Bachelor Thesis

27 July, 2017

jCloudScale Lambda

Automated Transformation of Java Applications to AWS Lambda

Stefan Würsten

of Stäfa, Switzerland (14-725-931)

supervised by

Prof. Dr. Harald C. Gall

Dr. Philipp Leitner

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Bachelor Thesis

jCloudScale Lambda

Automated Transformation of Java Applications to AWS Lambda

Stefan Würsten

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Bachelor Thesis

Author: Stefan Würsten, stefan.wuersten@uzh.ch

Project period: 20.02.2017 - 20.08.2017

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

This bachelor's thesis is the last milestone for my bachelor graduation. I would like to thank all those who supported me in the last three years of my studies.

I would like to thank Prof. Dr. Harald Gall for giving me the opportunity to write this bachelor's thesis at the software evolution and architecture lab at the University of Zürich. Special thanks go to my advisor, Dr. Philipp Leitner for his amazing support and guidance in the meetings over the whole period of five months. I had neither technological nor theoretical experience in the area of client-side middleware for cloud computing. Without such great support, it would be impossible to master such a challenging topic. The time spent writing this bachelor's thesis will remain in my memory as an educational and challenging time.

Abstract

Today, software developers often use cloud services to execute their applications. One possibility in the cloud is to run the code without managing any underlying resources. This approach is called serverless computing. For instance, the cloud provider offers a Function as a Service (FaaS) platform, where the program logic is executed as a function. The function is invoked with input parameters and returns a return value. Each invoked function is independent of any other previous executions, because persistent data are never saved in a serverless environment.

For a software developer, it is time-consuming to chain single functions together to form an application. In this thesis a framework is presented, called jCloudScale Lambda, which supports the developer in writing FaaS-based applications. The program is written as a regular Java application, and at runtime the framework transforms the code into the desired format from the cloud service. The goal is to provide a powerful, but also user-friendly framework. The framework follows an approach that was pioneered at the Vienna University of Technology with its JCloudScale framework.

First the used approach and system architecture are explained. Next, the functionality is illustrated with code snippets. In the qualitative evaluation, an existing project is refactored into a cloud-based application. In addition, some guidelines are provided on how to write a cloud-based application with jCloudScale Lambda. Next, the performance of the framework is measured in a quantitative evaluation. The startup and runtime performance are analyzed and compared to a regular application. Moreover, the effectiveness of the automated code transformation from the framework is investigated. Finally, the current conceptual and technical restrictions of jCloudScale Lambda are summarized.

Zusammenfassung

Softwareentwickler verwenden heutzutage immer häufiger Cloud Services, um ihre Applikationen auszuführen. Eine Möglichkeit dabei ist Code in der Cloud auszuführen, ohne dass sich der Entwickler um die darunterliegenden Ressourcen kümmern muss. Dieser Ansatz nennt sich Serverless Computing. Der Cloud Anbieter offeriert beispielsweise eine Function as a Service (FaaS) Plattform, auf welcher die Programmlogik als Funktion ausgeführt wird. Die Funktion wird dabei mit Eingabeparametern aufgerufen und gibt einen Rückgabewert zurück. Jede aufgerufene Funktion ist unabhängig von einer vorher aufgerufenen Instanz, da keine persistenten Daten gespeichert werden.

Für einen Softwareentwickler ist es aufwändig einzelne solche Funktionen zu einer Applikation aneinanderzureihen. Das in dieser Arbeit vorgestellte Framework, jCloudScale Lambda, unterstützt den Entwickler beim Schreiben von FaaS-basierten Applikationen. Das Programm wird als reguläre Java Applikation geschrieben und zur Laufzeit vom Framework in die gewünschte Form transformiert. Das Ziel ist es ein einfach zu bedienendes, aber mächtiges Framework anzubieten. Das Framework orientiert sich dabei am Ansatz, welcher das JCloudScale Framework der technischen Universität Wien bereits verwendet.

Zuerst werden die Vorgehensweise und die Systemarchitektur erklärt. Danach wird die Funktionalität anhand von Codebeispielen erläutert. In der qualitativen Evaluation wird zuerst eine bestehende Applikation umgeschrieben und eine Möglichkeit vorgestellt, wie man vorgehen kann, um eine jCloudScale Lambda basierende Applikation zu erstellen. Die Performance des Frameworks wird in der quantitative Evaluation gemessen. Es wird beispielsweise untersucht, wie sich die Performance zur Start- und Laufzeit im Vergleich zu einer regulären Applikation verhält. Zusätzlich wird die Effektivität der automatische Codetransformation durch das Framework untersucht. Schlussendlich werden die konzeptionellen und technischen Restriktionen von jCloudScale Lambda zusammengefasst.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Cloud computing is a fast-growing area in the information technology field. The use of cloud computing increases security, enables a simpler scaling of resources and allows a more efficient and timely roll out of patches and updates [Haeberlen12]. Cloud computing offers a highly agile environment with lower capital costs compared to a conventional Virtual Machine (VM) environment [Bruneo14]. Today there are a large number of Infrastructure as a Service (IaaS) providers such as Amazon Elastic Compute Cloud (EC2)[1], Microsoft Azure[2] or DigitalOcean[3]. Other information technology companies provide Platform as a Service (PaaS), e.g. Google Cloud[4] or OpenStack[5]. On top of these cloud infrastructures, Facebook, Google and Co. offer a wide range of Software as a Service (SaaS), which are a part of our daily life.

Another trend in software engineering is the microservice architecture. The microservice architecture has started gaining popularity because of a better flexibility, isolated components and a higher maintainable system [Dragoni16] [Dragoni17]. Amazon had recognized the trend and released a serverless computing platform called Amazon Lambda[6] in November 2014, which brings both approaches, cloud computing and microservice architecture, together. Serverless computing means that *"the application is run in stateless compute containers that are event-triggered, ephemeral and fully managed by a third party [Roberts16]."* Amazon Lambda uses the Function as a Service (FaaS) architecture, so the user can create his Lambda function without managing any underlying resources. The key difference between FaaS and PaaS is that the FaaS provider checks for each request if it is necessary to start a container with the application. After some time without a request, the provider will remove the container, so the scaling is entirely done by the provider. A developer does not have to care about how many requests per second a function receives, because the platform provider manages the scalability of that serverless function [Avram16]. The provider can manage the function, because it is stateless. Stateless means that an invocation of a function is independent of any other previous executions [Yan16].

To build an application with a FaaS approach, a developer writes stateless functions and chains them together. If a developer wants to create more than just a couple of serverless functions, an automated code transformation into the model, which the FaaS provider expects, is helpful [Spillner17a]. This is the challenge of cloud-based software engineering when integrating state-of-the-art cloud services, such as Amazon Lambda, with standard software development environments and existing programming languages and their infrastructures [Cito15]. Hence a framework is necessary that transforms the code, uploads the built file into the cloud, configures the serverless

[1] https://aws.amazon.com/en/ec2/
[2] https://azure.microsoft.com/en-us/services/virtual-machines/
[3] https://www.digitalocean.com/
[4] https://cloud.google.com/appengine/
[5] https://www.openstack.org/
[6] https://aws.amazon.com/releasenotes/AWS-Lambda/8269001345899110/

functions and returns the Uniform Resource Locator (URL) of the Representational State Transfer (REST)-ful endpoint.

The Serverless Framework[7] is an open-source Command Line Interface (CLI) tool, which helps developers for different programming languages such as Node.js or Java to build and deploy auto-scaling, pay-per-execution and event-driven serverless functions. However, a significant amount of knowledge about the chosen serverless cloud provider is necessary. Another promising approach uses PyWren[8], a Python-based prototype from the University of Berkeley in California. The main idea of this prototype is that a developer must only press the push to cloud button [Jonas17]. Currently, the functionality of the prototype is quite limited. Thus, today there is no framework with a user-friendly, but powerful concept to create and manage a FaaS-based application. Either the developer needs substantial knowledge about how a specific provider service works to configure the application, or the framework was built for general performance measurements of a FaaS model and the functionality is limited.

1.1 Contribution

The absence of an easily operable but powerful framework presents a motivation to close this gap. This thesis introduces a prototypical Java-based middleware framework called jCloudScale Lambda, which modifies a regular Java application into many small, independent serverless functions and uploads them into the cloud. The framework is currently limited to Amazon Web Service (AWS) Lambda on the provider side. However, the approach is not limited to Amazon and Java, it can be easily implemented on top of other programming languages and supports more cloud providers. With annotations, the developer can configure which part of the application should be deployed to the cloud as a serverless function. The written application appears like a common local Java application, but on application startup with Aspect Oriented Programming (AOP) and bytecode manipulation the application is modified. The general idea will follow the development model pioneered by JCloudScale, wherein cloud applications are built as local Java applications and cloud services are injected at application startup via bytecode transformation [Leitner12]. With JCloudScale, the business logic and implementation of the scaling behavior are separated [Zabolotnyi15].

With jCloudScale Lambda a developer does not require any specific knowledge about the cloud service. An understanding of the idea of serverless computing is sufficient. The first time, a basic configuration to set up the AWS credentials is necessary, but after that all instructions are annotation-based. For example, with annotations the developer can control which instance and class variables are only locally available, or if a variable should be passed by-value or by-reference to the cloud.

