided to the consumer is to use the providers applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings." [74]

- *Platform as a Service (PaaS)*: "The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment." [74]

- *Infrastructure as a Service (IaaS)*: "The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but

has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls)." [74]

However, caused by the rapid evolution of technologies, terms for new capabilities that can be provided by a platform provider were defined [75]. One of the terms that became popular in the context of *Serverless Computing* was *Function as a Service* (FaaS). A definition of *Serverless Computing* or *Serverless Architecture* is proposed as follows according to [104, 105]:

"**Serverless Computing** is a form of cloud computing which allows users to run event-driven and granularity billed applications, without having to address the operational logic."

This definition overlaps with *PaaS* [105] since here the user does also not control the underlying cloud infrastructure and can run custom applications. The operational tasks of serverless functions are mainly managed by the cloud platform provider. In contrast to *SaaS* where the operational tasks are also managed by the cloud provider, only simple operational tasks like assigned memory or CPU power can optionally be changed [104]. As can be seen in Figure 2.2 taken from [104] the serverless computing services *FaaS* and *BaaS* can be categorized between *PaaS* and *SaaS* when operational logic is considered.

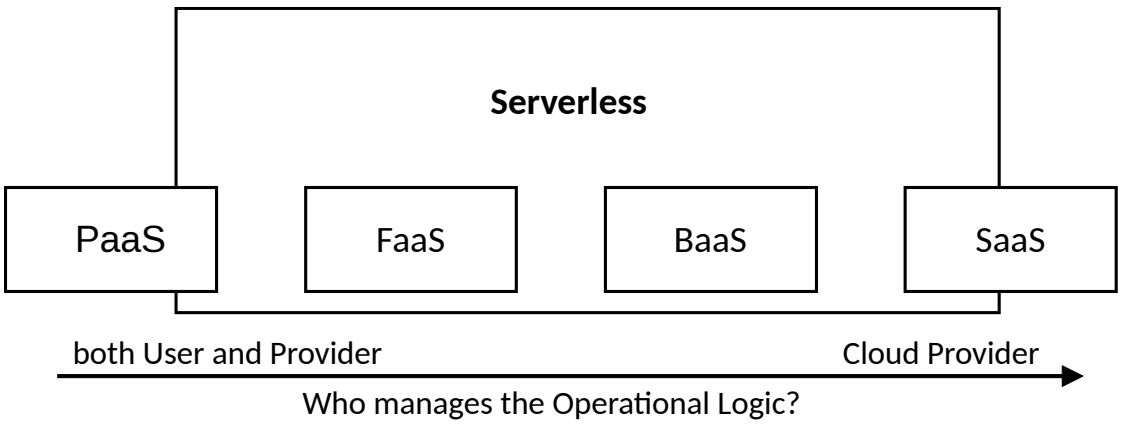



Figure 2.2.: Management of operational logic based on [104]

Another categorization is given in [9] (compare Figure 2.3) where the x-axis describes the time-to-live and the y-axis the ease of scaling. Here, *FaaS*, *BaaS*, which is sometimes also referred to as Mobile Backend as a Service (MBaaS) [77], and *SaaS* show serverless properties regarding the properties of the x- and y-axis, while *PaaS* is referred to be both server-less and server-aware. Even if there are some intersections with other cloud computing models, serverless computing is based on serverless functions which have to be stateless to enable scalability [2] which is a clear distinction from other cloud computing models. This influences the architecture of serverless applications which is

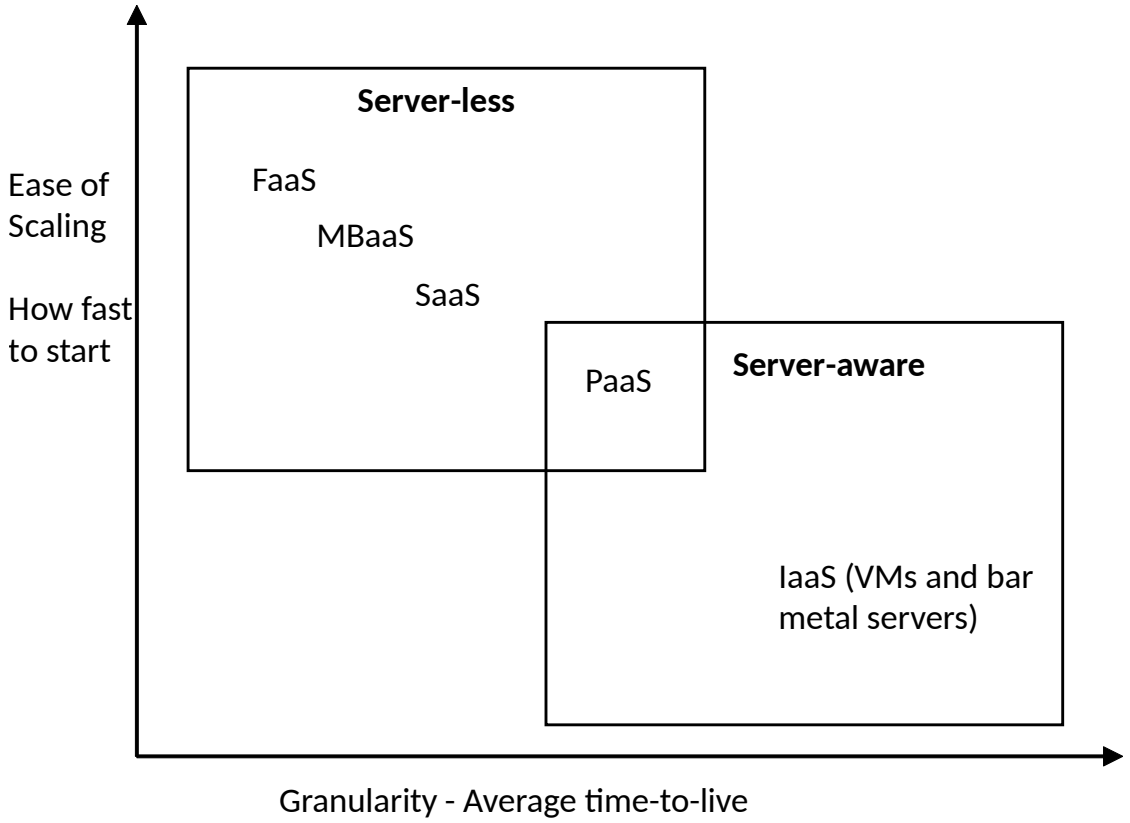



Figure 2.3.: Server-less vs server-aware based on [9]

based on these stateless serverless functions which are integrated with other cloud platform-specific services.

## 2.1.3. Advantages and Drawbacks

The usage of serverless applications has some benefits and drawbacks which are listed in this section.

### 2.1.3.1. Advantages of Serverless Computing

The scalability provided by serverless applications is a huge benefit. This enables the development of scalable applications that can potentially be used by many users without investing much money in the infrastructure before. Besides the ability to scale in serverless applications, obvious benefits result from the infrastructure of serverless applications which is abstracted away. So, developers can spend more time focusing on business features than on the code managing the deployment of their serverless application which is generally considered to be simpler than for other types of cloud service [57].

Furthermore, the costs when running the final application can be reduced since serverless applications have only to be paid when actually used as was shown in some case studies by Adzic and Chatley in [1]. However, this depends on the use case. While a serverless application might cost less for applications

with an infrequent but bursty workload [9] compared to other cloud computing models, an application with a constant workload [78] or extensive input and output operations might be more expensive [9, 40].

Other benefits result from the architectural style of serverless computing which is mainly based on serverless functions. These functions are designed to be stateless to enable scalability. This results in the following benefits identified in [105] where particularly the first and last aspects influence the costs of a running system directly:

- **Improved resource management:** Traditionally, the developer of the application has to decide how many resources are assigned to the application. The classical resource types that are deployed are virtual machines or containers. However, these units are very coarse and rarely fit the used resources which often results in an over- or under-provisioning of the resources. Serverless functions are more fine-grained which enables a closer match of the resources to the currently needed ones.

- **More insight and control:** Usually, the developer of the application is responsible for operational tasks. With serverless computing, the cloud platform provider is responsible for these tasks. This makes it easier for developers who do not have experience in operational tasks. Furthermore, the insights are given on a more granular level than for virtual machines or containers. Therefore, the developer can make decisions to improve the application based on these insights.

- **Granular Scaling:** In traditional models where for example virtual machines are used, these virtual machines act as a black-box. If only one part of the application is bottlenecked, the whole application has to be scaled. Extracting such bottlenecks requires some additional work which requires some experience. However, in serverless computing, these units can easily be scaled because of their fine granularity.

All in all, serverless applications can be beneficial for development and costs. While the development of a scalable application can be accelerated by the provisioned infrastructure, the costs can be reduced by the fine granularity of the application and the pay-as-you-go billing mode working on this fine granularity.

#### 2.1.3.2. Drawbacks in Serverless Computing

But there are also some drawbacks in serverless computing which are listed here.

- **Latency:** When a serverless function is invoked and no instance is running which is able to handle the request, a new instance has to be deployed first which causes a delay that is called a *cold start* [9, 70]. The platform

provider can also decide to scale the number of active instances up or down depending on the workload which can also cause *cold starts* when new instances are deployed [1].

Furthermore, latency is higher in serverless applications compared to traditional VM-based approaches since the statelessness of the serverless functions requires communication with other components over the network which causes additional delay [38].

There is also some research done in the area of the cold start of serverless functions. Manner et al. measured the cold start times on several platforms to identify factors that influence the duration of the cold start in [70]. Lin and Glikson tries to mitigate the cold start in [61] by keeping warm instances for the execution of serverless functions on the platform. Some research [91, 95] tries not only to reduce the number of cold starts but also to minimize the used resources of the cloud at the same time to avoid costs for the platform provider.

Another approach described in [16] introduces a client-side approach by using the application knowledge. If a cold start is expected to happen soon based on the application data in this client-side approach, a hint to the cloud platform is sent which can now pre-warm an additional execution environment to prevent the cold start. In [85], the cold start time was improved by reducing the deployment size of the serverless functions.

- **Vendor Lock-in:** A known problem in *Cloud Computing* is the vendor lock-in where vendors offer cloud-based services that have different services from one provider to another [82]. Using a serverless application can make the application tightly coupled to the cloud platform since usually other cloud platform services are run on the same cloud platform and can be easily connected with serverless functions [1, 117]. Therefore, moving a serverless application from one platform to another requires not only to adapt the serverless functions used but also to adapt the services and interfaces used by the serverless functions. Approaches for the migration of serverless platforms from one provider to another are given in [117, 118].

- **Loss of Control** Using a serverless platform involves giving up full control of the software stack. Therefore, the platform might not support the desired version of a programming language or library. This problem can be solved on some platforms that provide support for any programming language as long as it is packaged in a container image supporting a well-defined *Application Programming Interface* (API) [9].

- **Security** Also regarding the security, control is lost since users of serverless platforms have to rely on the security mechanisms of the platform

and cannot implement custom ones [87]. Furthermore, since serverless functions of the different users are executed on a shared execution environment on the cloud platform [9, 94], it is important that the serverless functions cannot break out of their isolated environment and gain access to other resources [116].

- **Life Span:** Also the lifespan of serverless functions is limited. The functions can only be executed for a certain time and are terminated after that time limit [1, 9, 40]. This limits the applicability of serverless functions when a single task needs more time, for example, when a task should keep an HTTP connection and receive information for a longer period of time [1].

- **Costs:** As already written in benefits, serverless computing is not always cheaper. Compared to other on-demand compute services running indefinitely like virtual machines or container runtimes, costs for a serverless application might result in higher costs if the workload for an application is constant [78]. Additionally, serverless applications often use some other platform-specific services that are charged separately, such as data storage services or HTTP gateways for the serverless functions, whose costs can become quite significant and are often underestimated [57]. Therefore, different models for the prediction of the performance are proposed in [22, 60, 66] to calculate and optimize the costs.

- **Lack of Tooling:** The remote execution of the code on commercial platforms results in a lack of tools where classical tools can often not be used. In particular, there is a lack of tool support for serverless applications regarding local execution environments and debugging [1, 116] which makes it harder to develop and test new applications locally.

## 2.2. Phases of Testing

Since this work is about the testing phase integration testing for serverless applications, this section gives a short overview of the different testing phases.

In software development, software is developed on different abstraction levels that are tested. This can be illustrated by the classical *V-model* which was first proposed by Rook in [89]. Figure 2.4 shows the V-model based on an illustration in [72] representing a software development process.

The software requirements are refined into specifications which results in an architecture describing the system which is followed by a more concrete design where the modules are specified. All these abstractions have to be tested where in general the following testing phases are considered:

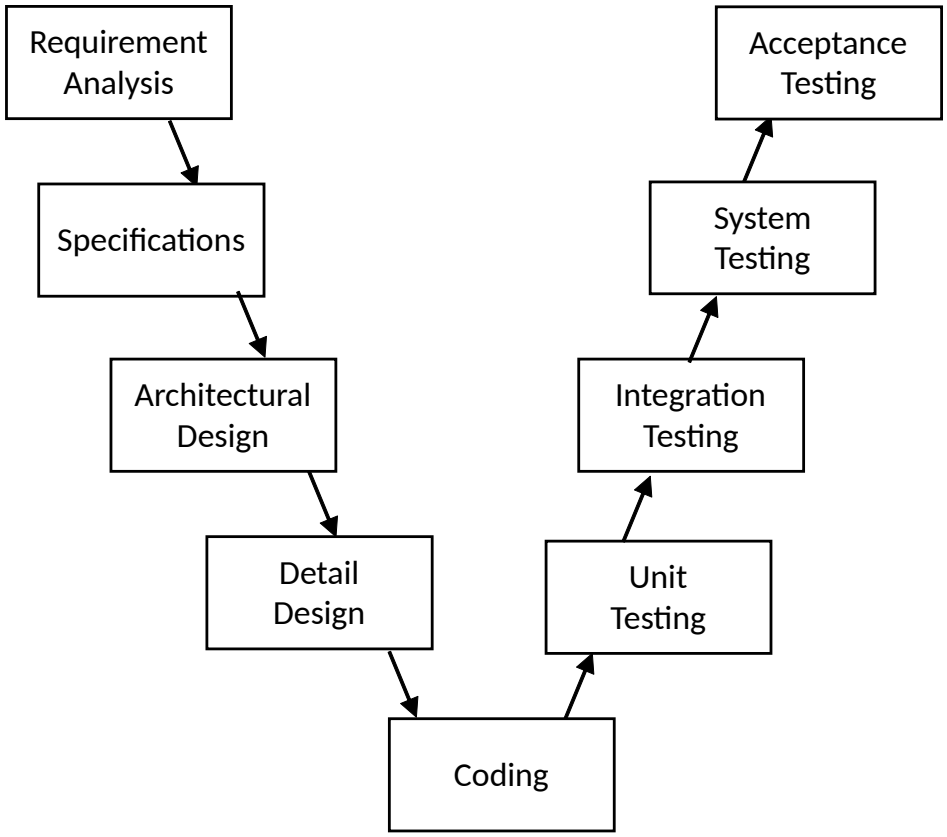



Figure 2.4.: Classical V-model based on [72]

- **Unit testing** *Unit testing* is also often called "Component Test" [115]. It focuses on each unit individually and ensures that it works properly as a unit [72, 115].
- **Integration testing:** *Integration testing* combines the units of the program and tests their correct interaction [72, 115] since it is nearly impossible for developers to consider all environments in which a component could be inserted during unit testing [43].
- **System testing:** *System testing* tests if the complete system complies with the specifications of the whole system [72, 115]. It does not only test the functionality of the system but also the performance of the system [43] to ensure that it also works when the system is used in a real-world scenario.
- **Acceptance testing:** *Acceptance testing* validates that the system meets the specified requirements of the customer [72, 115]. Even if all requirements were implemented correctly, it might happen that the requirements of the customer have changed or the requirements that were implemented were not the ones that the customer needed.

Depending on the software development process, these phases cannot always be sharply separated. They can also be extended by additional testing phases or by the repetition of testing phases like in agile software development.

For this work, the relevant part is the integration test which assumes that there are already components implemented that have to be combined. In particular, for serverless applications which consist of several serverless functions and other services provided by the cloud platform, the integration of these components has to be verified. Even if the components should be tested in isolation, it is hard to predict all possible combinations of a component and test them. Therefore, the integration testing phase is needed where their actual usage in combination with other components is tested.

Furthermore, *regression testing* is a quite common technique that is used in the maintenance phase of the software. It is performed on a modified program by using an existing suite of test cases to increase the confidence that the changes made are correct and do not affect the unchanged parts of the program adversely [90]. But also in recent methodologies like *DevOps* or approaches like *CI/CD* where automation is a key aspect, test cases are required which can be executed after code changes to ensure that the changes made do not break the existing functionality of the software.

In contrast to classical applications that are deployed on local machines, cloud applications run on cloud platforms. The loss of control caused by the remote execution often prevents the usage of existing testing tools, for example, to measure coverage metrics. Since direct access to some components running in the cloud is often not possible, these components must be treated as black boxes in the testing phases.

# Part II.

# Integration Testing for Serverless Applications

# 3. Modeling

The following chapter elaborates a model for serverless applications which represents the structure of the application and can be used as a basis for model-based testing. The content of this chapter is mainly based on the publication [108] where the author of this work is the main contributor. The aim of the model is to give a simple representation of the application which makes testing easier.

The main contribution of this chapter is given in Section 3.2. It shows the relevant characteristics of serverless applications that are needed for integration testing and demonstrates how these can be mapped to a model. Section 3.3 describes and shows how such a model can be created automatically to support the creation process of the elaborated dependency graph by applying it to a specific use case in Section 3.4.

Furthermore, related work is described in Section 3.5. It describes a similar approach of another paper, where a service graph is created automatically, and also other models focusing on the representation of cloud applications as well as the differences to the approach proposed here.

## 3.1. Motivation

Since in a serverless application, many interacting services are involved which can additionally be executed in parallel, behavior emerges which is hard to predict. To address this issue, a model is essential for visualizing and analyzing serverless applications. This model should enable the developer to detect components that have the potential to cause errors, in particular so-called hot spots where data are accessed in parallel by different components. Furthermore, this model should enable the creation of test cases. Since no such model existed for serverless applications before, this chapter introduces a model representing serverless applications in a human-readable way grasping the structure of the application to support the generation of test cases.

## 3.2. Model for Serverless Applications

Serverless applications consist of various resources offered by the cloud platform provider where the corresponding application runs. The primary resources of serverless applications are serverless functions. A serverless application is

created by implementing logic in serverless functions, which interact with other resources on the cloud platform. This section illustrates how to construct a dependency graph by focusing only on serverless functions and their connection to other serverless functions and resources. These resources are necessary for storing the application's data, processing and dispatching data, and orchestrating serverless functions. Therefore, these resources and their interaction with the serverless functions should also be captured by the model. This section demonstrates how these resources can be utilized based on the graph by either ignoring them or by adding them to the graph if suitable. Adding certain resources makes the graph more powerful which is useful for a deeper analysis of the application. Furthermore, this chapter illustrates how further useful information can be added to the graph by assigning this information to the nodes or arcs to extend its expressiveness and discusses the power of these extensions.

So, based on a basic graph representing only serverless functions, it is shown how the graph is extended to also represent data storage resources and other services like streams, queues, and more specific services like a mail service. Additionally, for data storage resources, it is shown how their kind of data access from a serverless function can be added to the model, and for serverless functions, how the information can be added whether a serverless function is invoked synchronously or asynchronously. Also, other aspects that can be assigned statically to the nodes are described. This includes the order of calls to services made from a serverless function, the number of asynchronous calls made by a serverless function to other resources, and the conditions under which a serverless function calls other resources.

Platform-specific settings can also be assigned to the nodes, comprising the access rights of resources and platform-specific settings for serverless functions like their timeout limit or the memory and computational resources assigned to them. Since data flow testing is also handled in this work, it is shown how data flow information can be added to the graph and how the serverless functions can be called at all by describing their interface. Finally, it is discussed how runtime data collected during the execution of the application can be assigned to the model. This includes the runtimes and call history of serverless functions, the usage of computational power and memory, and billing information.

### 3.2.1. Basic Serverless Dependency Graph

Serverless functions are the central resource of a serverless application which can be connected with each other in order to combine the logic implemented in the serverless functions. One possibility to invoke a serverless function directly from another serverless function is to pass an event to another serverless function (compare Figure 3.1) which finally processes the event.

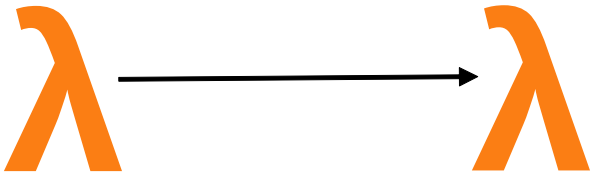


Figure 3.1.: Visualization of the invocation of a serverless function by another serverless function

But other resources can also connect serverless functions indirectly by being triggered by a serverless function and then triggering another serverless function. This can be done by accessing the resource directly by the serverless function. For instance, when a serverless function modifies data on a data storage resource (compare Figure 3.2) that triggers another serverless function informing it about the data change. Cloud platform providers support the functionality that a serverless function is triggered when data are written to a data storage resource. On *AWS*, this is accomplished with *DynamoDB* where, for example, a serverless function writes some data on the *DynamoDB* data storage resource and another serverless function can be configured to be triggered by the *DynamoDB* data storage resource. The triggered serverless function receives information about the state change in order to react to it.

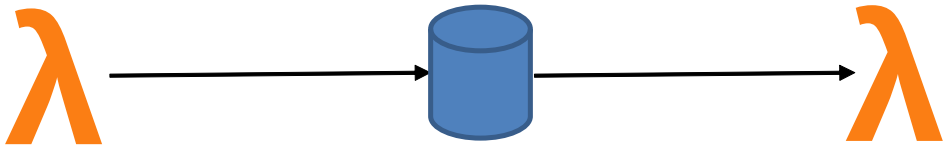


Figure 3.2.: Visualization of a basic data storage resource access of a serverless function

These dependencies build the basis for a first basic dependency graph which represents the serverless functions and visualizes how the invocation of a serverless function can trigger the execution of other serverless functions. So, the basic dependency graph represents just the serverless functions with their relations to the invocation of other serverless functions if one of the serverless functions of the graph is called. In this work, Definition 3.1 gives a basic definition of the dependency graph.

**Definition 3.1 (Basic Dependency Graph)**
*A basic dependency graph $G$ is defined as a set of nodes $N$ and a set of ordered pairs $A$. $A$ are ordered pairs of elements of $N$. In the model, a node $n \in N$ represents a serverless function, whereas an arc $(n_1, n_2) \in A$ represents a potential invocation of the serverless function $n_2 \in N$ resulting from the serverless function $n_1 \in N$.*

As a result, the structure of an application that only consists of serverless functions can be described by using this basic dependency graph. For example, if the command pattern is applied in the application, which is used to control

the invocations of other functions in response to the data received [92], it can be represented in a graph like shown in Figure 3.3.

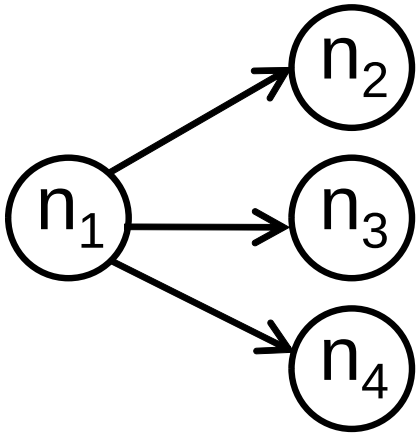



Figure 3.3.: Example of a basic graph representing the command pattern

The graph provides basic insights into the application and a compact overview of the event-driven architecture and the exchange of events between the components. But the graph can also be used for error detection and test case generation which is shown later in this work.
A recursive call of functions would result in a cyclic graph. Therefore, identified cycles have to be analyzed in order to avoid endless recursive function calls and their resulting enormous costs.

But the graph can also be used for a general identification of workflows. A simple algorithm, which is described in Algorithm 3.1 and which is based on a depth-first search, can be used to identify paths of an acyclic graph in a total order. Paths generated by this algorithm represent serverless functions in their total order of execution and can be used as a starting point for a coverage analysis of the application. However, such a path does not consider other nodes which are called in parallel if a node has several successors.

**Algorithm 3.1** Path identification in acyclic graph based on depth-first search

```
1  public static List<Node> DFS(Node node) {
       List<Node> path = new ArrayList();
3      path.add(node);
       while(node.hasSuccessor()){
5          node = node.getASuccesor();
           path.add(node);
7      }
       return path;
9  }
```

If a node has several successors that are invoked, the potential execution of the serverless functions can be given as a partially ordered set of orders since

for each successor another workflow is created whose relation to the other workflows cannot be guaranteed.

Furthermore, if a graph is cyclic, the algorithm has to be extended to detect cycles, e.g., by marking nodes as visited. However, this requires also a strategy for how cycles should be handled, e.g., by ignoring them or by generating paths with a maximal number of cycles. For example, an algorithm like in Algorithm 3.2 can be used where for each successor a new depth-first search is started. By marking nodes as visited and passing the workflows that caused the current search, cycles can be detected.

**Algorithm 3.2** Depth-first search for parallel workflows in cyclic graph

```
1       public static int numberOfWorkflows = 0;
        public static List<Node> DFS(Node node, List<Integer>
            previousWorkflows) {
3           List<Node> path = new ArrayList();
            List<Integer> workflows = new ArrayList(previousWorkflows);
5           int currentWorkflow = numberOfWorkflows++;
            workflows.add(currentWorkflow);
7           path.add(node);
            node.addVisited(currentWorkflow);
9           for(Node suc : node.getUnvisitedSuccessors(workflows,
                currentWorkflow)){
                path.addAll(DFS(suc, workflows));
11          }
            return path;
13      }
```

For example, the function *getUnvisitedSuccessors* in Algorithm 3.2 checks if the successors of the current node were already invoked by the current workflow. If a successor was already called by another node that is not part of the current workflow, the node should still be called since it is not part of a cycle. Figure 3.4 visualizes a graph with a cyclic dependency.

If the node $n_2$ is visited for the first time, a list of the previous workflows that are always triggered before its execution is available. Therefore, $n_4$ is considered as a valid successor even if it was called before by a workflow of $n_3$ which is not part of the passed previous workflows. When $n_2$ is visited a second time by the algorithm, $n_4$ contains already the information that it was invoked by a workflow being in a causal relationship to the current one. So, either the workflow can be stopped if it was already visited or a handling strategy can be used which decides if a following node should still be considered. However, this requires that the handling strategy is able to limit the potential number of paths. Useful handling strategies to limit the number of paths are according to [84] limiting the number of cycles traversed or the length of the paths.

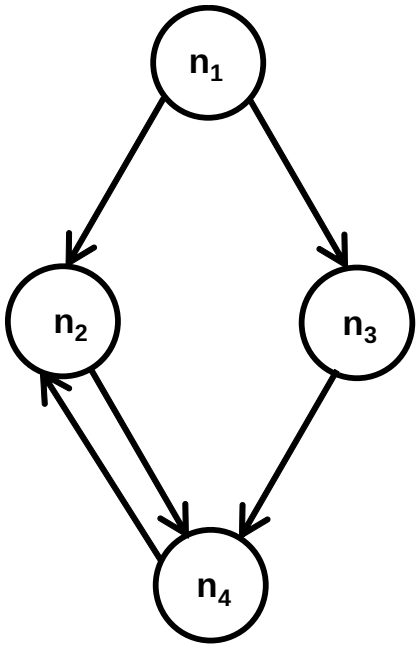



Figure 3.4.: Example for graph workflow

However, since the graph only indicates whether there is a relationship between the serverless functions, it is not guaranteed that the paths constructed by these algorithms can be executed, e.g., if an invocation depends on a certain condition, a successor might not be called.

### 3.2.2. Mapping of Resources to the Model

A serverless application usually consists not only of serverless functions but also of other resources that provide some services utilized by the application. The statelessness of serverless functions enables the cloud provider to dynamically parallelize function calls without violating the logic of the application. If some state has to be persisted, the state has to be saved in a resource since it is not guaranteed that the state can be retained within a serverless function. Therefore, resources like data storage resources, which keep data even if no serverless functions are running, are often used. But also other resources are often used for dispatching or processing data, orchestrating serverless functions, or offering an interface to the application. Most of these resources can not only be called by serverless functions but can also trigger serverless functions by passing an event to them. In the following, it is demonstrated how these resources can enrich the basic graph by either adding or ignoring these resources when building the graph.

#### 3.2.2.1. Data Storage

A serverless function does not guarantee that its state is kept if it is invoked again. For this reason, it has to save its state to data storage resources, which are usually databases, if persistency of data is needed. Since most applications have to persist data, data storage resources are integrated into the model which are defined in Definition 3.2.

> **Definition 3.2 (Data Storage in Graph)**
> *If a data storage resource has to be modeled in this dependency graph, the basic graph $G$ defined in Definition 3.1 must be extended to a graph $G = (N, D, A)$ by using an additional set of nodes $D$ where a node $d \in D$ represents a data storage resource.*
> *The set of arcs $A$ has to be extended by also allowing arcs $(d, n)$ and $(n, d)$ where $n \in N$ and $d \in D$.*

The arcs to a data storage resource indicate how the data storage resource is accessed. These are defined in Definition 3.3. If an arc directs to a data storage resource, it indicates that the data storage resource is accessed (e.g., an entry is created, read, updated, or deleted). Therefore, the arc does not indicate the data flow since data can be sent to the data storage resource or received by it depending on the type of access. It simply indicates the source of the interaction. However, when a data storage resource has an outgoing arc, it indicates that another resource, e.g., a serverless function is triggered, and data are sent to it.

> **Definition 3.3 (Access to Data Storage in Graph)**
> *An arc $(n, d)$ of a data-storage-extended graph $G = (N, D, A)$ of Definition 3.2, where $n \in N$ and $d \in D$, indicates that data are written to or read from a data storage resource $d$ by a serverless function $n$, whereas an arc $(d, n)$ represents a trigger of a data storage resource $d$ that creates an event handled by a serverless function $n$ if some data in the data storage resource $d$ were changed.*

If only the basic dependency graph is needed, the data storage resources can be omitted. This can be done by adding direct dependencies between serverless functions if these functions are connected via a data storage resource (e.g., arc $(n_1, d_1)$ and $(d_1, n_2)$ where $n_1, n_2 \in N$ and $d_1 \in D$ can be fused to a single arc $(n_1, n_2)$) (compare Figure 3.5).

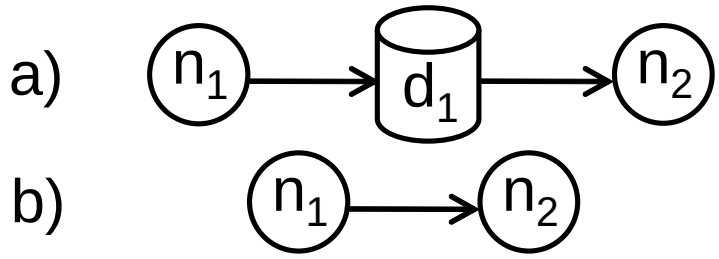



Figure 3.5.: Example of a graph containing a data storage resource (a) and its counterpart as a basic graph (b)

#### 3.2.2.2. General Approach for Other Resources

The same approach to represent data storage resources in the graph can be utilized for other resources that either trigger serverless functions or are accessed by them. These resources can for example be streams, queues, mail services, and other services provided by the cloud platform provider that can be connected to serverless functions. If a certain resource needs to be modeled, the graph can simply be extended by adding a new set of nodes representing the corresponding type of resource. The outgoing and incoming arcs are connected to the serverless functions invoking the resource respectively triggered by an event that was created by the corresponding resource. If the resources can be connected to other resources directly, the arcs have to be defined to support this relation. A definition for general resources is given in Definition 3.4

**Definition 3.4 (General Resources in Graph)**
*Other resources than serverless functions and data storage resources are modeled by extending the basic graph $G$ of Definition 3.1 or 3.2 by using an additional set of nodes $E$ where a node $e \in E$ represents a general resource.*
*The set of arcs $A$ has to be extended by also allowing arcs $(e, n)$ and $(n, e)$ where $n \in N$ and $e \in E$. An arc $(e, n)$ indicates that $e$ initiates the interaction with $n$ and vice versa for an arc $(n, e)$.*

However, if there are resources orchestrating the workflow of serverless functions, such as in Amazon's *AWS Step Functions*[5], it can be advisable not to omit all the serverless functions if only the basic model is intended to be used. For example, a step function can initialize a workflow as depicted in Figure 3.6 which consists of several serverless functions. These serverless functions could be abstracted away by simply using a node representing the orchestration of the serverless functions. But by modeling all serverless functions that are part of this resource in the dependency graph, the information of their potential order of invocations and the relations of the serverless functions is kept and can be visualized.

[5] `https://aws.amazon.com/step-functions/`

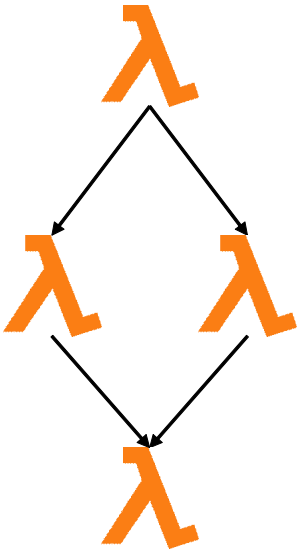


Figure 3.6.: Example workflow of a step function

Thus, the basic dependency graph can be easily extended if some resources need to be modeled. This gives new insights into the structure of the application and enables the detection and creation of architectural patterns. The additional nodes enable the creation of new test cases. In general, nodes and their relations can be used as a basis for coverage criteria, e.g., requiring the coverage of all nodes or all node-pairs [64], which can help in the identification of potential scenarios where the application is used. Specific coverage criteria are introduced in Chapter 4.

### 3.2.3. Characteristic Settings of Serverless Applications

Having only a dependency graph which consists of serverless functions and all the resources connected to them gives an overview of the structure of the application. For example, it provides a short overview of the serverless functions involved which is often relevant to identify workflows and data flows which can be identified by considering the model. However, finer details of the application which are relevant for further analysis of the application are missing. Therefore, this section discusses some additional useful aspects of the application that support the static model of the serverless application, and aspects that are relevant for the analysis of a serverless application depending on the purpose of analysis.

#### 3.2.3.1. Read/Write Access

Data storage resources are central resources in a serverless application to save the state of the application. However, the usage of data storage resources can also cause errors. In a serverless application, serverless functions can be executed in parallel by the cloud provider. This can happen if a workflow calls the same serverless function several times asynchronously which causes a new independent workflow or if an application interface is called several times in parallel. Both can result in parallel workflows where the order of data accesses can be relevant. However, if workflows running in parallel access the same data and at least one of the workflows writes data, the state and behavior of the application can deviate from its intended state and behavior. The actual behavior depends on the order of accesses to the data storage resources. Therefore, the kind of data access from a serverless function to a data storage resource is relevant knowledge of the graph. The graph can be extended to represent the kind of data access as defined in Definition 3.5 which indicates additionally if a read, write, or a read/write access is made to a data storage resource.

> **Definition 3.5 (Read and Write Access to Data Storage Resource)**
> *A data-storage-extended graph $G = (N, D, A)$ of Definition 3.2 is enriched with additional information of a set $I = \{r, w, rw\}$. Elements $i \in I$ representing read, write, or read/write accesses can be mapped to arcs $(n, d)$ where $n \in N$ and $d \in D$.*

Furthermore, the kind of data access can also be refined by not only sticking to the coarse information of *read* and *write* but also by replacing them with the basic data storage resource operations *create, read, update, and delete (CRUD)*. In this case, the set $I = \{r, w, rw\}$ is replaced by a set of $\{c, r, u, d\}$ and the permutations of their elements indicating the kind of CRUD operations used by the arcs (see Definition 3.6).

> **Definition 3.6 (CRUD Access to Data Storage Resource)**
> *A data-storage-extended graph $G = (N, D, A)$ of Definition 3.2 is enriched with additional information of a set $I = \{c, r, u, d\}$. Elements $i \in I$ representing create, read, update, or delete operations can be mapped to arcs $(n, d)$ where $n \in N$ and $d \in D$.*

Detecting problematic scenarios, such as a potential race condition illustrated in Figure 3.7 where two workflows access the same data storage resource, is a relevant scenario for developers. If both workflows access the same data, the data read by $n_{22}$, and thus the result of its workflow, depend on whether $n_{12}$ writes its data before or after the access of $n_{22}$.

Since calls to a serverless application can be started simultaneously several times, even write and read accesses of different instances of the same workflow

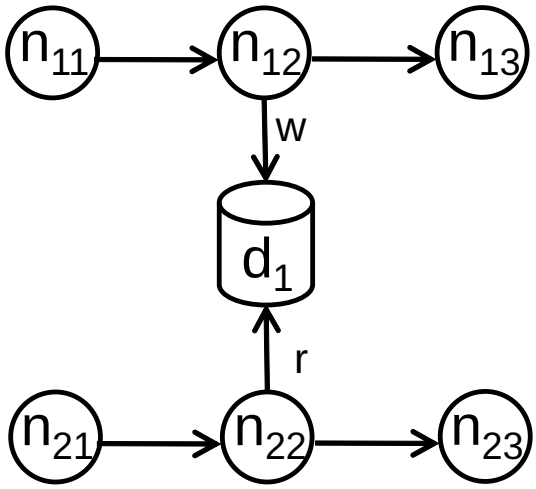



Figure 3.7.: Potential race condition in a graph with two workflows

(see Figure 3.8) can be critical. Different workflows could write and read the same data and thus provoke a race condition. But also a race condition of the same workflow can occur if $n_{12}$ is executed before the write access from $n_{11}$ to $d_1$ is finalized. This can happen if $n_{12}$ is called asynchronously before the write access from $n_{11}$ to $d_1$ is finalized or if the write access is finalized, but only eventual consistency is guaranteed.

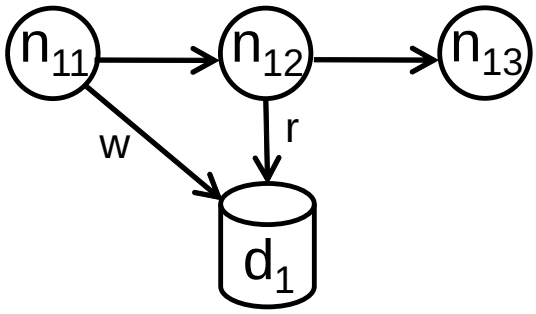



Figure 3.8.: Potential race condition in graph with one workflow

Using such a notation makes it easier to detect such hot spots, where critical data access situations are possible, and create test cases verifying that read and write access work in any order.

However, not all accesses to the same data storage resources use the same data. Therefore, annotating the arcs with constraints on data storage resource accesses supports the identification of the relevant accesses to data storage resources where race conditions might occur. If a key of a key-value database or a selection of a relational database is set as a constraint, intersections of accesses can be more easily detected. However, if the values constraining the access to the data storage resource can change depending on the input parameters of the function, detecting intersections becomes challenging. Therefore, instead of using this annotation, a worst-case assumption has to be used.

Even more annotations for databases are applicable. For example, an annotation of the consistency model of the database can help to detect problems in the database. If a database offers only a restricted consistency, the hot spots and their relations have to be handled more carefully.

The information about the type of data access to a data storage resource is not only relevant for the identification of hot spots but also for the identification of data flows. The basic dependency graph gives only the possibility to reconstruct workflows that result from an invocation of one resource. However, if there are data stored in a resource and used by another workflow, it is also relevant to see if this data flow can be visualized in the graph. By annotating the kind of data storage resource access, potential data flows can be identified. For example, if a serverless function writes data and another data flow reads data from the same data storage resource, a dependency between these two functions exists even if they are not invoked within the same workflow. Figure 3.9 shows an example of two workflows accessing a data storage resource. Without any annotations, a data flow between both nodes could not be seen. However, with the annotations of a read and a write access, a workflow between the serverless functions represented by the nodes $n_1$ and $n_2$ can be constructed where $n_1$ first writes data which are finally read by $n_2$.