To evaluate the usefulness and power of the framework, an existing project is refactored to advise under which circumstances outsourcing of a part of the application in a FaaS is useful. Some instructions are provided as to which steps are helpful to convert an existing, local program into a cloud-based application. Additionally, the overhead of an automated code transformation and the performance of the framework compared to a local execution is measured. The performance is compared at startup time and runtime. The performance of by-reference passing is compared to by-value passing, and the current restrictions are explained.

[7] https://serverless.com/
[8] https://pywren.io/

1.2 Thesis Outline

The thesis is structured as follows:

- Chapter 2 provides necessary background information. Basic terms such as cloud computing or AOP are discussed and the JCloudScale framework is introduced.
- Related work presenting similar analyses is summarized in Chapter 3.
- The framework jCloudScale Lambda is introduced in Chapter 4. The basic concept is explained and the architecture and implementation are illustrated.
- The evaluation part in Chapter 5 is divided into three parts. The first is a qualitative evaluation, where an existing project is implemented with the framework. Second, a quantitative evaluation measures the performance. The final part explains under which circumstances the utilization of the framework offers benefits.
- Chapter 6 concludes the thesis, summarizes the framework and the evaluation, and outlines possible future work.

Chapter 2

Background

The following chapter introduces basic terms and discusses the central topics on which this thesis is based. First cloud computing, FaaS and AOP are introduced and explained. Finally, the JCloudScale framework is presented.

2.1 Cloud Computing

Cloud computing is changing the business world. Much of the interaction on the Internet operates in a cloud environment. For several years there was no accepted overall definition of cloud computing, as all existing definitions only focused on one aspect of the technology [Geelan09] [Vaquero08]. In 2011, the National Institute of Standards and Technology (NIST) published one of the most cited and accepted definitions:

"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. [Mell11]"

These essential characteristics follow from the definition:

- **Geo-distribution and ubiquitous network access:** Cloud applications are generally accessible through the Internet. Any type of device with Internet access, such as a mobile phone or a computer, is able to access cloud services. Providers have geographically distributed data centers to achieve the best service utility [Zhang10].
- **Resource pooling:** The cloud provider has a homogeneous infrastructure that is shared between all users [Grobauer11]. Because of economies of scale the provider can offer a much cheaper infrastructure with his huge data centers [Armbrust10].
- **Rapid elasticity:** A user can acquire or release cloud computing resources within a short time [Armbrust09]. To a consumer it seems as if he can demand unlimited resources at any time [Mell11].
- **On-demand self-service:** An elastic application can automatically change the amount of used resources according to demand [Galante12]. No human interaction is necessary to acquire and release resources.
- **Measured service:** The usage of resources is monitored, measured, and reported based on utilization for both the provider and consumer of the service [IBM13].

Cloud computing is much more than just virtualization. A user does not have to know how the service is implemented. An understanding of what the service offers and how to operate it is sufficient [IBM13]. Cloud computing is not a new concept, it is related to technologies such as cluster computing, grip computing and distributed systems in general [Foster08]. Cloud users typically pay-per-use, which allows them to only pay for resources that they actually use. With a Service Level Agreement (SLA) the cloud provider commits to providing the service for a certain uptime [Vaquero08]. The terms of use stipulate how the compensation will be calculated if the downtime is higher than a pre-defined percentage. Amazon for example will pay out service credits as compensation if the downtime is more than 0.1% over a month [AmazonSLA17].

2.1.1 Types and Service Models

Usually four different cloud deployment models are distinguished [Mell11]. On the one hand, a cloud provider can offer a public cloud, which is available for a public target group [Ramgovind10]. On the other hand, a cloud infrastructure can be exclusively setup for a single organization, then the infrastructure is called a private cloud. The platform can be owned and operated by the organization or a highly specialized thirty party [Mell11]. Several organizations can use the same cloud infrastructure as a community to pursue common objectives. The community cloud is a public cloud with a limited number of known users [Dillon10]. The hybrid cloud is a mixture of at least two different cloud infrastructures from above. For example, an organization has a private cloud and supplements it with a public cloud to increase the computing capacity at short notice [Sotomayor09].

In these cloud infrastructures there are three popular cloud service models:

- **Infrastructure as a Service (IaaS)**
  With IaaS a user can create his own virtual server environment in the cloud, for example with Amazon EC2. Thus, the user has full control over the operation system [Bhardwaj10].
- **Platform as a Service (PaaS)**
  The provider offers a web-based application-development platform. Users deploy their applications with the provided on-demand tools to a platform such as Heroku[1]. The provider manages and controls the operating system [Lawton08] [Rimal09].
- **Software as a Service (SaaS)**
  The end-user uses a service such as Microsoft Office 365 Outlook[2], which is hosted by the provider in the cloud. The provider manages any underlying cloud infrastructure, and an end-user can utilize it without thinking about the technical aspects [Buxmann08].

There are other service models as well, such as Backend as a Service or Storage as a Service [Sareen13]. Containerization implementations such as the open-source project Docker[3] have also become more popular. With a Docker the application is completely isolated from the underlying operation environment [Bernstein14]. A VM needs a configuration for the operating system, whereas containers only require the libraries and settings to make software work [Docker17]. An advantage of containers compared to VMs is that the booting process is completed after a few seconds [Dua14].

The FaaS service model is a new trend in cloud computing. The big cloud players launched a FaaS environment in their clouds as an answer to the Docker technology, where serverless functions can be configured through a web interface or an Application Programming Interface (API).

[1] https://www.heroku.com/
[2] https://www.office.com/
[3] https://www.docker.com/

2.1.2 Function as a Service

Function as a Service (FaaS), also known as serverless computation, is a service model in cloud computing in which the cloud provider fully manages the underlying container as necessary to serve requests; the user pays based on a pay-as-you-go pricing policy [Miller15]. The serverless approach allows developers to write code without concerns about the runtime environment and resources [Spillner16a]. Serverless is an alternative way of creating backend applications without thinking about the dedicated infrastructure [Yan16]. A serverless function is a piece of software, a code snippet, on the cloud provider infrastructure, which can be started by either a triggered cloud event or calling a serverless function through network protocols such as a Hypertext Transfer Protocol (HTTP) request, or an API call. Examples for cloud events are e.g. a field in the cloud database has changed, a file has been uploaded, or a message is received over a messaging system [Malawski16]. The functions are generally stateless, so they lose all local states after the termination. If persistent behavior is desirable, a key-value store, database or file storage as an external stateful service is required [Spillner17b].

As illustrated in Figure 2.1, the FaaS service model is the logical conclusion of the evolution of sharing in the cloud [Hendrickson16]. In the first step, different users rented virtual machines in data centers as IaaS and did not have physical servers. Next, a full access to the operation system was not necessary and so PaaS became popular, where the provider offered a set of tools and technology for the user to configure the runtime environment. Finally, the runtime environment is now also shared with FaaS. A user can set some basic configurations and define the application, a serverless function; anything else is managed by the provider.

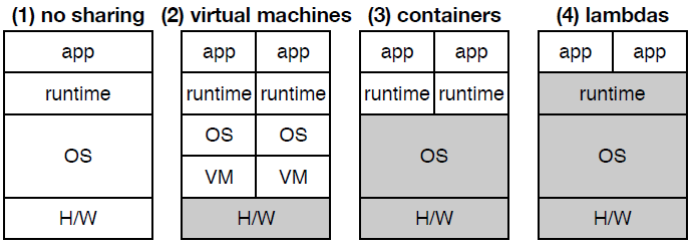


Figure 2.1: Evolution of sharing in the cloud [Hendrickson16]

As Hendrickson et al. illustrated, the median response time of AWS Lambda from a simulated load burst is only 1.6 seconds, whereas in Elastic Beanstalk, a PaaS application management platform, the load burst takes 20 seconds [Hendrickson16]. The FaaS model can scale quicker if the number of requests increases or decreases. To amortize the startup costs for creating a container, Amazon attempts to reuse the same container and to run multiple handlers with the same container whenever possible. For fast calculation in a millisecond range, the latency of AWS Lambda is normally between 75 and 150 milliseconds, compared to Elastic Beanstalk, which is more than five times faster [Hendrickson16].

Characteristics

FaaS has some characteristics that differ from other cloud computing approaches. For example, the user has no control over the execution environment, such as the underlying operating system, but he can use custom libraries with a package manager such as Maven[4] for Java [Malawski16]. The user can configure some runtime options such as language of the runtime environment, timeout, or the amount of Random Access Memory (RAM) that the function can allocate. Although a customization of the runtime environment is not possible, the developer has no more need to manage or maintain a server.

The execution of a handler is run in a sandbox, called container. The container ensures that each execution is independent from any other execution of the same function, so the interaction is stateless [Hendrickson16]. A container normally exists for several seconds or a few minutes, but not for longer. For example, AWS Lambda has a maximum execution limit of five minutes, after then the functions will be terminated by the provider. In contrast to Elastic Beanstalk, no configuration for scaling is necessary; the provider will start and terminate containers based on the usage. Hence, the key operational difference between FaaS (Amazon Lambda) and PaaS (Elastic Beanstalk) is scaling [Roberts16]. A FaaS is similar to a stored procedure that can be invoked.

FaaS Providers

The first platform where developers could write code at a functional level was Zimki in 2006 [Zimki06]. In November 2014 Amazon was the first major provider to offer a serverless based product. At the release AWS Lambda supported only Node.js as a language [Lambda14]. Over the years Amazon added more and more programming languages, and other cloud providers released their own products.