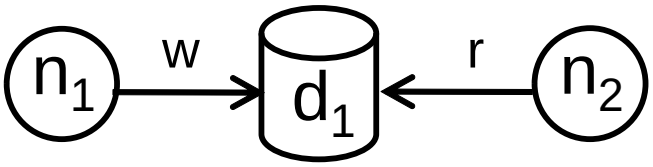



Figure 3.9.: Two workflows via data storage resource

#### 3.2.3.2. Synchronous Invocations

Serverless functions are invoked either asynchronously or synchronously. Asynchronous calls are based on an event-driven architectural procedure where an event is created and then handled by another resource. This results in an independent execution of the former code having created the call. On the other side, in synchronous invocations, the caller waits until the callee is finished. In the specification of the basic graph, there is no distinction between synchronous and asynchronous calls. Definition 3.7 shows how the basic graph can be extended where only relations between serverless function nodes are annotated with this kind of information.

**Definition 3.7 (Synchronous and Asynchronous Calls in Graph)**
*A basic graph $G = (N, A)$ is enriched with additional information of a set $S = \{sync, async\}$ where sync represents synchronous invocations and async represents asynchronous invocations. Elements $s \in S$ can be mapped to arcs $(n_1, n_2)$ where $n_1, n_2 \in N$ to represent the kind of invocation.*

However, the behavior of an application is influenced by the fact whether a serverless function calls another serverless function synchronously or asynchronously. If a synchronous invocation is made, the caller is more vulnerable to timeouts since its execution time depends on the callee whose answer the caller expects and wants to process.

Furthermore, billing is also a problem since the caller is running while waiting for a response. Therefore, both the calling and the called resources are charged for the time they are running which results in double-billing [10]. Synchronous function calls are typically not the best choice for the architecture of a serverless application and are, consequently, worth being highlighted. Accordingly, synchronous function calls are highlighted by extending the arcs initiating such a call with an additional element $sync$ for arcs. Figure 3.10 represents three serverless functions where one serverless function $n_1$ calls two other serverless functions $n_2$ and $n_3$. By annotating the arcs with an additional $sync$, it can be derived from the graph that no additional workflow is created.

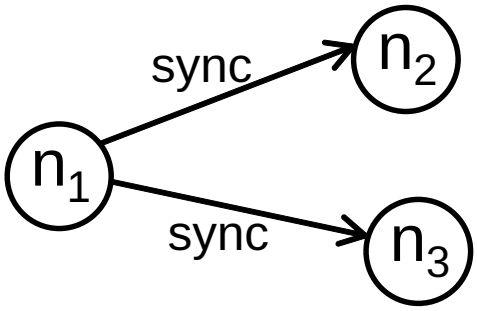



Figure 3.10.: Synchronous call to serverless functions

Only asynchronous calls of serverless functions can increase the number of parallel workflows. Therefore, the number of potential workflows that can be derived by a model can be reduced. If resources trigger serverless functions, these serverless functions receive an event which is an asynchronous communication and can also create several parallel workflows. For example, in Figure 3.11, a serverless function writes data to a data storage resource which triggers two serverless functions independently. Here, after the data were written by $n_1$ to the data storage resource $d_1$, the serverless functions $n_2$ and $n_3$ are triggered and run in parallel.

If the focus of the analysis is more on the identification of independently running serverless functions, synchronous calls of serverless functions can

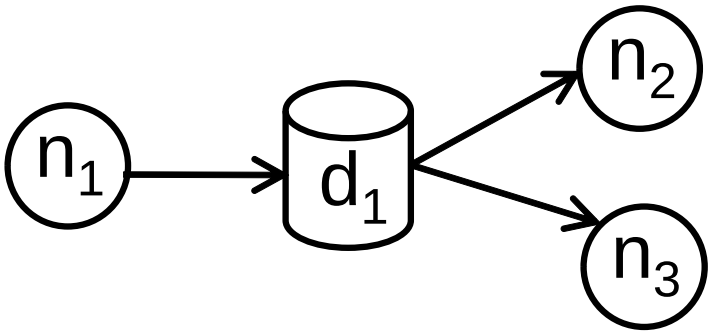



Figure 3.11.: Database triggers two serverless functions

also be omitted. Instead, only asynchronously called serverless functions are modeled which provides a more compact and compressed view of the model without losing the information of potential workflows. This is similar to the general approach of mapping resources to the graph, as shown in Figure 3.12 where the same scenario as in Figure 3.11 is shown without visualizing the data storage resource.

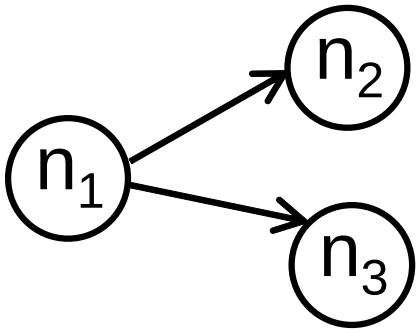



Figure 3.12.: Reduced model of database triggering two serverless functions

#### 3.2.3.3. Order of Calls

The graph does not indicate the order in which calls to other resources are made. Therefore, if the graph is analyzed for critical situations or potential workflows, more cases can be identified than are actually possible in any concrete run. As already shown in Figure 3.8, potential race conditions can be detected if the order of calls to resources accessing the same data storage resource is unclear. However, if the order of calls is known and a synchronous call is made before an asynchronous call, the potential for race conditions can be excluded in some cases. If in the example of Figure 3.8 $n_{11}$ writes its data synchronously to $d_1$ before $n_{12}$ is called either synchronously or asynchronously, no race condition occurs for the same workflow as long as only one instance of the workflow is used. Therefore, by annotating the order to each outgoing arc of a node facilitates the elimination of infeasible workflows which can simply be done by adding a number to the outgoing arc of a node representing a serverless

function. Then, this number indicates the order in which the relation is executed if the node is called. Of course, this can only be done if there is a clear order of calls within the serverless function, which is not always guaranteed.

#### 3.2.3.4. Multiple Asynchronous Calls

In order to analyze for parallel workflows or bottlenecks, dependencies are interesting where a method is not called only once asynchronously but several times. If a single serverless function invokes another serverless function several times asynchronously, this can be indicated by several arcs between these nodes. Such a model can be compressed by annotating the number of calls to an arc made by the caller to the callee. So, parallel workflows caused by a single node can be identified more easily without causing confusion in the visualization of the graph. Additionally, this enables the analysis of potential bottlenecks where the number of parallel calls can be throttled due to server limits.

#### 3.2.3.5. Conditions of Calls

While the dependency graph enables the creation of potential workflows, the number of generated workflows can be very large. This often includes also infeasible workflows that are generated since the information that a resource is called only under a certain condition is not available in the basic graph. Therefore, the worst-case assumption in a node has to be used that all related resources can be called if the node is called. This can be prevented by adding constraints to invocations if they are made under a certain condition. Similar to the order of calls, this information can be attached to the arcs of resources. This reduces the number of infeasible workflows generated by using the model and also helps identify workflows ending in the node since their conditions are not fulfilled. However, this requires a more sophisticated syntax, which is able to express constraints. Depending on the complexity of the constraint, this can make the graph less intuitive and harder to create why this approach is not considered in this work.

#### 3.2.3.6. Test-related Information

Since the model is later used for the creation of test cases, interfaces need to be added to enable direct invocation of the serverless functions. Furthermore, since in this work also data flow criteria are considered, support for data flow is added to the model.

**Interfaces** If a model should be used for testing, it is crucial to consider how the single nodes can be invoked. Therefore, it might be useful to assign an interface to each node representing a serverless function in order to execute the serverless function along with all the necessary data on the actual system.

**Definition 3.8 (Interfaces in Graph)**
*Each $n \in N$ of a basic graph $G = (N, A)$ can be enriched with additional information describing the interface of the serverless function, e.g., by using a regular expression describing the permitted parameters that are mapped to $n \in N$.*

**Data Flow** Not only the control flow but also the data flow is relevant when creating test cases. As will be discussed later in this work, the data flow can also be supported by the model. By assigning the locations where the data are defined and used which are passed between the nodes, data flows can be considered in more detail as defined in Definition 3.9 and illustrated in Figure 3.13.

**Definition 3.9 (Data Flow in Graph)**
*Each arc $(n_1, n_2) \in A$ of a basic graph $G = (N, A)$ is enriched with a set of $Defs$ and $Uses$ representing the locations where the data is defined for the last time in $n_1$ and where it is used in $n_2$ for the first time.*

*A data-storage-extended graph $G = (N, D, A)$ of Definition 3.2 has to be extended with Definition 3.6. For arcs $(n, d)$ where $n \in N$ and $d \in D$, $Defs_n$ represent the location where the transmitted data are defined in $n$ for the last time if $c$, $u$, or $d$ are assigned to the arc. If $r$ is assigned to the arc, $Uses_n$ represent the locations where the data are used in $n$ for the first time.*

*For arcs $(d, n)$ where $n \in N$ and $d \in D$, $Uses_n$ represent the locations where the transmitted data are used in $n$ for the first time.*

*If the graph is extended with other Resources $E$, elements $c$, $u$, or $d$ of $I$ can be assigned to arcs $(n, e)$ and $r$ to arcs $(e, n)$ where $n \in N$ and $e \in E$.*

For example, case *a)* of Figure 3.13 shows that the data were defined in $n_1$ in *line_10*, while the value is used for the first time in $n_2$ in *line_3*. Case *b)* shows a relation between a serverless function and a data storage resource where data are for the last time created in $n$ in *line_11* and passed to $d$. For such a case, a kind of CRUD operation is needed. Alternatively, also an *update* or *delete* is possible in such a constellation. Case *c)* requires a use even if the direction of the arc is from the serverless function to the data storage resource. The serverless function $n$ initiates the interaction and reads data from the data storage resource. These data are used for the first time in $n$ in *line_12*. Finally, case *d)* shows the relation when a data storage resource triggers another serverless function. The data which are passed here can only be used within $n$. Therefore, no additional annotation is needed. By combining these cases, e.g., data are created by a serverless function like in case *b)* which is used afterwards by another serverless function like in *c)* or *d)*, data flows can be constructed which are considered later in this work.

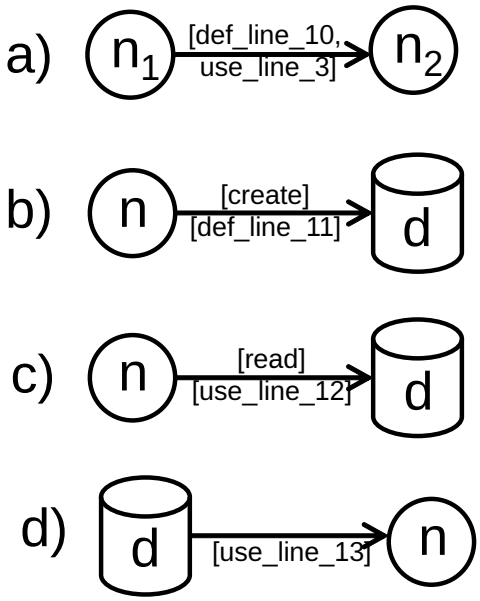



Figure 3.13.: Data flow between functions and data storage resources

If there are more locations in a serverless function where the data is defined for the last time or used for the first time, this information can also be added to the arcs.

#### 3.2.3.7. Platform Settings

Other resource-specific settings made to the cloud platform can also be assigned to the nodes of the dependency graph.

This includes the timeout limit which is set to indicate the maximum time a function is allowed to run before the function is terminated.

Furthermore, memory and computational power can be assigned to serverless functions which can also be stored as additional information in the nodes of the model.

This helps estimate the performance of the function and resource, as well as the associated cost consumption. However, this is particularly relevant if data are available providing information about the profile of a running application as described in 3.2.4 in order to evaluate if the corresponding serverless functions use their assigned resources efficiently.

Furthermore, in a serverless application, each resource has access rights assigned describing the rights it has to utilize other resources. By assigning only the necessary rights to each resource, unnecessary security breaches can be prevented. The dependency graph provides valuable insights into the rights needed for each serverless function. Therefore, if access rights are assigned to nodes and are displayed alongside, nodes with missing or unnecessary rights can be identified, improving the security of the application.

### 3.2.4. Runtime Data

In the preceding section, it was shown how serverless applications can be modeled and which static attributes can be added to the model to support the analysis. The additional information was either assigned to the nodes or the arcs of a model representing the architecture of an application without adding any dynamic data received from executions of the application. This section shows that also some runtime information that can be collected during test runs or production runs can be used in accordance with the model, and thereby enhance error detection.

#### 3.2.4.1. Execution Times

If data providing information about the execution times are available, usage statistics can also be annotated to the nodes indicating the time a function or another resource took to execute. If workflows are generated by using the model, their execution time can be estimated more easily by leveraging the annotated time. This enables a better estimation of future execution times and the identification of long-running tasks. Furthermore, the cold start time needed could also be added to enable a more intensive analysis helping estimate the execution time of workflows called for the first time. By comparing these values to the timeout limits of the functions, a risk analysis can be conducted, making the application more reliable.

#### 3.2.4.2. Usage of System Resources

Similar to the annotation of time settings, the memory and computational resources used by a serverless function for its executions can be assigned to it. This helps identify functions with a huge demand for system resources, such as memory or CPU power, whose performance might be worth improving. If the actual platform-specific computational or memory resources are available within the model as described in Section 3.2.3.7, the model can be evaluated to identify critical nodes that might be prone to terminate due to insufficient memory or CPU available. Furthermore, nodes having too many hardware capabilities assigned can be identified, potentially leading to cost savings if the execution is not time-critical since the reduction of system resources might result in a longer execution.

#### 3.2.4.3. Billing

Having functions that are only paid if they are actually used is a key characteristic of serverless computing. Therefore, in order to assess costs and explore optimization possibilities, the costs of runs can be assigned to nodes. This supports the identification of expensive functions, and this information correlates

with the data of time taken for a function execution and the hardware assigned to it.

#### 3.2.4.4. Call History

Another piece of interesting information that can be helpful for the evaluation of the model is the call history of the resources. By saving the number of calls or even the times when the resources were called, a usage profile can be created. Even more detailed is saving the calls made between resources, which is represented by the arcs, or even saving the call chains by assigning them to the corresponding resources. Of course, this requires a detailed instrumentation of the application which saves the relevant data to assign them to the model. If these data are available, a usage profile of the application can be created, and spots where improvement can be made or resources that are seldom used can be identified more efficiently.

## 3.3. Automatic Creation of Models

In the previous section, the definition of a basic dependency graph of a serverless application was given which represents the components of the application. This graph can be extended to represent some information, which can be useful for different use cases. While some of this information can be statically derived by analyzing the source code of the application, some other information has to be added to the graph dynamically during the execution of the serverless application and can change while the application runs. The latter one is the basis of a model for monitoring an application, displaying information, such as resource utilization or execution times. The following section describes how the creation of a basic model can be supported automatically. Furthermore, it demonstrates how its annotations and monitoring information can be added to the graph.

### 3.3.1. Structural Model

In the following, the creation of the model's structure is discussed by considering the infrastructure file that describes the application and the source code of the serverless functions.

#### 3.3.1.1. Basic Model

The basic model can be created by reading a file describing the infrastructure of the application. *AWS CloudFormation* can be used on AWS to describe the application and to deploy it. Also, the popular framework *Serverless Framework*, which can support several platforms to deploy serverless applications, provides

an infrastructure file, which describes the components of the application. When deployed on *AWS*, the infrastructure file is translated into the infrastructure file format of *AWS CloudFormation* where fewer settings are abstracted away.

The infrastructure file has to contain the description of all the resources that are deployed. Additionally, the source codes of the serverless functions are needed. If the infrastructure file or the source code is not available, the model can still be created manually by making worst-case assumptions.

All the resources being used in the application can be identified by parsing the infrastructure file. Even some relationships between the components can be identified, e.g., if a database triggers a serverless function, this relationship is defined in the infrastructure file. However, the relationship between serverless functions is not described in this file. In order to get the invocations made by serverless functions, their source code must be analyzed. This can be done by using a parser identifying the relevant code snippets and has to be done for all serverless functions. However, it is hard to decipher an invocation to a serverless function if it is not hard-coded and even impossible to get the invocation to a serverless function if it depends on a parameter given to the function. If so, possible functions have to be added manually. If the source code of the functions is not available, the relations of the functions have to be constructed manually or based on monitoring the execution of the app if supported by the serverless functions.

An example of a deployment file is shown in Listing 3.1 where a single function and a *DynamoDB* database are deployed as the components of the application. The basic graph representing this application contains the serverless function without the database while an extended version also contains the information that a database is available and its relations.

Even if there are policies available indicating which resources are allowed to be accessed by the serverless functions, these policies are often not set specific enough to extract the information of the accessed resources, like in the Listing 3.1 where the policy is given to access all *DynamoDB* databases by all kind of accesses. Therefore, the kind of database access and the database itself have to be parsed from the source code file of *Function1*.

Listing 3.1: Basic resources of a deployment file

```
1  {
2      "Description":"Simple test project",
3      "Resources":{
4         "Function1":{
5            "Type":"AWS::Serverless::Function",
6            "Properties":{
7               "FunctionName":"Function1",
8               "Handler":"com.serverless.demo.function.
                   Function1",
```

```
                "Runtime":"java8",
                "CodeUri":"./target/demo-1.0.0.jar",
                "Policies":[
                   "AmazonDynamoDBFullAccess",
                   "AmazonS3FullAccess",
                   "AWSLambdaBasicExecutionRole"
                ]
             },
          },
          "hashTable":{
             "Type":"AWS::DynamoDB::Table",
             "Properties":{
                "TableName":"myTable",
                "AttributeDefinitions":[
                   {
                      "AttributeName":"fileName",
                      "AttributeType":"S"
                   }
                ],
                "KeySchema":[
                   {
                      "AttributeName":"fileName",
                      "KeyType":"HASH"
                   }
                ],
                "BillingMode": "PAY_PER_REQUEST"
             }
          }
       }
    }
```

If a basic graph containing only serverless functions should be built and the application contains resources that are not serverless functions but are connected to serverless functions, a graph containing all resources and their relations has to be constructed first. Afterwards, the unwanted resources can be removed by connecting the predecessors and successors of the type serverless function resulting in a basic graph containing only serverless functions.

#### 3.3.1.2. Data Storage

Data storage resources utilized within the application should already be defined in the file describing the infrastructure. The invocations made to data storage resources by serverless functions, such as read and write accesses, can be

identified by analyzing the source code of the serverless functions. However, like for the identification of serverless function calls, if the parameters are not hard-coded, the relations to data storage resources cannot be created automatically. Hence, manual annotations can be useful. Constraints can also be identified by analyzing the source code of serverless functions with the same drawback.

Events sent by data storage resources to serverless functions to trigger them can be extracted from the configuration of the serverless function within the infrastructure file. Such an event configuration is exemplified in Listing 3.2 where a serverless function is triggered when a new object is stored on the data storage resource.

Listing 3.2: S3 triggering AWS Lambda function

```
"Events":{
    "GetResource":{
      "Type":"S3",
      "Properties":{
        "Bucket":{
          "Ref":"NameOfBucket"
        },
        "Events":"s3:ObjectCreated:Put",
        "Filter":{
          "S3Key":{
            "Rules":[
              {
                "Name":"suffix",
                "Value":".jpg"
              }
            ]
          }
        }
      }
    }
 }
```

#### 3.3.1.3. General Resources

Depending on the resource, its dependencies can be generated by the infrastructure file of the serverless application similar to serverless functions or data storage resources. The dependencies of gateways, queues, etc. are usually described there. The structure of a state graph, like the one used in Amazon's

*Step Functions*, is also described in the infrastructure file. By analyzing the states of the state graphs, the dependencies of the serverless functions used within the state graph can be constructed. If the connection of a resource to another resource or a serverless function is not described in the infrastructure file, this information has to be saved somewhere else which can then be used for the generation of the dependency.

### 3.3.2. Annotations of the Model

Not only the structure of the model but also the annotation of the model can be created automatically, which is discussed now. Some settings of the platform can simply be read from the infrastructure file. Usually, if they are not set, the standard values of the platform provider are used, which need to be known. Typical settings include access rights, timeouts, and throttling limits. Some other settings have to be read from the source code files of the serverless functions. The order of calls, conditions, and the identification of multiple calls have to be constructed by investigating the source code file. If external factors influence the control flow or if the structure of the source code is challenging to parse, it is quite difficult to calculate the orders correctly. Usually, it is easy to identify if a call is made synchronously or asynchronously by analyzing the command doing the invocation. However, the quality of all annotations that can be created automatically depends strongly on the quality of the parser interpreting the source code and has to be set manually if needed. An overview where the artifacts of the model can be identified is shown in Table 3.1. As soon as the information of an artifact has to be created from the source code, an automatic detection of the artifact for the model is not guaranteed. The concrete artifact used, e.g., the relation to a resource called, can depend on data passed to the serverless function, e.g., if the name of a serverless function to be called is passed as a parameter to the serverless function.

### 3.3.3. Monitoring

There are several metrics that can be monitored and assigned to the nodes and arcs of the model as shown in 3.2.4 which include execution times of the functions, the usage of memory and computational resources used during their execution, costs, and the call history of the resources. To assign these metric data to the nodes, a mapping of the resources of the model to the data collected during the execution of the application is needed. Once such a mapping is completed, the runtime information, which can for example be read from log information, can be evaluated, and the relevant information can be assigned to the corresponding nodes or arcs of the graph. For instance, if a node is mapped to a specific log stream, the log stream can be analyzed and its execution

Table 3.1.: Source of information for automatic generation of artifacts

| | Infrastructure File | Source Code | Full Automation |
|---|---|---|---|
| Serverless functions Nodes | ✓ | | ✓ |
| Data storage resources | ✓ | | ✓ |
| Relation to data storage | | ✓ | |
| Data storage trigger | ✓ | | ✓ |
| General resources | ✓ | | ✓ |
| Relations to general resources | | ✓ | |
| General resources trigger | ✓ | | ✓ |
| Kind of data storage access | | ✓ | |
| Sync/async invocation | | ✓ | |
| Order of calls | | ✓ | |
| Multiple asynchronous calls | | ✓ | |
| Condition of calls | | ✓ | |
| Interfaces | | ✓ | |
| Data flow information | | ✓ | |
| Platform settings | ✓ | | ✓ |

times, number of invocations, and costs be saved in the node and dynamically evaluated in the context of the graph.

### 3.3.4. Dynamic Generation of Graph

The same type of model can also be generated by evaluating the log data produced during test cases or in production. This results in a dynamically generated model where different approaches are applicable. Instead of creating the graph statically, the entire graph can be created dynamically. This requires an instrumentation of the application which produces meaningful enough log data that capture the resources of the application, their relations, and annotations. Additionally, the application must be executed with a profile covering as many cases as possible to create comprehensive log data. However, it is difficult to create enough cases that encompass all resources, relations, and annotations that are needed to produce the whole graph. Therefore, some relevant information can easily be missed.

A dynamic approach can also be combined with a static one. The graph can be created statically and be expanded by relations and annotations which are hard or impossible to create statically. For example, if a serverless function calls another resource depending on the input data it receives, a dynamic approach can better identify the relevant resources called in the given scenarios. Furthermore, worst-case assumptions can be reduced by evaluating the application dynamically if enough cases are available for the corresponding component.

## 3.4. Proof of Concept

To demonstrate the feasibility of the approach, a tool[6] was written being able to construct a model of a serverless application run on *AWS*. The serverless application uses *AWS Lambda* which is the implementation of serverless functions for this platform. The implementation supports *AWS Lambdas* and Amazon's NoSQL database service *DynamoDB* and Amazon's object storage *S3* as services. Furthermore, it supports the detection of write/read accesses, order of calls, and synchronous calls as annotations in order to demonstrate their benefits.

### 3.4.1. Use Case of Application

The evaluated application can be seen as a model in Figure 3.15. It comprises a data storage resource $d_1$ that triggers a serverless function $n_1$ as soon as a file is added to the data storage resource. The serverless function is designed to compress the uploaded file and store it back into the data storage resource $d_1$. Additionally, hash values for both the original file and the new file are created by invoking synchronously the serverless function $n_2$. These hash values are saved on *DynamoDB* $d_2$ and used by another serverless function $n_3$ which can be used to get hash value for the user which was called asynchronously by $n_1$ after the execution of $n_2$. The serverless functions were written in *Java* and utilize an *S3* object store along with a *DynamoDB* database.

### 3.4.2. Model Creation Tool

The tool creates the graph with its relations statically by using the infrastructure file and the source code files of the serverless functions (see Code in Listing 3.3). By reading a file written in *AWS SAM*, a superset of *AWSCloudFormation*, which describes the AWS infrastructure, the basic model including serverless functions and data storage resources is created. This model is expanded by additional relations which can be read from the source code files of the functions respectively the infrastructure file. Finally, if there are already log data available, e.g., containing the execution times or memory usage, the information can be added to the graph.

Listing 3.3: Code for graph generation

```
Graph createGraph(String pathOfResourceFile, String[]
    pathsToSourceCodeOfFunctions,
                              String[] pathsToLogFile) {
    Graph graph = new Graph();
    createBasicModel(pathOfResourceFile, graph);
```

[6] https://github.com/snwinz/ServerlessApplicationTool

```
6         for (String code : pathsToSourceCodeOfFunctions) {
              addRelationsFromSource(code, graph);
8         }
          addLogData(pathsToLogFile, graph);
10        return graph;
      }
```

A parser is applied to the source code files which is able to identify the relations of the resources and their annotations by identifying certain keywords that identify when another resource is accessed, if its kind of access is a read or write operation, and if the call was made synchronously or asynchronously. Since the code is processed sequentially, the order of the invocation can be detected which requires that the code is not nested. The parser applied here supports basic detection mechanisms for Java files with a straightforward structure. After applying the parser, a basic model is available containing serverless functions and data storage resources and their relations indicating the access type, the order of call and if a call was made synchronously or asynchronously (compare Figure 3.15). So, the model complies with the Definitions 3.1, 3.2, 3.3, 3.5, and 3.7.

This model can be enhanced by data that were created during runtime, which can be extracted from a log file. If a simpler version of the graph is needed, this can be done afterwards by removing the unnecessary nodes and annotations. The concept of the tool can be transferred to other platforms by adapting the parser for the infrastructure files of the corresponding platforms.

### 3.4.3. Evaluation of Example Application

In this section, an evaluation is conducted on the graph generated by the model creation tool for the test application. A visual representation of the generated graph of this application can be seen in Figure 3.14 which was created by Amazon's *AWS CloudFormation Designer*[7] by using the *AWS SAM* file. The graph has less information and lacks representation of potential runtime behavior between the components since this file is mainly used for deployment and does not evaluate the source code of the serverless functions. Only the relation from the *S3* data storage resource to a serverless function is shown, while the relationship from the serverless function to the *S3* data storage resource for the compression of the object saved there is not shown. Furthermore, the relations to the other serverless functions creating hash values for the objects and saving the value on a *DynamoDB* data storage resource and the relation to the serverless function informing the client are not shown. So, the graph does not show when a serverless function accesses the *S3* object store and the

[7] https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/working-with-templates-cfn-designer-overview.html

*DynamoDB* data storage resource or when the serverless functions invoke each other.

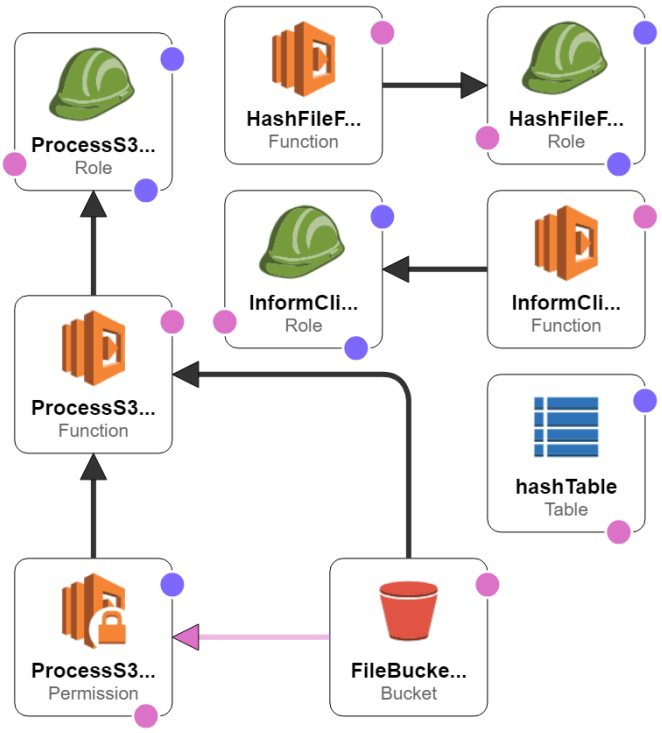



Figure 3.14.: *AWS SAM* representation of application in *AWS CloudFormation Designer*

In contrast to the visualization of Amazon's *AWS CloudFormation Designer*, the relations between the components are visualized in our model, which is depicted in Figure 3.15. Furthermore, the automatically generated model contains also information on the order of invocations, if data were read or written, and if the calls to other serverless functions are made synchronously or asynchronously.

Even for this very small application, there are several insights that can be detected by analyzing the graph, which are harder to detect otherwise:

- A subgraph of nodes $n_1$ and $d_1$ builds a cyclic graph, which could create an endless recursive call. So, if a file is added to the data storage resource $d_1$, $n_1$ is informed, reads the data, and uploads the compressed data again to the data storage resource. This would create another event handled by $n_1$ and so on. Therefore, settings have to be made for the data storage resource so that $n_1$ is only triggered for a certain kind of files or $n_1$ has to ignore compressed files.

- Another cycle can be found in the subgraph $d_1$, $n_1$, and $n_2$. Data of $d_1$ is read by $n_2$. However, this is not a problem since $d_1$ does not trigger an event when read accesses for *S3* data storage resources are made on this platform.

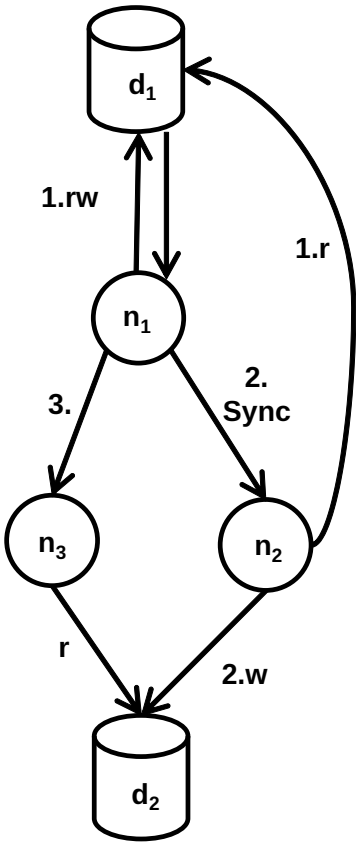



Figure 3.15.: Basic graph with some annotation of application to be modeled

- Both $n_1$ and $n_2$ access $d_1$, whereas $n_1$ also writes data. However, since the calls made by $n_1$ are ordered, the read access of $n_2$ is always after $n_1$ since the write access of $n_1$ is done synchronously (not shown in the model since this is the default for data store operations).

- $n_1$ reads and writes to a data store. Since there is no order given for this operation, it has to be ensured that these operations are either done in a fixed order or use different data, and the application behaves correctly if $n_1$ is called in parallel.

- $d_1$ is read and written in the application. It has to be ensured that the data are consistent if the application triggers $n_1$ in parallel after several files were uploaded at once.

- $d_2$ is written by $n_2$ and read by $n_3$. There is no race condition since $n_2$ is called synchronously by $n_1$. However, it has to be checked that the application behaves correctly if it is called in parallel.

- $n_1$ makes a synchronous call to $n_2$. Since $n_1$ has to wait for an answer (which can take some time since $n_2$ makes some data storage resource accesses), it has to be paid unnecessarily. Therefore, depending on its usage profile, it might be worth restructuring the application.

The example showed that it is possible to create the graph to some degree automatically and that the model introduced supports the identification of hot spots whose integration could be critical and behavior should be checked in more detail. Furthermore, it shows the most important typical flaws and potential problems that can be revealed by creating and analyzing the graph for even such a relatively simple application.

## 3.5. Related Work

Graphs for describing program behavior, both the static and dynamic properties of a program, have been well-known for a long time, e.g., control flow and data flow graphs. However, specific approaches tailored to serve serverless applications and their serverless functions are rarer and only available for the deployment process.

A similar approach for the creation of a model was introduced in [79] by Obetz et al., where the notion of an *extended service call graph* is introduced. This paper was published after my paper [108] and suggests a similar model as the one introduced in this chapter, where these components of the model of a serverless application are connected with interacting components. Furthermore, the paper introduces a tool where a static analysis of the components is done to identify the components of the application, similar to my approach described in 3.3.1. However, the tool of the paper does not identify dependencies to other components. This issue is tackled by Obetz et al. in a later paper [80], where the source code is also investigated to identify the dependencies, which complies with my approach in [108].
Additionally, Obetz et al. formalize the event-driven behavior of serverless applications by using operational semantics like Jangda et al. in [44], which is simplified in the graph introduced in this section by using a simple annotation as described in 3.2.3.2. Another formal approach was introduced in [35], where the interaction of serverless functions is formalized.

A component-based view for IoT architecture is introduced in [76] which was motivated by my work [110], the paper of Obetz et al. [80] and a third paper [47] by Jin and Offutt describing a general approach how testing criteria can be derived based on the architecture. While the component-based view for IoT architecture is also able to represent serverless functions, data storage resources, and other services, its focus is on IoT-specific information that is represented in the model. However, information on serverless functions like the kind of data storage resource access or where and how the data is used are missing.

In a bachelor thesis [99] supervised by the author of this work at the chair where he is employed, Sthamer analyzed *CAMEL* and *TOSCA* as cloud modeling languages to model serverless applications. However, these modeling languages focus on the deployment process of applications and not the logic, the control

flow, and the data flow between the components. *CAMEL* "describes itself as being similar to *TOSCA* with additional support during runtime" [99], whose "main goal is to enable the creation of portable cloud applications and the automation of their deployment and management" [17]. Extensions for serverless applications were introduced for *CAMEL* in [53] and for *TOSCA* in [114]. Moreover, Sthamer conducted a literature review to explore the models used to describe serverless applications. Only 9 of the 83 models he identified were written in a well-known modeling language, while only 30 models modeled the workflow of the serverless applications.

Leitner et al. introduced in [56] a graph-based model to depict the costs of microservice-based applications, including support for *AWS Lambda*. The model presented in this paper also relies on a graph but enhances it with information facilitating the simulation of total application costs. Lin and Khazaei also introduced a graph-based model for serverless applications to predict the response time and costs in [60]. Unlike the model introduced in this chapter, the models in [56, 60] require knowledge about the runtime behavior of the application and a service cost model. However, since both models are built on a similar graph representing the components, the model introduced here could help collect runtime data for costs and the workload as described in 3.2.4.3 and 3.2.4.4.

In [65], a service dependency graph for microservices is introduced, wherein services and their endpoints are depicted as nodes. Their tool uses *reflection* to obtain the service endpoints and service call information automatically. However, the approach is tailored to the architectural style of microservices and not of serverless applications whose components are normally smaller.

In [13, 14], an event sequence graph is introduced for modeling and testing the functional behavior of web services. However, this approach represents only the system behavior on a level of the interfaces of the web services and does not represent the components the system consists of.

Chan et al. introduced in [19] a graph where each node is a computing entity where more concrete details like services or IP addresses are assigned as attributes to the nodes. Predicates on the arcs between the nodes make attributes available from one node to the other. However, this graph does not represent a cloud application, but the computing entities provided in the cloud, which can be used and have to be used efficiently for incoming requests.

In [102], serverless functions are modeled using a dynamic Petri net model, which supports capacity planning for serverless functions. This model serves as a tool that can be used by developers to optimize the application time and costs.

Yussupov et al. introduced in [118] an approach to assess the portability of serverless applications, which is based on a deployment model. This deployment model, e.g., the deployment file of the *Serverless Framework*, is transformed into the *SEAPORTR CASE Model*, which is a generic serverless application

representation to abstract provider details away. In combination with the source code, this model can be used to assess the portability to other platforms. Furthermore, Yussup et al. introduced another vendor-agnostic modeling and deployment approach in [119], which relies on *BPMN* and *TOSCA*.

In [66], a performance model for serverless platforms is introduced. This model can calculate essential performance metrics that can estimate the performance of various workloads. It achieves this by modeling the components and their behavior with stochastic processes.

In contrast to the other models, the model provided in this chapter focuses not on deployment aspects but reflects the structure of the application with relevant aspects making it detailed enough to identify and visualize potential problems.

## 3.6. Limitations

The model introduced represents the application on the level of its components and the aspects which are characteristic of the serverless functions. It is not guaranteed that all aspects of the application are covered, but the model is flexible and can be extended with additional annotations at the resource level. For example, the annotations focus on the characteristics of serverless functions but also for other resources, like data storage resources, annotations can be added to the graph. Since the model aims to capture and model the most important aspects of the application, the model is only an abstraction of the serverless application

The tooling introduced in 3.4.2 is only able to create a model for serverless applications on *AWS* for *AWS SAM*, but the concept can be transferred to other platforms and infrastructure file templates by adapting the parser for the infrastructure files of the corresponding platforms.

Since the relations cannot be completely built from the infrastructure file, the source code of the serverless functions has to be parsed. The parser applied here supports basic detection mechanisms for Java files with a straightforward structure, but serverless functions can be implemented in many languages. Furthermore, the detection of relations in the source code is also limited since not all relations can be detected automatically from the source code. For example, if the name of a resource that is called is passed by an input parameter or the invocation depends on an input parameter, it cannot always be detected automatically which resource is called or if a resource is called at all (compare halting problem which is undecidable). So, both the missing information of the infrastructure files for relations and the difficulty of parsing these relations from the source code limit the applicability of the tool and is a drawback of this approach. However, if the application can be executed with a workload representing all use cases of the application, such a graph could also be created dynamically.

The example here shows only a small excerpt of the possibilities of such a model. However, the detection potential for problems depends on the use case and the representation of the application.

## 3.7. Summary

This chapter introduced an approach for the creation of a model that is designed for the support of serverless applications. The model is designed to represent the serverless application on the abstraction level of their components. A component, in this context, refers to a resource such as a serverless function or another service on the cloud platform that interacts with the serverless function. Potential annotations and information for a model were discussed in this section that can be assigned to the resources of a serverless application and which are potentially relevant to be modeled.

An approach was presented that implements the automatic creation of such a model based on the infrastructure file of the serverless application and the source code of the serverless functions. Based on a model of an example application. The evaluation potential of such a model was demonstrated based on an example application.

By using such a model, cloud providers could assist developers in various phases and help them produce systems of higher quality. In the following, the model and its abstraction level are used for the creation of potential coverage criteria and are used for the identification of potential control and data flows in serverless applications.

For the following work, the model is used as the representation of serverless applications and the basis for the testing process consisting of the identification of testing targets, the measurement of testing targets in an application, test case generation, and evaluation of the test cases.

# 4. Coverage Criteria

For testing the adequacy of systems, test cases are often generated which help to indicate if there are any faults within the system and if the system behaves as intended. Since there are usually infinite potential test cases for most systems, measurements are needed to check if the existing test cases are good enough and test the relevant aspects of the system. Serverless applications consist of several serverless functions combined with other services provided by a cloud platform provider which requires that their integration is tested properly. Therefore, the following chapter lists several coverage criteria that can be applied to serverless applications. The criteria focus on various aspects which can be tested. First, potential coverage criteria focusing on the control and data flow of the application for integration testing are introduced. Subsequently, criteria focusing on certain aspects of serverless applications, like their parallel execution of functions and limitations caused by their execution on the cloud are suggested and discussed. Furthermore, by modeling some example serverless applications, the data flow and control flow criteria are discussed. The content of this chapter is mainly based on my publications [107, 109–111].

## 4.1. Flow-based Adequacy

This section introduces some classical approaches for the coverage of applications based on the control flow and data flow and suggests some coverage criteria. They can be used for the evaluation of integration test cases of serverless applications.

The coverage criteria introduced here can be applied to the model that was introduced in the previous Chapter 3 which facilitates the testing process. So, existing test cases can be evaluated by mapping their execution to the model to see which components are tested. Furthermore, by evaluating the model, missing testing targets of the system under test can be identified, which can support the creation of missing test cases. The following section discusses how these criteria can be applied to a serverless application during integration testing and how they are supported by the model introduced in Chapter 3.

### 4.1.1. Control Flow Coverage

A classical testing approach, as already applied in [41], is the examination of an application where the control flow is represented as a graph. By building blocks

of statements that are mapped to individual nodes and connecting them with edges representing calls to the following statements, a graph can be created whose paths represent potential control flows of the application on the level of the components that can be invoked. An obvious solution for coverage criteria based on such a graph requires the coverage of all nodes, all edges, or all paths.

If the control flow is also considered at the code level for serverless applications, not only the integration of various services but also the structure within a function is tested which has to be known. This would make the test cases more complex and require code to be covered that should have already been tested explicitly in isolation during unit testing. Thus, since many parts of the application had to be tested at least twice, the size of a test case set would increase. Specifically, in a complex system where certain combinations of nodes must be tested, such as for the coverage of all paths, the size of test cases would increase tremendously. Some test cases would only be different in a small detail within a single node which does not influence any other component. Furthermore, the model representing the application would be too complex since not an abstract view of all the components is given. Therefore, building a graph on this fine-granular level produces a graph that is too complex and focuses not only on the relations between its components but also on the relations within its single components which is the primary focus of unit testing.

Therefore, the level of abstraction of the base model used for integration testing of the serverless application concentrates on the relations between the resources of serverless applications. Since individual serverless functions should have already been tested for their correctness in unit testing, relations between the resources are relevant which represent the communication between the resources whose integration has to be verified. Consequently, coverage criteria have to be adapted and introduced which can be applied on this abstraction level for serverless applications.

The model introduced in Chapter 3 represents a serverless application as a dependency graph and can be utilized to extract potential control flows. By using the dependency graph as a basis for the coverage, a model is available that is applicable to an existing system and can easily be adapted.

Taking inspiration from [64], where criteria are defined on module level, the coverage criteria as defined in Definition 4.1 are suggested for serverless applications which can be applied by using the previously defined model based on the Definitions 3.1, 3.2, 3.3, 3.4.

**Definition 4.1 (Control Flow Criteria)**

- *All-resources (AllRes) requires that every resource is executed at least once. A resource is the instance of a service like a serverless function or a data storage.*
- *All-resource-relations (AllRel) requires that every relation between resources is executed (e.g., all edges between the nodes are covered).*
- *All-resource-sequences (AllSeq) requires that every sequence of resources is executed (e.g., all paths of the graph are covered).*

By requiring the coverage of all resources, it is guaranteed that each resource is called at least once. This is particularly useful for continuous deployment to see if an instance that was added recently was deployed correctly and fulfills its basic functionality. An example is presented in Figure 4.1 where the criterion only requires that each resource is called at least once. Each resource that has to be covered is surrounded by a green circle. Consequently, it is not relevant who called a resource, for instance, if *fun2* was called directly or triggered by *db0* after *fun0* updated it, does not matter for the fulfillment of the testing target of the resource *fun2*.

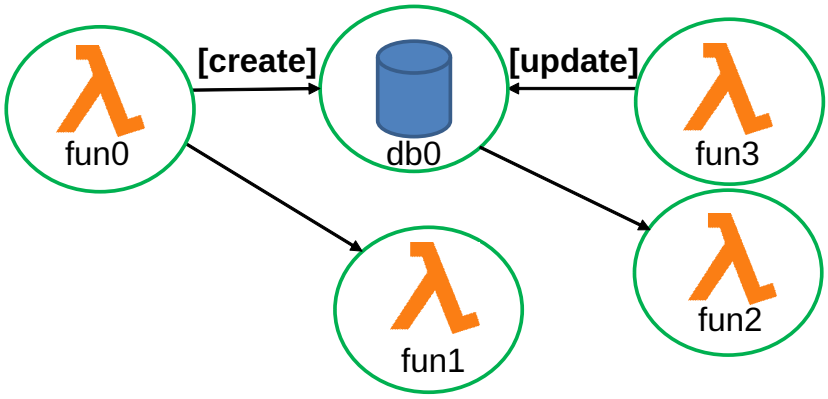



Figure 4.1.: Example for coverage of all resources

In contrast to *All-resources* (*AllRes*), coverage of all edges in the graph supports the testing of the communication of resources with their neighbors. This means that not only the individual deployment of resources is tested, but also the integration of each resource with all its neighboring resources is tested. In Figure 4.2 all the relations are marked green that have to be covered to fulfill this criterion. Even if some relations are also covered by *AllRes* when all resources are covered, not all relations are always executed. *All-resource-relations* (*AllRel*) now requires also that the call from *db0* to *fun2* is covered which is not always triggered when *fun0* or *fun3* is executed. However, for this criterion, it is not relevant who triggered the execution of a relation.

By using the model described, all potential workflows can be detected that are required to be covered to fulfill *All-resource-sequences* (*AllSeq*). Consequently,

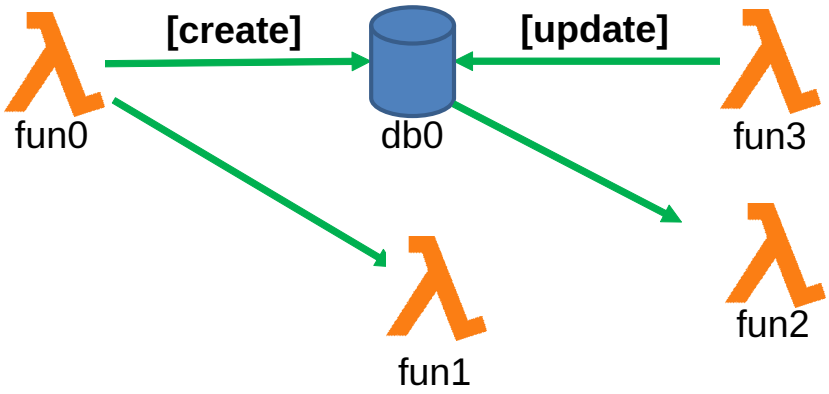



Figure 4.2.: Example for coverage of all relations between resources

it covers not only technical aspects but also the global context of the resource. Calculating all feasible paths can be challenging, and in some cases, impossible, especially if, for example, the graph contains a cycle. Applied to the previous example, all potential paths that are possible by invoking only one serverless function are highlighted as green arrows in Figure 4.3.

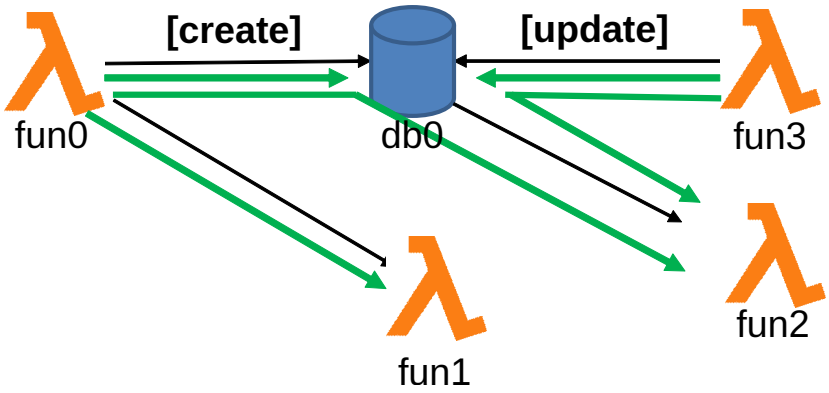



Figure 4.3.: Example for coverage of all paths between resources

If the *AllRes* criterion is to be supported by a model, the model must contain at least all resources of the serverless application. Similarly, if the *AllRel* should be supported by a model, all relations have to be modeled too. This is supported by the basic dependency graph shown in Section 3.2 if it is extended to support data storage resources and other general resources. However, if *AllSeq* has to be fulfilled, more information is useful to prevent the identification of paths on the graph as potential testing targets that are unfeasible. By adding information, like the call order of resources, constraints for calls, synchronous or asynchronous invocation types, the number of paths can be reduced.

Since *AllRel* does not require that resources without a relation are covered, *AllRes* is not subsumed by this criterion which requires that each single resource is covered.

### 4.1.2. Data Flow Coverage

The control flow criteria considered in the previous section focus only on the components of the application and the integration with other components However, they ignore the data used and passed between the components. By taking into account the data being used by different resources, the direct influence between the resources can be tested.

Typically, data needed by a serverless function are received through its parameters as input. However, if the complexity of a serverless application grows and more data are needed, these data are transferred by using data storage resources where the values are stored and processed later.

Similar to the control flow criteria, the data flow criteria consider interactions where several resources are involved. Thus, inspired by [64], data flow criteria are suggested in Definition 4.2 based on a model fulfilling the Definitions 3.1, 3.2, 3.3, 3.4, 3.5, and 3.9.

**Definition 4.2 (Data Flow Criteria *AllDefs* and *AllUses*)**
*If $x$ is a definition of a value within a serverless function that is used by another resource, then:*

- *All-resource-defs (AllDefs) requires that every $x$ is at least used once in another resource without being redefined before its usage.*
- *All-resource-uses (AllUses) requires that every $x$ is used by all usages of $x$ in other resources without being redefined before its usage.*

These coverage criteria can be applied to serverless applications and are relevant since data are transferred within a serverless application because of the statelessness of the serverless functions.

In order to use the model introduced in Chapter 3 for these coverage criteria, data flow has to be supported which is described in 3.2.3.6 where the graph is extended by assigning the definitions and uses of the data transmitted to the arcs of the graph.

A potential dependency graph is presented in Figure 4.4 where the data flow information is attached to the arcs. Data flow can occur between functions like between *fun0* and *fun1* where a potential definition and its usage is assigned to the arc. Additionally, data flow is also possible between serverless functions where another service is between them like a data storage resource. Here, *db0* keeps data which were defined in *fun0* or *fun3*. These data are used in *fun2* or *fun4*. The serverless function *fun2* is potentially triggered when data are written to *db0*. Therefore, no read annotation is needed here since the event triggered here can be interpreted as read access where data are passed to *fun2*.

In order to fulfill *All-resource-defs* (*AllDefs*), the definitions of *fun0* "def_line11" and of *fun3* "def_line12" have to be executed with a corresponding use. The definition "def_line11" of *fun0* is covered if the value is used in *fun1*, *fun2* or

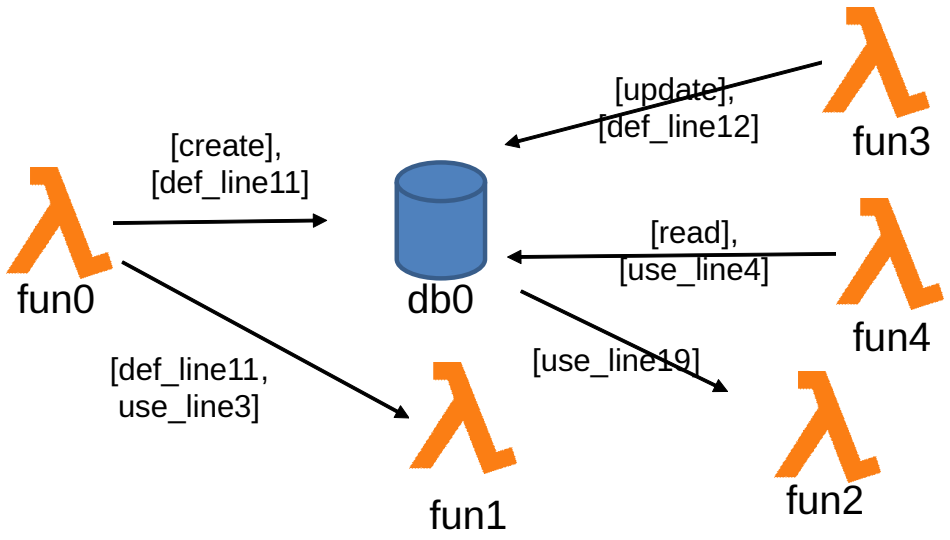



Figure 4.4.: Example for a model representing the data flow of a serverless application

*fun4* with the corresponding use, while the definition def_line12 of *fun3* can be used in *fun2* or *fun4*.

In contrast to *AllDefs*, *All-resource-uses* (*AllUses*) considers the coverage of uses explicitly. This results in a potentially higher number of test cases needed to fulfill this criterion as can be seen in the example shown in Figure 4.4.

The maximum number of test cases required for both criteria to be fully fulfilled is listed in Table 4.1 assuming each testing target is covered by only one test case where $I$ is the set of couplings (e.g., couplings via data storage resources or function invocations) and $D_i$ are the definitions and $U_i$ the uses of a coupling $i \in I$.

Table 4.1.: Testing targets of criteria

| Criterion | Maximal Number of Testing TargetsF |
|---|---|
| All-resource-defs | $\sum_{i \in I} \lvert D_i \rvert$ |
| All-resource-defuse | $\sum_{i \in I} \lvert D_i \rvert + \lvert U_i \rvert$ |
| All-resource-uses | $\sum_{i \in I} \lvert D_i \rvert \cdot \lvert U_i \rvert$ |

Serverless functions are often coupled externally by communicating through an external data storage resource like in Figure 4.4. When there is a data storage resource with many functions writing and reading data from it, and each test case is responsible for only one testing target, the number of test cases to fulfill *AllUses* can become significantly higher. Therefore, another coverage criterion was introduced by me in [111]. While the fulfillment of the *AllUses* criterion requires that all def-use pairs are covered, the weaker criterion *All-resource-defuse* requires only the coverage of all definitions with any use, and the coverage of each use with any definition as defined in Definition 4.3 based on a model created according to the Definitions 3.1, 3.2, 3.3, 3.4, 3.5 and 3.9.

**Definition 4.3 (Data Flow Criteria *AllDefUse*)**
*All-resource-defuse AllDefUse requires that:*

- *each definition of a value within a serverless function that is used by another resource is used at least once in another resource without being redefined before its usage.*
- *each usage of a value defined in another resource is used at least once in combination with any definition without being redefined before its usage.*

This criterion simply verifies that each use and definition was at least used once in combination with any definition respectively use and does not require that each combination is tested as in *AllUses*. For example, if a serverless function that uses a value of a data storage resource is added to a serverless application, only one additional test case is needed for the coverage of the use and any corresponding definition. However, *AllUses* required that each combination with all definitions has to be tested, which, particularly for complex systems, can significantly increase the number of required test cases.

Furthermore, *AllDefUse* is at least as strong as *AllDefs* since it requires also that all uses are at least called once in combination with a definition in contrast to *AllDefs*. So, the three test criteria result in the subsumption hierarchy as illustrated in Figure 4.5.

Since the testing takes place on the integration level, the criteria consider the general flow but can also be refined. For instance, considering also subsequent uses of values already used or all attributes of the data passed or even all records of a data storage resource, like in [51], would extend the criteria and make them more precise but less applicable. Furthermore, the criteria can also be refined by differentiating between the cause of the kind of use. While a *p-use* uses the value in a condition and reads only the value, a *c-use* uses the value for a computation whose result is used, e.g., assigned to another variable. The

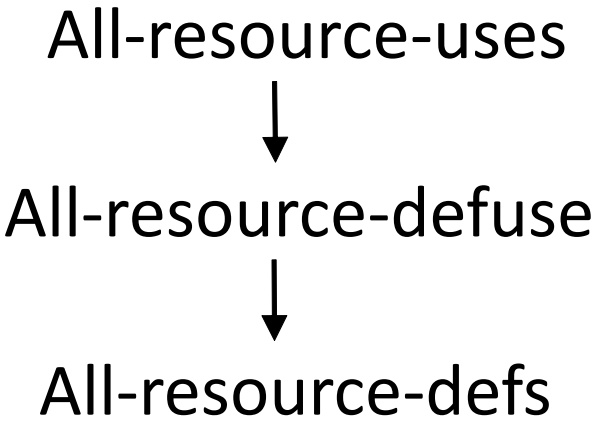



Figure 4.5.: Subsumption hierarchy of data flow criteria

criteria here just require that the value is used at all and do not differentiate between them but require both to be covered.

Since update and delete operations of data on data storage resources can transfer data to other serverless functions by changing the state of the data storage, these operations can also be considered as definitions. Therefore, a data flow is coupled via a data storage resource when a serverless function defines data, saves these data on this data storage, and finally these data are read and used by another running serverless function. However, while an update operation can create data on a data storage resource that can be used later, a delete operation can only change the state of the system if data are already saved on the data storage. Therefore, a data flow is considered to exist only if already available data are deleted and an attempt is made to access them later.

## 4.2. Adequacy of Characteristics

In addition to the control and data flow criteria introduced in the previous section, this section shows how various characteristics of a serverless application can be used to assess the adequacy of test cases and how this can be supported by the model. These criteria are intended to be used additionally to the existing test cases in order to gain confidence on the aspect the criterion focuses on.

### 4.2.1. Parallelism

Having instances of the same serverless functions running in parallel is characteristic of serverless applications since the cloud platform provider can decide independently when to start new instances. However, parallelism can cause errors which have to be detected. This is discussed in this section.

Figure 4.6 represents a model where two serverless functions *fun1* and *fun2* access the same database. Since both serverless functions are triggered

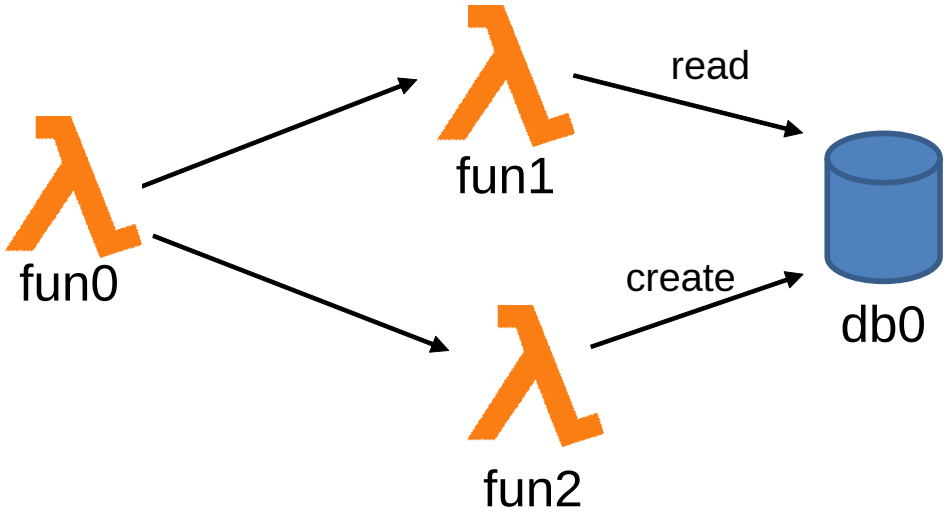



Figure 4.6.: Example for a model representing the data flow of a serverless application

asynchronously by *fun0*, it is not clear which serverless function accesses the database first. If both functions access the same data, it has to be tested if the system behaves correctly even if the order of access is switched. This race condition is caused by the workflow which does not assure which access to the data storage resource occurs first.

Additionally, since the cloud platform provider decides when to start new instances running the serverless functions, it is also possible that several instances of the same serverless functions accessing a data storage resource are running simultaneously. For example, this can happen, if the execution of a serverless function fails which makes the cloud platform provider restart the serverless function. Additionally, this scenario can occur if a serverless function is called several times, for example, if there is a high workload by users calling *fun0* in the example.

Such a scenario affects all data storage resources where data are written and read at least once. Therefore, these hot spots are easy to detect in a model if read and write accesses are available for the data storage resources modeled by simply considering the predecessors of a data storage resource.

In order to test the criticality of the hot spots, testing read and write accesses to the data storage resource in all possible orders on the same data increases the confidentiality of the system. This should also include the context of the workflow. A typical error is a lost update where data are read first from a data storage resource and are updated based on these data. If another function updates the same data in between these operations, this update is lost. Such a potential situation can be detected statically by considering the dependency graph. If such a situation is detected and considered valid, test cases should at least cover the lost update scenario in order to verify the correct behavior.

Since workflows can easily be triggered in parallel, all operations writing data should at least be executed twice on the same data in test cases and the data should be read afterwards to check if the data are still valid. If an

operation is implemented as idempotent, executing the same operation twice has no consequences. However, if the operation is not idempotent, a potential error can be triggered by such test cases.

Testing the hot spots for all possible orders requires an identification of all these situations. This can be done by simply checking the data storage resources and the kind of access to them. More complicated is the implementation of a framework that executes the serverless functions in different orders and ensures access to the data storage resources in different orders, in particular, if the accesses are part of a more complex workflow involving multiple serverless functions.

### 4.2.2. Execution Time

There are several factors influencing the execution time of serverless functions that can be considered for testing and the creation of criteria. The cold start time required for provisioning a container where the serverless function can be executed, if the function is not already loaded [70], is characteristic of serverless applications. Therefore, some executions of serverless functions take longer if a new runtime environment is deployed for the request to a serverless function.

But the execution time is also influenced by the machine to which the serverless function is assigned to run. Even if the developer configured that there is not much computing power and memory assigned to a serverless function, the platform provider can decide to load the container of the serverless function to another machine which is ideally faster in the execution of the container. This results in unpredictable execution times. Therefore, if the application is tested on a faster or slower platform than is actually used in production, potential failures can be obfuscated. These failures are either time-out errors resulting from a longer execution time or race conditions, where the ordering of the workflow is disturbed by different execution times. Thus, race conditions can occur when data are used by several workflows where at least one writes data.

Therefore, the following coverage criteria for testing a serverless application are suggested.

- *All-min-time* requires that every serverless function has to be executed with its minimum time at least once.

- *All-max-time* requires that every serverless function has to be executed with its maximum time at least once.

- *All-max-min-time* requires that every serverless function has to be executed with its minimum and maximum time at least once.

In order to apply the criteria, a dynamic measurement of the time behavior of the functions is necessary after the deployment. Additionally, time profiles are needed to categorize the times measured, especially to estimate the minimal execution time.

While the *all-min-time* criterion can probably be achieved by warming up the serverless functions and calling them without any additional delay by assigning them to the fastest machines on the cloud platform, *all-max-time* can be triggered by adding a delay at the end of the function execution which delays for the maximal possible time before a time-out exception is thrown. If there are background processes running within the container, an additional delay at the end of the function execution can help to finish these executions and therefore lead to another behavior.

Furthermore, the criteria could be refined by adding the delay before calls to other services. This refinement aims to create more varied scenarios in the presence of race conditions

Since the different execution times might also be influenced by the context of the workflow in which the serverless functions were triggered, the serverless functions can also be tested on a finer level by measuring the time profile for a certain context (e.g., a certain workflow).

However, these criteria consider only how the serverless functions are executed and not their context. In order to identify potential errors, the context of the execution is relevant. The criteria can be fulfilled by calling each serverless function individually without considering a potential combination of serverless functions where the timing could cause a change in the behavior. Therefore, it makes sense to combine these criteria with relevant test cases, such as end-to-end test cases, where several serverless functions are called, increasing the likelihood of provoking unintended behavior. Applying the *all-max-time* criterion on end-to-end test cases can also help assess if the request is handled fast enough.

### 4.2.3. Access Rights

Security is a crucial aspect of a serverless application. Access rights are assigned to serverless functions regulating that only resources are accessed and used that are actually needed. Thus, security breaches are intended to be prevented by only assigning these rights which are actually needed by a serverless function. This becomes especially relevant when the system is integrated with other components. It is generally easier to grant a function more rights than necessary for the system to operate. Therefore, it is important to check if the access rights assigned to a serverless function are really necessary.

Thus, a simple test for the adequacy of test cases requires that each right assigned to a serverless function is used at least once. However, this requires a fine-granular hierarchy of potential access rights assignable to the serverless

functions. Otherwise, if there is the possibility to grant a serverless function all available rights, this single right can easily be covered by just calling the function once. Therefore, a hierarchy of assignable access rights has to be modeled, which is as coarse as needed but as fine-grained as possible.

This coverage criterion does not test the functionality of a serverless application but the security. During the creation of test cases for the fulfillment, access rights might be identified which were missing. However, if no test case is created that uses a certain access right, it is still not guaranteed that this access right is not needed.

In order to test the access rights, the possibility to track the usage of the access rights is necessary. This has either to be supported by the cloud platform provider or implemented manually. However, the latter requires a more extensive modification of both the cloud platform and serverless application in order to precisely observe the usage of access rights.

### 4.2.4. Error Handling

When moving the execution of the application to a cloud platform provider, the responsibility for handling errors also shifts to the platform provider. In case of a thrown error of an asynchronously invoked serverless function, the cloud platform provider typically attempts to reexecute the function several times. However, the reexecution of a serverless function can be problematic, especially if the serverless function is not idempotent.

Not only errors caused by the logic of the function but also errors thrown by the cloud platform provider can occur. Two specific types of errors for serverless functions are the shortage of a resource (e.g., memory) used by the serverless function and exceeding the time limit assigned to the serverless function. Since it cannot be guaranteed that these errors never occur for a serverless function, it makes sense to test whether the system can handle such an error if it occurs.

Therefore, the following coverage criteria are suggested:

- *All-restored* requires that each serverless function fails at least once and is successfully reexecuted.
- *All-failed* requires that each serverless function fails at least once and fails in all of its reexecutions.
- *All-failed-and-restored* requires both all-restored and all-failed to be fulfilled.

Only the serverless functions of the application need to be known for these criteria, which is given by the basic model introduced in 3.2. *All-restored* simulates the reexecution of a serverless function. Here, it is crucial to understand which statements of the serverless function were executed before its failure. If the serverless function has already communicated with other

resources or triggered some events, it is more likely that a repeated execution of the serverless function reveals some unintended behavior.

Similarly, for *All-failed* it is relevant which other components have already been triggered. Since the serverless function does not influence the system state when no other component was called and is designed to work if all of their dependencies were called, calling only a part of the dependencies would be more promising. For instance, if a serverless function writes data to two different data storage resources, the failure before the second access is more promising since this could potentially provoke an inconsistent state of the system. By requiring that the following calls of the serverless functions also fail, the aim is to test if the system remains in a consistent state even if the function was reexecuted by the cloud platform or the user.

## 4.3. Static Model-based Analysis

This section creates models of applications found on *GitHub* and creates potential test cases based on these models without considering the concrete execution behavior or the implementation of the application. As an indicator of their potential and their strength, it is shown which parts of the applications are covered by criteria in common and which parts are not necessarily covered by some criteria. Therefore, some models of applications of serverless applications on *GitHub* are selected and based on their model the coverage potential of the criteria is discussed.

### 4.3.1. Selection of Real World Applications

Several real-world applications were used in order to investigate the applicability of different coverage criteria for serverless applications. The selection process of the applications is described here. The models support the identification of interactions between components to see whether the different coverage criteria also cover these parts.

The program used for the identification of the projects, an overview of the projects identified, the models created, and the data generated can be found online on *GitHub*[8]. Furthermore, the models are visualized in Appendix A.

As the probably most popular publicly available code hosting platform, *GitHub* was searched for applications that are publicly available. The focus was on the most popular cloud computing platform *Amazon Web Service* (AWS) and its usage with the *serverless framework* which is used to deploy the resources used in a serverless application. The framework uses the file with the name 'serverless.yml' to describe the resources of the application. The file name was used as a filter for the search on *GitHub*. Additionally, the filter was configured

[8] `https://github.com/snwinz/ServerlessApplicationIdentification`

to select only files containing the keywords 'AWS', 'functions', 'handler', and 'events' in order to filter only these projects which use the FaaS implementation *AWS Lambda* of Amazon's ecosystem.

The search, conducted on September 18, 2019, utilized the *GitHub* REST API and successfully identified 1000 files which is the maximum number of files that can be returned by the interface. The REST API 100 returns results per request whose answer is paginated. Therefore, 10 different endpoints had to be called to get all the 1000 results. Each request returns the total number of files that are searched which was different for some endpoints if it was called for the first time. In order to reduce the probability that the endpoints worked on different data, all endpoints were called several times until all endpoints returned the same number of total projects.

These files are part of 909 different projects. Since the number of projects changes on *GitHub*, a repeated search now would probably give another set of results. The projects were sorted by the number of stars given which is a rough indicator of their popularity. Projects not representing standalone serverless applications, e.g., incomplete or deprecated projects, projects only offering boilerplate code, projects not written in English, or projects being part of the deployment process, were ignored. The first ten projects fulfilling these criteria were taken in order to avoid bias, and we used them for the investigation here. *Node.js* was used by eight of these projects as a runtime environment for the serverless functions, whereas *Kotlin*[9] was used by two applications as programming language.

Furthermore, for this work, three other applications were additionally analyzed which were used for the evaluation in Chapter 7. These applications use also *Node.js*.

### 4.3.2. Modeling of Applications

A model was built for each of the identified applications by applying the approach described in Section 3.3.1 and [79] according to the Definitions 3.1, 3.2, 3.3, 3.4, 3.5, and 3.9. This approach was utilized by analyzing the infrastructure file 'serverless.yml' applied by the *serverless framework* to identify the resources used in the application and to get the dependencies between them. Furthermore, the source code of the serverless functions was analyzed manually to identify additional dependencies to external services and cloud platform-specific services that are used. Thus, the dependencies between the resources that facilitate the identification of relevant interactions could be modeled.

The relevant resources identified to be modeled were serverless functions (here AWS's *Lambda functions*), data storage resources (here *Amazon DynamoDB* and *Amazon S3*), and queues (here *Amazon Simple Queue Service*). Gateways

[9] `https://kotlinlang.org/`

were identified to handle HTTP requests (here *Amazon API Gateway*) but were not modeled explicitly since they are tightly coupled to the serverless functions. Furthermore, calls made to cloud-platform-specific services were also modeled. These resources were modeled as nodes of a graph in the model, whereas the calls made to these cloud-platform-specific services were modeled as arcs. If a serverless function was triggered by an event created by a resource, this was also modeled as an arc from the resource to the serverless function.

As discussed in Section 3.2.3, annotations were added to the arcs to provide additional information about the application. For calls made to data storage resources, the annotations were extended to indicate if they were used for reading or writing. If the data storage resource was used for writing, a more detailed description was added by describing if the access creates, updates, or deletes data. This additional information aimed to provide a better overview of the data storage resource interactions.

The applications consisted mainly of resources provided by the cloud platform provider. The integration of resources from other providers is usually not supported natively. These resources have to be accessed by an interface, which is mostly an HTTP interface. In general, accessing the resources directly within the same ecosystem is faster. The number of resources used in the applications investigated are listed in Table 4.3.2 where $A11$, $A12$, and $A13$ are the applications evaluated in Chapter 7. The resources displayed comprise data storage resources, queues, and HTTP gateways. Furthermore, the number of relations between the resources is listed. Since some serverless functions access the same resource several times, for example, for reading and updating data, the number of interactions between the resources is higher. If an interaction with a third party takes place, this is also listed. This occurs, for example, if an external billing service was used. Furthermore, the number of platform services used which are not already recorded in another column is listed.

| | Serverless Functions | Data Storages | Queues | HTTP Gateways | Internal Relations | Internal Interactions | Third party Interactions | Platform service Interactions |
|---|---|---|---|---|---|---|---|---|
| A1[10] | 5 | 2 | 0 | 3 | 7 | 7 | 2 | 0 |
| A2[11] | 11 | 2 | 0 | 10 | 10 | 13 | 0 | 5 |
| A3[12] | 6 | 1 | 0 | 6 | 5 | 5 | 1 | 0 |
| A4[13] | 13 | 3 | 0 | 12 | 24 | 36 | 0 | 5 |
| A5[14] | 4 | 1 | 0 | 4 | 4 | 4 | 0 | 0 |
| A6[15] | 6 | 2 | 0 | 5 | 7 | 7 | 2 | 0 |
| A7[16] | 3 | 1 | 0 | 2 | 4 | 5 | 0 | 0 |
| A8[17] | 3 | 1 | 1 | 1 | 4 | 7 | 1 | 0 |
| A9[18] | 2 | 1 | 1 | 0 | 4 | 5 | 2 | 0 |
| A10[19] | 6 | 1 | 0 | 5 | 6 | 7 | 2 | 4 |
| A11[20] | 15 | 2 | 0 | 15 | 17 | 18 | 1 | 0 |
| A12[21] | 6 | 2 | 2 | 3 | 12 | 15 | 0 | 1 |
| A13[22] | 19 | 3 | 0 | 19 | 56 | 82 | 0 | 0 |

Table 4.2.: Resources used in the applications

[10] `https://github.com/serverless/scope`

### 4.3.3. Control Flow

The control flow criteria defined in Section 4.1.1 use a dependency graph and its structure to determine the coverage of the coverage criteria defined. The criteria demand that the graph should be covered systematically. In contrast to classical programs, serverless applications are event-driven based on serverless functions. The applications run in the cloud and use several services provided by a cloud platform provider or a third party which makes it harder to identify the parts of the application that are relevant for its integration. Therefore, the discussion here focuses on how the criteria support integration testing.

#### 4.3.3.1. All-resources

This criterion requires that each unit of the dependency graph has to be used in order to fulfill this criterion. So, it can be ensured that the components were deployed on the cloud platform. However, the coverage of resources does not guarantee that the integration with other resources works properly or that interactions with other resources work as intended.

Test cases initialize the workflow of the application by calling a gateway connected to a serverless function or by calling a serverless function directly. For the test cases, it was assumed that a gateway is always used if available, considering both the gateway and the serverless function as one component which has to be covered. Therefore, all HTTP gateways are covered if all serverless functions were covered. To cover other resources, such as a data storage resource, a serverless function has to be called which accesses this resource.

Since the serverless functions are executed by the cloud platform provider, their execution can be stopped while running. This happens when functions run longer than set in their configuration on the cloud platform or use more memory than allowed. Therefore, it is not guaranteed that a function executes all statements if it is invoked. So, if a serverless function is invoked and fails, the resource is considered covered even if not all statements were executed, potentially missing calls to other relevant resources.

[11] https://github.com/aws-samples/aws-iot-chat-example
[12] https://github.com/AnomalyInnovations/serverless-stack-demo-api
[13] https://github.com/tqhoughton/chat-app-serverless
[14] https://github.com/aletheia/serverless-url-shortener
[15] https://github.com/serverless/forms-service
[16] https://github.com/nicholasgubbins/Serverless-Image-Resizer
[17] https://github.com/spinscale/serverless-owntracks-kotlin
[18] https://github.com/a0viedo/slackper
[19] https://github.com/ajurasz/ascii-less-gallery
[20] https://github.com/millslm/Jackal
[21] https://github.com/debrajpaul/serverless-cake-ordering-system-
[22] https://github.com/anishkny/realworld-dynamodb-lambda

The minimal number of test cases fulfilling a $100\%$ coverage of the resources for the criterion on the models was created considering that the function execution might fail. The coverable resources were all part of *Amazon's* ecosystem.

While Table 4.3 lists the percentage of dependencies with other resources that are covered if all resources are covered, Table 4.4 shows the percentage of the interactions that are covered.

A dependency simply indicates if there is a relation from one resource to another, for example, by triggering or invoking it, or by accessing some data. It does not indicate how often the resources are accessed. An interaction exists for each interaction from one resource to another resource, e.g., if a serverless function can be called several times from another serverless function, for each call, there exists an interaction, but in total, there is only one dependency between the resources.

If the criterion *All-resources* is fulfilled, more than $20\%$ of all the $160$ dependencies of all example applications are covered while less than $16\%$ of all the $211$ interactions are covered. In particular, for applications with many serverless functions accessing the same resource, like application $A13$, many dependencies and relations that should be covered are missing.

#### 4.3.3.2. All-resource-relations

Since the integration of different resources is located on the relations between the resources, the *AllRel* criterion requires that all arcs of the graph are covered. Depending on the representation as a dependency graph, a criterion fulfills this requirement if each arc of the model is covered. Therefore, it is important to clarify how many relations between the same nodes are possible. In the models, only one arc between nodes was used indicating that a component initiates the interaction with another component. This makes the graph more readable and represents that there is, in general, a dependency between these resources which has to be tested. By adding annotations to the arcs, the information is given that there are different ways of interacting with the other components. If the criterion *AllRel* is fulfilled without considering alternative interactions, not all interactions are necessarily covered. Table 4.5 shows the minimal percentage of interactions if only all dependencies are covered. So, for all the $211$ interactions of the applications, only $135$ ( $65\%$) are guaranteed to be covered if all relations are covered.

In order to verify a proper integration of all relations used, the coverage of each interaction is more suitable. The number of interactions depends strongly

Table 4.3.: Minimal coverage of dependencies if *All-resources* is covered

| A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | A9 | A10 | A11 | A12 | A13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.28 | 0.2 | 0.2 | 0.16 | 0.25 | 0.28 | 0.25 | 0.5 | 0.75 | 0.5 | 0.11 | 0.58 | 0.053 |

Table 4.4.: Minimal coverage of interactions if *All-resources* is covered

| A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | A9 | A10 | A11 | A12 | A13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.28 | 0.15 | 0.2 | 0.11 | 0.25 | 0.28 | 0.2 | 0.28 | 0.6 | 0.42 | 0.11 | 0.46 | 0.036 |

on the modeling of the dependency graph. Thus, more detailed approaches could be implemented, covering even more interactions. Therefore, in the following work, the applications are modeled to represent all interactions. So, the coverage of all interactions is required and enabled on such a model.

#### 4.3.3.3. All-resource-sequences

By covering only the interaction between two resources once, consequences caused only under certain circumstances or in a later stage of the application are not necessarily revealed. Therefore, by covering all paths of resources that can be called in a workflow, not only the interaction between two resources but also the interaction between several resources can be tested. This increases the variety of data exchanged during the interaction and checks also later states of the execution where the error might become visible.

However, for the applications investigated, the paths are very short. There were only three applications with workflows that contained more than two resources where the longest workflow contained five resources. This criterion subsumes the *AllRel* criterion and needs only additional test cases if workflows exist where the length of a workflow contains more than two resources. Therefore, the applicability of this criterion is limited, especially since the workflows of the applications investigated don't have any branches. This limits the variety of test cases. Furthermore, even if the *AllRel* criterion focuses on the coverage of single relations, most relations being part of a long workflow cannot be executed without triggering another relation.

However, if an application reflects complex business workflows that are more connected, the usage of this coverage criterion would generate more coverable paths.

The serverless applications investigated are mainly designed around data storage resources like in the example in Figure 4.7. The data storage resources are accessed to read and write data without triggering any other events. Additionally, the kind of data access is assigned to the relations and the location of

Table 4.5.: Minimal coverage of interactions if *All-resource-relations* is covered

| A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | A9 | A10 | A11 | A12 | A13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1.00 | 0.76 | 1.00 | 0.66 | 1.00 | 1.00 | 0.8 | 0.57 | 0.8 | 0.85 | 0.94 | 0.86 | 0.36 |

the usage. If there are several interactions between resources, an additional arc can be added between them representing the interaction.

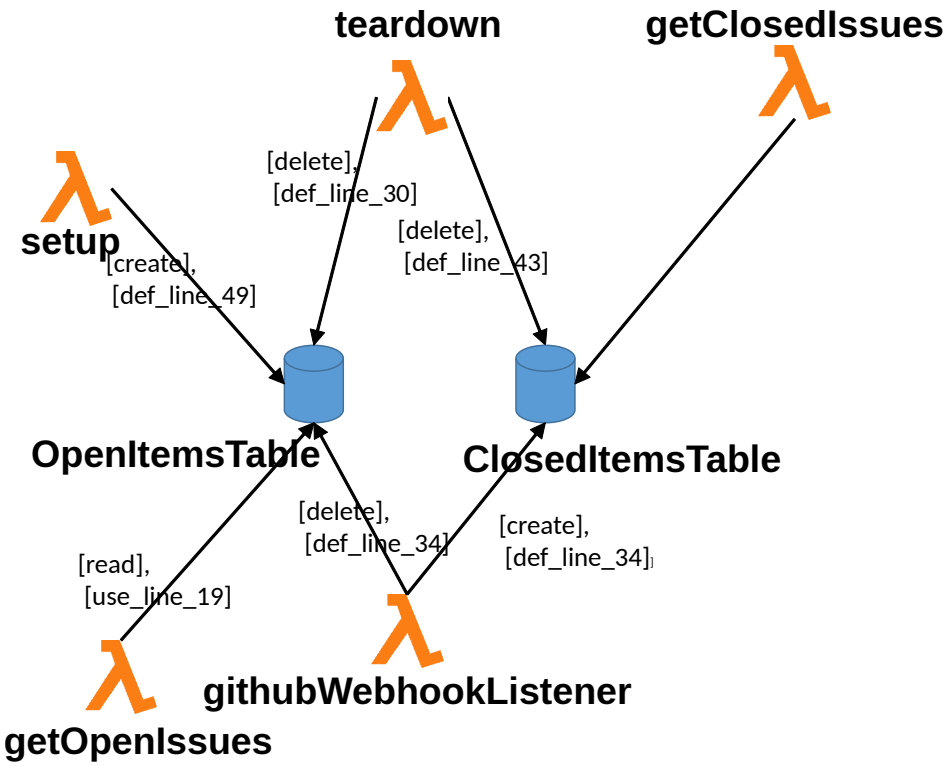



Figure 4.7.: Example of application *A1* with data storage resources at its center

There is only one scenario in the applications investigated where a function calls another function directly, and only three applications with a workflow with more than two resources in the same workflow. Therefore, the number of potential paths is limited. As for the *AllRes* criterion, the number of coverable parts can be increased by considering more fine-grained interactions between the resources. However, not every path that can be constructed is executable in the application, particularly when loops are involved, e.g., by functions being called recursively. The creation of executable paths can be supported by adding relevant information about the control flow to the nodes, like setting boundaries for the loop.