Provider	**Languages**	**Classification**
AWS Lambda	Node.js, Java, Python, C#	Commercial service
Google Cloud Functions[5]	Node.js	Commercial service
Azure Functions[6]	Node.js, C#	Commercial service
OpenWhisk[7]	Node.js, Swift, Binary (Docker)	Commercial service, open-source
Hook.io[8]	Node.js, ECMAScript, CoffeeScript, Python	Commercial service, open-source
OpenLambda[9]	Python	Research prototype, open-source
IronFunctions[10]	Binary (Docker)	Open-source

Table 2.1: FaaS implementations with the supported languages and classification

Table 2.1 presents an overview of some popular products. The products do not use a homogeneous approach of implementation. Some of them use the Docker container technique as the basic element of implementation, such as IronFunctions or OpenWhisk, while others such as Microsoft or Google design their own environment according to their respective infrastructure. As

[4]https://maven.apache.org/
[4]https://aws.amazon.com/lambda/
[5]https://cloud.google.com/functions/
[6]https://azure.microsoft.com/en-us/services/functions/
[7]https://developer.ibm.com/openwhisk/
[8]https://hook.io/
[9]https://open-lambda.org/
[10]https://www.iron.io/

a consequence of the diversity of the underlying techniques, the ways how to code a serverless function differ. If a developer chooses a product, he cannot easily change to another product due to the vendor lock-in [Roberts16]. Vendor lock-in means that the current code depends on a specific vendor product. To switch vendors, the implementation has to be redesigned, and tools for deployment, monitoring etc. have to be updated.

Advantages and Disadvantages

Each service model has benefits and drawbacks. One of the most important advantages of FaaS is that scaling is completely done by the provider and the user only pays for the consumed computing time. Multi-threading problems are handled by the provider and the developer does not have to worry about them. Villamizar et al. evaluated the costs of Amazon Lambda compared to a regular, monolithic application and microservices. They calculated the cost per million of requests. Depending on the scenario, the monthly costs can be reduced by 50% [Villamizar16]. Making a general statement is hard, because the application architecture of a serverless program is fundamentally different to a regular application and so the costs per million of requests are not meaningful in metric terms. With the FaaS model, the developer can more focus on creating the application and does not spend time on environment configuration and scaling, so the development costs are reduced [Roberts16]. Another benefit is that for example OpenWhisk has a marketplace, where a developer can share solutions for recurring problems.

The largest drawback is that currently many limits exists on the provider side. For example, for AWS Lambda there is a limit of 1000 concurrent executions over all functions per Amazon region. Some providers raise limits on request [AmazonLimit17b]. Another problem is that the execution limit of five minutes is too short for some calculations. Testing and debugging is difficult. Most of the platforms that provide FaaS are not open-source, so a developer only has some log files for debugging. The platform is like a black box. A developer cannot write integration tests on the platform, so if a test is necessary he must write his own tests over the API. To manage a set of functions over the web interface or the REST-ful API, one needs to handle each function separately. Serverless functions cannot be grouped, or new versions cannot be deployed, with a few clicks for a group of functions [Roberts16]. Another drawback is that the web interface and the API do not operate in identical ways. For example, with Amazon Lambda, a developer can add a REST-ful endpoint over the web interface with two clicks. With the API ten requests are required to create the default configuration of a REST-ful API endpoint. The documentation only explains basic problems, anything else is not explained.

FaaS is a new field in cloud computing. Each approach has its benefits and drawbacks, but in the future some drawback will be resolved [Roberts16]. For example, the community of AWS Lambda launched an open-source project called docker-lambda[11], which is a local environment that replicates the functionality of AWS Lambda. The debugging is much easier in a local environment. If the providers expand the documentation with technical information about their implementation, and the providers or the community release more such helpful tools, programming serverless applications will become much easier and more comfortable.

[11] https://github.com/lambci/docker-lambda/

2.2 Aspect Oriented Programming

Today the dominant programming paradigm is Object Oriented Programming (OOP), where the software system is built by decomposing the problem into objects [Elrad01]. AOP is another programming paradigm, which allows the separation of cross-cutting concerns [Kiczales97]. Most applications contain a common functionality that is used in many modules for example logging or caching. *"Such functionality is generally described as cross-cutting concerns because it affects the entire application, and should be centralized in one location in the code where possible. [Microsoft17]"* With the OOP approach a developer cannot (easily) centralize recurring problems in the code, so another programming paradigm is necessary. Tracing is an often used example, which is implemented with the AOP programming paradigm. As showed in Listing 2.1 for Java, with conventional OOP the developer must explicitly set logging information for entering and leaving a method to build a trace.

```
public void methodA(){
    logger.info( "Enter 'methodA'" );

    methodB();

    logger.info( "Leave 'methodA'" );
}
```

Listing 2.1: Logging a trace with OOP

In each method, a developer has debug code for logging the trace, making the code harder to maintain and understand. AOP is a structural extension of the existing OOP possibilities, for simpler realization of cross-cutting concerns [Elrad01]. Repetitive code is outsourced into methods in a separate file, called aspects, which have pointcuts as interceptor points. In Listing 2.2 the object-oriented code is refactored into aspect-oriented code.

```
@Aspect
public class myAspect{

    @Before( "call( * *.methodA(..) )" )
    public void doBefore ( JoinPoint joinPoint ){
        MethodSignature signature = joinPoint.getSignature();
        logger.info( "Enter '" + signature.getMethod().getName() + "'" );
    }


    @After( "call( * *.methodA(..) )" )
    public void doAfter ( JoinPoint joinPoint ){
        MethodSignature signature = joinPoint.getSignature();
        logger.info( "Leave '" + signature.getMethod().getName() + "'" );
    }
}
```

Listing 2.2: Logging a trace with AspectJ (AOP)

Aspects are the key element of AOP. With AOP, it is possible to program cross-cutting concerns in a modular way. A developer has the usual benefits of a modular architecture, such as a code that is more readable and easier to develop and maintain [Kiczales01]. Aspects work fundamentally different compared to subprograms and subroutines. Aspects isolate the implementation of a feature such as logging from the business logic. A subprogram or subroutine only moves the code into another method, but structurally the code is still connected and the different concerns are not separated [Elrad01]. Popular programming languages such as C/C++, C#,

Java or JavaScript do not natively support AOP, but external libraries such as jQuery AOP[12] or AspectC[13] add aspect-oriented features to the language.

2.2.1 AOP in Java

AspectJ[14] is the most popular implementation of AOP, a Java extension. The core of the extension is the ajc compiler, which weaves the defined aspects and Java code together into coherent Java bytecode. There are different ways to do the weaving. The simplest approach is compile-time weaving, where the source file and aspects are compiled together in a class file. In post-compile time weaving, a jar file is modified with the aspects. Load-time weaving is done if the class loader loads a class file.

AspectJ uses a dynamic join point model, in which join points are principled points in the execution of the program [Kiczales01]. A join point is identified by the signature of a method, constructor or field. Join points are implicitly given, so they exist for each method, constructor and field access. A pointcut has a pattern that is checked for each joint point. If a join point fits to the pattern, the advice body is executed. Each joint point has an advice in AspectJ. The advice tells when a join point is reached, for example before or after the execution of method code. Possible advice types are before, after, around a method execution, construction call, a get/set of a field or if an exception handler is executed [AspectJ17]. An advice body can modify or replace the functionality of a method [Kiczales01]. The pointcut *call(* *.*(..))* is executed if any method is called. The wildcards in the previous example can be replaced by an explicit name of the component. The package name, class name, method name, method arguments (type of a specific argument or number of arguments), method annotations or return type [AspectJ17] serve as identification. For example *call(int Point.*(..))* is a pointcut for all methods in the class *Point*, which has an integer as a return type.

2.2.2 Advantages and Disadvantages

A benefit of AOP is that the components with different concerns are physically separated, so the code is easier to maintain and adapt. For example, if a user has the authorization to enter a part of an application, security checks are distributed over the whole code. An aspect centralizes the authorization check into a separate module, so there is no code duplication in each module. This means that the code is easier to reuse and extend. The non-intrusive logging and tracing functionality is a helpful tool to understand a program, and is also a perfect example for AOP [Laddad03].

A drawback of AOP is that the weaving generates some overhead, which influence the performance [Johnson05]. In case of AspectJ, the Java bytecode is modified and the overhead of the interception is small. With other AOP frameworks in other languages the interception mechanism may have a significant influence on the performance [Johnson05]. Performance is important, but not the only facet. The traceability of what the code does is more complex, because a developer cannot see which part of the code the aspect will modify. Hence, testing the code needs more effort. A developer does not know, how a particular method is handled by an aspect. Most debugging tools do not support AOP, for example, a developer cannot set a breakpoint in an advice body with standard Java debugging tools in a common Integrated Development Environment (IDE). To find unwanted side effects is hard if a pointcut is not configured correctly. Understanding the program flow is more complex than in an object-oriented application [Laddad03].

[12] https://github.com/gonzalocasas/jquery-aop/
[13] https://www.cs.ubc.ca/labs/spl/projects/aspectc.html
[14] https://eclipse.org/aspectj/

In their study on exception detection and handling using AOP, Lippert et al. reduce the code of error detection and handling by a factor of four in their reimplementation of the JWAM framework with AspectJ [Lippert00]. The findings are that the code is more tolerant to changes in the specifications, better for incremental development, and the program texts are cleaner and easier to reuse. Newbie programmers should be careful with using AOP, because they can create anti-patterns.

2.3 JCloudScale

JCloudScale[15] is an open-source, research prototype developed at the Vienna University of Technology. JCloudScale is a Java-based middleware framework for building elastic cloud applications. The user writes regular Java applications and defines a scaling policy, and the framework deals with the cloud [Zabolotnyi15]. Thus, the framework will dynamically manage the cloud resources and release and acquire virtual machines if necessary. It is perfectly suited to build multi-threaded, computation-heavy, memory-heavy and elastic applications. JCloudScale runs on top of IaaS clouds. Because of the implementation, the framework is independent of the chosen provider and no vendor lock-in exists. The framework tries to combine the advantages of both IaaS and PaaS.

JCloudScale is a Java-based framework, which is controlled over annotations. The code distribution and scaling related code is introduced at application startup time, using bytecode manipulation and AOP [Leitner12]. A developer adds a Maven dependency and sets the configuration for the used IaaS platform before he uses the framework. After that the developer defines some cloud objects using annotations, and he can determinate if a parameter should be passed by-value or by-reference, or whether an instance should be deployed to the cloud. JCloudScale even supports file dependencies. With a class annotation, the developer defines the required files, which should be deployed with the code to the cloud.

For debug and testing reasons JCloudScale supports a local deployment model, where separate local Java Virtual Machine (JVM) instances are created. Thus, no distribution happens over multiple hosts. The communication between remote hosts and local applications is performed over JClouds API[16] for machine startup and shutdown. Anything else is communicated over a message queue [JCloudScale17]. JCloudScale requires some user-defined scaling policies to known how many hosts are required. For simple scaling there are some default events such as RAM or Central Processing Unit (CPU) usage, but the developer can define and trigger custom events.

The developers' user study shows that the model pioneered by JCloudScale has advantages for a developer compared to native IaaS and PaaS implementations. For example, the amount of code is reduced, because almost all code for interacting with the cloud is unnecessary. An important finding was that for some developers it is still difficult to set the scaling policy correctly [Zabolotnyi15], moreover, a more powerful way to handle faults is desirable.

jCloudScale Lambda adapts the pioneered development model from JCloudScale. The approach that a cloud application is written as a regular Java application and cloud services are injected at application startup is converted into a model that is based on a FaaS model.

[15] https://github.com/xLeitix/jcloudscale/
[16] https://jclouds.apache.org/

Chapter 3

Related Work

Serverless computing has been, and still is, an active field of research. This chapter summarizes an existing serverless framework and previous research.

3.1 Podilizer

The Podilizer[1] is an open-source, research command line prototype developed at the Zürich University of Applied Sciences, School of Engineering. The aim of the tool is to transform regular, monolithic Java code to AWS Lambda units [Spillner17a]. If a developer has an existing project and would like to use a serverless architecture from a common FaaS provider without reimplementing the project, Podilizer helps to transform the code. At the moment, the whole project is transformed and the user cannot configure, which part should be deployed into the cloud or to set special configuration (more RAM or a higher timeout) for a specific method.

The input is a valid Java 7 project. As output the Podilizer creates a new Java project, where the whole business logic is outsourced into different serverless functions and the local application invokes only serverless functions in the correct order. Additionally, a jar file is built for each Lambda function. Optionally the tool deploys and configures the functions on AWS [Spillner16b]. So far not all features from the entire Java language specification is supported, for example a method should not use the *this* context, or the file system would be unavailable. Instance variables and method arguments are so far always passed by-value. If a method that is deployed to the cloud is invoked, first an input object is created and the current instance state and parameter are saved. The input object is serialized and a request is sent to the REST endpoint in a JavaScript Object Notation (JSON) format. The response is a JSON object, which is the serialized result of the execution. The result object contains the return value of the invoked method and the instance variable values, where the updated value is saved in the local application [Spillner17a].

Podilizer is implemented in Java with Google's Java Parser and the Abstract Syntax Tree framework[2]. During the transformation the tool changes the visibility of variables and methods and adds a public no-argument constructor if a no-argument constructor does not exist. Outsourcing a simple calculation is currently possible. A problem for more complex applications is that a multiplicity of restrictions exists. An existing regular, monolithic project normally requires some refactoring because the developer team decided to optimize the usage of network resources, and so getter and setter methods are not REST endpoints and under certain circumstances some values are not correctly setted or getted. The Podilizer team has done preparatory work on how

[1] https://github.com/serviceprototypinglab/podilizer/
[2] https://github.com/javaparser/javaparser/

a Java program must be modified to create a set of stateless functions. jCloudScale Lambda has enhanced the approach from Podilizer how the code is modified.

3.2 PyWren

The University of Berkeley in California developed a Python-based, open-source framework, called PyWren[3]. Their motivation was the barrier for an average scientific user, which currently is too high to deploy his code to the cloud [Jonas17]. A serverless execution model provides a more user-friendly approach to run distributed applications. In the model pioneered by PyWren, the serverless function is deployed and invoked on runtime with some input and returns of an output object. PyWren uses Amazon Simple Storage Service (S3) as storage unit to transfer the input and output over the network. So far, the prototype only supports AWS Lambda on the provider side. A serverless function only has method arguments as input, hence it is so far not possible to pass instance or class variables. The model is captivating due to its simplicity. A drawback is that there is no caching mechanism, so for each invocation the serverless function is newly deployed. Due to the Amazon limits and the long setup time for the custom python runtime, PyWren is optimized for task with a duration of more than 20 seconds [Jonas16].

The PyWren team evaluated the pioneered model with some performance benchmarks. One focus was on whether the S3 storage is the bottleneck of the current system architecture. However, that proved not to be the case; in the test cases with an image processing pipeline, the selected solution scales up to 2800 simultaneous functions with 60 GB/s read and 50GB/sec write performance.

The PyWren team showed that it is possible to provide a user-friendly framework. For simple, but computationally intensive calculations, PyWren is the perfect choice to deploy code to the cloud. The developer only has to press the "deploy to cloud" button. jCloudScale Lambda adapts the simple handling for developers, so that a developer without cloud computing background can use the framework without first having to read a long documentation.

3.3 Serverless Framework

The Serverless Framework[4] is an open-source CLI tool written by a full-time development team using Node.js. The framework currently supports Amazon AWS, Microsoft Azure, IBM OpenWhisk and Google Cloud Platform as serverless cloud providers, and Node.js, Python, Java and Scala as programming languages [Serverless17a]. The tool helps to manage the lifecycle of applications, with a serverless architecture to build, deploy, modify and delete serverless functions. The framework extends the basic functionality of the cloud providers. For example, the framework makes it possible to group functions to manage them more easily [Serverless17b]. The Serverless Framework is only an extension for the Graphical User Interface (GUI) and CLI of the providers for easier management of serverless functions. Additionally, a developer has a collection of example codes. The framework does not transform existing regular code into serverless code.

The community extends the framework with plugins, for example someone adds a local environment for testing, or a functionality to chain functions in a queue [Serverless17b]. The framework helps to easier manage Lambda functions, but a fundamental knowledge about the underlying service is necessary. For example, to configure a serverless function in the Amazon cloud a developer must set the permission using the Amazon syntax. There are some step-by-step

[3] https://pywren.io/
[4] https://serverless.com/

tutorials and a detailed documentation about the framework [Serverless17c], but none of these explain why, for example, the created user needs administrator permission for AWS. The Serverless Framework is a layer between the application from the user and the service provided in the cloud. On the one hand, the framework simplifies the handling of serverless functions, by means of providing helpful features for managing them. On the other hand, a developer does not understand what the framework does if he has never written native serverless functions. If an issue occurs, the exception is hard to understand, and a developer without cloud computing experience cannot solve the issue [Brown16].

If a developer decides to use the Serverless Framework for a project, he must first choose the provider, because the configurations and the code syntax differ for each provider. A developer cannot easily change the provider later due to the vendor lock-in, so a reimplementation of the code is necessary. The Serverless Framework is a tool that adds helpful managing functionality for a developer.

Compared to jCloudScale Lambda, the Serverless Framework is more a managing tool. During runtime the Serverless Framework has no influence on what happens. The goal of jCloudScale Lambda is to eliminate the need for substantial serverless knowledge. jCloudScale Lambda transforms regular code into serverless code, so that a developer does not need to know the syntax of a specific provider.

3.4 Serverless Chatbot

The University of Illinois at Urbana-Champaign and the IBM Watson Research Center jointly developed a chatbot using a serverless platform. The chatbot consists of an interface with different REST endpoints [Yan16]. A chatbot requires to chain functions together. For example an audio file is sent from a user as input; the first service converts the audio file into text. The next service routes the user input to the serverless function, which can handle the input. Then another service extracts the needed parameters from the text to invoke the final service, which handles the input. Some services may be called external services, for example a weather service. Finally, the output is converted into the desired format, text or audio. The chatbot is implemented on IBM OpenWhisk, but the chosen architecture can be adapted to most other cloud providers. The code is currently not open-source, but the team plans to release the code.

The chatbot answers simple questions about the weather, the current date, or acts as an alarm service. The chatbot has similarities to the AWS Alexa Skill Software Development Kit (SDK)[5]. The presented chatbot can be easily extended with own functionalities, but the current core code is not performance optimized. The bot uses existing services such as IBM Watson Dialog Service[6] for understanding and responding to user questions, the IBM Speech to Text Service[7] to convert audio into text or IBM Watson's Entity to get the name of the city from a string. The challenge is to build a module-based application, which can be extended with more modules and external services.