Most of the applications offer several interfaces to call the application. By executing several of these interfaces in a single test case, interactions can be tested where several workflows access the same data. However, the number of potential paths which can be built this way is not limited. Therefore, the number of paths that should be tested in this way should be chosen systematically and is not part of this criterion. Since the number of test cases for *AllSeq* is hard to calculate, it is not considered in more detail in the following work. In order to cover only these paths which are connected in some way, data flow criteria can be used which are discussed in the following section.

### 4.3.4. Data Flow

Considering the control flow of an application doesn't ensure that the data being processed in one serverless function is used by another serverless function and influences its behavior. If the data flow of the application is considered, the consequences caused by a resource can be tracked and checked how the data influenced the execution. Even though values can be passed from one function to another function, as usually done in procedural programs, values are passed directly only once in the applications investigated here. The values created and used by other functions were stored in data storage resources which can be considered as a global memory being used to pass values. Therefore, the data storage resources are part of the data flow and test cases have to use the values stored there to check the proper integration of the resources. Other resources like queues can also be considered as data storage resources which save data until they are accessed and used.

#### 4.3.4.1. All-resource-defs

When applying this criterion to the applications, a definition and a use of the value must be detected. A definition of a value that is used in other resources can either be passed directly, such as when a function is invoked, or by writing values to a data storage resource which are later read. The usage of a value can be interpreted as using the passed value or reading the value from the data storage. Reading the value from a data storage resource includes both direct access to the data value and the access to information using this data, e.g., a set of data containing the corresponding value.

If data are passed via a data storage resource, the data are either created, updated, or deleted. The definition does not explicitly handle the case if a value is deleted from a data storage. Therefore, since the deletion is a write access, it is demanded that a deleted value is tried to be accessed at least once after its deletion to fulfill this criterion. In the applications investigated, there were only two workflows where data were stored in a data storage resource and accessed within the same call afterwards. However, if several workflows of the application are used in combination, there are many cases where the result of a workflow is influenced by another one. Thus, test cases can be created that test the data flow. The minimal number of test cases needed to fulfill this criterion where each test case focuses only on the coverage of one definition as testing target is listed in Table 4.6. However, since the criterion focuses

Table 4.6.: Testing targets needed to fulfill *AllDefs*

| A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | A9 | A10 | A11 | A12 | A13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 2 | 3 | 19 | 2 | 5 | 2 | 5 | 3 | 1 | 12 | 5 | 13 |

only on data flows whose data are used within the application, data flows to other services that are not used by the application itself are not required to be tested. Therefore, some interactions are not necessarily covered by this criterion. Table 4.7 shows the number of interactions that are at least covered if the criterion is fulfilled. At least $136$ ( $64\%$) of all the $211$ interactions of all applications are covered if the criterion is fulfilled covering all definitions.

#### 4.3.4.2. All-resource-uses

The criterion *AllUses* is stricter by demanding that all serverless functions using the data defined elsewhere have to use the data at least once. For most of the applications, a test case has to make at least two requests to the application to cover such a single def-use pair. However, not all serverless functions access data. Such serverless functions are not covered by this criterion and are not covered at all. Table 4.8 shows the number of testing targets needed for the applications to cover this data flow criterion where each definition and use has to be covered individually. If a data entry is read by several serverless functions, the number of testing targets can increase drastically. Thus, each entry has to be read for each write access of a function which can lead to a drastic increase of test targets needed to fulfill this criterion. In particular, applications $A4$ and $A13$ need many test cases because of their variety of read accesses. If each testing target is covered in a separate test case, the number of test cases is bigger than for *AllDefs*. Here, for all applications $502$ testing targets have to be covered compared to $77$ where four applications do not have more testing targets caused by their limited number of read accesses to the data. However, the additional number of testing targets results also in the coverage of additional interactions which is shown in Table 4.9 which increases drastically compared to the interactions covered by *AllDefs*.

#### 4.3.4.3. All-resource-defuse

The insights gained from the two criteria inspired the creation of the third criterion which was already introduced in Section 4.1.2. Since the testing targets on some applications increased drastically, a less strict criterion can be helpful. By requiring that each definition and each use is at least used once with a coupled use respectively a coupled definition, the same number of interactions is covered as for *AllUses* shown in Table 4.9. The number of testing targets for each application is shown in Table 4.10. While for some

Table 4.7.: Minimal coverage of interactions if *AllDefs* is fulfilled

| A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | A9 | A10 | A11 | A12 | A13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1.00 | 0.61 | 0.8 | 0.61 | 0.75 | 1.00 | 0.8 | 1.00 | 1.00 | 0.28 | 0.77 | 0.8 | 0.5 |

Table 4.8.: Testing targets needed to fulfill *AllUses*

| A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | A9 | A10 | A11 | A12 | A13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 7 | 6 | 121 | 4 | 5 | 4 | 5 | 3 | 2 | 38 | 14 | 288 |

Table 4.9.: Minimal coverage of interactions if *AllUses* or *AllDefUse* is fulfilled

| A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | A9 | A10 | A11 | A12 | A13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1.00 | 0.69 | 1.00 | 0.88 | 1.00 | 1.00 | 0.8 | 1.00 | 1.00 | 0.42 | 1.00 | 0.93 | 1.00 |

applications, like the applications *A4*, *A11*, and *A13*, the number of testing targets is drastically less than for *AllUses*, the number of testing targets is for some applications higher than for *AllUses*. These applications have only one possible use for their definitions why the total amount of def-use-pairs is quite low. While such a coupling requires only one testing target for *AllUses*, the criterion *AllDefUse* requires two testing targets for such a case. However, for such a coupling, a test that can be used for the coverage of the testing target of *AllUses* can also be used for the coverage of both corresponding testing targets of *AllDefUse* since this test case covers both the definition and the use of the defuse-pair. As an example, Figure 4.8 shows a scenario where values can be saved on a data storage resource by the serverless function *fun0* and be read by *fun1*. While *AllUses* has only one testing target, namely the def-use pair consisting of "def_line_13" of *fun0* and "use_line_42" of *fun1*, *AllDefUse* has two testing targets: the coverage of "def_line_13" of *fun0* with any use and the coverage of "use_line_42" of *fun1* with any definition.

## 4.4. Related Work

Like for the models described in Section 3.5, there exist some specific coverage criteria for applications in the cloud. For the component-based view for IoT applications in [76] by Nagalakshmi and D'Souza, some coverage criteria are given, among others a *lambda coverage* which focuses on the coverage of all serverless functions of the IoT application which have to be triggered by another resource. Such coverage is subsumed by the criteria *AllRel* of this work. The other criteria are also specific to the IoT architecture by focusing on the potential paths that can be executed in the IoT application.

Table 4.10.: Testing targets needed to fulfill *AllDefUse*

| A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | A9 | A10 | A11 | A12 | A13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 8 | 5 | 31 | 4 | 7 | 4 | 7 | 5 | 3 | 18 | 10 | 56 |

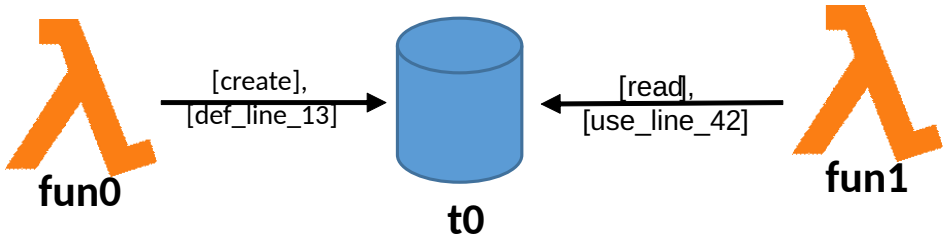



Figure 4.8.: Example for coupling via data storage

Some criteria are suggested in [19] by Chan et al. which are based on a model representing the computing entities. However, the target of the criteria is to test the utilization of the computing entities and not the functional correctness of the application.

In [18], Cannavacciuolo and Mariani introduced criteria for smoke testing which is a rough test to check if the basic functionality of a system is working. The criteria focus on the availability of the services deployed, their connections among each other, and the protocol used.

In general, the coverage criteria found focus mostly on the obvious coverage of nodes and their arcs which strongly depends on the model which represents the application.

Furthermore, also in [28, 29] serverless applications were collected by focusing on different sources listing existing serverless applications. However, the approach used here does not rely on existing serverless applications but uses the API of *GitHub* to identify serverless applications based on the source files used which also enables the identification of new serverless applications. In [30], Eskandani and Salvaneschi conducted a similar search as in this work by searching for serverless applications created for the *Serverless Framework* on *GitHub*. However, the search was executed after the search executed for this work but provides a bigger number of serverless applications.

## 4.5. Limitations

Of course, there are some limitations considering the criteria introduced here in isolation. Several criteria are introduced in this chapter, but it is not verified yet if they detect errors at all. They are based on the assumption that the system is executed under certain aspects which ideally reveals a potential error or deviation from the intended behavior of the system. This aspect will be handled in Chapter 7.

Furthermore, even if the selection process of the serverless applications was tried to be executed neutrally, it cannot be ensured that the applications are representative of all serverless applications. Additionally, the model representing the serverless applications is only an abstraction of the applications. Therefore, it cannot be guaranteed that certain aspects are captured. However,

the static analysis inspired the author of the work to some improvements to the criteria. So, in the following work, *AllRel* is stricter by focusing on all interactions between resources, and an additional criterion, *AllDefUse*, was introduced helping limit the number of test cases compared to *AllUses*.

## 4.6. Summary

This chapter introduced some coverage criteria that focus on serverless applications. In particular, control flow and data flow criteria which are based on the model of the previous chapter, are introduced and their potential for integration testing of serverless applications is considered. While the number of testing targets to be covered for the fulfillment of the criteria *AllRes* and *AllRel* is quite low, the number of testing targets for *AllSeq* can be infinite depending on the structure of the application. Since the number of testing targets of the stricter data flow criterion *AllUses* can be drastically larger than data flow criterion *AllDefs*, the new criterion *AllDefUse* was introduced whose number of testing targets is not drastically larger than *AllDefs* but whose testing targets are still stricter than *AllDefs*.

# 5. Measurement of Coverage Criteria

In order to apply coverage criteria practically, it must be possible to measure if the testing targets of the coverage criteria are covered when a test case is executed. Therefore, this chapter answers how control flow and data flow criteria can be measured in serverless applications. Furthermore, the measurement of coverage criteria is needed for the automatic generation of test cases in Chapter 6.

At the beginning of this chapter in Section 5.1, various distributed tracing tools are compared which were evaluated in the context of a master thesis [26] which resulted in a paper [27]. The performance of some selected tools was measured and their applicability for the measurement of coverage criteria is discussed.

Since data flow for the coverage criteria has to be measured, a model of serverless functions is built which shows how data can be transmitted in serverless functions by investigating some existing serverless applications to adapt the corresponding interfaces. On several serverless applications found on *GitHub* it was investigated how the serverless functions interact with their environment.

The chapter then presents a general approach that the measurement of the coverage criteria can be implemented in general by using serverless functions. This approach is implemented in an application, which is able to automatically instrument the source code to measure these criteria on some example applications. Furthermore, the impact of this measurement on common workflows of serverless applications is examined. The limitations of various distributed tracing tools in the context of coverage criteria measurement are also discussed

The content of this chapter is mainly based on the content of the papers [110, 111].

## 5.1. Distributed Tracing on AWS

In the context of Christina Eder's master thesis [26], which was supervised by the author of this work, distributed tracing tools were investigated in terms of execution time, memory usage, and initialization duration which resulted in a common paper [27] whose content is used in this section.

Various distributed tracing tools were investigated and applied for some scenarios to evaluate their performance. However, it turned out that these tools have limitations, particularly in their inability to measure data flow.

Tracing the data flow is an important capability for testing to check if the data were actually used. Since this important capability is not supported by a tool, this gap is filled by a tool introduced in this work which enables besides the tracing of workflows also the tracing of data flows. This tool is introduced in the subsequent section.

Distributed tracing focuses on the detection of the workflow of an application. This is typically utilized for monitoring where various metrics such as latency and memory usage are measured, but without focusing on specific testing targets.

### 5.1.1. Efficiency of Frameworks on AWS

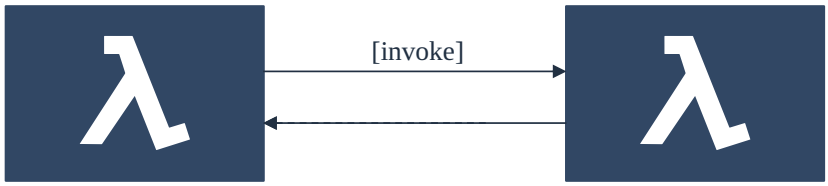
For the comparison of the execution times, three tools were chosen. *Zipkin*[23], recognized as the first highly scalable open source distributed tracing tool [96], was selected for this purpose. It uses a framework-based instrumentation. While in a framework-based instrumentation, an instrumented version of the application replaces the original one, an agent-based instrumentation enables tracing automatically by using technologies like library hooks or monkey patching [96]. Furthermore, *OpenTelemetry*[24] (OTel) and *SkyWalking*[25] were selected as agent-based instrumentation frameworks. *OpenTelemetry* is a standardization approach that is used by many frameworks identified. As a tracing backend, *Jaeger*[26] was chosen for *OpenTelemetry*.

The tools were compared on five microbenchmarks which comprise a function invocation (mb1), the access to external services using HTTP (mb2 and mb3), and the usage of the platform-specific data storage resource DynamoDB (mb4 and mb5) (see Figure 5.1). The HTTP request and the DynamoDB operation were subdivided into a GET and a POST request, as well as a read and a write operation to enable more detailed analysis.

A detailed description of the settings is available on *GitHub*[27]. The serverless functions were invoked using `Node.js 16.x` with 256 MB memory. The tracing backends needed were run on `AWS t2 EC2` instances running in the same region.

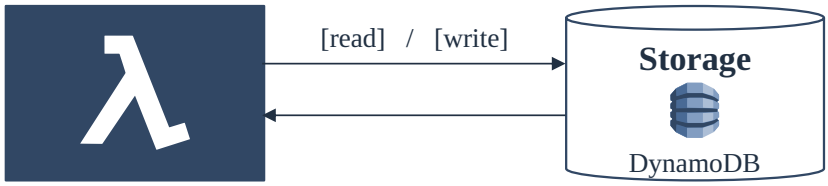
Only for *OpenTelemetry* no additional instrumentation was needed. By using AWS Distro for OpenTelemetry (ADOT) as an additional layer, no further manual instrumentation was needed to collect data. In contrast, *Zipkin* and *SkyWalking* required the code to be adapted. *SkyWalking* required to start an agent and to wrap the function handler of the serverless function. However,

[23] `https://zipkin.io`
[24] `https://opentelemetry.io`
[25] `https://skywalking.apache.org`
[26] `https://www.jaegertracing.io`
[27] `https://github.com/ch-eder/distributed-tracing`




(a) Microbenchmark 1: Function Invocation (mb1)


(b) Microbenchmark 2 and 3: HTTP GET request (mb2) and HTTP POST request (mb3)




(c) Microbenchmarks 4 and 5: DynamoDB read operation (mb4) and DynamoDB write operation (mb5)

Figure 5.1.: Microbenchmarks used for evaluation

the implementation of serverless functions is still considered experimental by the provider of the tool. *Zipkin* provided no libraries for capturing AWS SDK activities why they were implemented in the context of the master thesis by following the granularity of other Zipkin-provided realizations.

For each instrumentation and microbenchmark $100$ warm function invocations were considered. Each group of $100$ invocations was divided into $10$ blocks to mitigate the risk of running on one machine only. The configuration of each serverless function was updated before running a new block, which forces the following call to run in a newly started container.

In contrast to the execution time and memory usage, the initialization duration was investigated using an empty function as an additional microbenchmark (mb0). In the case of using tracing, the empty function embraces the inclusion of necessary libraries and dependencies without executing any statements in its body.

For the results, a uniform distribution was assumed following the central limit theorem [54] and a significance level $\alpha$ with the value $0.05$ was chosen.

The results of the evaluation are given in Table 5.1 showing the average overhead. Nearly all execution times of the tools were higher than without instrumentation. Taking into consideration the median, even all execution times were higher. However, the sample size is based on $100$ runs why not for all tools a significant overhead could be shown. Furthermore, the memory usage for all microbenchmarks was higher whereas the table shows the overall overhead

of all microbenchmarks. Also, the initialization duration is significantly longer than without using distributed tracing.

Table 5.1.: Efficiency overheads of the selected distributed tracing tools. Overheads highlighted in gray are statistically significant.

| | Benchmark | Zipkin | OTel | SkyWalking |
|---|---|---|---|---|
| Runtime | mb1 | 6.37 % | 4.52 % | 38.29 % |
| | mb2 | 36.08 % | 70.85 % | 48.09 % |
| | mb3 | 6.31 % | 17.15 % | 8.15 % |
| | mb4 | 4.88 % | 7.17 % | 73.04 % |
| | mb5 | 0 % | 21.25 % | 85.79 % |
| Memory usage | mb1-mb5 | 7.51 % | 91.06 % | 18.90 % |
| Initialization duration | mb0 | 70.22 % | 575.35 % | 208.04 % |

### 5.1.2. Measuring Coverage Criteria with Distributed Tracing

Distributed tracing is a tool that allows users to track the resources accessed during a request. The focus of the most common and available tools is on the visualization of the data in a human-readable way [15]. Thus, monitoring the applications enables us to see where the performance and the costs of running the application are affected. However, since monitoring lists the components that are called by a request, they can also be used for the measurement of some coverage criteria. This requires that all calls and not only a sample size of the calls made are collected which is often applied in distributed tracing. Since the distributed tracing tools log the resources called and the relation between them, the criteria *AllRes* and *AllRel* can be measured by them.

Data flow criteria are not supported by distributed tracing. They require measurement of the uses and definitions of the values passed which is not implemented in the distributed tracing tools. Furthermore, not all paths can be clearly identified by distributed tracing. If a serverless function writes data to a data storage resource which triggers another serverless function, the resources are captured which is enough for *AllRes*. Depending on the framework, it was also possible to create all the relations which was enough for the fulfillment of criterion *AllRel*. However, *AllSeq* cannot be fulfilled since the complete path cannot be unambiguously identified since there is no information passed about the serverless function initially triggering the event.

Therefore, a more specific implementation is needed to measure the data flow. Such an implementation is introduced in the following.

## 5.2. Execution Time Measurement of Criteria

The following section discusses how the measurement of coverage criteria can be implemented. Since serverless applications use not only functions but also other resources, the interactions with different kinds of resources have to be covered. As the coverage criteria focus not only on the coverage of program code but also on the coverage of resources and the data processed by these resources, support for the ecosystem of the platform provider is needed to measure control and data flow. However, no general support is given by the cloud platform providers to support such coverage criteria. Therefore, this section shows how these criteria can be measured by modifying the serverless functions, which are often the only resources where individual logic can be implemented by developers within the running application, and by evaluating the corresponding logs of the serverless functions, which are used to log information about the current state of execution. Thus, these criteria can be implemented individually on each FaaS platform by adapting the serverless functions and evaluating the log file where the information needed for the coverage analysis is stored.

### 5.2.1. All-resources

The coverage of all resources requires not only the coverage of all serverless functions but also of other resources provided by the cloud platform provider. While the coverage of all serverless functions can be implemented by adding a log statement with the name of the function when the function is entered, the measurement of other resources is often not supported by the cloud platform provider. However, the coverage of other resources can be measured indirectly by adding a log statement to the serverless function before the corresponding resource is called containing the name of the resource being called. If the cloud platform provider supports the possibility to see when and if a resource was called, this information can be used instead. Additionally, if supported, a return value of the method calling a resource can indicate if the call was successful and the resource is available. Finally, all resources that were covered can be read from the log file. If more than one resource is between two serverless functions of a workflow where none of them is also a serverless function, the coverage of these resources is more complicated. By using logs of the serverless functions and by analyzing potential workflows between the functions, resources having been used can be identified if there is only one potential workflow or all workflows use the same resources. By reading the state of data storage resources, it can also be evaluated if the resource was used. However, this required access rights for the evaluation tool to the data storage resource and an additional evaluation phase.

### 5.2.2. All-resource-relations

Covering all relations between the components needs additional information. While all outcoming arcs, which represent calls to other resources, of serverless functions can be covered by logging the names of the serverless functions and their resources being called at the corresponding statements, incoming arcs are more difficult to track since the source of the call is unknown to them. If an identifier of the source is not passed by the platform provider to the function, the application has to be adapted so that this additional information is passed. For example, this can be done by adding the name of the resource to the payload. For example, a message queue can have an additional field in its data passed where its name is saved. If the event is finally passed to another serverless function or forwarded by another resource, the name of the message queue is still available. Thus, a relation can be covered by the following serverless function which now knows the previous resource or if there is only one resource between, the serverless function can create two relations: from the message queue to the resource between, and from the resource between to the serverless function. If the calling resource depends on a dynamic parameter, the identification of feasible relations is harder or even impossible. However, this wasn't the case for any application considered in the context of this work. Furthermore, if a workflow contains a chain of at least two resources not interrupted by a serverless function, the coverage of these resources has to be estimated by analyzing the dependency graph and its workflows.

### 5.2.3. All-resource-sequences

In order to measure the coverage of all paths, it must be possible to reconstruct the previously executed workflow belonging to a resource being called. This requires that calls, which are made from one resource to another resource, pass the information of the current workflow. For example, if a resource is called several times within the same workflow or by another workflow, it would not always be clear which path was actually executed. Therefore, a causal order of the calls has to be logged. This information about the resources already being called can be passed by attaching it to the data transmitted between the resources. Therefore, the data being passed depends also on the number of resources called within a workflow. Alternatively, a unique identifier can be created for each resource called which is passed to each resource being called. By logging this information together with the ID of the current serverless function, the paths being used can be reconstructed from the log without passing the complete path between each resource.

If, for example, a lambda function writes a value to a database, it has to save its identifier and a unique identifier of its call too. If the data storage resource triggers another lambda function, this function is able to extract the value if

the changed value is passed. Finally, all paths covered can be reconstructed by reading the log file and creating the paths recursively.

However, if data are deleted and the resource triggers a serverless function, measuring the path is more difficult. A wrapper for the delete operation to the data storage resource is needed. One possible approach is to write the data first temporarily to the data storage resource with the information of the actual path. If the resource triggers also other serverless functions when data are written, these functions have to be instrumented to detect if the event was actually triggered by a delete operation and ignore such a wrapped delete event. After the write operation, the delete operation can be called which now passes the values of the deleted entries to the serverless function being triggered. The triggered serverless function can now extract the data of the workflow. However, the concrete implementation here depends on the application and the data being passed when a trigger is executed after data were changed.

### 5.2.4. All-resource-defs

Values from a serverless function to another serverless function can be transferred in two ways. If a value is passed from one serverless function to another serverless function directly, the data of the definition can be attached to the payload of the event. If the value of the payload is used, the usage of the definition can be logged by simply extracting the information from the payload. This requires that the serverless functions are adapted to log this information before a use is executed. If there is another resource between, e.g., a queue or a stream, the same approach can be applied. If serverless functions don't call each other directly but use a data storage resource for their interaction where one function saves data and the other one uses them, the information of the data of the definition can be attached to the data storage resource. If the value is read by another serverless function, its usage can be confirmed in the log similar to the usage of a payload. However, if data are deleted from a data storage resource, a wrapper keeping deleted data until they are accessed for the first time has to be used. This requires also that uses of data received from the data storage resource are adapted. When the data are read, the data should contain the original definition but be handled as having no data when used in the source code. This requires additional instrumentation for each use too.

### 5.2.5. All-resource-uses

If for each definition not any use has to be covered but all uses, the procedure is similar to the one of *AllDefs*. The same information is passed from one resource to another resource. The treatment for the usages differs from *AllDefs* in that when a definition is used, the usage is logged together with its definition. So, by evaluating the log files, the def-use pair can be clearly identified.

### 5.2.6. All-resource-defuse

Since *AllDefUse* requires that all definitions and uses have to be covered, the approach of *AllUses* can be applied since this approach logs both uses and definitions which can be read from the data being logged. So, only the evaluation of the logs recorded differs from *AllUses* where both definitions and uses have to be extracted from the data.

### 5.2.7. Parallel Execution

Tracking the parallel execution is more complicated since this requires that the causal order of accesses to a data storage resource can be tracked. If accesses to a data storage resource from different functions should be logged, this can be done by adding additionally the information of the caller to the data by using a unique ID that identifies the function. If another function accesses the data, the same information is added to the data. When the data is requested, the function using the data has to request the identifier information and log it before eventually the log can be evaluated. However, the data storage resource has to be locked for such an access because the data have to be read first and then updated. The IDs saved in the data storage resource represent the order of accesses. If the same function should access a data storage resource in a different order, the calls made by the function must save an individual ID. If only test cases are executed, a framework could also be used guaranteeing that the steps of test cases are executed in certain orders by executing them sequentially. Thus, the data storage resource had not to be locked explicitly.

## 5.3. Measuring Data Flow of Serverless Functions

The data flows of serverless functions with their environment are relevant for the creation of test cases in order to cover the dependencies to other resources of the system. These data flows to other services can influence the state of the system. Therefore, various serverless functions were investigated in order to see how the execution process of a serverless function is influenced by external data flows to use these insights for the testing framework. Since there aren't many applications currently available using serverless functions, *GitHub* was searched for serverless functions and the applications found were analyzed similarly to the search in Section 4.3.1. Based on these insights, a model was built in 5.3.3 showing the factors influencing the execution of serverless functions.

### 5.3.1. Selection of Serverless Functions

The decision was made to investigate serverless functions that are publicly available on *GitHub*. The aim was to construct a model of factors influencing

the execution of serverless functions. *GitHub* was searched for application focusing on *Amazon Web service* (AWS) that are deployable by the *Serverless Framework*[28]. This framework is often applied to describe the components of serverless applications and deploy their serverless functions. *GitHub REST API*[29] was used for the search which provides, in contrast to *GHTorrent*[30], the possibility to search for filename and keywords contained in the files.

Files with the name 'serverless.yml' were selected which is the standard filename used by the *Serverless Framework* describing the infrastructure of the application. Furthermore, the request to the API was set to filter for the keywords 'AWS' and 'handler' to ensure that serverless functions were used on *AWS*. Thus, files could be identified that are usable on *AWS Lambda* and contain serverless functions that are deployable by the *Serverless Framework*.

The API provided $1000$ results for files describing serverless applications with the search conducted on July 30, 2020. Since the REST API could only return $100$ results per request whose answer was paginated, $10$ different endpoints had to be called to get all the $1000$ results. The response contained the total number of potential results for the request. Since this total number varied for some endpoints, when an endpoint was called for the first time, it was possible that the data accessed by the endpoints differed. However, if all endpoints were called several times, finally all endpoints returned the same number of potential results. Therefore, all the endpoints were called until each endpoint had the same number of potential results assuming that this makes it more likely that the same data were accessed.

The files identified are part of $527$ different projects. Since new serverless applications have been added since then, a new search would probably result in a different outcome. The projects were sorted by their number of stars which roughly indicates the popularity of a project. Afterwards, the first $27$ projects were analyzed resulting in $323$ serverless functions, after functions of projects had been sorted out which were not completely implemented, just boilerplate code or just a very simple demonstration. Furthermore, only projects were investigated with serverless functions written in JavaScript which was the most popular language used in the applications identified (see Table 5.2) in order to stay in the same domain of a programming language.

The program and the data collected with its analysis can be found online[31].

### 5.3.2. Investigation of Data Flows

Various serverless functions and their structure were investigated to identify factors influencing the execution of serverless functions which have to be

[28] `https://www.serverless.com/`
[29] `https://docs.github.com/en/rest`
[30] `https://ghtorrent.org/`
[31] `https://github.com/snwinz/ServerlessApplicationSearcher/releases/tag/v1.0`

Table 5.2.: Languages used in applications

| Language | Number of projects |
|---|---|
| JavaScript | 459 |
| Python | 59 |
| Java | 6 |
| dotNet | 3 |

considered for the interaction of serverless functions with their environment. Therefore, all types of calls requiring dependencies not running within the container of the serverless function were noted.

The calls that were made outside the serverless functions were categorized into calls made to platform-specific services, e.g., data storage resources, and calls made to external services like calls made via HTTP. If the service was a data storage resource service of Amazon (i.e., *DynamoDB* or *S3*) where data are saved or read, a note was taken if data were written (e.g., creations, updates, or deletions) or read. The direct invocation of other serverless functions was noted too.

Additionally, it was checked if the arguments passed to a serverless function are used at all. This indicates the direct influence of an external call. Furthermore, how the usage of services influenced the return values of the functions was evaluated.

Therefore, the return values of functions using services were categorized in *state*, *value*, and *nothing* as follows:

- *state:* if the return value just indicates that the execution was successful by returning a simple state message.
- *value:* for return values which were calculated depending on the input of other resources and contain more information than the pure state of the successful execution.
- *nothing:* if no return value was returned at all.

Since usually the state of a service call is returned if its call fails, all return values investigated were influenced by the services called.

This categorization helps to see if there is a data flow between the calling resource and the callee or if the function is just called and only its successful execution is relevant for the caller (e.g., an orchestrator or a user).

Finally, the functions were checked for the usage of global variables which were set in a previous call and potential background processes.

### 5.3.3. Potential Influential Factors of Serverless Functions and Practical Impact

Based on this investigation, a model of the interfaces of a serverless function was built describing both the influential factors to its execution coming from its environment and the influential factors of a serverless function to its environment (see Figure 5.2). In general, a serverless function gets the information needed to be processed by its arguments assigned to the parameters of the serverless function and can return a value. But there were also other factors identified influencing the execution and the result of a serverless function.

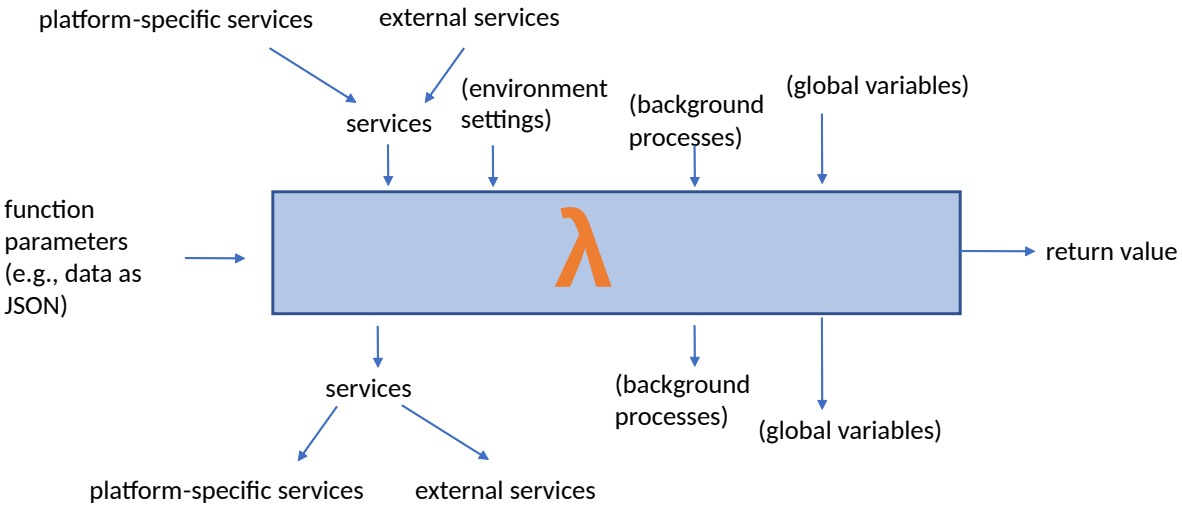



Figure 5.2.: Data interfaces of serverless functions

Environment variables set for the configuration of the system can also be accessed by a serverless function during its execution and influence its execution. A typical example of such a variable is the name used for a data storage resource. These variables cannot be set directly within a serverless function and are usually set when a new version of a serverless function is deployed. Therefore, they are not really relevant for this data flow evaluation since they can be considered as a constant factor after deployment.

Another factor that influences the execution process of serverless functions are processes running in the background. A serverless function can return a value and terminate without having background processes completed. These processes are continued if the same runtime environment is started again. This can influence the state of the application even if the background process was started by another invocation, e.g., by changing global variables, calling external services, or terminating the execution. In the serverless functions investigated here, such processes starting in the background could not be identified.

A similar behavior arises when a global variable is used and changed during the execution of the serverless function. If so, the value set by a previous call in the runtime environment could influence the state of the application. If the

global variable is always instantiated the same way when a function is started, it doesn't influence the system. However, such a behavior implemented in the serverless functions investigated where a previous call influences the value of a subsequent one could not be identified. This behavior could be utilized in order to cache some data.

Besides these factors, there is also the possibility that a serverless function gets data to be processed by calling a service. The service can be platform-specific, like a data storage resource access, or external, like a call to an HTTP API. Of course, services can also be used to store data.

### 5.3.4. Analysis of the Interfaces of Serverless Functions

The analysis revealed that nearly all the functions relied on services. Therefore, it is not enough to test only the function in isolation, but also other dependencies have to be considered, and their integration be tested.

306 ($\approx 94.44\%$) of the functions investigated used at least one service. A popular kind of service containing and preserving the state of an application are data storage services. More than half of the functions (162) used at least one platform-specific data storage resource like *DynamoDB* or *S3* where 142 of these ($\approx 87.65\%$) used at least once *DynamoDB*. 68 of these functions read data from and 77 changed data on *DynamoDB*. This underscores the importance of platform services, in particular data storage resources.

Furthermore, all return values of the functions using services were influenced by the service called. Only 40 functions ($\approx 12.38\%$) didn't return anything explicitly at all, whereas nearly every third function (104) returned a value. The majority of the functions (179) returned a value indicating the successful execution of the function. A return value indicating only if the execution was successful is an indicator that the return value is only used for its orchestration, e.g., reexecuting the function by an orchestrator, or to indicate to the caller that the function invocation failed. The parameters of a serverless function are instantiated by the events triggering the serverless functions and had also an effect on the execution of the serverless functions. 271 functions ($\approx 83.90\%$) used the parameter for its execution. However, there were only 13 serverless functions ($\approx 4.02\%$) invoking another serverless function directly.

Even if this analysis on *GitHub* cannot be representative of all serverless applications since many projects on *GitHub* are personal and inactive [50], it still shows the different usages of serverless functions and its reliance on services, particularly data storage resources.

### 5.3.5. Measurement Implementation

The previous investigation highlighted the various ways in which serverless functions can exchange data with their environment. Even if there are multiple

factors that can influence the internal data flow during execution, the most relevant factor is the interaction with services. Consequently, the data flow of these services can influence the state of the system if data are written. If data are read, the execution process depends on an external state. Therefore, the state of the application resides in the data flow within the application and the state of other services, mostly data storage resources. The services where the information is passed to, can process, save, or route the information to another service. The most common service detected in this investigation was *DynamoDB* which is a key-value storage. In general, each of the services called by a serverless function is a potential data storage resource where data could be saved and read by other parts of the system. Therefore, while developing a serverless application, it is not enough to test only the serverless function in isolation, but also the influence of the data that was changed to the system. Thus, not only the relations between services are tested, but also a direct influence between different services. To address this need, a framework was introduced and implemented for measuring data flow coverage in a serverless application while focusing on different kinds of data flow: data flow between serverless functions via a service, here *DynamoDB*, data flow via return values and data flow via function invocation demonstrating the usage of a parameter set by an event. In addition, the workflow to measure this data flow and the small execution time overhead of this measurement is presented.

### 5.3.6. Criteria Instrumentation

The implementation of the criteria discussed in the previous section is based on the execution time implementation described in section 5.2. The relevant locations in the source code were identified and instrumented to enable the coverage of some testing targets.

The entry point of a serverless function is used particularly for the control flow criteria. Here, a log statement records for the *AllRes* criterion that the serverless function was called. The *AllRel* criterion checks the event passed for data indicating that the serverless function was called by another resource. This happens if the serverless function is triggered by another resource. If such a resource is found, this information is recorded in a log statement. For *AllSeq* also all incoming requests were checked for data having some tracking information. If so, the current serverless function was attached to the trace and recorded.

Furthermore, the tool instrumented the code which is executed before another serverless function is invoked. *AllRes* instruments that the serverless function called is recorded while *AllRel* records the relation. However, since *AllRel* needs to pass the actual trace, the instrumentation adds the actual trace including the current function to the payload which is passed to the serverless function called.

Additionally, the tool instruments the location before data are accessed on a *DynamoDB* data storage resource by recording the name of the data storage resource for *AllRes* and the relation from the serverless function to the data storage resource for *AllRel*. However, *AllSeq* can only record for read accesses the current trace without any further adaptions. When data are written or deleted, the instrumentation attaches information to the payload of the current trace. So, if the data storage resource triggers another serverless function, this information can be read and evaluated. Since in the case of a delete operation, the data is not deleted, the tool instruments also all uses of values received from a read access. If the value received has a mark that it was actually deleted, the value is replaced as if it didn't exist.

Based on the findings from section 5.3, the data flow criteria for different kinds of data flows were implemented. These criteria were primarily implemented for serverless functions which are coupled via a data storage resource, here *DynamoDB*, by supporting write and read operations. Additionally, support for the delete operation to measure the usage of deleted values was implemented.

The locations where values received from a data storage resource are used are particularly relevant for data flow criteria. These locations were instrumented similarly for all criteria. It is checked if some information about the definition of the value was passed too. If so, a combination of the location of the definition and current use is saved, while *AllDefs* records only the coverage of the location of the definition.

The same approach was applied for serverless functions which are invoked by other serverless functions by attaching the relevant information to the payload. Even if direct function invocation wasn't used very often in the analysis, this behavior is relevant, since the callee uses the information passed via a parameter of the event which is a common scenario, especially when a data storage resource triggers a serverless function when a new entry is added. If the callee returns a value, the usage of the value returned is also instrumented similarly in the caller.

In order to have the information available, the locations where definitions take place are instrumented. This is done for variables used for accesses to *DynamoDB* and serverless function invocations. If a function returns a value, also an identifier of its definition is attached making it possible for the caller to interpret the source of the result. This is only implemented if the return value is an object in JSON format where the information can easily be attached to an additional field.

It's worth noting that if another data format is used, the usage of the additional information has to be adapted in a way that other resources using the return value can handle the additional information. For example, if a string value should be returned, it would only make sense to attach the identifier of the definition if all services using the return value can handle it, such as by parsing the relevant value by a delimiter.

Finally, the data flow criteria and *AllSeq* require replacing the delete operations with write operations. The write operations add a marker to the corresponding value to indicate that the value was deleted and the location of deletion respectively the previous sequence of calls. Additionally, each use of the values received is checked for markers if the value is actually deleted. If a marker is identified, an empty entry is returned to indicate that the entry is not available. Furthermore, the information of the marker is evaluated to record the previous sequence of calls or the combination of the definition of the deletion and its usage.

If a variable of a serverless function is used to pass information via a coupling, each of its definitions before the coupling is instrumented adding the information of the concrete definition to the variable. Thus, when the variable is used, the latest definition of the variable can be read. The usage of variables is implemented similarly. Each usage of a variable coming from a coupling is instrumented by logging its usage. This allows a usage after a coupling to be identified later.

### 5.3.7. Workflow for Measuring Coverage

The first major step of the application whose coverage has to be measured is the instrumentation of the serverless functions. In the tool, only the source code of the application is needed which is read by a parser. *ANTLR*[32] was used to create a parser for JavaScript. Other languages have to be implemented explicitly in the framework if more support is needed. The parser was adapted to identify relevant parts of the source code. These parts are the entry point of the function, locations where an interaction with another resource takes place, and the locations where variables are defined and used. These parts are enriched with JavaScript source code according to section 5.3.6 which adds relevant information to the variables and adds log statements to the source code. The generated source code has to be deployed and run afterwards, e.g., by using test cases.