During developing the chatbot development team noticed conspicuous behavior. Creating a single serverless function is easy, but to chain the functions together is a difficult task. Therefore, a framework or tool that support the developer is desirable. Another feature that may be useful is a debug tool in the cloud environment (or an emulated local environment) to set breakpoints, which makes debugging much faster. The chatbot, as a use case of serverless computing with serverless functions, shows the possibilities of services such as IBM OpenWhisk or AWS Lambda, but with some disadvantages. An important drawback is that chaining functions together is cumbersome,

[5]https://developer.amazon.com/alexa-skills-kit/
[6]https://www.ibm.com/watson/developercloud/dialog.html
[7]https://www.ibm.com/watson/developercloud/speech-to-text.html

but a real-world application is a conglomerate of small units. Most cloud providers, for example Amazon, designed serverless functions for simple trigger-based jobs such as adding some metadata to a S3 bucket [Jackson17]. In November 2014, when Amazon, as the first serverless function provider, released its Lambda platform, the developer team did not consider that Lambda facilitates the building of large, chained applications. Amazon recognized the problem and added a beta support for AWS Lambda to AWS X-Ray[8] in April 2017 [AmazonXRay17]. X-Ray is the Amazon analysis and debugging tool for distributed applications.

The serverless chatbot is a complex example of a possible serverless application. The findings during the development of the chatbot are an inspiration for the core idea of jCloudScale Lambda. The jCloudScale Lambda middleware framework tries to eliminate some of the problems explained.

[8] https://aws.amazon.com/de/xray/

Chapter 4

jCloudScale Lambda Framework

As already stated, building a native serverless application is difficult. There is no middleware framework for FaaS based application that is on one side powerful and on the other side user-friendly. jCloudScale Lambda tries to fill the gap with a framework that is simple to handle but yet powerful with a large set of functionalities. On high-level, it follows the concept of JCloudScale with a serverless implementation based on the FaaS model. The framework uses an approach that is a mixture of the two existing research projects PyWren and Podilizer, as already explained in Section 3.1 and 3.2. Compared to the Serverless Framework, a developer does not need a significant amount of knowledge about the used cloud provider to configure a serverless function. jCloudScale Lambda ports the idea of JCloudScale to AWS Lambda and considers the findings of related serverless projects. The basic functionality is similar to JCloudScale, and a non-trivial feature is also realized with the pass by-reference. Other features such as file dependencies or data store integration are currently not implemented.

jCloudScale Lambda is Java-based and uses Amazon as provider, because AWS Lambda is one of the most popular serverless platforms and is not in a beta phase like many other provider services. The framework can be integrated into a project as a Maven dependency. The developer must only configure some settings once; after that all instructions are annotation-based. In the git repository, a configuration guide exists under *jcs_lambda/docs/Configuration.md*, which explains how to set up the framework the first time.

The framework itself uses the MojoHaus Maven Plugin[1] to automatically change the compiler to ajc instead of using the default compiler. The ajc compiler weaves the code and aspects together; at application start up the code is deployed to the cloud and the cloud services are injected via bytecode transformation. AOP and reflection are important technologies to attain the goal of a slim framework.

Amazon provides two possibilities to interact with its services in addition to the GUI. On one hand there is the AWS CLI[2] and on the other hand the AWS SDK for Java[3]. The CLI offers the basic functionality of the service with simple commands. The SDK has more features, but is sometime more complex to operate. jCloudScale Lambda uses the Java SDK, because the CLI returns the result as string and not in a suitable format such as JSON or Extensible Markup Language (XML).

In this chapter, the functionality of the framework, and how to use it and the high-level architecture are explained.

[1] https://www.mojohaus.org/aspectj-maven-plugin/
[2] https://aws.amazon.com/cli/
[3] https://aws.amazon.com/sdk-for-java/

4.1 System Architecture

As illustrated in Figure 4.1, in a regular Java application a method is called from an invoker, the method calculates something and returns a value if the method has not a void return type.

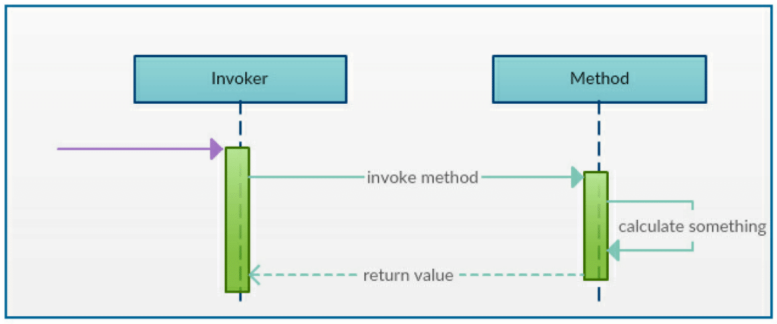


Figure 4.1: Method invocation of a regular Java application

The model of the local method invocation must be adapted for methods that are executed in the cloud. Figure 4.2 depicts a sequence flow diagram on high level of abstraction, which explains what the jCloudScale Lambda framework does at runtime. Each method that should be deployed to the cloud has a *CloudMethod* annotation. If such a method is invoked, the original method body is not executed. The method passes all required instance and class variables from the context and method arguments to the CloudManager. The CloudManager knows to which REST-ful endpoint the current method belongs.

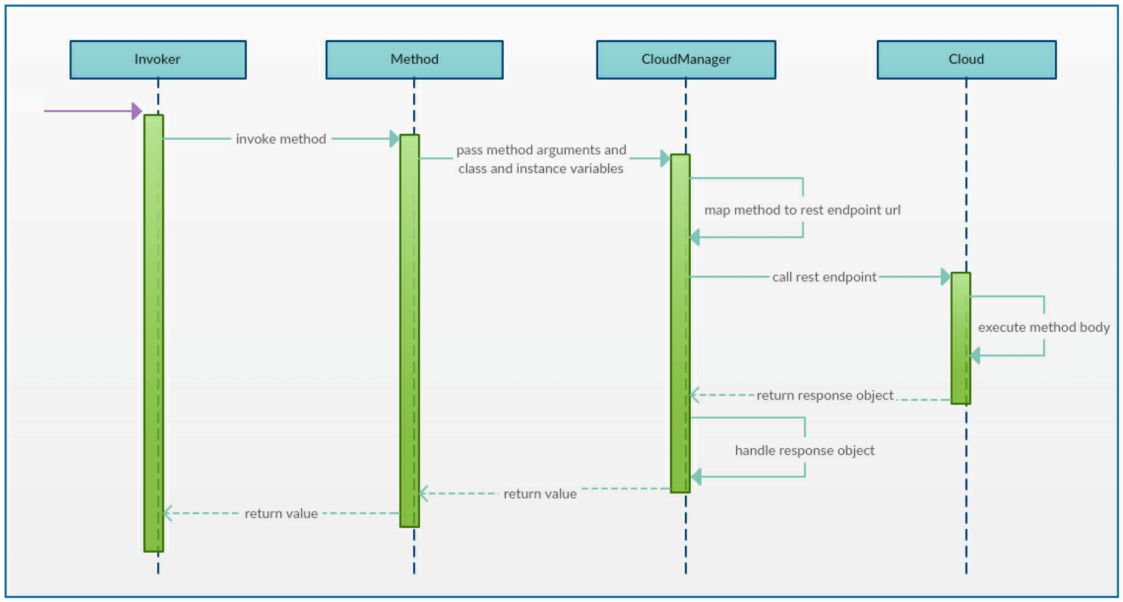


Figure 4.2: Method invocation of a method with a *CloudMethod* annotation from jCloudScale Lambda

The CloudManager creates a request object and, if necessary, a message queue for passing by-reference variables is started. Next the REST-ful endpoint is called with the serialized request object, which contains the class and instance variables and the method arguments, and the CloudManager waits for a response. In the cloud the original body of the method is executed. The response of the cloud functions is either a stack trace, or the return value of the method. The CloudManager deserializes the response object and handles all possible errors, for example a timeout or an exception from the cloud. The CloudManager only throws unchecked exceptions, because usually the issues cannot be handled at runtime. If no error occurs, the return value is returned to the invoker.

If the Java application starts, some initialization and tasks are necessary, so that the transformed application works with as little overhead as possible at runtime. The *StartUp* annotation is responsible for initialization of the framework before anything else is executed. The startup process of the framework is illustrated in Figure 4.3. First all methods with a *CloudMethod* annotation are searched with the org.reflections[4] library. An aspect replaces the method body of each method found with a proxy, which calls the CloudManager as explained in Figure 4.2. Each method is registered by the CloudManager, so that the manager can forward a request at runtime to the cloud.