In contrast to the implementations of [46, 81] where a global array is used to log usages, serverless applications are distributed. Therefore, *CloudWatch* was used for saving the information of usages, definitions, calls, etc. However, in contrast to a global array where the values could be read directly, the log files have to be downloaded and evaluated explicitly. This is done after the execution of test cases. Monitoring the coverage of a system during its execution could be supported by scripts constantly polling and evaluating these logs or sending the records to a backend which evaluates them when received.

The log files can simply be parsed for the record where valid def-use pairs were recorded within a serverless function if the data flow coverage of a serverless function should be measured. Similarly, for the control flow criteria,

[32] `https://www.antlr.org/`

the records for the call of resources, relations, and sequences can be checked and counted. Since also parts of the sequences are recorded, it must be considered that there are more records than sequences actually called. If the actual number of sequences is relevant, a unique number could be added to each trace to only select the longest sequence.

### 5.3.8. Measurement with Model Support

The implementation of this tool whose execution time is investigated in the subsequent section measures the coverage without requiring prior knowledge of all the testing targets. Each potential variable has to be instrumented for the data flow coverage and each interface where data are potentially transmitted must also be instrumented. Furthermore, the data that are recorded, like function names and the name of the previous calling resource, are often read from variables passed, such as shown in Listing 5.1. Here, the parameters *event* and *context* are evaluated for potential tracing data. It is examined if the serverless function was called directly by another serverless function (lines 1-3) or if it was triggered by a $DynamoDB$ data storage resource (lines 4-15). Furthermore, the name of the current serverless function is extracted from the variable $context$.

Listing 5.1: Parameter evaluation for *AllUses*

```
if (event.funcDefAU != undefined) {
    console.log('#AU_SU_' + event.funcDefAU + '_##' + context.functionName
        + '_line14_event');
  }
if (event.Records != undefined) {
    if (event.Records[0] != undefined) {
        if (event.Records[0].dynamodb != undefined) {
            if (event.Records[0].dynamodb.NewImage != undefined) {
                if (event.Records[0].dynamodb.NewImage.funcDefAU !=
                    undefined) {
                    let funcDefAU = event.Records[0].dynamodb.NewImage.
                        funcDefAU.S;
                    console.log('#AU_SU_' + funcDefAU + '_##' + context.
                        functionName + '_lineevent_14');
                }
            }
        }
    }
}
```

This is an additional overhead that could be avoided if more information about the application and the context in which the resources are used is known. So, if the source code is assigned explicitly to only one serverless function, function names that are saved can be hard-coded. Furthermore, if relations between

the functions and potential data flows between the resources are known, less instrumentation has to be done since the relation can be hard-coded if only one relation is possible. This makes the instrumented application both more readable and requires fewer operations to be executed.

## 5.3.9. Execution Time Evaluation

This section evaluates the execution time of the code instrumented by the framework to measure the coverage of the criteria. The execution times of the instrumented code are compared to the execution times of the original code without any instrumentation. The instrumentation that was added to the source code contains no hard-coded names of serverless functions, other resources, or relations. A version for comparison was taken where *OpenTelemetry* with *Jaeger* is used for distributed tracing to compare it to a common monitoring framework. Furthermore, another version used *X-Ray*[33] which is the monitoring solution of *AWS* to compare it to a monitoring solution of a cloud platform provider.

Three different scenarios were evaluated covering different kinds of data flow between serverless functions.

### 5.3.9.1. Scenarios

The first scenario (Figure 5.3) is a serverless function calling another serverless function where the callee uses a value of the caller. The serverless function called returns a value to the calling serverless function which results in another coupling. Therefore, there exists a def-use pair between the caller and the callee and the callee and the caller. The first def-use pair is a typical example of a serverless function using the argument of an event, like when the serverless function is invoked directly by another one or triggered via an event by another resource, whereas the second def-use pair is an example of a coupling through a return value.

The execution times of the calling function on the cloud side are measured since the caller calls the callee synchronously and has to wait for its answer. Thus, the calling function takes also longer when the execution of the callee takes longer.

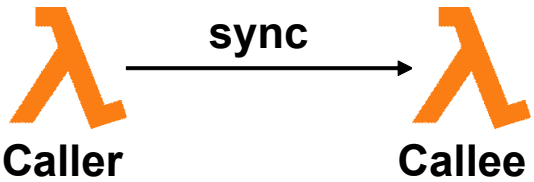



Figure 5.3.: Model of function calling another function

[33] https://aws.amazon.com/xray/

The second scenario (Figure 5.4) is a scenario with a write operation to a *DynamoDB* data storage resource whose value is read by another serverless function. Both functions are called synchronously by another serverless function whose execution time is measured on the cloud side. The read operation is set to be a consistent read which guarantees that the latest value of the data storage resource is read. Otherwise, only eventual consistency is guaranteed where potentially the latest write operation is not read. Thus, both serverless functions are coupled by a value on the *DynamoDB* storage.

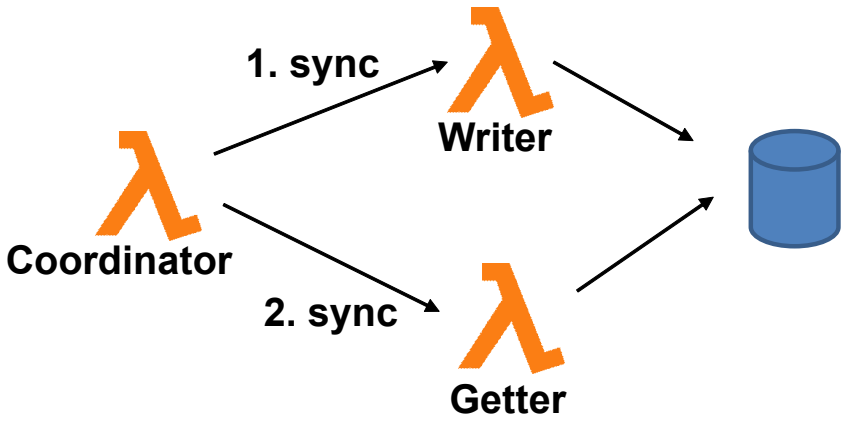



Figure 5.4.: Model of scenario for coupling via a write operation

The third scenario (Figure 5.5) is similar to the second one, but a value written to a data storage resource is deleted by a serverless function. This value is read by another serverless function with a consistent read like in the previous scenario. Both the deleting and reading functions are coordinated by a serverless function calling these functions synchronously. All values were written to the data storage resource before the coordinator function was called.

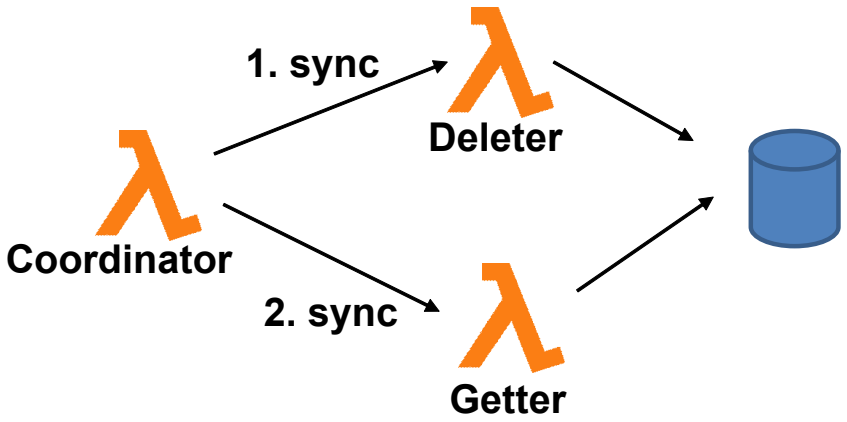



Figure 5.5.: Model of scenario for coupling via a delete operation

#### 5.3.9.2. Execution

All tests were run between September 30 and October 1, 2023. Each of these scenarios was executed with its original source code and a version instrumented by the tool for each of the data flow and control flow criteria. Furthermore, it was verified that each def-use pair was tracked correctly. A version using *OpenTelemetry* was executed for comparison to an existing monitoring framework that supports serverless computing. Similarly to Section 5.1.1, *Jaeger* was used as the backend. Furthermore, Amazon's *X-Ray* was used for comparison which is the monitoring solution of Amazon that can be applied to trace and visualize the resources used by requests.

Therefore, each scenario was executed with nine different versions. The nine versions were tested for each scenario in one test run to ensure that the utilization of resources of the cloud platform is similar. Each run was divided into $100$ blocks where in each block each version was executed eleven times. If the execution of one block finished, a block of the next criterion was executed. Thus, the next block of a version could only be executed when the previous block of all other versions is already finished which avoided that all blocks of a version were executed when the cloud platform was slow or fast.

Before each single execution of a block, the configuration of the serverless functions was upgraded. Thus, a cold start of the serverless function could be enforced for each first run of a block since the platform deploys a new runtime environment if the description of a serverless function is changed. This reduced the risk that a serverless function is only running on the same machine. Furthermore, the memory assigned was set to the maximal size of $10240$ MB enforcing a deployment to a fast machine. Otherwise, slower machines could be compared to faster ones since sometimes if slower resources are assigned to a serverless function, faster ones are assigned by the cloud platform provider [32, 67]. Since logs got lost when the description of a serverless function was changed immediately after its execution, the script used waited for 10 seconds after each execution block.

All in all, $1000$ warm runs were compared for each of the versions of the scenarios and checked that the coupling was tracked.

For all instrumentations, the effect size was calculated indicating how strong the overhead of the instrumentation was. Also, the significance was calculated similarly to Section 5.1 by assuming a uniform distribution following the central limit theorem [54]. The significance level $\alpha$ was set to $0.05$.

#### 5.3.9.3. Results

The results of the first scenario showed that there is only a small difference in the execution times if a function was instrumented compared to the function without instrumentation, as can be seen in Figure 5.6 where a box plot is shown with the median execution time. The instrumentation is only limited to the

addition of a value to the parameter and a few log messages where there were only small differences in the instrumentation added for these criteria. The original code was significantly faster than the instrumented code. However, the criterion *AllRel* was even faster on average for the test runs than without instrumentation, but slower when the median is compared. This can be explained through some outliers in the data. The other instrumentations of the coverage criteria had an effect size between $0.09$ and $0.19$ which is still a small effect according to Cohen. However, the instrumentation with *OpenTelemetry* is much slower with an effect size greater than $2.1$ indicating a strong effect. Also, *X-Ray* was much slower with an effect size of more than $0.49$ which is still a normal effect size but nearly a large one according to Cohen [21].

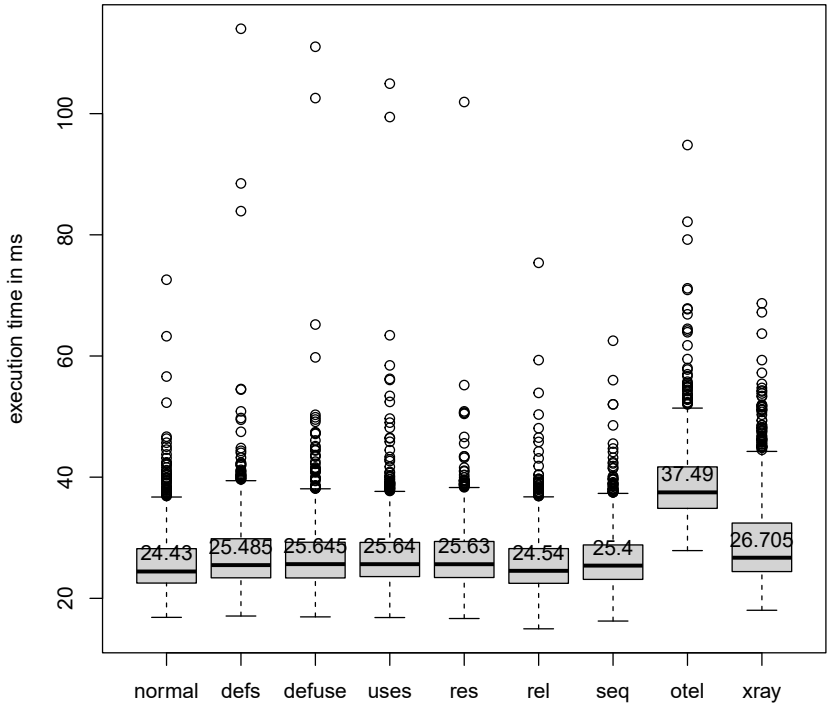



Figure 5.6.: Box plot of the execution times of caller scenario

The second scenario showed also no relevant differences in its execution times (compare Figure 5.7). The data flow criteria and *AllSeq* saved additional data to the data storage resource containing information of the current serverless function. The criteria themselves have only a few differences in their implementations by logging the concrete statement of the usage of a definition for *AllUses* and *AllDefUse*. The median execution time for this scenario was quite similar for all criteria implementations. The effect sizes of the criteria implementations were quite low with no more than $0.06$. The code without instrumentation was not significantly faster than the instrumented one of the

criteria. Some criteria were even faster than without instrumentation resulting in a negative effect size. Reasons for this can be some outliers, the small effect of the additional instrumentation on the execution time, and optimization in the background of the runtime environment. In contrast, *OpenTelemetry* showed an overhead of more than 22 ms on average and of nearly 20 ms for its median. The effect size was moderate with around $0.35$ while *X-Ray* had a large effect size of more than $0.70$.

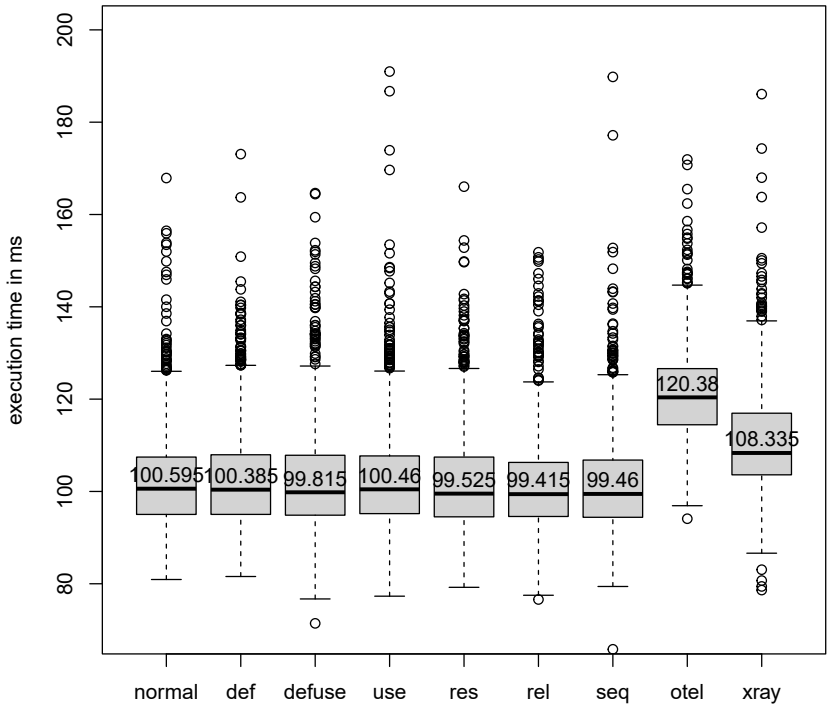



Figure 5.7.: Box plot of execution time of writer scenario

The last scenario showed also only a small difference in its execution time (compare Figure 5.8). The scenario is quite similar to the previous one, but the instrumented versions of this scenario don't use a delete operation, but a write operation for the data flow and *AllSeq* criteria. Additionally, the entries on the data storage resource are overwritten which is also a source of additional execution time on the data storage resource caused by additional entries. Therefore, this scenario depends more on the service than the previous scenario where the same write operation was used with additional values whereas here the delete operation is replaced by a write operation. All instrumentations, except *AllDefUse* and *AllRes*, were significantly slower than the original one. Similarly to the previous scenario, the effect size is quite low with less than $0.12$ for all criteria. Both the overhead of *OpenTelemetry* and *X-Ray* were quite high with and large effect size of more than $1.61$ respectively $0.78$. There were

only small differences in the execution times of the instrumented scenarios compared to the original ones which shows that the execution time of the instrumentations is not so relevant for these scenarios.

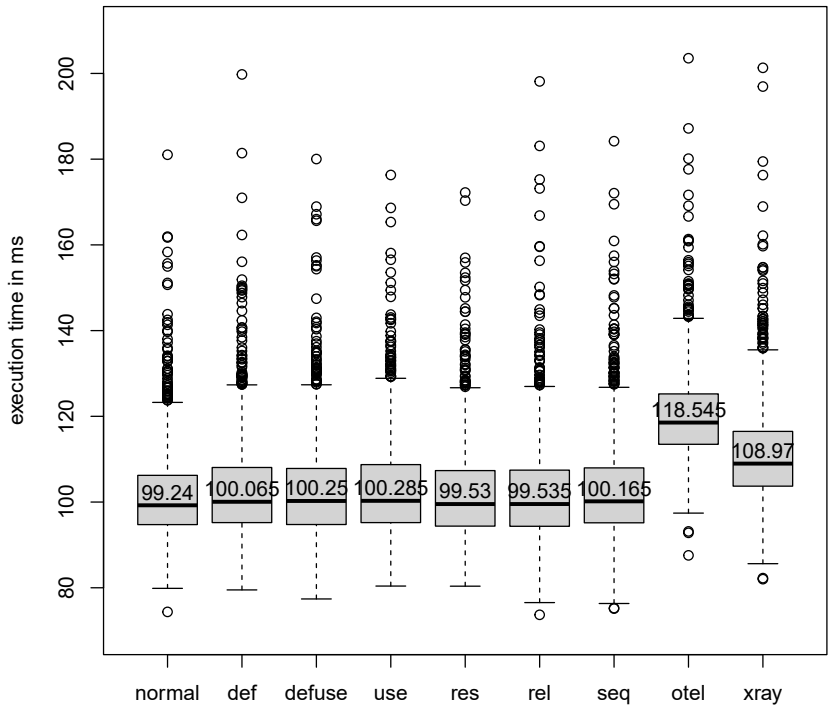



Figure 5.8.: Box plot of execution time of deleter scenario

Even if these scenarios are very simple, they cover all the interfaces where an additional instrumentation is needed. Therefore, the execution of the instrumentation of a more complex function would only take much longer if it used more services or required more instrumentations to log the definitions and usages of variables used by its interfaces.

## 5.4. Related Work

Lin et al. showed in their work [63] an approach to track the causal order for serverless applications on Amazon. Their tool *GammRay* adds a unique ID into the payload when invocations to serverless functions are made, records these data, and processes them offline to build the causal order between the components of the application. Furthermore, also performance data are collected. However, their tool is not able to track the data flow between serverless functions. In [62], their work is extended by adding a multi-cloud support where for each cloud a local instance collects the data which are

propagated to the other instances. This approach could also be applied to the approach in this chapter if another service than *CloudWatch* should be used to collect the data or data should be collected for several cloud platform providers or regions.

Sreekanti et al. [98] added an additional layer between serverless functions and the data storage resource service to improve fault tolerance. This is in contrast to our work where we do not need an additional layer but simply instrument the source code of serverless functions when data storage resources are accessed to measure the data flow.

The measurement of def-use-pairs was also described in [46, 81] by using a global table where the calls of definitions and uses are recorded and accessed during execution time. However, because of the stateless architecture of serverless applications such an approach would require a central instance where data are both saved and read which would reduce the execution time. Furthermore, their approach uses a table that records the last location of a definition which is associated with a use to reconstruct the def-use-pair. Since serverless functions can be executed in parallel, such an approach could not reproduce the corresponding definition of a usage of a value if, for example, the serverless function of the use is called by two other serverless functions defining the value at the same time, since one of the last definitions would be overwritten.

The Lamport timestamp [55] and the vector clock [31, 73] are classical approaches that provide the possibility of a partial ordering of the events. However, these approaches are based on stateful processes whose state is updated if the process is invoked while our approach considers the order and data flow of the components of a serverless application which are based on stateless serverless functions.

Furthermore, also in [28, 29], which were published after the work of the author of this work, serverless applications were collected and investigated but without focusing on the data flow. Their investigation confirmed the popularity of *JavaScript* and *Python* as the most popular programming languages, and that the usage of data storage resources is quite common in serverless applications.

## 5.5. Limitations

The model provided for the influential factors of serverless functions has the limitation that it cannot be guaranteed to be complete. However, the author of this work discovered no other factors in the applications investigated which makes the possibility of important other factors not so relevant for the instrumentation of a testing framework.

The serverless applications considered are also not fully representative of all applications. By selecting them by avoiding bias, their representativeness was

improved. Furthermore, in the selected applications the focus was on the kinds of interactions between the components.

The instrumentation of the tool is limited to JavaScript and is based on the recognition of certain keywords indicating an interaction with another component. For additional support for other keywords or languages, the tool can be extended. Since the instrumentation has to identify entry and exit points as well as variables depending on them, the data flow of the source code has to be analyzed. This is only possible to a certain degree since it is limited by the halting problem. Therefore, the tooling gives a suggestion that has to be confirmed or adapted by the developer.

Furthermore, it cannot be guaranteed that the scenarios selected for measurement are fully representative. The path length, for example, was only one which makes *AllSeq* pass the same amount of tracking information as *AllRel*. If there is a longer path, *AllSeq* requires additional overhead which can have an effect on the execution time. However, the additional execution time can only be caused by the length of the path of the predecessors which is passed and by the calculation of a unique ID which is logged to identify the current workflow.

The information logged is saved by using Amazon's *CloudWatch*. However, the capacity of this logging service is limited if too many logs are saved at the same time. Therefore, if more capacity is needed, another backend for logging is needed which can for example be done by deploying a dedicated logging service with a higher capacity.

It cannot be guaranteed that the runs made are fully comparable. Since the execution time depends on the platform provider and its capabilities assigned for the execution, it can happen that some runs are faster than others. Therefore, the $1000$ warm runs measured for each criterion and scenario were divided into $100$ individual blocks which were executed alternately with the block of the other criteria of the same scenario. So, if the cloud platform provider has a phase where faster or slower resources are assigned for execution, this deviation is more likely to affect other criteria of the scenario as well, not just a single criterion.

## 5.6. Summary

This section introduced a model representing the influential factors for the execution of serverless functions. An evaluation with real-time applications showed that the most relevant influential factors are the input to the serverless functions via parameters and other services, in particular data storage services. Since tool support for the criteria introduced in Chapter 4 did not exist previously, an approach for measuring these criteria was introduced and implemented for the relevant influential factors identified. This approach is also needed in the following Chapter 6 for the creation of automatic test cases to measure which testing targets are covered by a test case (compare Figure 6.1).

The evaluation of the tool showed that the instrumentation does not need much overhead in its execution time in contrast to other monitoring frameworks that added a significant overhead when running. Therefore, it could also be run within a running system for monitoring the control and data flows.

# 6. Automatic Test Case Generation

This chapter shows how integration test cases can be created automatically for serverless applications considering certain coverage criteria. Creating test cases can be a time-consuming task, which is why automation is helpful. Particularly, when many test cases are required, as in regression testing, where the goal is to ensure that the system's behavior remains unchanged, an automatic generation of test cases can help save time. Additionally, since the subsequent Chapter 7 requires a substantial number of test cases to evaluate the efficiency of the test criteria, an objective way for the creation of test cases was needed. Therefore, in this chapter, an approach for the automatic generation of test cases for serverless applications is introduced based on a model describing the structure of the application according to the model introduced in 3.2. This model is used for the identification of potential testing targets. Based on these targets, a tool is implemented that shows for each testing target potential serverless functions that have to be invoked to fulfill the corresponding testing target. Since the behavior of the serverless functions often depends on the data they receive, the tool generates input data for the serverless functions automatically and checks if the generated data cover the corresponding testing target. The content of this chapter is based in parts on [112].

## 6.1. Settings for Serverless Integration Tests

This section investigates how integration test cases for serverless applications are structured and what is necessary for building them automatically. Typically, test cases are executed to uncover errors and to gain confidence in the behavior of a system by showing that no errors occurred during their execution.

### 6.1.1. State of Application

The state of an application before the execution of a test case is in general a relevant aspect for the reproducibility of test cases. This reproducibility is needed to compare different test runs to each other. Additionally, if a test case fails, the ability to replicate the error simplifies the debugging of the error.

Even if serverless functions are mainly stateless, the relevant state influencing their behavior resides in the data received by the serverless functions and their operational environment, such as data storage services. This makes the state of the serverless application also relevant for the execution of test cases.

The state at the beginning of the test case can only be neglected if serverless functions that don't get any input from other services are executed by a test case since the execution is not affected by the state. However, since usually some stateful services are used by serverless functions, as shown in Section 5.3.3, the execution of a test case can be influenced by the state of the system and has to be appropriately set.

There are many potential data combinations that could influence the execution of a serverless function if the execution of a test case is influenced by the state of the application. So, if a certain behavior of a serverless function should be executed, the environment has to be set into the desired state which would trigger this behavior.

This state can be achieved by executing serverless functions influencing the state of the system. If serverless functions influencing the state of the system are known, these functions can be executed systematically within a test case to provoke a certain behavior. Therefore, it makes sense to identify both serverless functions changing the state of the system and to identify functions that are influenced by the state of the system. If this knowledge was not available for the test case generation, serverless functions could only be called randomly without knowing if their execution has any effect on the execution of another serverless function. However, if these dependencies are known, potentially through a dependency graph, potential relevant serverless functions needed for a certain testing target can be identified.

A model, such as the model of Chapter 3, can be used to represent these dependencies. This graph would represent all functions and resources with their relations describing how and if the state of the other component is changed. Thus, the components influencing the execution of a serverless function can be identified by checking the relevant successors and predecessors of a serverless function. This enables the desired behavior to be tested and verified by setting the system into a state by calling serverless functions before the actual behavior to be tested is invoked.

### 6.1.2. Execution Steps

A test case describes some operations that have to be executed in a predefined order on the system to trigger a certain behavior. For the generation of test cases for serverless applications, these operations are considered as calls made to serverless functions in a certain order since the logic of the application resides in these functions. So, the serverless functions can be seen as interfaces that have to be called by the tester in order to execute the intended behavior. If there is an explicit interface for the serverless function, like an HTTP interface, this gateway can also be used for the invocation of the serverless function assuming that the interface calls the serverless function correctly. Therefore,

one or more serverless functions have to be called to execute the logic of the system and to test a specific behavior.

Most serverless functions have parameters that are used by the serverless functions themselves and influence the execution of the function as shown in Section 5.3.3. Therefore, concrete values for these parameters have to be set for these functions in order to make a test case both executable and reproducible.

In order to generate the input data for serverless functions, the format of the input parameters should be known since randomly trying possible arguments, especially complex ones, is not very effective. A schema describing the functions with its input can be used like in [7] where Swagger was used for the definition of the RESTful APIs to be tested.

Consequently, the execution steps of a test case consist of serverless functions being called in a certain order with concrete input data. Therefore, both callable serverless functions and the structure of the data that can be passed have to be available if test cases should be generated automatically.

### 6.1.3. Testing targets

Test cases can be generated randomly without the intention to test a certain aspect or with the intention to cover a certain testing target, such as ensuring that a particular behavior of the application or specific line of code is executed.

Simply calling randomly serverless functions of a serverless application in a random order with random data could by chance also trigger some relevant behavior that has to be tested. However, this approach would generate too many test cases which might not be meaningful. Moreover, it is unclear when to halt the test generation and how to identify relevant test cases. Therefore, rules are necessary to determine the behavior that the test cases should cover.

There are several factors that can be considered for the selection of test cases. One factor is the selection of test cases that reveal errors. However, it is practically not possible to know all test cases revealing errors or test cases that would detect future errors that are introduced by refactoring and would be detected by regression testing. Therefore, a more viable approach for deciding whether to include a test case in a test suite is through the usage of coverage criteria. When a test case fulfills an uncovered testing target of a coverage criterion, it should be added to the test suite. For instance, a common scenario in unit testing is the execution of a specific line of code. But also coverage criteria focusing on the data flow are prevalent. Since the test case generation should create test cases systematically, testing targets are needed which represent the behavior a test case should trigger and test.

### 6.1.4. Measurement of Coverage Criteria

In order to verify if a test case covers the behavior that is intended to be tested, the execution of this behavior has to be measured. When the behavior to be measured are testing targets of coverage criteria, this measurement can be achieved by instrumenting the code of the serverless functions which logs relevant information when the serverless functions are executed. The instrumentation can be done by simply adding log statements indicating that a relevant part of the serverless function was executed.

Data flow criteria require a more complex instrumentation. The information where data were created has to be passed which requires additional instrumentation in the serverless functions creating these data and the serverless functions using these data. The instrumentation has to log the uses of the data passed combined with the origin of the data which can be used to verify to see if a test case has covered its testing target successfully. A practical implementation of the measurement of the control and data flow criteria is already shown in Chapter 5.

### 6.1.5. Test Oracle

A general problem in testing is the creation of test oracles where the result of a test case is tested for correctness. In [12], test oracles are categorized into three categories: specified test oracles where a specification is available, derived test oracles where the correct behavior is derived from different artifacts like documentation or previous system executions, and implicit test oracles where incorrect behavior is determined for obvious cases where the behavior must be incorrect, such as buffer overflow.

Specified and derived test oracles require detailed system information, which is often specific and not easily generalizable. Therefore, only implicit test oracles that show obvious errors in the system can be applied without extensive knowledge of the domain. But even implicit errors may be seen as acceptable [12], e.g., if a serverless function might throw an error because the parameters passed to it are not correct. Therefore, predictions about the desired behavior of an application cannot be made without having some domain knowledge. Nevertheless, a practical approach can be applied for regression tests where the results of a previous version of the application can be used as a reference oracle. If the test cases are used in stress testing, their outcomes can also be determined by executing the test cases sequentially without generating much load on the system and by interpreting these outcomes as the desired behavior.

Since the state of serverless applications resides in resources like data storage services and the data passed, the test oracle must compare these data. So, test cases can be checked by comparing the results of calls and the state of stateful components, such as data storage services, of the application after the

execution of the test case to the ones generated in a previous run. However, if the execution of a test case returns an error message, it has to be checked manually if this behavior is intended which is not always clear if no specification is available. Potential values that have to be checked can be the return values of the serverless functions, the logs produced, or other stateful services. Depending on the test case, it is enough to check the state of only a few stateful components of the application instead of all. Therefore, dependencies of the components are useful information that indicate potential stateful services that can be affected by the test case.

### 6.1.6. Requirements for a Suitable Model

The test case generation is based on a model which has to support the following aspects for an automatic test case generation:

- **Representation of resources:** The resources, particularly serverless functions, are relevant for the identification of potential workflows and data flow. The resources must be available in order to identify potential testing targets for *AllRes* and for the representation of dependencies of the application.

- **Type of resource:** The concrete type of the resource is relevant for the construction of workflows and data flows.

- **Dependencies between resources:** Dependencies are required to identify potential workflows and data flows of the serverless application. Furthermore, *AllRel* requires them for the identification of its testing targets.

- **Interfaces of functions:** Interfaces are needed for the creation of test cases to actually run functions.

- **Defs of data transmitted:** The defs are needed for the identification of all testing targets of the data flow criteria. Furthermore, the measurement of the defs requires the location of the defs for the instrumentation of the source code.

- **Uses of data transmitted:** Similarly to defs, the uses are also needed for the identification of the data flow criteria and their locations are required to instrument the source code for the measurement of the uses.

- **Mapping of source code to resources:** The source code of the serverless functions must be assigned to the resources representing serverless functions to enable the instrumentation of the source code for the measurement of criteria.

- **Locations of invocations of resources:** The information where the code of serverless functions calls another resource is relevant in order to adapt these interfaces to pass the relevant data needed for the measurement of the criteria.

- **Type of data storage access:** By knowing the type of data storage access of a dependency, potential data flows can be constructed. Furthermore, def-use-pairs coupled via a data storage resource can be identified which is relevant for the identification of all testing targets of the data flow criteria.

These aspects are supported by the different dependency graph versions introduced in Chapter 3 with the Definitions 3.1, 3.2, 3.3, 3.4, 3.6, 3.8, 3.9 and are used as a basis for the automatic test case generation. As shown in Chapter 3, the creation of the representation of the resources, the type of resources, the dependencies and the type of the data storage resource access can be automated to a certain degree if the deployment file and source code of the application are available. This deployment file can also be used for assigning the source code to the corresponding serverless functions. Furthermore, in Chapter 5 the identification of defs and uses, and the identification of the locations of resource invocations are supported by analyzing the source code. This information is identified generally for each function but can be assigned to the interacting dependency. However, the interfaces of the serverless functions are more complex where the format of the input has to be known or derived by analyzing the usage of the potential input.

The aim of the automatic test case generation for serverless functions in this chapter consists of two phases. In the first phase, relevant serverless functions have to be selected and their order of invocations be set. The second phase fills the function parameters with meaningful data which finally represents the behavior that is intended to be tested.

## 6.2. Test Generation Approach

In the subsequent section, the behavior that is intended to be tested is represented by the testing targets of the data flow and control flow criteria introduced in Chapter 4. Since *AllSeq* produces potentially an unlimited number of potential paths that can be created and the interactions are also covered by *AllRel*, *AllSeq* is not supported by the implementation.

Based on these criteria, approaches for the identification of the execution steps for potential test cases are described. Since the statically generated test steps need to be executed with input data, also potential approaches for the identification of the input data for these execution steps are introduced.

### 6.2.1. Static Generation of Test Case Structure

Each testing criterion requires that serverless functions are identified as the execution steps that have to be called in order to fulfill its testing targets. By analyzing the dependencies of calls between the resources of serverless functions, potentially through a dependency graph, a test case structure can be built. This structure comprises an ordered sequence of serverless functions identified for invocation with specific input data to potentially cover the testing target. However, the structure identified does not guarantee that a scenario exists where the testing target with the right input data is triggered, but the structure calls at least resources that are related to the statement that has to be fulfilled to trigger the corresponding testing target.

The testing targets for the criterion *AllRes* are all the resources of the application while the testing targets for the criterion *AllRel* can be derived from the dependencies between the resources which are available. If the resource to be covered is a serverless function, this single serverless function can simply be used as the test case structure. Otherwise, if a relation for *AllRel* or a resource for *AllRes* which is not a serverless function has to be covered, the first serverless functions preceding the relation respectively the resource can be used as the test case structure.

In contrast to control flow criteria, the identification of the testing targets for the data flow criteria also requires the knowledge of potential definitions of data passed and their uses. This knowledge can be derived from the dependencies between resources where this information can be attached. If a serverless function, for example, passes data to another serverless function, the relation representing this dependency contains potential locations of the data definitions or if data are used by a serverless function from a data storage, the corresponding read relation contains also the information where the data are used. If this information is available, all combinations of definitions and uses can be identified by considering all potential following uses of a definition. These are the testing targets for the criterion *AllUses*.

The testing targets of the criteria *AllDefs* and *AllDefUse* can be simply calculated by counting the definitions and uses assigned to the relations.

There are several general scenarios for potential data flows which can be represented by a test case. A serverless function or one of its following components creates data on a stateful component which is finally used by another serverless function. This use can be covered by a serverless function having to be called explicitly which results in an additional serverless function being called for the test case. If the serverless function containing the use is triggered by the stateful component, no additional serverless function is needed for the test case structure, and it is enough to simply call the serverless function triggering the create operation.

Another scenario is that data are passed from one serverless function to another one directly or via other services between. If so, only the first serverless

function has to be selected for the test case. However, if a serverless function invokes another serverless function that returns a value that is used by the caller, the test case consists only of the caller.

If there exists a definition in a test case that calls a delete operation on a data storage, an additional function has to be called which tries to create the data which should be deleted. Therefore, all serverless functions are identified which could create such data which have to be deleted later. Each of these functions can be combined with the existing structure of the test case of the testing target and represents a potential data flow.

Since testing targets of the criteria *AllDefs* and *AllDefUse* are more general by just requiring that any uses or definitions are enough to be covered for corresponding definitions or uses, a potential test case for the fulfillment of the corresponding testing target is built by each of the potential definitions and uses. Here, the structures of test cases identified for the *AllUses* criterion can be used and assigned to the testing targets.

The state of the application is also crucial for the execution of a test case in some cases. Therefore, it can make sense to change the state of the application if the test case uses some data of stateful services. Similar to a potential state change before the execution of a delete operation, these structure of test cases can be extended by adding additional functions creating some data for the stateful components.

Additionally, if a test case uses some stateful resources, not only the return values of the serverless functions but also the state of the stateful resources should be checked to see if a test case passed or failed. These resources can also be identified statically by investigating the resources that are potentially affected by the execution. Therefore, the dependencies available can be used to suggest resources that should be checked.

This approach does not cover testing targets of the control flow criteria like components or relations that are not reachable by serverless functions. If such components or relations have to be covered, an additional serverless function can be added to the serverless application which is able to call these. A data storage resource that is manually filled and triggers a serverless function, for example, is not covered by this approach. However, if a serverless function is added that saves data on the data storage, both the data storage resource itself and the relation between the data storage resource and the relation to the function called by it can be covered.

### 6.2.2. Dynamic Input Data Generation

While the structure of test cases can be created statically, it is not clear which input data are required to fulfill the testing target that is intended to be tested. Therefore, it is necessary that the source code is instrumented to measure the coverage of the test cases in order to evaluate different input data. Furthermore,

these functions have to be invoked with data that fulfill the intended testing target to be covered. The information of the execution has to be evaluated and used to adapt the data for the test case.

Various strategies for the selection of data are suggested. If there is only one function to be called in a test case structure, the strategy is always the same. A random value of the input format is chosen, and the function is executed. After that, the coverage of this test case is evaluated, i.e., it is checked if the testing target was covered. If the testing target was not covered, the procedure can be repeated.

However, test cases for the coverage criteria investigated here consist of several serverless functions only if the intention of the test case is that there exists a data flow between some of these serverless functions. This means that they are coupled via a stateful component like a data storage resource and the functions have to access the same data to trigger the data flow. Therefore, the input parameters have to be adapted in order to access the same data. If the parameters are chosen randomly according to the input format, the likelihood is very small that the same data are accessed.

Therefore, strategies are introduced in order to adapt the parameters to increase the likelihood that the same data are accessed assuming that the input and output data are in JSON format having some key-value pairs. While the data of the first serverless function of a test case structure are still generated randomly, the values of the key-value pairs of the following functions can be generated by using the following strategies.

- **Same Input**: Use the value of a key-value pair of a previous input with the same key.
- **Random Input**: Use the value of a random key-value pair of a previous input.
- **Same Output**: Use the value of a key-value pair of a previous output with the same key.
- **Random Output**: Use the value of a random key-value pair of a previous output.
- **Same Values**: Use the same random value for all values.

Workflows typically require that the same data are accessed, e.g., if a value is created which is used later in the workflow. The heuristics listed here can improve the likelihood that the same data is accessed if the same value which is relevant for the data access already occurred in some input or output data. While the very primitive strategy *Same Values* sets all values to the same value, the other strategies provide a broader range of potential input data by only selecting one certain value. If a value is chosen depending on the same key of a

previous input or output, the variety of inputs is also smaller than for choosing a random value of a previous input or output.

If only the values of input key-value pairs are used, the input data for the test cases can be created before their execution since these data do not depend on output data which depend on the executions of serverless functions. Otherwise, the data have to be adapted dynamically while running the test cases.

Afterwards, the testing target has to be checked which requires that the execution measures the coverage of the coverage criteria. Depending on the coverage, the data of the test case are calculated again, and the test case is reexecuted without considering any results of the previous run.

## 6.3. Tool Implementation

The static and dynamic generation of test cases described in the previous section is implemented in a tool whose context is given in Figure 6.1 where the parts of the tool have a gray background.

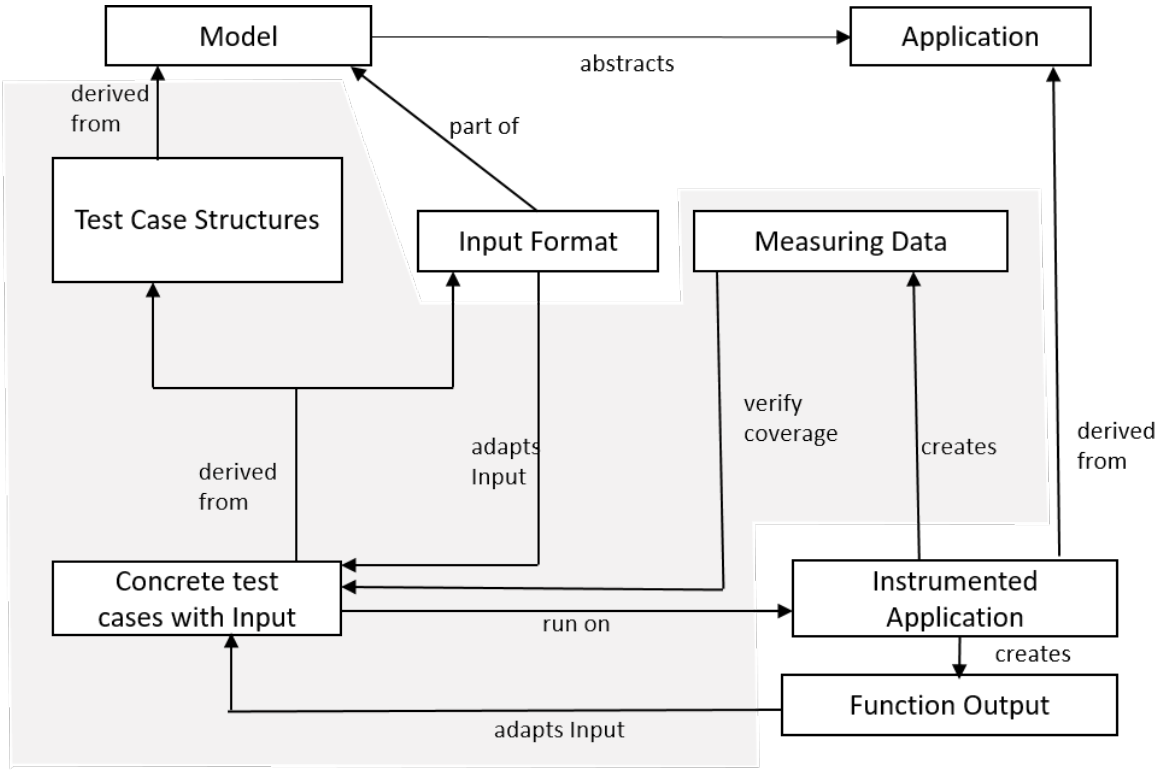



Figure 6.1.: Context of tooling (gray) in test case generation

The test case generation is based on a model that represents the serverless application and contains the necessary information for the automatic creation of test cases as listed in Section 6.1.6. Each of the resources, such as serverless functions or data storage services, is described by a node in this model. The model can be seen as a dependency graph that describes if one resource uses another resource, e.g., a serverless function calls another serverless function

or reads data from a data storage. Furthermore, if data storage services are accessed, the relation to the data storage resource can be annotated with the CRUD operations create, read, update and delete which is relevant for the construction of data flows.

Additionally, for the representation of the data flows between the resources, the definitions of values passed between serverless functions and their uses are assigned to the relations. This information is needed for the first phase of the test case generation where testing targets are identified and the structures of potential test cases are calculated. For the second phase, a potential input format for the nodes is needed which has to be assigned to the nodes representing serverless functions. The tool supports JSON as input format where the range of potential inputs for the values of a JSON key is represented as a regular expression or a range of integers.

So, a model according to the Definitions 3.1, 3.2, 3.3, 3.4, 3.6, 3.8, 3.9 is supported.

The first phase creates potential test case structures for the testing targets of the coverage criteria according to Section 6.2.1. Therefore, the testing targets are identified by walking the graph for potential testing targets. For each testing target, potential orders of serverless functions according to the model are suggested.

Test cases for testing targets of control flow criteria are created by finding the first serverless function calling the resource or the relation to be covered according to the general approach of the previous section. Additionally, if the serverless function identified reads data from a data storage resource, additional functions are searched on the graph that create data on the data storage. Each of these functions is combined with the existing test case and suggests a new test case structure for the testing target.

The testing targets for the data flow criteria *AllDefs* and *AllDefUse* are identified by walking the graph and checking the nodes for definitions and uses that were defined.

For the *AllUses* criterion, the graph is walked, and successor nodes are identified which could potentially use the value in order to find all combinations of definitions and uses as testing targets. The structures of test cases for the data flow criteria require that a serverless function defines some data that are used by another serverless function. Therefore, all potential combinations of serverless functions are identified, which could cover these testing targets, by walking the graph. These combinations are assigned as potential test cases to the corresponding target.

Additionally, if a data flow testing target has to cover a definition based on an update operation, all scenarios where the updated data can potentially be created by a single serverless function are created and suggested as test case structures. If the first serverless function of a test case structure tries to read data from a data storage resource, additional structures are created which also

try to create data before the read operation is executed. Thus, several structures of test cases for the fulfillment of the testing target are potentially suggested which have to be selected individually for the next phase where the data are set for the functions.

Furthermore, all resources whose state or output could be influenced by the execution of a test case are suggested by the tool. These resources should be checked to see if a test case passed or failed which is not part of the tool since this requires additional domain knowledge and potential outputs like time-dependent values are often not constant.

For each of the test cases, measurement statements are created which have to be covered later and be evaluated if the testing target assigned to the test case was fulfilled. If additional operations are added to the test case structure that do not focus directly on the testing target, measurement statements are created which express if these additional serverless functions have an influence on the serverless functions actually covering the coverage target.

Afterwards, the input data for the functions can be created which requires the model to be annotated with the potential input of the function which can be added as a regular expression. Furthermore, since the fulfillment of the criteria has to be checked afterwards, the tool requires that the fulfillment of the criteria can be measured.

The dynamic part of the tool executes the functions with the input data created on the platform and checks afterwards if the testing target was fulfilled. Therefore, an instrumented version of the application with the measurement statements has to be deployed before running the test case data generation.

The execution can be adapted so that the strategies for the input data generation introduced in the previous chapter are applied with a certain probability for each key-value pair. If for a key-value pair no strategy is assigned according to its probability, the input data are generated randomly. If the strategy uses an output value, the test case saves the information of the position and key of the output value as an abstract description of the value to use it dynamically to be able to reproduce the test case. After each run, the state of the application should be reset due to reproducibility, why the possibility was added to add a function that resets services having a state and which is executed before each run.

So, the tool creates test case structures from a model which are used for the creation of concrete test cases which are evaluated against the application and adapted if not successful.

## 6.4. Evaluation

The tool was applied with the strategies implemented on two serverless applications which were selected on *GitHub* to show the applicability of the approach.

### 6.4.1. Model Description

On September 11, 2021, GitHub[34] was searched for serverless applications running on *AWS* and using *serverless framework*[35] which is a tool commonly used by developers [57]. The same tooling was applied as in Section 4.3.1 and Section 5.3.1.

Furthermore, the serverless functions of the applications should run on *Node.js* and the deployment with the *serverless framework* be possible without many adaptions of the application. The data and tooling for the selection and the following test case creation is available online[36].

The first application[37] uses several serverless functions reading and writing data on a data storage service (compare Figure 6.2).

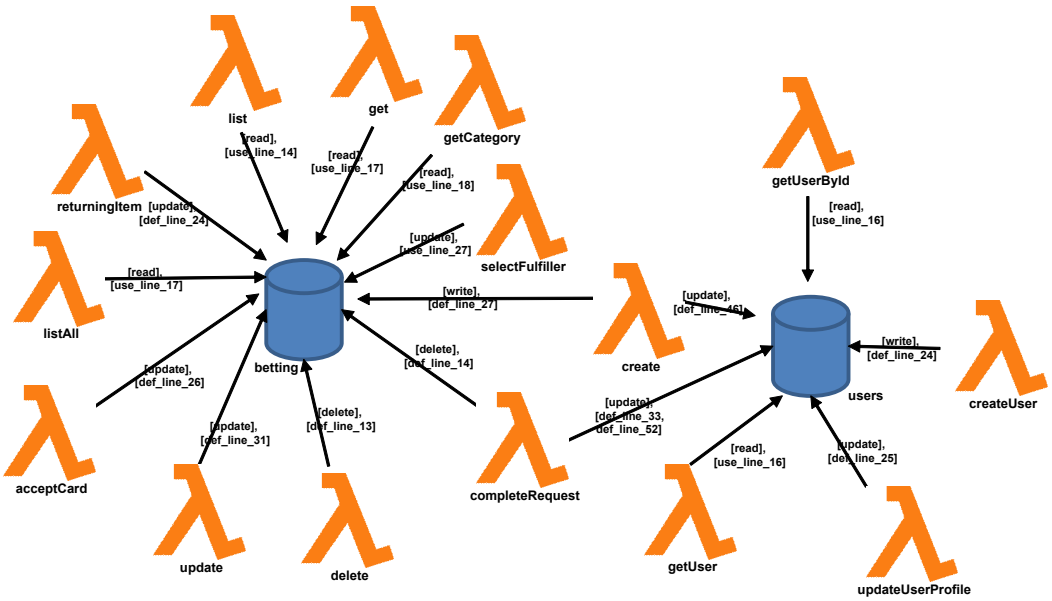



Figure 6.2.: Data-centric application

The second application investigated[38] consists not only of serverless functions accessing data storage services, but also of other services like a stream, queue, and email service which are involved in the data flow (compare Figure 6.3).

Models of these applications were built shown in Figure 6.2 and Figure 6.3 which contain the relevant information to identify the testing targets and to build structures for potential test cases.

If an arrow accesses a data storage, the type of access is assigned to the arrow. Additionally, the lines of the serverless functions influencing the data flow are assigned to the arrows in order to identify potential data flows between the

[34] https://github.com/
[35] https://www.serverless.com/
[36] https://github.com/snwinz/ServerlessCoverageTesting/releases/tag/v0.6
[37] https://github.com/milllslm/Jackal
[38] https://github.com/debrajpaul/serverless-cake-ordering-system-

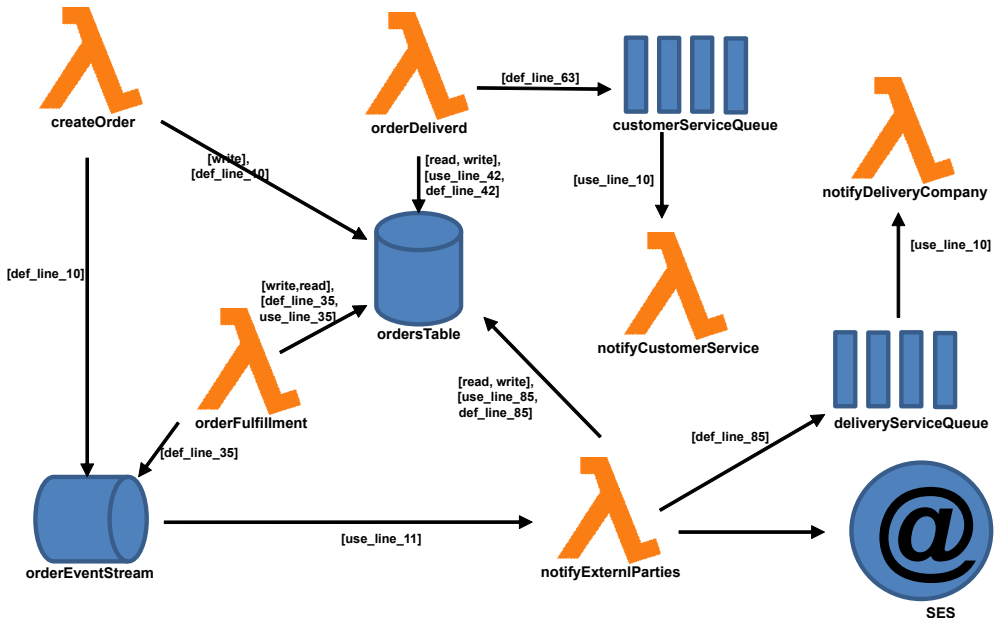



Figure 6.3.: Application with workflow

serverless functions. So, the model complies with the Definitions 3.1, 3.2, 3.3, 3.4, 3.6, 3.8, 3.9.

Since no specification for the input values of the serverless functions was given, it was examined how the input values were used. If they were used and compared to a fixed value, the input was defined as the values which were used for the comparison and one value which was not used for comparison. If the input was used as a string, it was specified as a regular expression representing any combination of characters of the Latin alphabet and numerical digits. And if the input was used as an integer, the input was specified as a value between $0$ and $10000$.

### 6.4.2. Test Case Creation

These models of the applications that were selected on *GitHub* were used for the creation of testing targets. The tool implemented identified all testing targets and suggested for each of them all potential test case structures. These test case structures consisting of one or more serverless functions to be invoked were tried to fill with input data automatically. Furthermore, the applications were instrumented in order to measure the coverage.

Since the code was nested and distributed in several files, the instrumentation was done manually. A bug was found during the instrumentation of the second application. The serverless function "notiyExernlParties" [sic] started a write operation to the "ordersTable" and the invocation to the "deliveryServiceQueue" asynchronously without waiting for them to finish. Therefore, the serverless functions can be finished before these operations are executed. If the same

runtime environment of the serverless function is invoked again, these operations are continued which could obfuscate this bug. However, also here some other operations would be started in the background without being finished. All data flow criteria have at least one testing target which could not have been covered if this bug was not fixed. Therefore, this bug was fixed in order to make the creation of these testing targets possible. However, this shows that by considering the testing targets potential bugs can be detected.

The coverage of the application which could be reached with automatically created test cases can be seen in Table 6.1.

Table 6.1.: Coverage of applications

| | Targets Created | Targets Covered | Structures of TCs | Covering Structures |
|---|---|---|---|---|
| **Application 1** | | | | |
| AllResources | 17 | 17 (100%) | 44 | 42 (95.45%) |
| AllRelations | 18 | 18 (100%) | 24 | 23 (95.85%) |
| AllDefs | 12 | 9 (75.00%) | 62 | 26 (41.94%) |
| AllDefUse | 18 | 16 (88.89%) | 124 | 71 (57.26%) |
| AllUses | 38 | 24 (64.16%) | 62 | 26 (41.94%) |
| $\sum$ | 103 | 84 (81.55%) | 316 | 188 (59.49%) |
| **Application 2** | | | | |
| AllResources | 11 | 10 (90.91%) | 25 | 10 (14.29%) |
| AllRelations | 15 | 12 (80.00%) | 68 | 12 (17.65%) |
| AllDefs | 5 | 4 (80.00%) | 64 | 6 (9.38%) |
| AllDefUse | 11 | 9 (81.82%) | 128 | 10 (7.81%) |
| AllUses | 16 | 8 (50.00%) | 64 | 8 (9.38%) |
| $\sum$ | 58 | 43 (74.14%) | 394 | 46 (11.68%) |

The table lists the number of testing targets for each of the criteria for both applications and how many of them could be covered. Since for some testing targets several potential test case structures are built statically, the number of them is listed. Furthermore, the numbers of structures that could be filled with data to fulfill the corresponding testing target of the test case structure are listed. The tool tried to generate input data up to $10$ times for each test case structure by applying for each of the inputs the same strategy. This approach managed to create an overall coverage of the criteria of around $81.55\%$ and $74.14\%$ for the first and second applications. While in the first application around $59.49\%$ of the created test case structures could be filled with data, the second application could create valid input data for around $11.68\%$ of the test case structures.

The reason for the latter is that some serverless functions of the second application invoke relations that are relevant for some testing targets only

when other serverless functions were already executed. Therefore, only a few of the proposed test case structures could be filled with data covering the corresponding testing target at all.

The number of testing targets that were covered by a single strategy can be seen in Table 6.2.

Table 6.2.: Uniqueness of strategies of testing targets

| | Other Successful Strategies | | | | | | |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | $\sum$ |
| **Application 1** | | | | | | | |
| No Strategy | 0 | 0 | 0 | 0 | 0 | 51 | 51 |
| Same Values | 1 | 4 | 19 | 8 | 0 | 51 | 83 |
| Same Input | 0 | 4 | 17 | 8 | 0 | 51 | 80 |
| Random Input | 1 | 0 | 18 | 8 | 0 | 51 | 78 |
| Same Output | 0 | 0 | 0 | 0 | 0 | 51 | 51 |
| Random Output | 0 | 0 | 3 | 8 | 0 | 51 | 62 |
| **Application 2** | | | | | | | |
| No Strategy | 0 | 0 | 0 | 0 | 0 | 19 | 19 |
| Same Values | 0 | 0 | 0 | 0 | 0 | 18 | 18 |
| Same Input | 0 | 0 | 0 | 0 | 1 | 19 | 20 |
| Random Input | 0 | 0 | 0 | 0 | 1 | 19 | 20 |
| Same Output | 6 | 17 | 0 | 0 | 1 | 19 | 43 |
| Random Output | 0 | 17 | 0 | 0 | 1 | 19 | 37 |

Here, the last column shows how many testing targets could be covered by the strategy at all. The other columns indicate the uniqueness of the strategy. A cell here lists the number of testing targets that are covered by the criterion while the testing target is also covered exactly by the number of other strategies given in the column. For example, there are four testing targets for the first application and the strategy "Same Input" where also one other strategy (here "Same Values") was able to create input data dynamically which covered the corresponding testing targets. But no testing target is exclusively covered by the strategy "Same Input". In contrast, the strategy "Same Values" covers exclusively one testing target that could not be covered by another strategy.

While for the first application the strategy "Same Values" is the most successful one, it is the least successful for the second application. Vice versa, the most successful strategy of the second application "Same Output" is the least successful one for the first application. While in the first application, many testing targets can be triggered by the same input data, in the second application the focus is more on the output.

In Table 6.3, the same data are listed for the test case structures created for the testing targets where additionally the number of total runs is depicted

which was needed for the fulfillment of all the test case structures that could be filled with data covering the corresponding testing target.

Table 6.3.: Uniqueness of successful test cases and runs needed

| | Other Successful Strategies | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | $\sum$ | runs |
| **Application 1** | | | | | | | | |
| No Strategy | 0 | 0 | 0 | 1 | 0 | 76 | 77 | 81 |
| Same Values | 4 | 21 | 71 | 12 | 0 | 76 | 184 | 185 |
| Same Input | 1 | 14 | 61 | 12 | 0 | 76 | 164 | 171 |
| Random Input | 1 | 6 | 71 | 11 | 0 | 76 | 165 | 274 |
| Same Output | 0 | 0 | 0 | 1 | 0 | 76 | 77 | 77 |
| Random Output | 1 | 3 | 10 | 11 | 0 | 76 | 101 | 173 |
| **Application 2** | | | | | | | | |
| No Strategy | 0 | 0 | 0 | 0 | 2 | 18 | 20 | 26 |
| Same Values | 0 | 0 | 0 | 0 | 1 | 18 | 19 | 21 |
| Same Input | 0 | 0 | 0 | 0 | 2 | 18 | 20 | 27 |
| Random Input | 0 | 0 | 0 | 0 | 2 | 18 | 20 | 24 |
| Same Output | 9 | 15 | 0 | 0 | 1 | 18 | 43 | 51 |
| Random Output | 2 | 15 | 0 | 0 | 1 | 19 | 37 | 98 |

Depending on the application and its relation of the input and output data to the data stored in the application, either the output or input approaches were more successful. Of course, if the input or output values are chosen randomly from a set of previous inputs or outputs, the expected number of runs until a successful execution is larger. In general, if a test case was successful, not many runs were necessary to create these data. Therefore, it is advisable to try other testing structures of the same testing target first where fewer runs were made before a new run is started for the test case structure if it is intended to find valid test cases quickly.

Furthermore, the approach to extend test case structures with additional serverless functions creating data that are used by the original identified serverless function of the test case structure resulted in 19 testing targets which could be covered only by these templates which all belonged to *Application 2*.

For the testing targets that could not be covered, it was investigated why no test case could be created, which would fulfill these testing targets. Two testing targets could not be covered since the workflow of a function changing a value in a data storage resource changes the same value again. Thus, the read access to the value depends on the time which cannot be handled by the tool which waits ten seconds after each invocation of a serverless function in order to make sure that all operations were executed. Three testing targets could not be fulfilled even if the relevant input data of a function were available in

the input or output values of the runs. These testing targets were covered if more runs were executed which would parse the corresponding values from the input and output data. However, there were 16 testing targets where an input value of a serverless function corresponds to a relevant ID of a data entry that was not available in input or output data. Thirteen testing targets could not be covered by applying always the same approach. However, if the approaches were not applied on all the input values, e.g., by setting to use an approach with a probability less than 1 for an input, these targets were also fulfilled after some runs.

## 6.5. Related Work

In [71], Manner et al. created simple templates for unit tests of serverless functions by investigating the log files of running systems. This requires a productive system where the application is already used by users producing a workload triggering the serverless functions of the application. This is not a requirement for the approach described in this chapter of the work where the test cases are created based on a model. Furthermore, the templates created in [71] create unit tests, not integration testing as generated by the approach of this chapter.

A test case generation based on the *Web Services Description Language specifications* (WSDL) was introduced in [8] by Bai et al.. Similarly to the approach here, the framework considers also test cases where not only a single operation has to be invoked, but also combinations of operations. These required operations are identified based on data dependencies that have to be given. However, since WSDL is only an interface description language, such an approach could only represent the serverless functions with their interfaces but not their connected services and their relations among each other.

Integration test cases focusing on the coverage of control flow criteria were created automatically by using the REST APIs of applications in [7]. However, this work does not focus on serverless applications and the specific coverage of testing targets also including data flow coverage. Although coverage criteria considering the data flow for integration are by no means new (compare [20, 33, 97]), there is no work available yet creating such test cases for serverless applications automatically.

## 6.6. Limitations

The possibility of creating test cases automatically depends strongly on the application and the provided set of test case structures that can be derived from the application. Therefore, it might happen that in serverless applications with a completely different structure, the automated generation of test cases

is not as successful as in this chapter. However, the approach can be adapted by adding new test case structures or new strategies for the generation of the input data of the serverless functions.

Since the approach here is based on a heuristic, it can only be checked if the test case structure with its input data generated was successful. If the data generated don't cover the testing target, new data are generated and the process is repeated. However, there is no termination condition when to stop this process. Here, the generation stopped after $10$ repetitions. These were enough repetitions to identify valid input data for each testing target which could be covered by the approach applied here. Testing targets that could not be covered were analyzed manually where it was identified that more repetitions wouldn't have identified valid input data.

## 6.7. Summary

This chapter showed how integration test cases for serverless applications can be built and created automatically based on a model representing the serverless application. For each of the testing targets identified, an individual test case is generated which generates a more specific coverage in contrast to a pure random testing approach.

The approach identifies potential test case structures first which could cover certain testing targets. Such a test case structure consists of one or more serverless functions which have to be invoked sequentially. In another step, these test case structures are invoked with data that were generated with different strategies and measured if the corresponding testing target was covered. In the applications investigated here, not many runs were needed if a testing structure could finally cover a testing target with the data generated.

Developers can benefit from this approach by having a larger number of test cases generated automatically which is in particular useful for regression and stress testing. Furthermore, by generating test cases automatically a more objective set of test cases can be generated which is finally used in the evaluation of Chapter 7.

# 7. Evaluation of criteria

While the previous chapters introduced coverage criteria and approaches to measure these criteria and create test cases automatically, it is not clear if these coverage criteria help reveal faults. Coverage criteria are often introduced and suggested as a valuable objective in test case creation. Even if it sounds reasonable to use them, their potential is often not investigated.

Therefore, this section investigates the control flow criteria and data flow criteria for their potential to expose faults. Three applications are investigated for their potential to reveal fault by using mutation testing. Some faults are injected into the application and measured how different test sets can detect the faults injected. The content of this chapter is mainly based on my publication [113].

## 7.1. Methodology

In this section, a quantitative approach was chosen to investigate the efficiency of various coverage criteria. A test pool consisting of test cases built randomly and test cases covering different testing targets was created. Based on this test pool, random test suites and test suites fulfilling certain coverage criteria were built. Mutation operators were used to inject faults in three serverless applications to assess the effectiveness of these test suites. These mutated applications were run against the test suites. The amount of detected injected faults indicates the efficiency of the test suites. The coverage criteria, mutation operators, and applications investigated are described in this section. Furthermore, it is shown how the test cases and test suites were generated for the evaluation and how the oracle worked which was used. Lastly, the procedure describes how test cases are executed against mutated applications and how it is measured.

### 7.1.1. Application Selection

The two applications identified in the previous Chapter 6 based on a search on *GitHub* were selected. Additionally, a third application was selected which is used in two other papers. This application was both the most complex one in [3] and the one with the most functions in [93] which makes it appropriate to have another more sophisticated example application.

A model of these three applications was built containing the relevant information for the identification of the testing targets and for the creation of potential structures for test cases which were built similar to the model creation in Section 6.4.1. This model is based on the Definitions 3.1, 3.2, 3.3, 3.4, 3.6, 3.8, 3.9.

This included that services and serverless functions are represented by nodes while arrows indicate that nodes interact. For data storage services, also the kind of CRUD operation is annotated. The line of code influencing the data flow is also assigned to the arrows. Additionally, for the creation of automatic test cases, potential input data were required which were derived from the usage of the input parameters. The input data could be modeled in JSON syntax for all applications. Any possible combination of characters of the Latin alphabet is assigned to the input value if the input is used as a string, a potential value between $0$ and $10000$ if it is used as an integer. If the input value is compared to fixed values, all fixed values and another value are assigned as a regular expression. While the first application[39], *App1*, and third application[40], *App3*, use several functions reading and writing data on data storages (see Figure 7.1 and Figure 7.3), the second application[41], *App2*, triggers also other services, such as stream, queues and an email service, in its data flow (see Figure 7.2).

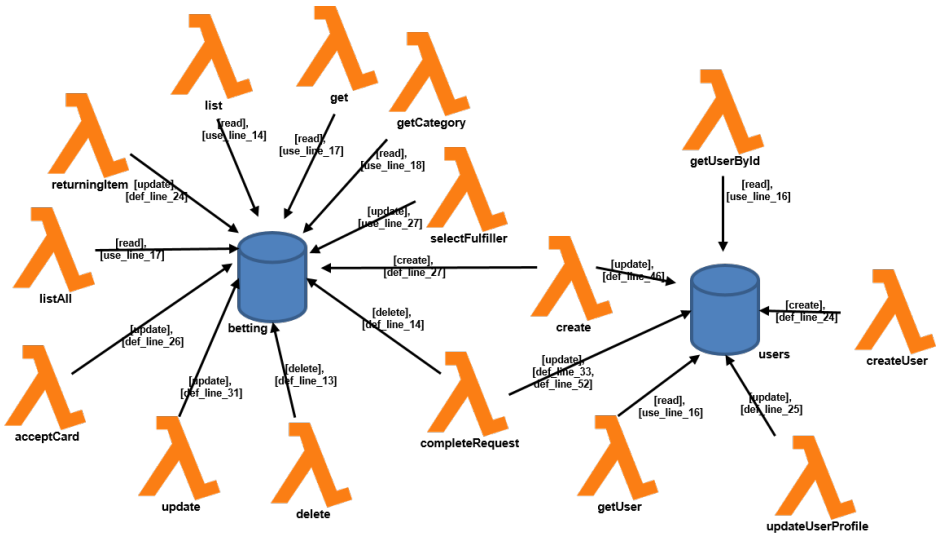



Figure 7.1.: Data-centric application identified via GitHub search

## 7.1.2. Test Case Generation

The models of the applications were used for the test case generation. The approach described in Chapter 6 was used, which enables the creation of many test cases that are less biased than manual test cases. This approach uses the

[39] https://github.com/millslm/Jackal
[40] https://github.com/anishkny/realworld-dynamodb-lambda
[41] https://github.com/debrajpaul/serverless-cake-ordering-system-

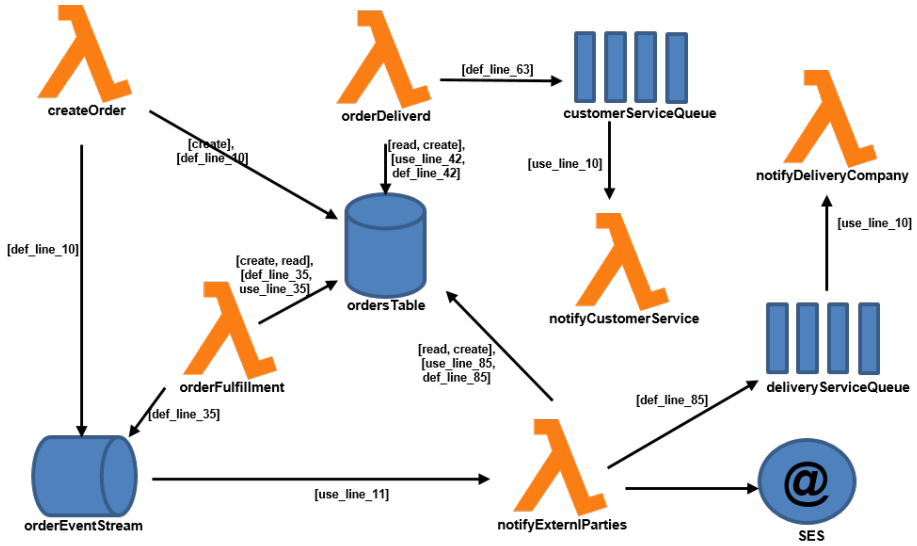



Figure 7.2.: Application with workflow identified via GitHub search

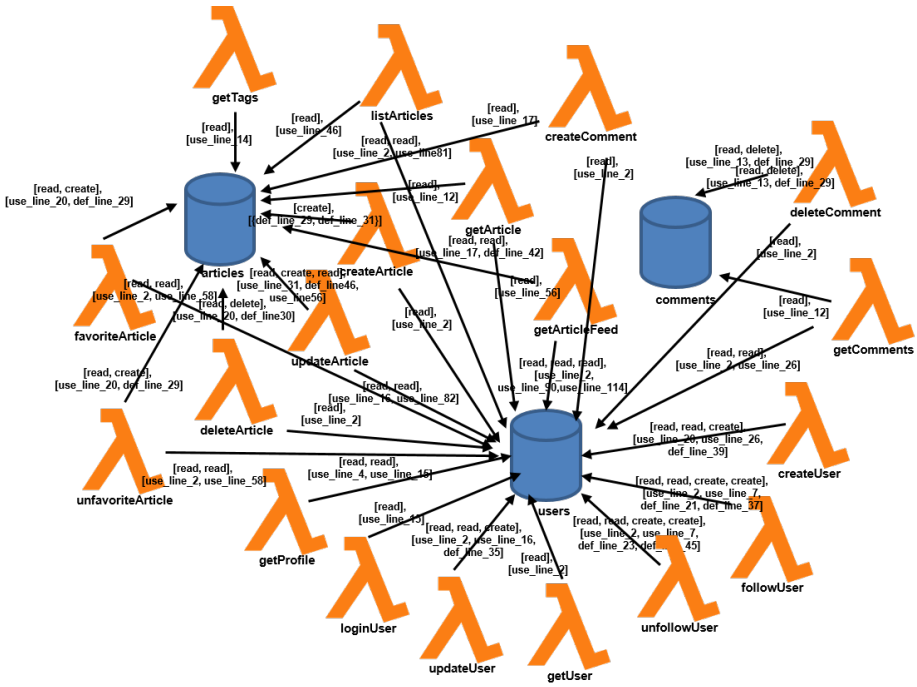



Figure 7.3.: Data-centric application identified in other research papers

model to identify potential orders of functions that could be called to fulfill certain testing targets. These structures of serverless functions have to be called with values assigned to their parameters and checked if the testing target was covered. The input values were generated randomly or filled with data that could be derived from input or output values of previous invocations of serverless functions of the same test case. For example, if a serverless function returns a key-value pair whose key is used in a subsequent call of the test case, an attempt is made to use the value of the key-value pair in the subsequent call. Additionally, before each test case, a serverless function is called deleting all the data on the data storage. This ensures that a test case is executed within the same state and is reproducible. Furthermore, the applications are instrumented to measure if a test case fulfills a testing target. So, it can be checked if a potential test case fulfills the testing target that it should cover. If a test case does not fulfill the testing target, the test case has to be adapted by using other input data.

However, there were some testing targets where test cases could not be generated automatically covering these testing targets. For these testing targets, test cases were created manually by calling the least possible number of serverless functions with data that are necessary to fulfill the testing target.

A random test case generation was also implemented, which creates test cases randomly based on the model. A random test case is generated by selecting a certain number of serverless functions from the application randomly which are called by the test case. These functions are filled with random input data based on the possible input defined in the model. So, test cases with a certain number of serverless functions can be created that have valid input data for the serverless application to be executed. These test cases can be used for comparison later.

Since $App3$ requires for most of the functions an authentication, a test user was added whose token is used if a serverless function requires an authentication. So, if the key of a key-value pair of the input represents an authentication, the token of the test user is assigned to it or, if available, a token, which is created in one of the functions called within the same test case, is extracted from the output and assigned to the key. Thus, it is ensured that the logic of a serverless function is called, and the request is not rejected immediately.

Even if calling serverless functions can result in an erroneous state, this state has to be detected. Therefore, a test case oracle is needed, which indicates how the test case should behave. Simply checking for errors thrown by the test case is not enough since errors thrown can also be an intended behavior [12] or the application behaves slightly differently without throwing an error.

Since test cases are needed for the evaluation to detect injected faults, the original application is chosen as a test oracle producing the intended behavior. For each test case, the results and logs generated by the functions were recorded and assigned to the test case. Since the outputs are not constant when the test

case is executed again, the dynamic parts of the outputs have to be identified and ignored.

However, some parts of the output, which are not constant, can also be derived from the output data of previous calls. Therefore, when the intended behavior of the test cases is created, it is checked initially for each output value of each key-value pair if the value is contained within a previous output of a serverless function of the test case. If a similar value is identified, such as an ID that is created in a previous serverless function of the test case and returned by it, the test case should assume that this is the intended behavior and check this dynamic value in its executions by extracting the corresponding data from the outputs.

In order to eliminate other dynamic values, the test cases were executed two times and their results were compared. By extracting parts with a minimum length of three characters from the first result which are also present in the second string, potential constant parts of the results were created. However, some parts, which were present in both results, were not constant. For example, time-related data in a result might not be recognized as a dynamic component if the test case is executed on the same day. These parts must be removed too. This is done by identifying the characters that usually surround the dynamic part and removing these parts. Usually, the dynamic parts were surrounded by a time- or id-related keyword in the test cases created which helped to identify and remove the dynamic parts.

### 7.1.3. Test Suite Generation

Several test suites for the experiment were created that are used for the evaluation of the coverage criteria.

For each coverage criterion, a test suite was created focusing on the fulfillment of that criterion. Each test suite contains for each feasible testing target a test case focusing on the coverage of this specific testing target. Therefore, the size of each test suite is equal to the number of feasible testing targets for the corresponding criterion. If a testing target could not be covered automatically, the corresponding test case was created manually by the author. Furthermore, since two def-use-pairs of *App2* were infeasible, they were not considered as being coverable. The number of testing targets and the number of test cases created automatically is shown in Table 7.1. It shows for each application and each testing criterion the number of testing targets. The number of testing targets that could be covered automatically by a test case by using the approach of Chapter 6 is listed and the number of testing targets that required a manual-built test case is listed. So in total for each application between $73\%$ and nearly $93\%$ of the testing targets a test case could be produced automatically.

Additionally, for each of these test suites, ten additional test suites were built which contain the same number of test cases. Each of these test suites was

Table 7.1.: Automatic created test cases

| | Testing Targets | Automatically Covered | Manually Covered |
|---|---|---|---|
| **Application 1** | | | |
| All-Resources | 17 | 17 (100%) | 0 (0%) |
| All-Relations | 18 | 18 (100%) | 0 (0%) |
| All-Defs | 12 | 9 (75.00%) | 3 (25.00%) |
| All-Defuse | 18 | 16 (88.89%) | 2 (11.11%) |
| All-Uses | 38 | 24 (64.16%) | 14 (36.84%) |
| $\sum$ | 103 | 84 (81.55%) | 19 (18.45%) |
| **Application 2** | | | |
| All-Resources | 11 | 10 (90.91%) | 1 (9.09%) |
| All-Relations | 15 | 12 (80.00%) | 3 (20.00%) |
| All-Defs | 5 | 5 (100%) | 0 (0.00%) |
| All-Defuse | 11 | 11 (100%) | 0 (0.00%) |
| All-Uses | 14 | 14 (100%) | 0 (0.00%) |
| $\sum$ | 56 | 52 (92.86%) | 4 (7.14%) |
| **Application 3** | | | |
| All-Resources | 22 | 22 (100%) | 0 (0.00%) |
| All-Relations | 56 | 48 (85.71%) | 8 (14.29%) |
| All-Defs | 14 | 11 (78.57%) | 3 (21.43%) |
| All-Defuse | 57 | 43 (75.44%) | 14 (24.56%) |
| All-Uses | 244 | 163 (66.80%) | 81 (33.20%) |
| $\sum$ | 393 | 287 (73.03%) | 106 (26.97%) |

built by creating for each of the test cases of the original suite a random test case with the same number of serverless functions as the original test case. So, the test suites have the same number of invocations of serverless functions which made the functions and suites more comparable and extended the pool of potential test cases.

All the test cases created for specific testing targets and the test cases created randomly for the other suites were added to a test case pool which is used for the creation of the test suites used for the evaluation.

Test suites of different sizes were created focusing on the fulfillment of the coverage criteria investigated. Therefore, for each coverage criterion and each $x$, where $x$ is a number between $1$ and the maximum number of feasible testing targets, $200$ test suites were created which cover at last $x$ testing targets of the corresponding coverage criterion to have a broader space of test suites fulfilling a certain coverage value. The algorithm shown in Algorithm 7.1 was applied which is adapted from [6].

**Algorithm 7.1** Test Suite Generation from Test Pool

**Require:** Test Pool $P$, Target Coverage $t$
  $suite \leftarrow \emptyset$
  $suiteCoverage \leftarrow 0$
  **while** $suiteCoverage < t$ **do**
    $tc \leftarrow$ random tc of $P$
    $P \leftarrow P \setminus \{tc\}$
    $newCoverage \leftarrow$ coverage of $suite \cup \{tc\}$
    **if** $newCoverage > suiteCoverage$ **then**
      $suite \leftarrow suite \cup \{tc\}$
      $suiteCoverage \leftarrow$ coverage of $suite$
    **end if**
  **end while**
  **return** $suite$

Even when the test cases focus only on one testing target, other testing targets are also often covered by them. Thus, the number of test cases required for the fulfillment of the coverage criteria is usually less than the number of all testing targets and a variety of potential test suites can be built by these basic test cases. For each of the test suites created, similar test suites with a similar number of test cases were created. These new test suites were constructed by randomly selecting test cases from the test pool without rejecting any test case in order to have test suites of equal size and make them more comparable since the same number of test cases are used.

### 7.1.4. Mutation Operators

In order to inject faults into the application, mutation operators are used that focus on the integration. A mutation operator changes the application by deliberately injecting a fault into the program. The mutated version of the application is called a mutant. A test suite or test case kills the mutant if it detects the fault. Therefore, by executing mutants against test suites, the quality of the test suites is assessed [45]. Thus, mutation testing can be used to predict the detection effectiveness for real faults [5, 6, 49].

Since integration uses the interfaces of the components, the operators have to focus on them. In [23, 24, 106] mutation operators were introduced focusing on interfaces of programs written in the programming language C while in [39], Ghosh and Mathur applied some to CORBA-IDL and in [88], Rodríguez-Baquero and Linares-Vásquez applied some on JavaScript. However, serverless applications consist of several serverless functions integrated with other services. The relevant interfaces of serverless functions that influence their execution were identified in Section 5.3.3 and are shown in Figure 7.4. These interfaces are potential interfaces where mutation operators can be applied.