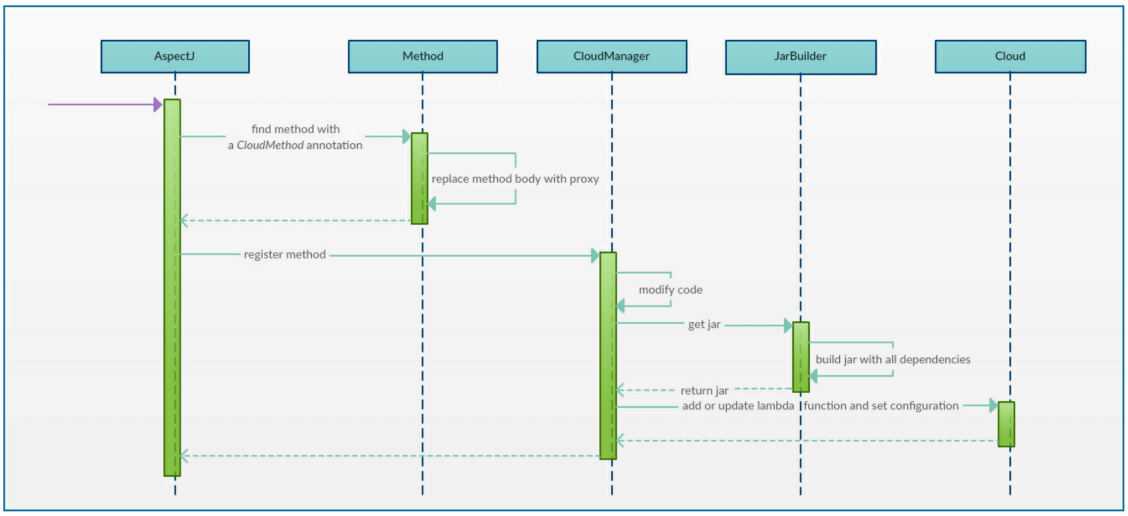


Figure 4.3: Application startup process from jCloudScale Lambda

For each method, the CloudManager has to create two Data Transfer Object (DTO) classes, a request and a response class. A third class is also created, the REST endpoint. All these classes created at runtime are compiled and added to the local running application. The endpoint class has one method, which has the serialized request class as input and the serialized response class as output and is configured as the start point of the serverless function in AWS Lambda. Figure 4.4 depicts a process flow diagram, which explains the life-cycle of an invoked Lambda function. If a user sends a request to a REST-ful endpoint, then Amazon starts the Lambda function. First the input is deserialized. Then, the endpoint class uses reflection to initialize the context and set the by-value passed instance and class variables. After the initialization, the method is executed and the return value is processed further. Last the return value of the invoked method is serialized, the serialized object is sent as response to the invoker and the Lambda function is terminated from Amazon.

[4]https://github.com/ronmamo/reflections/

The framework checks if every serverless function is up-to-date in the cloud. The project is up-to-date, iff the last modification date is smaller than the date of the last deployment process of the framework for each file in the *src* folder. If not, a jar file of the whole project with all Maven dependencies and the dynamic created files is generated. The CloudManager has a list of all Lambda functions. If not all Lambda functions already exist in AWS, the project is always deployed to the cloud. The jar file is uploaded to Amazon S3 and the configuration of each Lambda function is updated, or if the Lambda function does not already exist, a new serverless function is created.

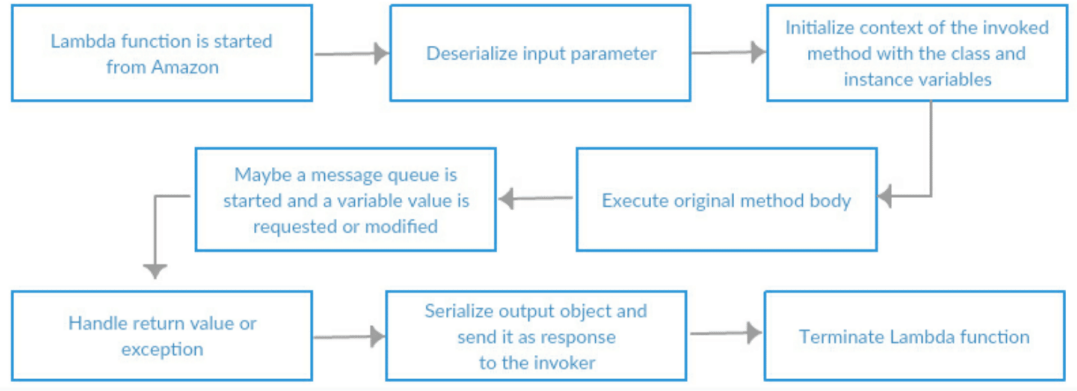


Figure 4.4: Life-cycle of an invoked Lambda function

At startup time, the CloudManager gets a list of all methods that feature a *CloudMethod* annotation. For each method, the CloudManager creates a *CloudMethodEntity* object. A *CloudMethodEntity* has all relevant information such as package name, class name, return type or the URL of the REST-ful endpoint that is required from the framework at startup or runtime. First the raw information must be converted into the accepted format of the *CloudMethodEntity*. For example, each cloud method has a full qualified name, which is a hash value of the concatenated string of the package, class, method name and parameter types. The framework always uses internally this unique name to prevent problems with ambiguous names. If a class has two overloaded methods, the explained identifier is always unique in a single project. The full qualified name of a method with a prefix defined in the configuration file is the name of a Lambda function. If a developer has multiple jCloudScale Lambda based projects and uses the same AWS account, he should use different prefixes. Otherwise, if both projects use the same package, class and method name, the identifier is not unique. Next the *CloudMethodEntity* collects some information about the class to which the method belongs. An important information is if a class has a by-value or by-reference variable and the name and type of that variable. If a timeout or memory value is specified in the *CloudMethod* annotation, the value is picked. Otherwise the default value from the configuration file is taken. In a final step, the URL of the REST-ful endpoint is saved if the serverless function already exists in AWS Lambda. Now enough information is available to deploy the serverless functions to AWS Lambda. If a new serverless function is created, the URL of the REST-ful endpoint is saved. The last piece of missing information is now collected and the framework is ready for the runtime phase.

At runtime, an aspect forwards each invocation of a method with a *CloudMethod* annotation to the CloudManager. The CloudManager gets the join point from AspectJ as a parameter. The join point contains information about the current context, i.e. which object was invoked, and the passed arguments from the invoked method. The CloudManager checks if the class from the current context has a by-reference variable. If yes, then the message queue is started. The request DTO is now created and filled with the values, a HTTP request is started and the response is han-

dled. jCloudScale Lambda also supports methods with a void return type. For the framework, a void return type is a special case, because the response DTO cannot have a variable with a void type. The CloudManager changes the void return type internally to a string and always returns nothing. A serverless function with a void return type can be useful because it can calculate a new value of a by-reference variable using the message queue.

The current architecture of jCloudScale Lambda follows these principles:

- jCloudScale Lambda uses already existing frameworks where possible and does not reinvent the wheel.
- The current implementation supports Amazon only, but the core idea should be as easy to adapt to other cloud providers as possible. Sometimes the concrete design decision is more on reasonable performance and simpler handling, and not on the perfect abstraction for all cloud providers.

If the application crashes, a developer should receive as much exact exceptions as possible. Without custom exceptions the framework often throws a *InvocationTargetException*. A developer cannot find the mistake in his code with such a vague exception. jCloudScale Lambda has some custom exceptions, which should help to localize the issue. The developer receives a custom exception for example if the credential is invalid or the by-reference passing fails because the developer tries to pass a variable of a generic type. If a serverless function does not terminate successfully, a *CloudRuntimeException* is thrown in the local application and the stack trace from the cloud is displayed in the local console. A list of all custom exceptions is available under *jcs_lambda/docs/Annotations_and_Exceptions.md*.

4.2 Functionality

The core functionality of the framework is that an execution of a method can be outsourced into the cloud. A developer writes regular, local Java applications and annotates the methods that should be deployed to the cloud with *CloudMethod*. If it is necessary and the required memory or timeout time differs from the default values in the configuration file, the annotation properties' memory and timeout can be used. The method arguments and the return value are always passed by-value. The framework requires a *StartUp* annotation for the *public static void main* method and any other method, which can be the start point of the application (for example a simple JUnit[5] test). The annotation is responsible for the initialization of the framework. If no *StartUp* annotation is set, the application starts as a regular, local Java application. However, if a method with a *CloudMethod* annotation is invoked and the framework is not already initialized, an exception is thrown. The application can be started as a regular Java application without using jCloudScale Lambda if the AspectJ compile-time weaving is disabled in the Maven configuration.

The current, local value of an instance and class variables are normally not available in the cloud. A developer can annotate an instance and class variables to define if and how the variable is passed to the serverless function. The *Local* annotation is the default case if no other jCloudScale Lambda annotation is used for a class or instance variable. The cloud has no access to the variable value, so the value is only available in the local application. The *ReadOnly* annotation means that a variable value is passed by-value if the method is invoked. The value is available as read-only in the cloud, so a value change only influences the current cloud instance and the value is never updated in the local application. If it is necessary to have write access to an instance or a class variable, the *ByReference* annotation is required. If a serverless function requires the value

[5] http://junit.org/

of such a variable, the current value is automatically requested from the local Java application over the Amazon Simple Queue Service (SQS)[6], after which the value is updated in cloud. An updated value is automatically sent to the local application if the value in the cloud is changed. Any communication between cloud and local client is serialized with GSON[7], so a read-only and by-reference variable and the return value must be serializable. The above information is summarized in Listing 4.1, which shows how to use the jCloudScale Lambda annotations.

```
public class TestObj{
    @Local
    private int a;

    @ReadOnly
    private int b;

    @ByReference
    private int c;


    @CloudMethod( timeout=30, memory=1536 )
    public int sum ( int d ){
        return b + c + d;
    }
}

@StartUp
public static void main ( String [] args ){
    TestObj testObj = new TestObj();
    testObj.sum( 4 );
}
```

Listing 4.1: Sample code of jCloudScale Lambda

Passing By-Value

If an instance or class variable has a *ReadOnly* annotation, the value of the variable is serialized if the method is invoked. A copy of the value is sent to the cloud, so the variable is passed by-value. Exactly at that moment, where the method is invoked, the current value of the local application is copied. After the serverless function starts, the copy in the cloud and the original variable in the local application are not linked. If the serverless function changes the value, the updated value is never transmitted to the local application.

A by-value passing is preferable if the serverless function does not modify the value. If the method modifies a variable value, a developer should always try to ensure that the method returns the modified value to the local application as a part of the return value.

The *Matrix* class in Listing 4.2 is an example for by-value passing. The class has multiple methods each of them is a matrix operation such as solving a linear equations, finding the transposition of a matrix or calculating a matrix multiplication. The class has three instance variable: the number of rows, the number of columns and a double dimensional array with the values of each matrix element. After a matrix object is initialized, the instance variables are never modified. If a method calculates a new matrix, the return value is always a new created object. In this example, the *multiplication* method is outsourced into the cloud. The serverless function needs access to each instance variable, but only with read access. So each instance variable gets a *ReadOnly* annotation.

[6]https://aws.amazon.com/sqs/
[7]https://github.com/google/gson/

```
public class Matrix{
   @ReadOnly
   private int r; // number of rows
   @ReadOnly
   private int c; // number of columns
   @ReadOnly
   private double[][] data; // r-by-c array


   public Matrix (){
      // empty no-argument constructor
   }

   // other constructors

   // create and return the transpose of the invoking matrix
   public Matrix transpose ();

   // return x = A^-1 b, assuming A is square and has full rank
   public Matrix solve (Matrix rhs);

   // return C = A * B
   @CloudMethod
   public Matrix multiplication (Matrix B){
      Matrix A = this;
      if (A.c != B.r){
         throw new RuntimeException("Illegal matrix dimensions.");
      }

      Matrix C = new Matrix(A.r, B.c);
      for (int i = 0; i < C.r; i++){
         for (int j = 0; j < C.c; j++){
            for (int k = 0; k < A.c; k++){
               C.data[i][j] += (A.data[i][k] * B.data[k][j]);
            }
         }
      }

      return C;
   }
}
```

Listing 4.2: Example of by-value passing a variable

Passing By-Reference

The by-reference passing of variables is a non-trivial feature of jCloudScale Lambda. Compared to by-value passing the developer requires a significant amount of knowledge of how this feature is implemented in the framework, in order to avoid problems caused by lacking comprehension. A developer must understand why the explicit set and get is necessary and how to correctly initialize a by-reference variable. In some cases, by-reference passing of a variable is necessary. For example, if a variable value must be sent to the local application immediately and the value cannot be sent as return value, because waiting until the method is terminated would take too long. During the execution of a serverless function, the function cannot communicate with other instances in the serverless environment. However, during the execution a calculation maybe requires some information from other instances executed in parallel. A by-reference variable is like a shared variable in multi-threaded applications.

If a class has multiple methods with a *CloudMethod* annotation and one of these is invoked, all instance and class variables from the current context with a *ReadOnly* annotation are always passed to the serverless function even if they are not used. However, a memory-heavy passed

variable has an influence on the initialization time of a serverless function. Compared to a by-value passed variable the value of by-reference passed variable is only loaded from the local application if the value is required in the cloud instance. If not all methods with a *CloudMethod* needs access to such a variable, the by-reference passing is the preferred solution.

Each object with a by-reference variable has a unique identifier. When the local application starts a serverless function, the unique identifier from the *this* context is sent as input parameter to the serverless function. If the serverless function requests the value of a variable, the variable name and unique identifier of the context are sent over the message queue; the local application can unambiguously identify the correct object context of the request and sends the value of the variable to the cloud.

In Listing 4.3 we have a non-primitive, complex instance variable *point* with a *ByReference* annotation. If the method *modify* overwrites the current point with a new one, everything works without a problem. However, if, as in the example shown, the method only modifies the *x* coordinate of an existing point, the changed value is not sent to the client over the message queue. The problem is that AspectJ has no feature to capture a modification in a variable, which is compounded by different sub-variables. jCloudScale Lambda implements an explicit get and set, where a developer can bypass such problems with compounded variables.

```
public class TestObj{
    @ByReference
    private Point point;

    @CloudMethod
    public void modify (){
        Explicit.get( this, "point" );
        point.x = 3;
        Explicit.set( this, "point" );
    }
}


public class Point {
    public int x;
    public int y;
}
```

Listing 4.3: Explicit get and set with a by-reference variable

Another important aspect is that by-reference variables should never be initialized directly in the class definition. A by-reference variable should always be initialized in a constructor or method. Otherwise the value is initialized once locally, and then again in the cloud with the same value. The current local value, which may be different to the initialization value, is overridden. When the developer does not directly initialize by-reference variables, the cloud will automatically load the current, local value if the variable is required in the cloud.

```
public class BadClass{
    @ByReference
    private int y = 3;


    public void setY ( int newY ){
        y = newY;
    }


    @CloudMethod
    public int sum ( int z ){
        return y + z;
    }
}
```

```
public class GoodClass{
    @ByReference
    private int y;

    public GoodClass (){
        // empty no-argument constructor
    }

    public void initialize (){
        y = 3;
    }

    public void setY ( int newY ){
        y = newY;
    }

    @CloudMethod
    public int sum ( int z ){
        return y + z;
    }
}

@StartUp
public static void main ( String [] args ){
    BadClass badClass = new BadClass();
    badClass.setY( 7 );
    badClass.sum( 5 );

    GoodClass goodClass = new GoodClass();
    goodClass.initialize();
    goodClass.setY( 7 );
    goodClass.sum( 5 );
}
```

Listing 4.4: Initialization of a by-reference variable

Figure 4.4 illustrates the problem of a by-reference variable initialization directly in the class definition. In the *BadClass* the variable y is initially 3 and then changed to 7. If the serverless function in the cloud starts, a *BadClass* object is initialized. The initialization of the *BadClass* object is done with the no-argument constructor, and thus the variable value of y is set to 3. The initialized value from y is sent to the local application and the local value of 7 is overwritten. The *GoodClass* shows how to avoid such problems. The variable is initialized during the execution of the method *initialize* and not in the no-argument constructor; there the same error would occur, as the no-argument constructor is always called if an object is initialized in the cloud. The serverless function in the cloud starts, and during the execution of the *sum* method the current value of y is requested from the local application.

Synchronous and Asynchronous Execution

Normally the execution of a serverless function with jCloudScale Lambda is synchronous. This means that the local Java instance waits for the response from the cloud; until then the thread that invokes the serverless function is blocked and will not execute any other code. If it is necessary to execute multiple calculations at the same time, for example to split a calculation into different smaller sub-calculations, a multi-threaded application is required. Each thread invokes a serverless function. If a developer requires an asynchronous execution of a serverless function, he can

use either an own thread for the asynchronous execution or the Java future implementation. An example of a multi-threaded calculation that use jCloudScale Lambda framework is available in the git repository under *jcs_lambda/code/evaluation/quantitative/ 5.8_evaluation_prime_number/*.

Chapter 5

Evaluation

This chapter is split into three parts. The first part is a case study where an existing GitHub[1] project is refactored into a cloud-based application. In the quantitative part different metrics of the framework are measured and compared. In the last part, the findings are summarized and the conceptual and technical restrictions of the current approach are explained.

5.1 Qualitative Evaluation

In the qualitative evaluation, an existing non cloud-based application is refactored and deployed to the cloud with jCloudScale Lambda. Next some general guidelines are provided on how to write a cloud-based application with jCloudScale Lambda.

5.1.1 Case Study: Cloud Migration

In the case study, an existing project is chosen and the code is refactored into a cloud-based application. First a suitable project must be selected. The selected use case is the Monte Carlo algorithm, which is computation-heavy and can be split into multiple independent threads. The Monte-Carlo-Pi project[2], an existing GitHub project, is a multi-threaded application that approximates Pi. The goal of the case study is to show how a developer can use jCloudScale Lambda to outsource an expensive calculation of the application into the cloud.

First the existing system architecture is reviewed. The author of the project has obeyed common object-oriented guidelines, so the code was well developed. There is a thread pool, and each thread verifies for a defined number of points if they are inside or outside of the circle. The calculation is illustrated in Figure 5.1. The program counts how many points are in the blue area. The result of all threads is summed and divided by the total number of points. Each thread checks the same number of points and there is no communication between the threads during the calculation.