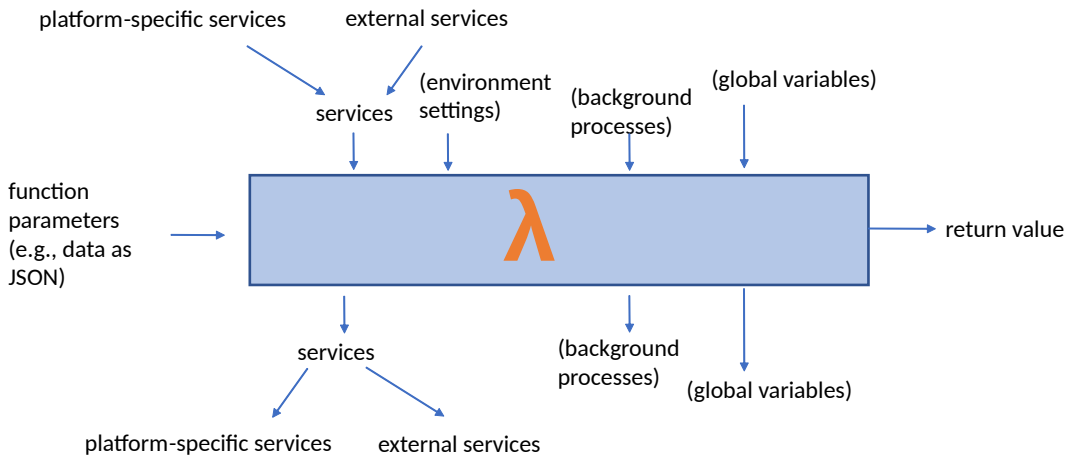



Figure 7.4.: Data interfaces of serverless functions

In general, the mutation operators described in [23, 24, 39, 88, 106] either change or remove potential data which are transmitted between the components. Specifically, when the data are changed, several strategies can be applied. Based on these interfaces and existing mutation operators, the following mutation operators were defined changing the data transmitted which are usually transmitted as key-value pairs in JSON:

- **DelKey:** Delete key of key-value pair
- **DelVal:** Delete value of key-value pair
- **RepVar:** Replace value of variable with a random value
- **NegVar:** Negate boolean value

- **RemRet:** Remove return statement
- **RemCal:** Remove call to other resource

These mutation operators were applied both on incoming data and outgoing data of the serverless functions and adapted all JSON key-value pairs that could be derived from the source code.

A general problem for mutation testing is that a new version of the program is needed for each mutant. This problem was handled by instrumenting the source code with feature flags which activated a specific mutation only if it was activated by an environment variable. For each interface, a statement was added which created code by using the current location as the condition for the activation of a mutation. The location is passed to a general function whose result is code which is executed as shown in Listing 7.1.

Listing 7.1: Usage of function for dynamic code generation

```
let code = mutate("someInterface_0");
eval(code);
```

The code is generated by a function like shown in Listing 7.2 where the environment variables of the runtime environment are evaluated, and the code is generated based on them. So, for each mutant only environment variables have to be changed instead of deploying the whole application again.

Listing 7.2: Function for dynamic code generation depending on location

```
export function mutate(location) {
    let code = "";
    const locationIdentifier = process.env.locationIdentifier;

    if (locationIdentifier === location) {
      const mutationType = process.env.mutationType;
      const variableName = process.env.variableName;

      switch (mutationType) {
        case "DelVarValue":
          code = `${variableName} = undefined;`;
          break;
        case "DelVarKey":
          code = `delete ${variableName};`;
          break;
        case "RepVar":
          const variableValue = process.env.variableValue;
          code = `${variableName} = ${variableValue};`;
          break;
        case "IncrVar":
          code = `${variableName}++;`;
          break;
```

```
            case "DecrVar":
24            code = `${variableName}--;`;
              break;
26          case "NegVar":
              code = `${variableName} = !${variableName};`;
28            break;
            default:
30            break;
          }
32      }
        return code;
34    }
```

For all three applications, 2790 mutants were created. The types of mutants used are listed in Table 7.2 where for *App3* the most mutants were generated.

Table 7.2.: Number of mutants

| | Application 1 | Application 2 | Application 3 | $\sum$ |
|---|---|---|---|---|
| DelKey | 258 | 68 | 534 | 860 |
| DelVal | 258 | 71 | 534 | 863 |
| RepVar | 223 | 71 | 534 | 828 |
| NegVar | 35 | 0 | 10 | 45 |
| RemRet | 33 | 10 | 67 | 110 |
| RemCal | 18 | 12 | 54 | 84 |
| $\sum$ | 825 | 232 | 1733 | 2790 |

### 7.1.5. Execution

The goal of the investigation is to see which coverage criteria are most suitable for integration testing. Therefore, the test cases had to be executed for each mutant individually. In order to avoid the execution of test cases being indicated as failed caused by an undetected dynamic behavior, they were executed before and after the execution on all mutants to ensure that they still work if no fault was injected. Similar to the test case generation process, each test case was executed in the same state of the applications. Therefore, a function was executed before each execution which deleted all the data saved in the application. Additionally, a test user which was also created in the test generation process was created for *App3* whose authentication token was available for the test cases. When a test case was executed and failed, the test case was repeated to ensure that there was a logical cause for the failure and not a temporary problem, like a network error or a resource shortage on the platform provider side. For each mutant, all test cases that killed the mutant were saved. Furthermore, all the testing targets covered by a test case

were also saved and assigned to the test cases. So, a test pool with test cases where all information of their mutant detection potential and coverage could be created. This test pool was used for the creation of further test pools by using Algorithm 7.1. Since the mutants that were killed by each test case were already available, the mutants killed by each suite could be generated quite easily. All data generated and used for the evaluation are available online[42].

## 7.2. Evaluation

In the following section, the results of the test suites created are evaluated.

Initially, the number of mutants, which the suites created focusing on certain coverage criteria, and their randomly generated counterparts are listed. These test cases form the basis of the test pool. This pool is used for the creation of test suites whose test cases are either selected randomly or based on criteria that are also evaluated. Finally, by applying an approach that selects the test cases of the pool based on different coverage criteria, test suites are built and evaluated.

### 7.2.1. Test Suites for Test Pool

The number of mutants and their types killed by the test suites created for specific coverage criteria, as well as the best results of their randomly generated counterparts are listed in Table 7.3. In *App3*, for example, $365$ of the mutants built with the mutation operator *DelVal* could be detected by the test suite built fulfilling the *AllUses* criterion. The best of the ten random test suites which were built with a similar size could only detect $225$ mutants of the $534$ mutants (see Table 7.2 for the number of total mutants).

More mutants were killed by most of the criterion-specific test suites than by their best randomly generated counterparts. For only a few mutant types, the best randomly generated test suite killed more mutants. This happened in particular for mutants of type *RemRet* where return statements were removed. Since the criterion-based test cases focus more on the regular data flow between the serverless functions, these test cases mostly contain input data which are based on data created by serverless functions called previously. As a result, return values invoked when the input data of the serverless function have no relation to a previously invoked serverless function were not called as often as when the function was called directly in a random test case. Return statements that were executed if an exception occurred were more frequently triggered by random test cases since their input data often reference invalid data of the application.

[42] `https://github.com/snwinz/ServerlessCoverageTesting/releases/tag/v0.7`

Table 7.3.: Coverage of test suites (max number of covered mutants of the best random suite in brackets)

| | All Res | All Rel | All Defs | All DefUse | All Uses |
|---|---|---|---|---|---|
| **App1** | | | | | |
| DelKey | 136 (115) | 136 (114) | 156 (136) | 178 (159) | 182 (170) |
| DelVal | 136 (115) | 136 (114) | 156 (136) | 178 (159) | 182 (170) |
| RepVar | 57 (49) | 57 (49) | 123 (71) | 139 (84) | 147 (104) |
| NegVar | 15 (12) | 15 (13) | 15 (16) | 18 (17) | 19 (17) |
| RemRet | 14 (12) | 14 (12) | 12 (14) | 15 (15) | 16 (15) |
| RemCal | 8 (7) | 8 (8) | 13 (10) | 16 (10) | 16 (11) |
| $\sum$ | 366 (308) | 366 (306) | 475 (367) | 544 (443) | 562 (487) |
| **App2** | | | | | |
| DelKey | 42 (35) | 42 (35) | 37 (34) | 39 (35) | 39 (35) |
| DelVal | 45 (36) | 45 (36) | 40 (35) | 42 (36) | 42 (36) |
| RepVar | 45 (27) | 47 (30) | 45 (27) | 44 (31) | 44 (31) |
| NegVar | 0 (0) | 0 (0) | 0 (0) | 0 (0) | 0 (0) |
| RemRet | 9 (7) | 7 (8) | 5 (8) | 6 (8) | 6 (8) |
| RemCal | 4 (3) | 4 (3) | 3 (3) | 4 (3) | 4 (3) |
| $\sum$ | 145 (108) | 145 (109) | 130 (107) | 135 (113) | 135 (113) |
| **App3** | | | | | |
| DelKey | 162 (130) | 265 (168) | 214 (162) | 284 (169) | 314 (195) |
| DelVal | 190 (152) | 312 (196) | 253 (190) | 334 (198) | 365 (225) |
| RepVar | 172 (132) | 327 (181) | 271 (171) | 361 (180) | 373 (206) |
| NegVar | 1 (1) | 10 (1) | 10 (1) | 10 (1) | 10 (1) |
| RemRet | 16 (13) | 26 (18) | 15 (16) | 29 (18) | 32 (24) |
| RemCal | 18 (15) | 41 (20) | 35 (18) | 45 (21) | 48 (22) |
| $\sum$ | 559 (440) | 981 (583) | 798 (558) | 1063 (585) | 1142 (672) |

Additionally, some serverless functions in *App2* lacked data flow to other components but were still invoked by other functions. These functions were not required to be called by the data flow criteria which made it easier for the randomly created test suites to cover them.

## 7.2.2. Test Suite Pool

In order to have a greater set of test suites, the test cases of the test suites created here were used as a test pool. This test pool was used to build test suites to answer the following questions:

- Question 1: Are criteria-based test suites better than randomly created test suites?
- Question 2: Which criterion kills most mutants in total?
- Question 3: Which criterion kills most mutants per testing target?
- Question 4: Do the selection algorithms, which select test cases based on several criteria, kill more mutants per test case than other criteria?

Since the number of mutants killed by the test suites was not normally distributed, the Mann-Whitney test was chosen with a significance level of 1% to compare them and measure their effect size if the null hypothesis had been rejected.

### 7.2.2.1. Comparison to Random Test Suites

The number of the mutants killed by the test suites, which were generated systematically for each criterion and their counterparts of randomly generated test suites, was compared. Figure 7.5 visualizes the comparison of the test suites generated for *AllUses* and their counterparts generated for *App3* where the interpolated lines indicate that *AllUses* kills more mutants if the test cases were selected focusing on this criterion.

Test suites built to fulfill a certain number of testing targets of a criterion were compared to the sets of their randomly generated counterparts. For all applications and criteria, the systematically generated test suites detected significantly more mutations than their randomly generated counterparts. Additionally, for the data flow criteria, the effect size was greater than $0.5$, which indicates a strong effect according to [21], and supports the hypothesis. However, for the control flow criteria, only the test suites for *App2* and the criterion *AllRel* had an effect size greater than $0.5$. For the other applications, the effect sizes ranged from $0.11$ to $0.28$, indicating only a small effect. Furthermore, the test suite sets built to only fulfill a certain number of criteria were compared with their randomly built counterparts to see which suites are better.

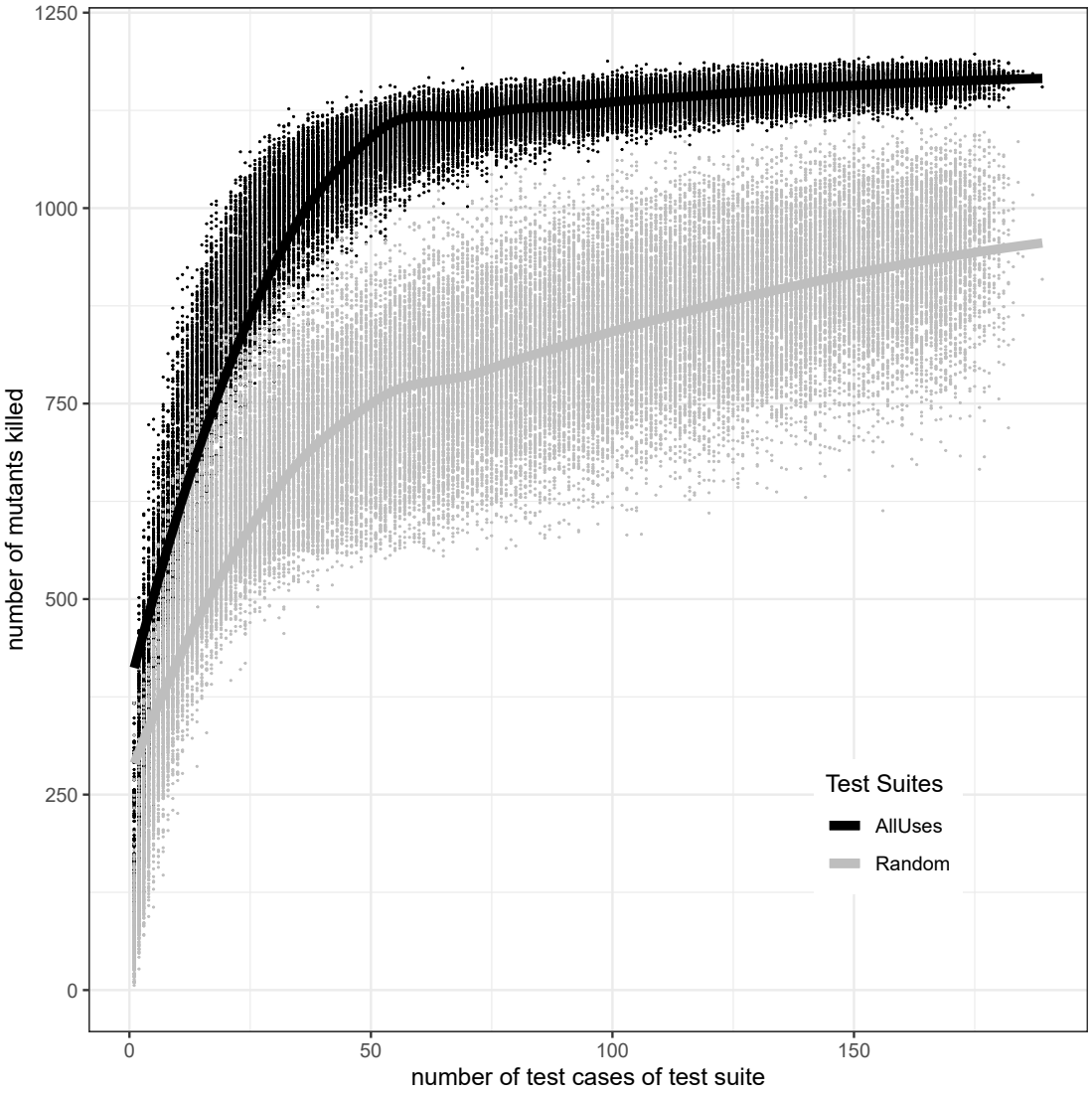



Figure 7.5.: Comparison of test Suites for *AllUses* and random created suites of *App3*

Here, in contrast to control flow criteria, the data flow criteria indicated to have significantly better test suites compared to their randomly built counterparts even when the criteria were not fulfilled yet for all applications. So, even when only half of the testing targets were fulfilled, the data flow criteria were significantly better than their randomly built counterparts. The control flow criteria could only be evaluated to be better than their randomly built counterparts if all of their testing targets were fulfilled.

Additionally, the same approach was executed for the comparison of the test suites where the manually generated test cases were not used. Here, also a significant advantage of the systematically created test cases compared to the random ones was shown.

Also, when the number of mutants was reduced to mutants on the interfaces where direct communication to other components took place, the criteria-based test suites were significantly better for the test suites built with and without manually added test cases.

#### 7.2.2.2. Comparison of Fulfilled Criteria

All criteria are compared to each other by considering their test suites that fulfilled $100\%$ of their testing targets (see Table 7.4). Since the data flow criteria build a subsumption hierarchy, it could be expected that the stricter data flow criteria killed more mutants than their subsumed ones. There was a strong effect which could be shown for most of the cases while *AllDefUse* showed no significant improvement compared to *AllDefs* in *App2*. This could be due to the architecture of *App2* where the number of potential uses for each definition is less than in the other applications. Additionally, the workflow of *App2* triggers several serverless functions that cause the coverage of additional testing targets within a test case. Therefore, the stricter coverage criteria can be fulfilled by executing relatively few test cases, in contrast to the other applications where the stricter coverage criteria have to execute more test cases to be fulfilled. This often results in higher coverage.

Thus, it could not be demonstrated that data flow criteria perform in general better than control flow criteria since the detection potential depends heavily on the application.

#### 7.2.2.3. Efficiency of Criteria per Testing Target

Since the more sophisticated test criteria require more testing targets to be fulfilled, it is relevant to know if the simpler test criteria detect mutants earlier with fewer testing targets. Therefore, the test criteria were compared where more testing targets had to be covered with the criteria where fewer had to be covered (see Table 7.5). The test suites of the criteria with fewer testing targets were chosen where $100\%$ of their testing targets were covered and compared to the test suites of the other criteria where the same number of

Table 7.4.: Comparison of detected mutants per fulfilled criterion (row) with other fulfilled criteria (column). ++ = strong effect, + = medium or small effect but significant, - = no effect for applications *App1*,*App2*,*App3*

| | All Res | All Rel | All Defs | All DefUse | All Uses |
|---|---|---|---|---|---|
| AllRes | | -,-,- | -,++,- | -,++,- | -,+,- |
| AllRel | -,+,++ | | -,++,++ | -,++,- | -,++,- |
| AllDefs | ++,-,++ | ++,-,- | | -,-,- | -,-,- |
| AllDefUse | ++,-,++ | ++,-,++ | ++,-,++ | | -,-,- |
| AllUses | ++,-,++ | ++,-,++ | ++,++,++ | ++,++,++ | |

testing targets were covered. The results showed a strong effect that *AllDefs* was always performing better than other criteria when the same number of testing targets were fulfilled. When *AllDefUse* is compared to *AllUses*, no effect could be shown for *App2* but for *App1* and *App3* effects were shown. Since the data flow criteria built a subsumption hierarchy, the less strict criteria select test cases first which are more different from the stricter criteria. This enables them to kill more mutants faster if there are enough test cases to select what didn't apply to *App2*.

Table 7.5.: Comparison of detected mutants of fulfilled criteria (row) with criteria with same number of testing targets (column). ++ = strong effect, + = medium or small effect but significant, - = no effect for applications *App1*,*App2*,*App3*

| | All Res | All Rel | All Defs | All DefUse | All Uses |
|---|---|---|---|---|---|
| AllRes | | +,+,++ | | -,++,- | -,+,- |
| AllRel | | | | -, ,- | -, ,+ |
| AllDefs | ++,++,++ | ++,++,++ | | ++,++,++ | ++,++,++ |
| AllDefUse | ,-, | ++,-, | | | +,-,++ |
| AllUses | | ,-, | | | |

#### 7.2.2.4. Combination of Criteria

Based on these insights, test case selecting algorithms were implemented which selected test cases based on the criteria that reveal most mutants per testing targets first. However, this does not automatically mean that the faster algorithms have fewer test cases. Since several testing targets could be covered by a test case, a test suite of a coverage criterion could contain more test cases than a test suite of another coverage criterion covering the same number of testing targets.

Three selection algorithms were created, and their results were evaluated. The first algorithm chosen for the selection of test cases was *AllDefs* since it performed better, as can be seen in Table 7.5, than all other criteria. Furthermore, this criterion required the least testing target to be fulfilled. If *AllDefs* was fulfilled, another coverage criterion was chosen for selection. For the first selection algorithm *AllDefUse* was chosen, for the second *AllRes* and for the third *AllRel* which killed more mutants per test case compared to *AllUses* in at least one application. Finally, test cases were added based on *AllUses* if still possible. Algorithm 7.1 was adapted and is described in Algorithm 7.2, where $c_1$, $c_2$, and $c_3$ are the coverage criteria that are used for the test case selection. Criterion $c_1$ is assigned to *AllDefs*, and criterion $c_3$ to *AllUses*. For the three different selection algorithms, criterion $c_2$ is assigned either to *AllDefUse*, *AllRes*, or *AllRel*. When a test case is rejected since it does not increase the coverage of $c_1$ or $c_2$, it is not removed from the pool, since the test case might be relevant for the fulfillment of testing targets of other criteria ($c_2$ respectively $c_3$).

Similar to the creation of test suites based on coverage criteria as described in Section 7.1.3, for each optimized approach and each $x$, where $x$ is a number between $1$ and the maximum number of feasible testing targets, $200$ test suites were created which cover at last $x$ testing targets of the corresponding optimized approach.

**Algorithm 7.2** Optimized Algorithm for Test Suite Generation

**Require:** Test Pool $P$, Target Coverage $t$, Coverage Criterion $c_1$, $c_2$, $c_3$
  $suite \leftarrow \emptyset$
  **for all** $c \in \{c_1, c_2, c_3\}$ **do**
    $suiteCoverage \leftarrow$ coverage regarding $c$ of $suite$
    $totalCoverage \leftarrow$ coverage regarding $c_3$ of $suite$
    **while** $suiteCoverage < 100\%$ and $totalCoverage < t$ **do**
      $tc \leftarrow$ random tc of $P$
      $newCoverage \leftarrow$ coverage regarding $c$ of $suite \cup \{tc\}$
      **if** $newCoverage > suiteCoverage$ **then**
        $suite \leftarrow suite \cup \{tc\}$
        $suiteCoverage \leftarrow$ coverage regarding $c$ of $suite$
      **end if**
    **end while**
  **end for**
  **return** $suite$

The results showed for all created test suites of the algorithms that they revealed significantly more mutants per test case across all three applications with a small effect size ranging from $0.05$ to $0.17$ compared to the test suites built based only on *AllUses*. While the selection algorithm with *AllRes* killed most mutants on average per test case for all applications, all of them performed better than the test suites created based only on *AllUses* (compare Table 7.6).

Table 7.6.: Average of additional mutants killed for all sizes of test suites

| | App1 | App2 | App3 |
|---|---|---|---|
| OptimizedDefUse | 9.04 | 3.07 | 25.92 |
| OptimizedResources | 11.38 | 3.17 | 36.53 |
| OptimizedRelations | 10.69 | 3.12 | 17.01 |

Figure 7.6 visualizes the comparison of the optimized test suites taking the *AllRes* criterion as the second selection criterion and the test suites generated for *AllUses* where on average the optimized test suites killed more mutants per test case.

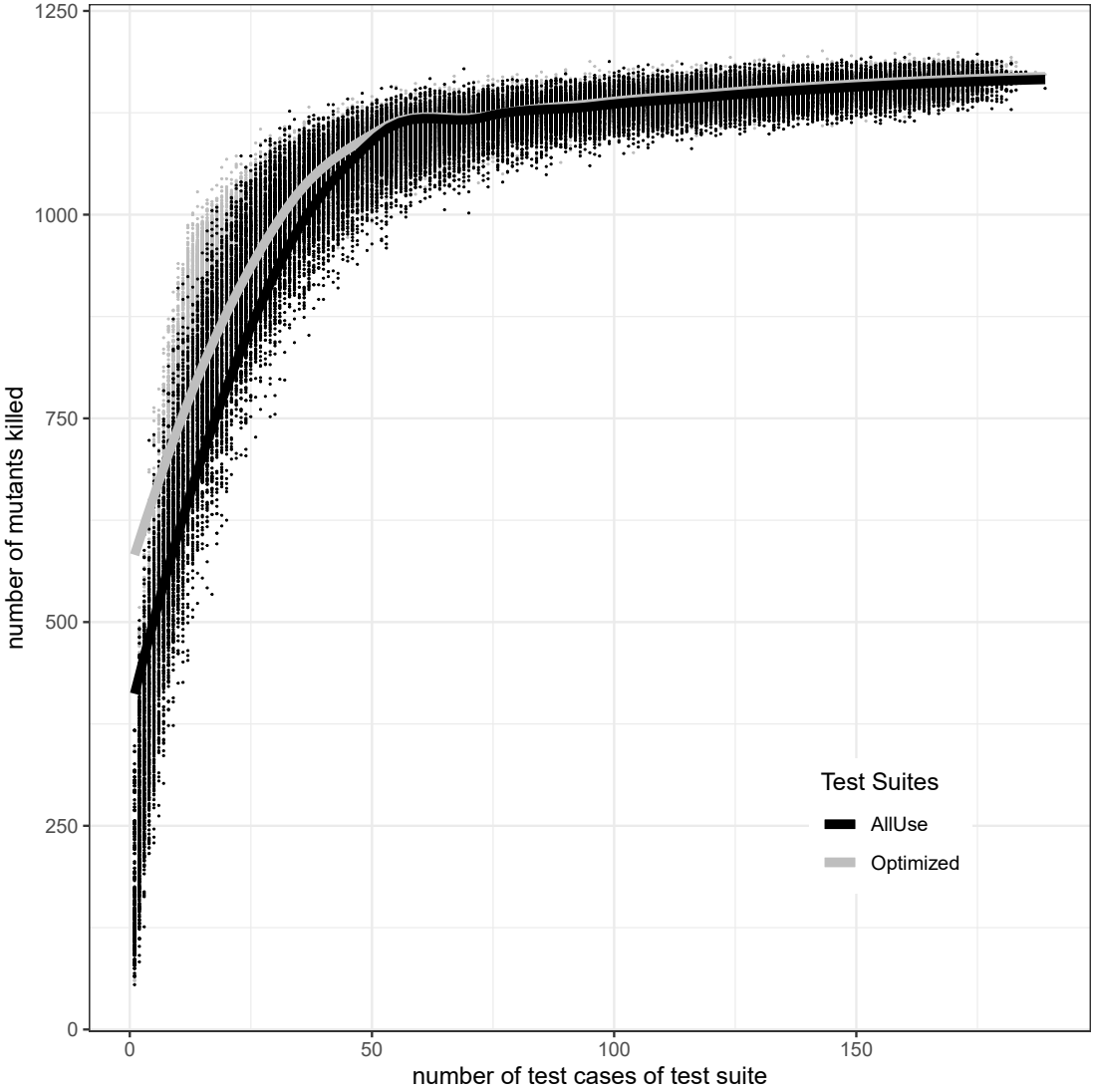



Figure 7.6.: Example of mutants killed per test case for *App3*

## 7.3. Related Work

To the knowledge of the author of this work, there are no evaluations for coverage criteria for serverless applications done yet.

However, in [34] and [42] data flow and control flow criteria were evaluated on some simple programs with faults on the code level and not on the integration level for serverless applications. In [34] the faults were already known while the faults in [42] were injected individually which makes them prone to bias. Their results showed also that both data and control flow criteria are quite effective. However, no best criterion could be identified since the effectiveness of the criterion depends on the application. A similar approach was applied in [101] where deterministic and random generated input data for test cases were also compared by using mutation operators.

According to Andrews et al. in [6], these approaches of [34, 42, 101] have a similar approach by generating a test pool of test cases first, running the faulty versions of the programs on all the test cases, and observing which test cases detected which fault. Finally, the fault detection potential of certain test suites that were generated based on the pool is evaluated [6] which matches the approach applied in this work.

## 7.4. Limitations

Just like in the other sections, a limitation lies in the selected applications as they cannot represent all serverless applications. Therefore, an additional application was added which was also used in other papers as a more complex example. Nevertheless, all applications showed a better performance of test suites selected based on coverage criteria compared to random ones.

Since some test cases for the criteria-based testing targets were created manually, it could be claimed that these test cases are more sophisticated than necessary. However, in order to prevent additional coverage, these test cases were generated with the minimum number of serverless functions necessary to be invoked. Furthermore, another evaluation was made where the test suites were built without manual test cases. Here, the criteria-based test suites were also significantly better than the random ones.

The number of mutants is limited to a certain set which does not necessarily represent all kinds of faults and their importance. There are equivalent mutants that show the same behavior as the original application and subsumed mutants. A mutant is a subsumed mutant when each test case killing the subsumed mutant would also kill the subsuming mutant [37]. So, a subsumed mutant is not killed by any test case that is not killing the subsuming mutant. While the equivalent mutants are only a problem for the execution time of the experiment since only the absolute numbers of mutants killed were used for the experiment and not a percentage of the mutants killed, according to [83] subsuming

mutants can distort the comparison. However, identifying them is an NP-complete problem [4]. Therefore, it cannot be guaranteed that the mutants created are representative but all of them were created objectively and all test cases were executed on them, making it a good basis for comparison. Additionally, it was also not relevant what percentage of the mutants were killed, but the absolute number of mutants killed could be used for comparison among the criteria themselves.

## 7.5. Summary

This chapter shows that coverage criteria help create better test cases than simply invoking some random function. However, it could not be shown that data flow criteria are generally better than control flow criteria since the detection rate depends strongly on the application. Test suites covering all def-use-pairs detected the most faults which were injected by mutation operators in two of three applications. The other application could be tested successfully by test cases built with control flow criteria because of the architecture of the application. Some criteria showed to be more efficient for the selection of test cases when the focus of the test cases is to reveal as many faults as possible. This inspired us to create two selection algorithms that used this insight and were more efficient in detecting faults per test case.

Thus, our work can help developers select test cases more efficiently to detect potential faults earlier and build test suites having a reasonable size.

# Part III.

# Conclusion and Outlook

# 8. Conclusion and Outlook

This chapter summarizes the work and gives an outlook on future work.

## 8.1. Summary

The focus of this work was on coverage criteria supporting integration testing for serverless applications.

Therefore, a model was needed to support integration testing which was investigated in research question 1.1. This included the creation of a model which can represent the components of a serverless application and their dependencies. The model introduced formalizes the serverless application as a graph whose nodes represent serverless functions and whose arcs represent the dependencies between the serverless functions. The introduced graph can be easily extended for different purposes, in particular for testing. The extensions include, inter alia, the representation of data storages and the kind of access to them, general resources of a serverless application, e.g., the access to other services, the interfaces of serverless functions, and the representation of the data flow.

Research question 1.2 investigated how a model of a serverless application can be created automatically based on existing code. Here, a tool was developed that can be used to create a model of a serverless application based on the source code and the deployment file of the serverless application. But also the limitations of the tool were discussed, e.g., that the tool has to be adapted for different programming languages individually.

By introducing potential coverage criteria for integration testing that can capture the interaction between the components of serverless applications, research question 2 was answered. Definitions were given for three control flow and three data flow criteria. However, the control flow criterion, which focuses on the coverage of all paths, and the data flow criterion, which focuses on the coverage of all combinations of definitions and uses, turned out to not be applicable to all kinds of applications. If an application becomes too large or complex, the number of feasible test cases required for its coverage could also become excessively large or unclear. Furthermore, other potential testing criteria were discussed focusing on parallelism, execution time, error handling,

and security of serverless applications.

It was shown how the testing targets of control and data flow criteria can be measured in general on serverless applications to answer research question 3.1. A concrete implementation for *AWS* was presented which supports the required instrumentation of the source code of the serverless functions. Furthermore, research question 3.2 could be answered by showing how the instrumentation of the serverless functions influenced the performance of the serverless functions. By applying the instrumentation to some scenarios, it was demonstrated that the instrumentation for the measurement of the coverage criteria added only a small overhead to the execution time of the serverless functions while other monitoring frameworks added a significant overhead during execution. In contrast to other monitoring frameworks, the approach introduced here is able to measure sequences of calls between the services and the data flow between serverless functions which are coupled by another data storage resource.

Furthermore, an approach was demonstrated for automatically creating integration tests based on a model, addressing research question 4. The approach was able to create test cases for control flow and data flow criteria introduced in this work by identifying potential test cases for the specific testing targets of the coverage criteria first. These test cases are run on a serverless application measuring the coverage. If the intended testing target of a test case was not covered after a run, its input data were adapted or the test case was rejected after several adaptions. Through this approach, more than $70\%$ of the testing targets for the sample applications could be covered. This approach can alleviate the tedious task of creating test cases, especially for regression testing or load testing where many test cases are needed.

These automatically generated test cases were also used in the final chapter of the thesis where the coverage criteria were evaluated to answer research question 5.1. It was shown by injecting faults into the application using mutation operators that the test suites fulfilling the control flow criteria or data flow criteria are significantly better for the detection of the injected faults compared to randomly created suites.

Research question 5.2 asked which coverage criterion revealed the most injected faults. Of course, the strictest data flow criterion *AllUses* subsuming the other data flow criteria revealed more faults when fulfilled than the other data flow criteria. However, it performed better than the control flow criteria for only some applications, not all. So, there was not a single best testing criterion since the success of the criteria depended also on the application.

Furthermore, since some test criteria resulted in faster coverage per testing target, an algorithm was introduced that combined several coverage criteria

and helped to cover the application faster.

This work highlights the important role of coverage criteria in the creation of test cases to improve the detection of faults and introduces tool support tailored to the specific needs of serverless applications.

## 8.2. Future work

Future work is possible in several areas. In general, the tools could also support other programming languages and other cloud platforms could be integrated for them. As for now, only basic support for JavaScript is available. In particular, the support for the detection of relations to other resources and the detection of statements that need to be instrumented for measuring the coverage of the criteria can be further improved to broaden the range of support.

Furthermore, the visualization of serverless functions could be better supported by the cloud platform providers. As for now, the tooling introduced here supports only a local visualization of the model. Cloud platform providers could extend their services by also providing visualization support for the applications deployed. By combining it with a dynamic analysis of the source code and deployment file, developers using the cloud platform could analyze the structure of their applications in a contemporary way.

The coverage criteria introduced in this work can also be further refined or extended in future work. For example, the data flow criteria could be refined by considering all attributes of data that are passed. The criteria introduced here focused on the control flow and data flow of applications. But also other basics for criteria focusing on the parallelism or security aspects could be investigated further in future work. For example, by checking the access rights assigned to all of the resources in a serverless application, the security could be enhanced. This would also require additional evaluation of the coverage criteria by assessing the test cases based on the new criteria.

Furthermore, the support of the cloud platforms could be better for measuring the data flow. This could be achieved by providing frameworks supporting the instrumentation of the code and by providing services that are able to pass additional data flow information that could be used for system monitoring or test case measurement.

Also, the adaption of the coverage criteria and tooling for microservices could be considered. In a microservice architecture, different small services communicate with each other. In contrast to serverless applications, only consisting of small serverless functions instead of services, microservices administrate and keep their state by having their own data storage. By measuring the data flow between different microservices and considering their uses, it might be possible to identify problems better.

# Bibliography


[1] G. Adzic and R. Chatley, "Serverless computing: Economic and architectural impact," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE'17. ACM, Aug. 2017.

[2] Z. Al-Ali, S. Goodarzy, E. Hunter, S. Ha, R. Han, E. Keller, and E. Rozner, "Making serverless computing more serverless," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, Jul. 2018.

[3] K. Alpernas, A. Panda, L. Ryzhyk, and M. Sagiv, "Cloud-scale runtime verification of serverless applications," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, Nov. 2021.

[4] P. Ammann, M. E. Delamaro, and J. Offutt, "Establishing theoretical minimal sets of mutants," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, Mar. 2014.

[5] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the 27th international conference on Software engineering - ICSE '05*. ACM Press, 2005.

[6] J. Andrews, L. Briand, Y. Labiche, and A. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, Aug. 2006.

[7] A. Arcuri, "RESTful API automated test case generation with EvoMaster," *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 1, pp. 1–37, Feb. 2019.

[8] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen, "WSDL-based automatic test case generation for web services testing," in *IEEE International Workshop on Service-Oriented System Engineering (SOSE'05)*. IEEE, 2005.

[9] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*. Springer Singapore, Jun. 2017, pp. 1–20.

[10] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, and O. Tardieu, "The serverless trilemma: Function composition for serverless computing," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2017*. ACM Press, 2017.

[11] D. Bardsley, L. Ryan, and J. Howard, "Serverless performance and optimization strategies," in *2018 IEEE International Conference on Smart*

*Cloud (SmartCloud)*. IEEE, Sep. 2018.

[12] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, May 2015.

[13] F. Belli and M. Linschulte, "Event-driven modeling and testing of web services," in *2008 32nd Annual IEEE International Computer Software and Applications Conference*. IEEE, 2008.

[14] ——, "Event-driven modeling and testing of real-time web services," *Service Oriented Computing and Applications*, vol. 4, no. 1, pp. 3–15, Mar. 2010.

[15] A. Bento, J. Correia, R. Filipe, F. Araujo, and J. Cardoso, "Automated analysis of distributed tracing: Challenges and research directions," *Journal of Grid Computing*, vol. 19, no. 1, Feb. 2021.

[16] D. Bermbach, A.-S. Karakaya, and S. Buchholz, "Using application knowledge to reduce cold starts in FaaS services," in *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC20)*, 2020.

[17] A. Brogi, J. Soldani, and P. Wang, "TOSCA in a nutshell: Promises and perspectives," in *Advanced Information Systems Engineering*. Springer Berlin Heidelberg, 2014, pp. 171–186.

[18] C. Cannavacciuolo and L. Mariani, "Smoke testing of cloud systems," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Apr. 2022.

[19] W. Chan, L. Mei, and Z. Zhang, "Modeling and testing of cloud applications," in *2009 IEEE Asia-Pacific Services Computing Conference (APSCC)*. IEEE, Dec. 2009.

[20] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil, "A formal evaluation of data flow path selection criteria," *IEEE Transactions on Software Engineering*, vol. 15, no. 11, pp. 1318–1332, 1989.

[21] J. Cohen, "Statistical power analysis," *Current Directions in Psychological Science*, vol. 1, no. 3, pp. 98–101, Jun. 1992.

[22] 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.

[23] M. E. Delamaro, J. C. Maldonado, A. Pasquini, and A. P. Mathur, "Interface mutation test adequacy criterion: An empirical evaluation," *Empirical Software Engineering*, vol. 6, no. 2, pp. 111–142, 2001.

[24] M. Delamaro, J. Maidonado, and A. Mathur, "Interface mutation: An approach for integration testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 3, pp. 228–247, Mar. 2001.

[25] M. Dudjak and G. Martinovi, "An API-first methodology for designing a microservice-based Backend as a Service platform," *Information Technology And Control*, vol. 49, no. 2, pp. 206–223, Jun. 2020.

[26] C. Eder, "A comparison of distributed tracing tools in serverless applications," Master Thesis, University of Bamberg, Kapuzinerstraße 16, 96047 Bamberg, Dec. 2022.

[27] C. Eder, S. Winzinger, and R. Lichtenthäler, "A comparison of distributed tracing tools in serverless applications," in *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, Jul. 2023.

[28] S. Eismann, J. Scheuner, E. v. Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "The state of serverless applications: Collection, characterization, and community consensus," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 4152–4166, Oct. 2021.

[29] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "Serverless applications: Why, when, and how?" *IEEE Software*, vol. 38, no. 1, pp. 32–39, Jan. 2021.

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

[31] C. Fidge, "Logical time in distributed computing systems," *Computer*, vol. 24, no. 8, pp. 28–33, Aug. 1991.

[32] K. Figiela, A. Gajek, A. Zima, B. Obrok, and M. Malawski, "Performance evaluation of heterogeneous cloud functions," *Concurrency and Computation: Practice and Experience*, p. e4792, Aug. 2018.

[33] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1483–1498, Oct. 1988.

[34] P. G. Frankl and S. N. Weiss, "An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria," in *Proceedings of the symposium on Testing, analysis, and verification*, ser. TAV-4. ACM, Oct. 1991.

[35] M. Gabbrielli, S. Giallorenzo, I. Lanese, F. Montesi, M. Peressotti, and S. P. Zingaro, "No more, no less," in *Lecture Notes in Computer Science*. Springer International Publishing, 2019, pp. 148–157.

[36] J. Gao, X. Bai, and W.-T. Tsai, "Cloud testing-issues, challenges, needs and practice," *Software Engineering: An International Journal*, vol. 1, no. 1, pp. 9–23, 2011.

[37] A. Garg, M. Ojdanic, R. Degiovanni, T. T. Chekam, M. Papadakis, and Y. Le Traon, "Cerebro: Static subsuming mutant selection," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 24–43, Jan. 2023.

[38] B. C. Ghosh, S. K. Addya, N. B. Somy, S. B. Nath, S. Chakraborty, and S. K. Ghosh, "Caching techniques to improve latency in serverless archi-

tectures," in *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*. IEEE, Jan. 2020.

[39] S. Ghosh and A. P. Mathur, "Interface mutation," *Software Testing, Verification and Reliability*, vol. 11, no. 4, pp. 227–247, 2001.

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

[41] J. C. Huang, "An approach to program testing," *ACM Computing Surveys*, vol. 7, no. 3, pp. 113–128, Sep. 1975.

[42] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria," in *Proceedings of 16th International Conference on Software Engineering*, ser. ICSE-94. IEEE Comput. Soc. Press, 1994.

[43] M. Jaffar-ur Rehman, F. Jabeen, A. Bertolino, and A. Polini, "Testing software components for integration: a survey of issues and techniques," *Software Testing, Verification and Reliability*, vol. 17, no. 2, pp. 95–133, Nov. 2007.

[44] A. Jangda, D. Pinckney, Y. Brun, and A. Guha, "Formal foundations of serverless computing," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–26, Oct. 2019.

[45] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, Sep. 2011.

[46] Z. Jin and A. J. Offutt, "Coupling-based criteria for integration testing," *Software Testing, Verification and Reliability*, vol. 8, no. 3, pp. 133–154, Sep. 1998.

[47] Z. Jin and J. Offutt, "Deriving tests from software architectures," in *Proceedings 12th International Symposium on Software Reliability Engineering*, ser. ISSRE-01. IEEE Comput. Soc, 2001.

[48] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Menezes Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2019-3, Feb. 2019.

[49] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. ACM Press, 2014.

[50] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining GitHub," in *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*. ACM Press, 2014.

[51] G. M. Kapfhammer and M. L. Soffa, "A family of test adequacy criteria for database-driven applications," *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 5, pp. 98–107, Sep. 2003.

[52] K. Kritikos and P. Skrzypek, "A review of serverless frameworks," in *Proceedings of the 4th Workshop on Serverless Computing (WoSC)*. IEEE, Dec. 2018, pp. 161–168.

[53] K. Kritikos, P. Skrzypek, A. Moga, and O. Matei, "Towards the modelling of hybrid cloud applications," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, Jul. 2019.

[54] S. G. Kwak and J. H. Kim, "Central limit theorem: the cornerstone of modern statistics," *Korean journal of anesthesiology*, vol. 70, no. 2, pp. 144–156, 2017.

[55] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[56] P. Leitner, J. Cito, and E. Stöckli, "Modelling and managing deployment costs of microservice-based cloud applications," in *Proceedings of the 9th International Conference on Utility and Cloud Computing - UCC '16*. ACM Press, 2016.

[57] P. Leitner, E. Wittern, J. Spillner, and W. Hummer, "A mixed-method empirical study of Function-as-a-Service software development in industrial practice," *Journal of Systems and Software*, vol. 149, pp. 340–359, Mar. 2019.

[58] V. Lenarduzzi and A. Panichella, "Serverless testing: Tool vendors' and experts' points of view," *IEEE Software*, vol. 38, no. 1, pp. 54–60, Jan. 2021.

[59] V. Lenarduzzi, J. Daly, A. Martini, S. Panichella, and D. A. Tamburri, "Toward a technical debt conceptualization for serverless computing," *IEEE Software*, vol. 38, no. 1, pp. 40–47, Jan. 2021.

[60] C. Lin and H. Khazaei, "Modeling and optimization of performance and cost of serverless applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 615–632, Mar. 2021.

[61] P.-M. Lin and A. Glikson, "Mitigating cold starts in serverless platforms: A pool-based approach," *arXiv e-prints*, Mar. 2019.

[62] 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.

[63] W.-T. Lin, C. Krintz, R. Wolski, M. Zhang, X. Cai, T. Li, and W. Xu, "Tracking causal order in AWS lambda applications," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, Apr. 2018.

[64] U. Linnenkugel and M. Mullerburg, "Test data selection criteria for (software) integration testing," in *Systems Integration '90. Proceedings of the First International Conference on Systems Integration*. IEEE Comput.

Soc. Press, 1990.

[65] 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.

[66] N. Mahmoudi and H. Khazaei, "Performance modeling of serverless computing platforms," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2020.

[67] M. Malawski, K. Figiela, A. Gajek, and A. Zima, "Benchmarking heterogeneous cloud functions," in *Euro-Par 2017: Parallel Processing Workshops*, D. B. Heras and L. Bougé, Eds. Springer International Publishing, 2018, pp. 415–426.

[68] A. Mampage, S. Karunasekera, and R. Buyya, "A holistic view on resource management in serverless computing environments: Taxonomy and future directions," *ACM Computing Surveys*, vol. 54, no. 11s, pp. 1–36, Jan. 2022.

[69] J. Manner, "A structured literature review approach to define serverless computing and Function as a Service," in *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. IEEE, Jul. 2023.

[70] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in Function as a Service," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion, 4th Workshop on Serverless Computing (WoSC)*. IEEE, Dec. 2018.

[71] J. Manner, S. Kolb, and G. Wirtz, "Troubleshooting serverless functions: a combined monitoring and debugging approach," *SICS Software-Intensive Cyber-Physical Systems*, vol. 34, no. 2-3, pp. 99–104, Feb. 2019.

[72] S. Mathur and S. Malik, "Advancements in the v-model," *International Journal of Computer Applications*, vol. 1, no. 12, pp. 29–34, 2010.

[73] F. Mattern, "Virtual time and global states of distributed systems," in *Parallel and Distributed Algorithms*. North-Holland, 1989, pp. 215–226.

[74] P. Mell and T. Grance, "The NIST definition of cloud computing," National Institute of Standards and Technology, Gaithersburg, Tech. Rep., 2011.

[75] C. Miyachi, "What is cloud? it is time to update the NIST definition?" *IEEE Cloud Computing*, vol. 5, no. 3, pp. 6–11, May 2018.

[76] S. R. Nagalakshmi and M. D'Souza, *Coverage Criteria Based Testing of IoT Applications*. Springer Nature Switzerland, 2024, pp. 101–116.

[77] A. B. Nandyal and M. Rafi, "Determining feature gaps of open source cloud platforms for mobile backend as service (MBaaS) in enterprise mobile applications," in *2020 IEEE International Conference on Distributed Computing, VLSI, Electrical Circuits and Robotics (DISCOVER)*. IEEE, Oct. 2020.

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

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

[80] M. Obetz, A. Das, T. Castiglia, S. Patterson, and A. Milanova, "Formalizing event-driven behavior of serverless applications," in *Service-Oriented and Cloud Computing*. Springer International Publishing, 2020, pp. 19–29.

[81] A. Offutt, A. Abdurazik, and R. Alexander, "An analysis tool for coupling-based integration testing," in *Proceedings Sixth IEEE International Conference on Engineering of Complex Computer Systems. ICECCS 2000*. IEEE Comput. Soc, 2000.

[82] J. Opara-Martins, R. Sahandi, and F. Tian, "Critical review of vendor lock-in and its impact on adoption of cloud computing," in *International Conference on Information Society (i-Society 2014)*. IEEE, Nov. 2014.

[83] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. L. Traon, "Threats to the validity of mutation-based test assessment," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, Jul. 2016.

[84] M. Pezze and M. Young, *Software Testing and Analysis*. Wiley, 2007.

[85] H. Puripunpinyo and M. Samadzadeh, "Effect of optimizing java deployment artifacts on aws lambda," in *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, May 2017.

[86] E. Rinta-Jaskari, C. Allen, T. Meghla, and D. Taibi, "Testing approaches and tools for AWS lambda serverless-based applications," in *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*. IEEE, Mar. 2022.

[87] M. Roberts and J. Chapin, *What Is Serverless?* O'Reilly Media, 2017. [Online]. Available: http://www.oreilly.com/programming/free/what-is-serverless.csp

[88] D. Rodríguez-Baquero and M. Linares-Vásquez, "Mutode: generic JavaScript and node.js mutation testing tool," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Jul. 2018.

[89] P. Rook, "Controlling software projects," *Software Engineering Journal*, vol. 1, no. 1, p. 7, 1986.

[90] G. Rothermel and M. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 529–551, 1996.

[91] R. B. Roy, T. Patel, and D. Tiwari, "Icebreaker: Warming serverless functions better with heterogeneity," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. ACM, Feb. 2022.

[92] P. Sbarski, *Serverless Architectures on Aws: With Examples Using Aws Lambda*. Manning Publications, 2017.

[93] J. Scheuner, S. Eismann, S. Talluri, E. van Eyk, C. Abad, P. Leitner, and A. Iosup, "Let's trace it: Fine-grained serverless benchmarking using synchronous and asynchronous orchestrated applications," 2022.

[94] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: A survey of opportunities, challenges, and applications," *ACM Computing Surveys*, vol. 54, no. 11s, pp. 1–32, Jan. 2022.

[95] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 205–218. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shahrad

[96] Y. Shkuro, *Mastering Distributed Tracing: Analyzing performance in microservices and complex systems*. Packt Publishing Ltd, 2019.

[97] A. Spillner, "Test criteria and coverage measures for software integration testing," *Software Quality Journal*, vol. 4, no. 4, pp. 275–286, Dec. 1995.

[98] V. Sreekanti, C. Wu, S. Chhatrapati, J. E. Gonzalez, J. M. Hellerstein, and J. M. Faleiro, "A fault-tolerance shim for serverless computing," in *Proceedings of the Fifteenth European Conference on Computer Systems*. ACM, Apr. 2020.

[99] S. Sthamer, "Analysis of cloud modeling languages for serverless applications," Bachelor Thesis, University of Bamberg, Kapuzinerstraße 16, 96047 Bamberg, Mar. 2021.

[100] J. Surbiryala and C. Rong, "Cloud computing: History and overview," in *2019 IEEE Cloud Summit*. IEEE, Aug. 2019.

[101] P. Thevenod-Fosse, H. Waeselynck, and Y. Crouzet, "An experimental study on software structural testing: deterministic versus random input generation," in *[1991] Digest of Papers. Fault-Tolerant Computing: The Twenty-First International Symposium*. IEEE Comput. Soc. Press, 1991.

[102] R. Tolosana-Calasanz, G. G. Castañé, J. Á. Bañares, and O. Rana, "Modelling serverless function behaviours," in *Economics of Grids, Clouds, Systems, and Services*. Springer International Publishing, 2021, pp. 109–122.

[103] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," Mar. 2021.

[104] E. van Eyk, A. Iosup, S. Seif, and M. Thömmes, "The spec cloud group's research vision on FaaS and serverless architectures," in *Proceedings of the 2nd International Workshop on Serverless Computing*, ser. Middleware '17. ACM, Dec. 2017.

[105] E. van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uta, and A. Iosup, "Serverless is more: From PaaS to present cloud computing," *IEEE Internet Computing*, vol. 22, no. 5, pp. 8–17, Sep. 2018.

[106] A. M. R. Vincenzi, J. C. Maldonado, E. F. Barbosa, and M. E. Delamaro, "Unit and integration testing strategies for C programs using mutation," *Software Testing, Verification and Reliability*, vol. 11, no. 4, pp. 249–268, 2001.

[107] S. Winzinger, "Towards coverage criteria for serverless applications," in *11th Central European Workshop on Services and their Composition (ZEUS)*, 2019.

[108] 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.

[109] ——, "Coverage criteria for integration testing of serverless applications," in *13th Symposium and Summer School On Service-Oriented Computing*, 2019.

[110] ——, "Applicability of coverage criteria for serverless applications," in *2020 IEEE International Conference on Service Oriented Systems Engineering (SOSE)*. IEEE, Aug. 2020.

[111] ——, "Data flow testing of serverless functions," in *Proceedings of the 11th International Conference on Cloud Computing and Services Science - CLOSER,*, INSTICC. SciTePress, 2021, pp. 56–64.

[112] ——, "Automatic test case generation for serverless applications," in *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, Aug. 2022.

[113] ——, "Comparison of integration coverage criteria for serverless applications," in *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, Jul. 2023.

[114] M. Wurster, U. Breitenbucher, K. Kepes, F. Leymann, and V. Yussupov, "Modeling and automated deployment of serverless applications using TOSCA," in *2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE, Nov. 2018.

[115] R. S. Yadav, "Improvement in the v-model," *International Journal of Scientific & Engineering Research*, vol. 3, no. 2, pp. 1–6, 2012.

[116] M. Yan, P. Castro, P. Cheng, and V. Ishakian, "Building a chatbot with serverless computing," in *Proceedings of the 1st International Workshop on Mashups of Things and APIs*, ser. Middleware '16. ACM, Dec. 2016.

[117] V. Yussupov, U. Breitenbücher, F. Leymann, and C. Müller, "Facing the unplanned migration of serverless applications: A study on portability

problems, solutions, and dead ends," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, ser. UCC '19. ACM, Dec. 2019.

[118] V. Yussupov, U. Breitenbücher, A. Kaplan, and F. Leymann, "SEAPORT: Assessing the portability of serverless applications," in *Proceedings of the 10th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, 2020.

[119] V. Yussupov, J. Soldani, U. Breitenbücher, and F. Leymann, "Standards-based modeling and deployment of serverless function orchestrations using BPMN and TOSCA," *Software: Practice and Experience*, Jan. 2022.

# Part IV.

# Appendix

# A. Models of Applications of Section 4.3.2

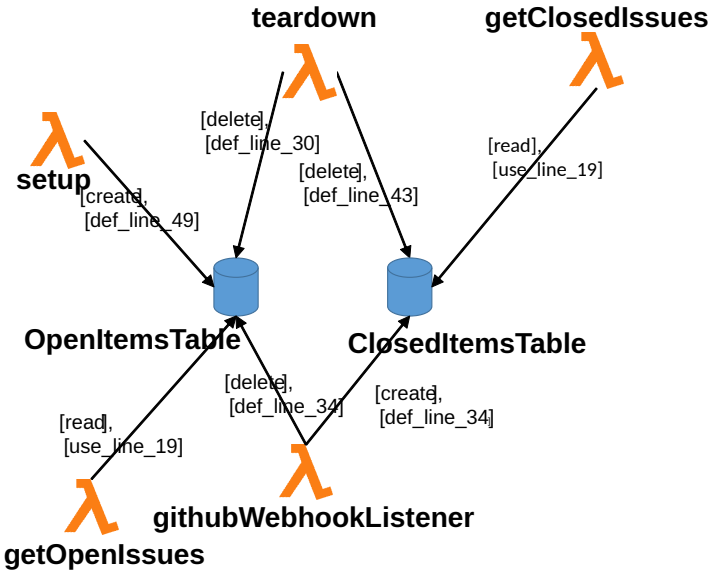



Figure A.1.: Model of application *A1*

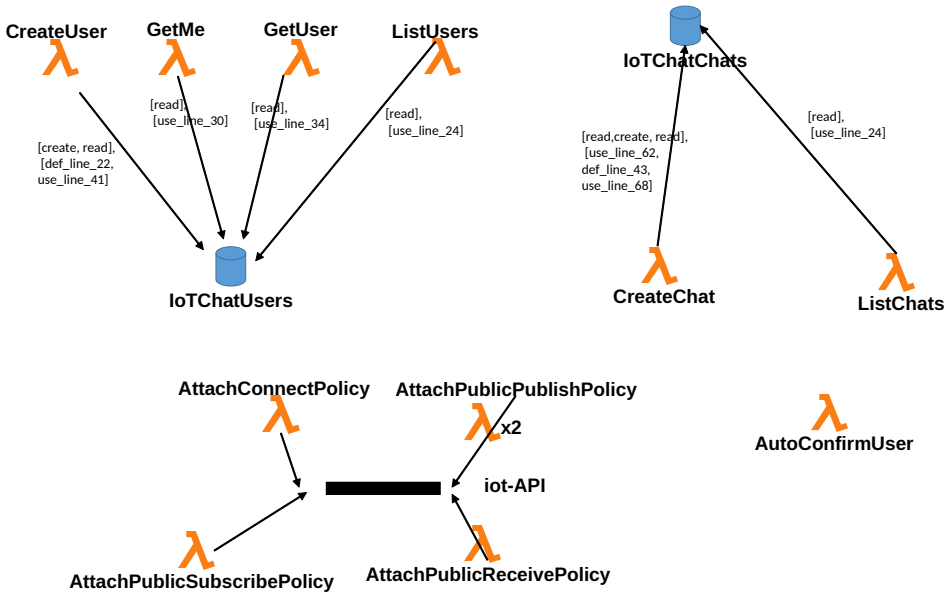



Figure A.2.: Model of application *A2*

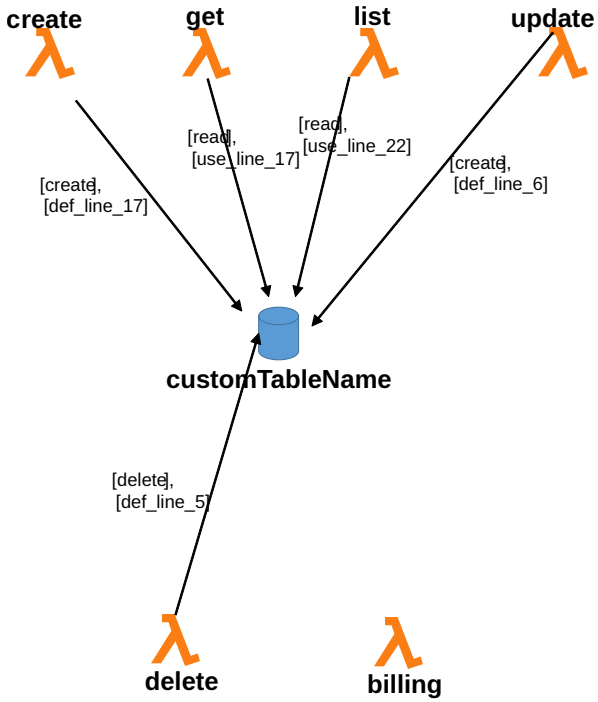



Figure A.3.: Model of application *A3*

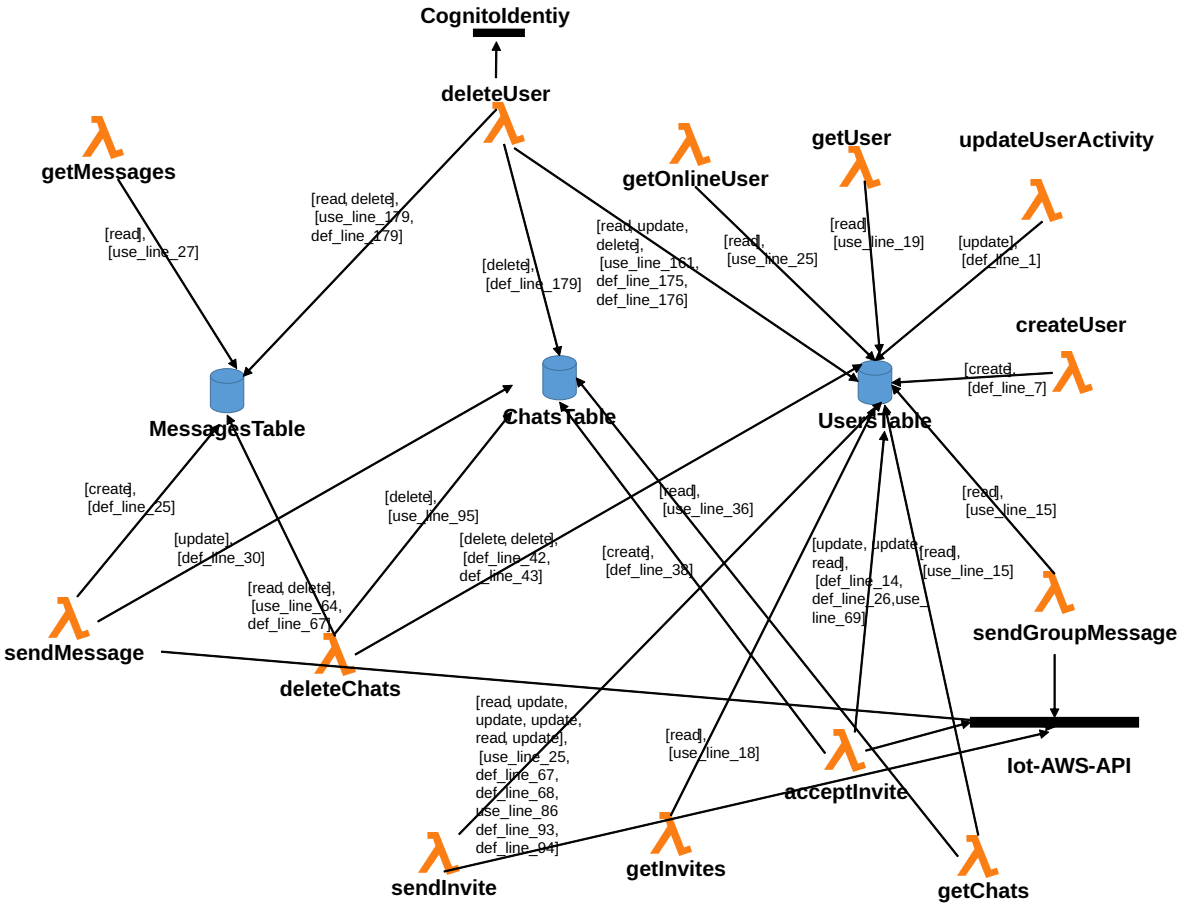



Figure A.4.: Model of application *A4*

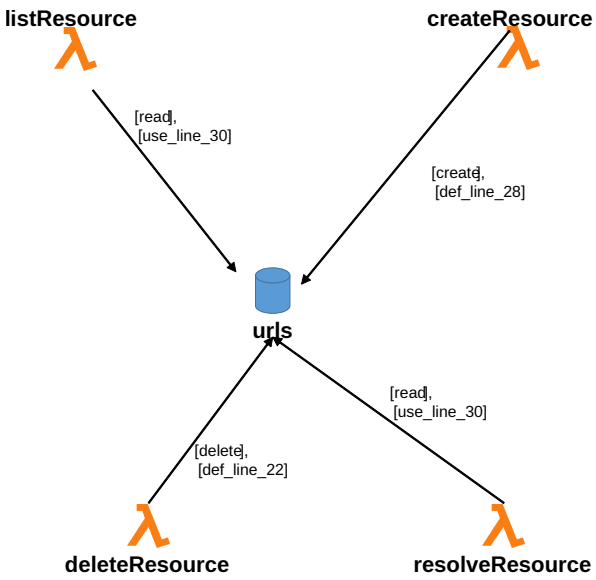



Figure A.5.: Model of application *A5*

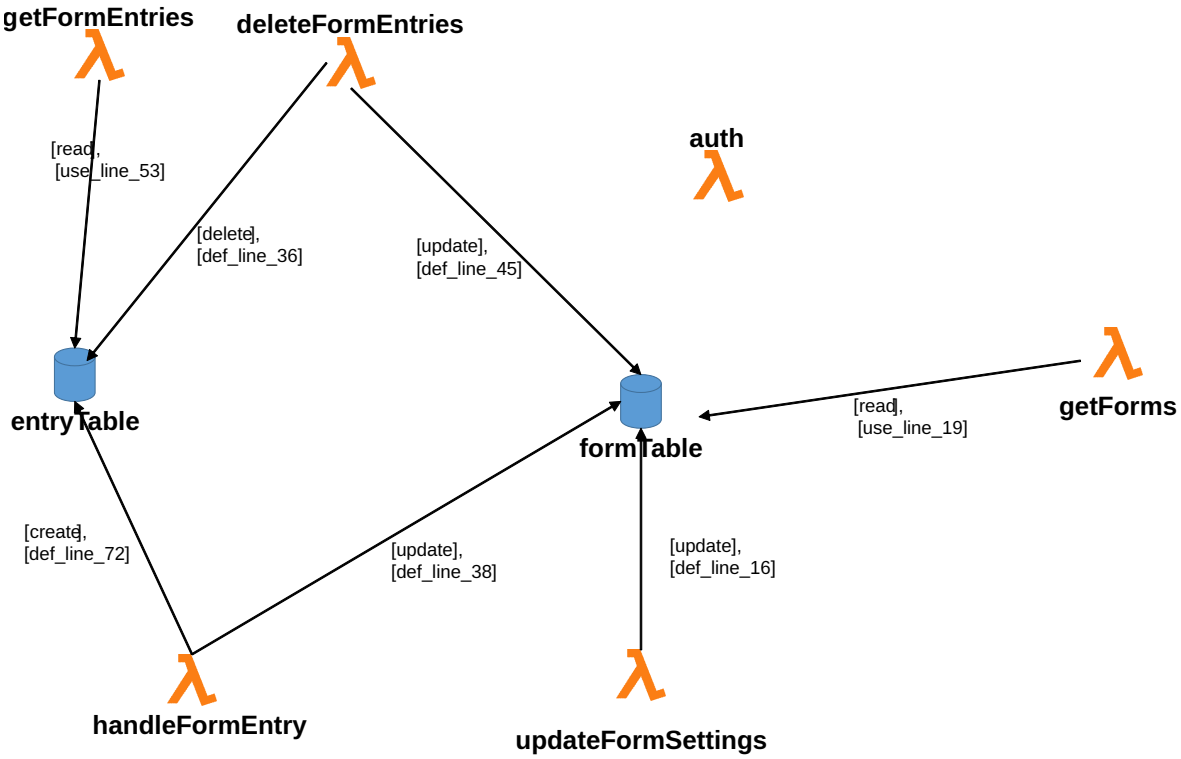



Figure A.6.: Model of application *A6*

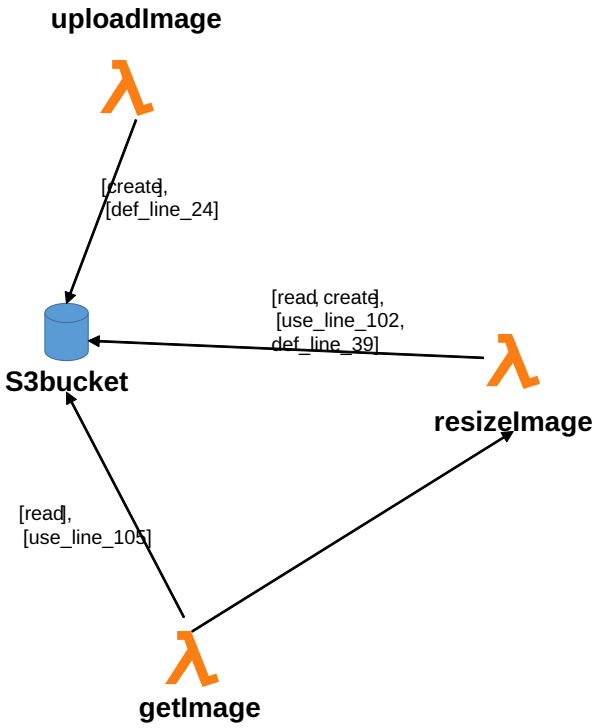



Figure A.7.: Model of application *A7*

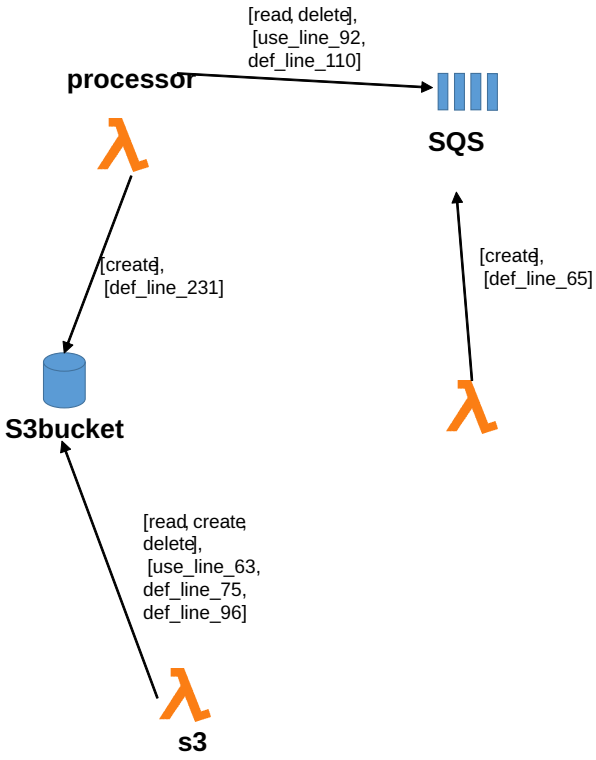



Figure A.8.: Model of application *A8*

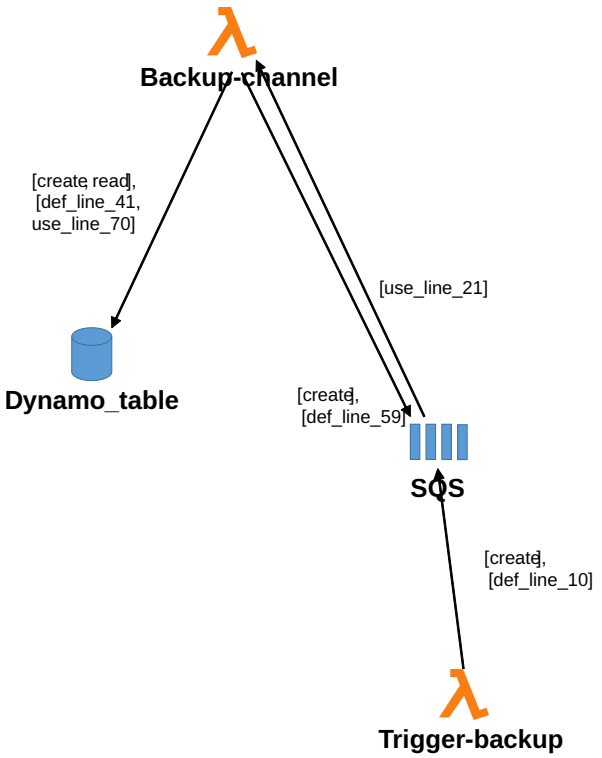



Figure A.9.: Model of application *A9*

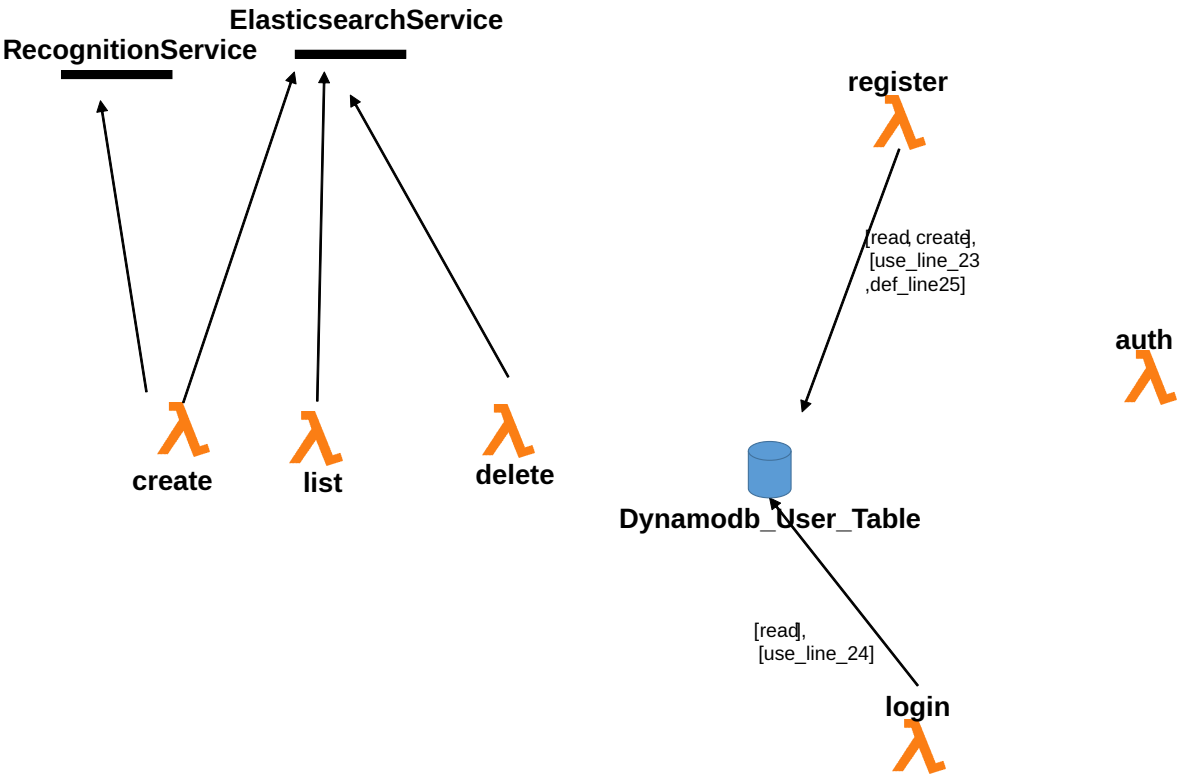



Figure A.10.: Model of application *A10*

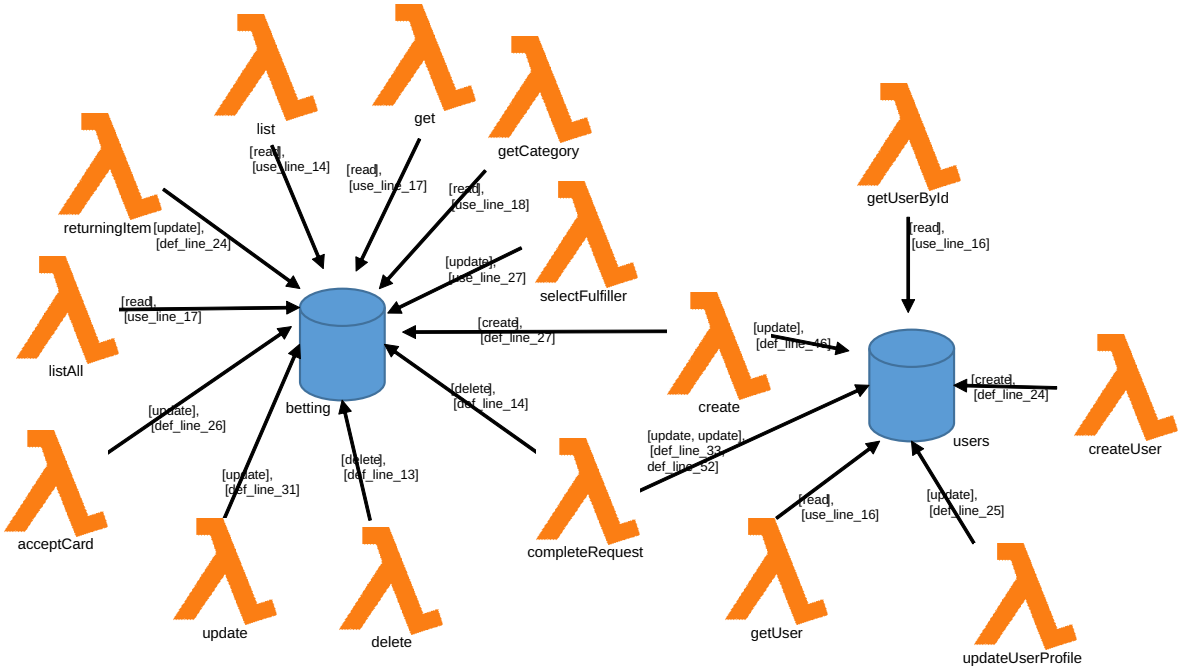



Figure A.11.: Model of application *A11*

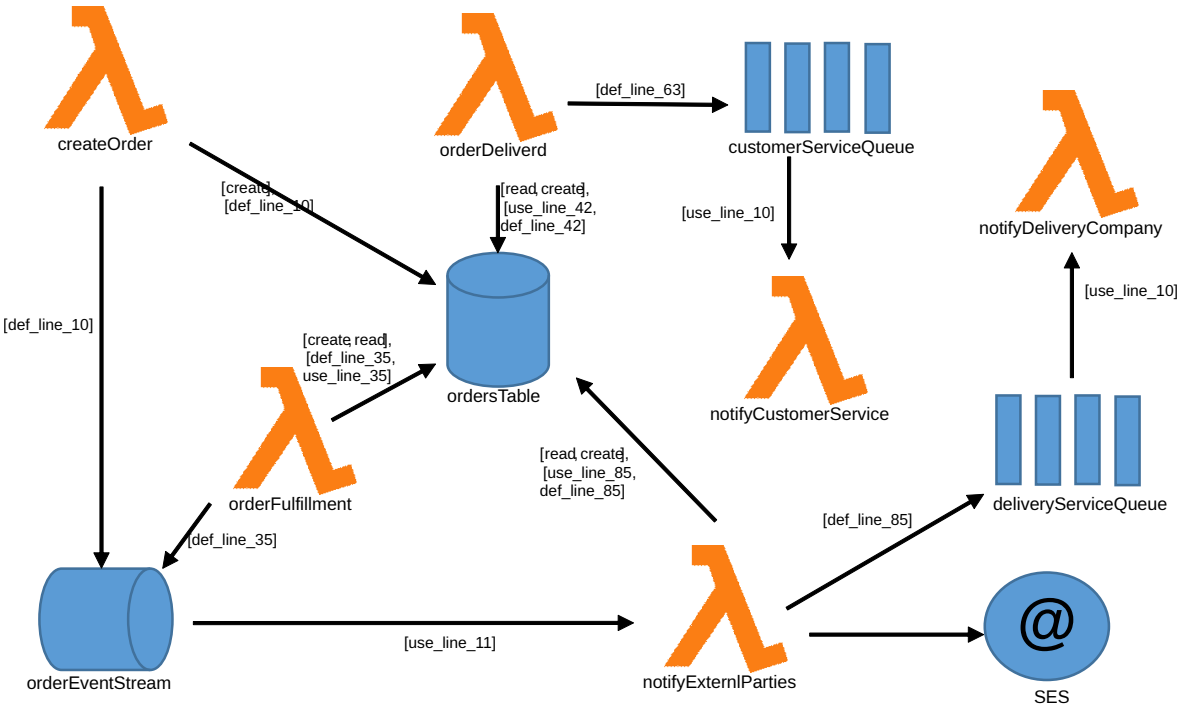



Figure A.12.: Model of application *A12*

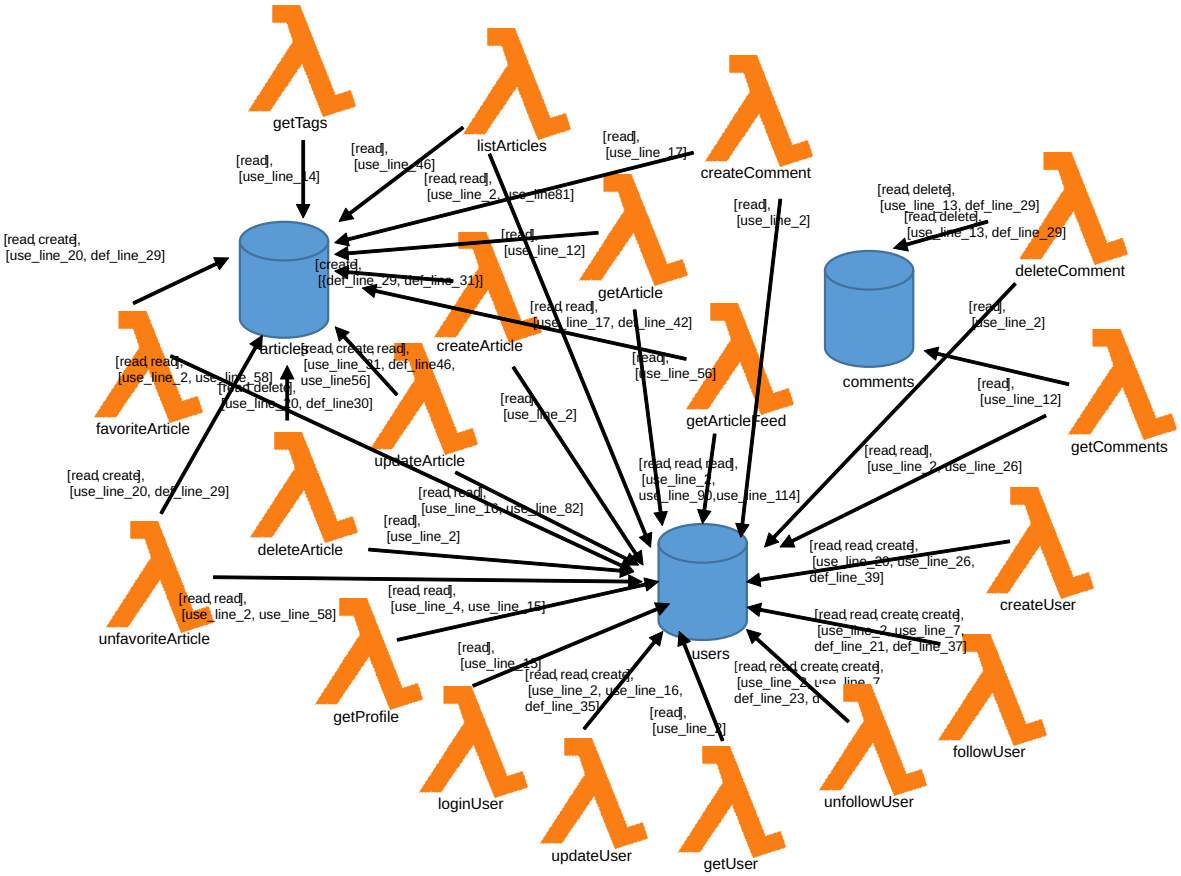



Figure A.13.: Model of application *A13*