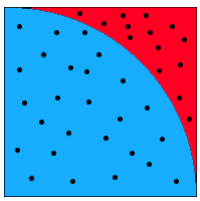


Figure 5.1: Monte Carlo experiment

First there is the decision as to which part of the application should be deployed to the cloud. Here the choice is simple, because we want to outsource the computation-heavy calculation from the local threads into the cloud. The original code of the *MyThread* class, which does not use jCloudScale Lambda, is illustrated in Listing 5.1. The *MyThread* class has only one public method, which controls the calculation for a defined number of points. This method requires a *CloudMethod* annotation. Next for each instance variable from the *MyThread* class we

[1] https://github.com/
[2] https://github.com/ChrisMitov/Monte-Carlo-Pi/

check if they are used in the cloud. All instance variables are used in the method. Except the *numPointsInCircle* variable, the serverless function only requires read access, so for these variables we can use the *ReadOnly* annotation. For the last variable an inspect is necessary if a *ByReference* annotation is required, or if the modified value can be returned at the end of the method execution without by-reference passing. The number of points in the circle are the return value of the method, so a *ByReference* annotation is not required for the variable *numPointsInCircle*. Furthermore, no *ReadOnly* annotation is required, because if the method starts, the value of the variable is always 0. Thus, a jCloudScale Lambda annotation is not required for the *numPointsInCircle* variable. The rewriting of the application is finished and the program is now jCloudScale Lambda compatible.

```
public class MyThread implements Callable<Long> {
   private int nameThread;
   private long sideSquare;
   private long pointsOfThread;
   private long numPointsInCircle;
   private boolean quit;

   // here are a constructor and the methods isInCircle and
       randomNumber, which do not use the instance variables

   public Long call (){
      if ( !quit ){
         System.out.println("Thread" + nameThread + " started" +
             "Points " + pointsOfThread + " !");
      }

      numPointsInCircle = 0l;
      for (long i = 0; i < pointsOfThread; i++){
         long x = randomNumber(0, sideSquare);
         long y = randomNumber(0, sideSquare);
         Point point = new Point(x, y);

         if ( isInCircle(sideSquare / 2, point) )
            numPointsInCircle++;
      }

      if ( !quit ){
         System.out.println(nameThread + " is finishing! Points in
             circle: " + numPointsInCircle + " !");
      }

      return numPointsInCircle;
   }
}
```

Listing 5.1: Code of the *MyThread* class without using jCloudScale Lambda

5.1.2 Guidelines

The Monte Carlo experiment is a small application, but a similar process can be used for larger applications. In this section, an iterative approach is explained to making a local application jCloudScale Lambda compatible, so that an expensive calculation can be executed in the cloud. The high-level process is illustrated in Figure 5.2.

First the developer should ensure that the local application has as few bugs as possible, because debugging in the cloud is harder than locally. Next, the developer should consider which parts of the application make sense to outsource. The use of jCloudScale Lambda is worthwhile if multiple calculations can be done in parallel, the application requires significant amounts of RAM, or the computation takes a long time and can be divided into small sub-calculations. The

sub-calculations should be as independent of other calculations as possible. For example, a gravitation simulation of the galaxy is not suitable, because after each calculation step the calculation unit requires the results of all other calculations executed in parallel. As a result, the synchronization costs are higher than the benefits of serverless computing.

The developer adds the Maven pom.xml and the XML file from jCloudScale Lambda in the resource folder to the project. For each method that should run in the cloud, an iterative process begins. The developer takes a method that should be deployed to the cloud, and ensures that the method runs in a multi-threaded part of the application. If not, the application should be refactored. Each thread invokes the method once, and the calculation is done in parallel in the cloud. The method gets a *CloudMethod* annotation. For each instance and class variable in the class to which the method belongs, the developer must check if the variable is used in the cloud. If not, the developer could add the *Local* annotation for a better understanding. Moreover, the question is if a by-value, read-only passing is sufficient, or a by-reference passing with write permission to the variable is required. Sometimes a developer can prevent the use of a by-reference variable by making a few small changes in the code. For example, if during the execution a method modifies an instance variable value, the method can return the value and the invoker updates the new value in the local application.

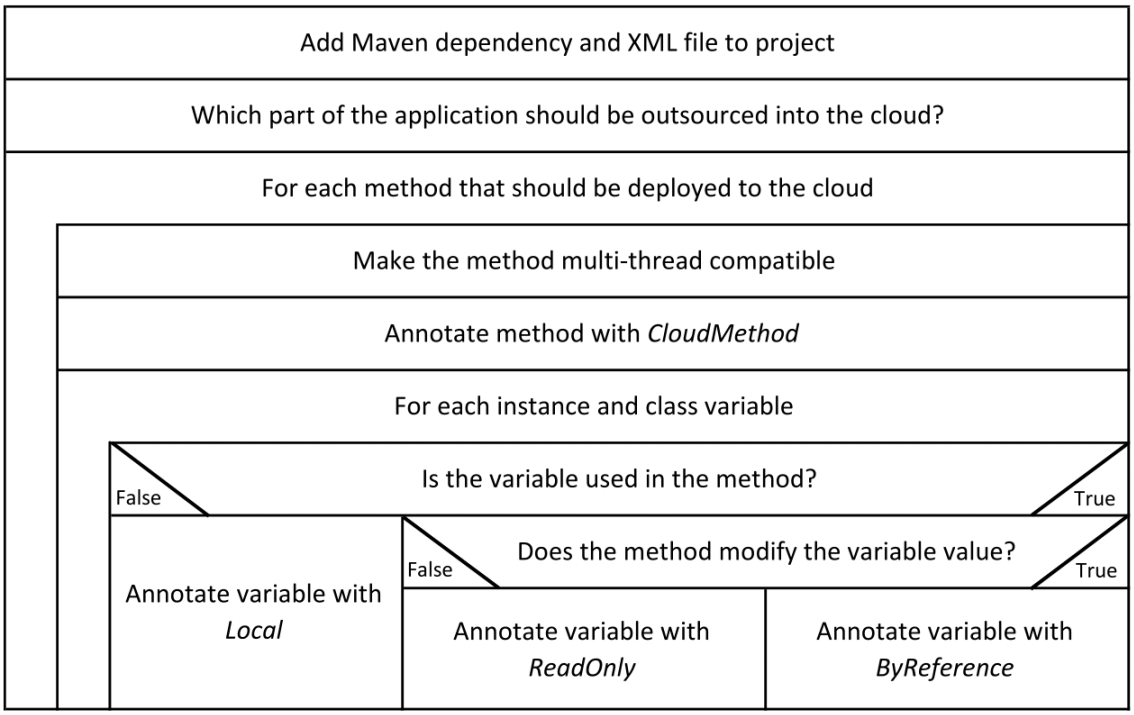


Figure 5.2: High-level process on how to make a local application jCloudScale Lambda compatible

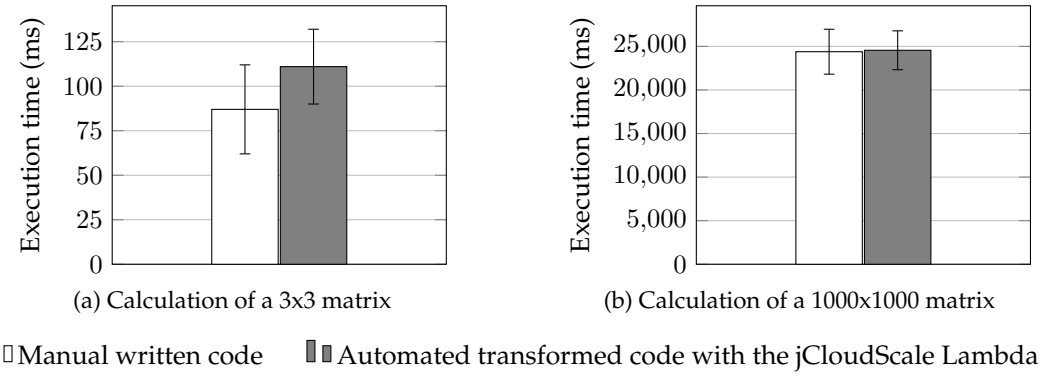
A regular Java application and a jCloudScale Lambda based application do not have the same code structure. In other words, a migration from a regular application to jCloudScale Lambda in more complex projects is not a trivial matter; the subtleties are crucial, so that the application works as desired and does not throw exceptions. Sometimes a developer needs less time to write a new application, as many problems do not occur as often, and to plan for the beginning of a serverless application architecture is simpler than refactoring an existing project.

5.2 Quantitative Evaluation

In the quantitative section of the evaluation, the transformation overhead, startup and runtime performance is measured. All experiments have been executed on an Intel Core i7-3770 processor[3], a quad-core with a 3.4 GHz clock speed and a turbo speed of 3.9 GHz. The computer has a Windows 10 64-bit operating system and 16 GB DDR3 RAM. The computer is connected to the Internet with 40 Mbit/s download and 10 Mbit/s upload speed.

5.2.1 Overhead of Automated Transformation

The framework transforms Java code into another code structure that is required by AWS Lambda. The question is how efficient this process is compared to a manual written solution from a developer. In Figure 5.3 the performance of the matrix multiplication from a manual written code is compared to an application that is transformed with jCloudScale Lambda. In a first case, a 3x3 matrix is calculated. The result is illustrated in 5.3a; the execution time with the automated code transformation from the framework is 30% higher. The reason is that the framework must handle special cases such as errors. If the calculation is done in a few milliseconds, the extra checks from the framework have a large influence on the execution time. Figure 5.3b shows the calculation time of a 1000x1000 matrix. The overhead of the automated code transformation is now less than one percent of the execution time. However, the automated code transformation with jCloudScale Lambda is more user-friendly, and the additional execution time is negligible.



(a) Calculation of a 3x3 matrix

(b) Calculation of a 1000x1000 matrix

Figure 5.3: Manual written code vs. automated transformed code with jCloudScale Lambda

5.2.2 Startup Performance

The startup time is the time period during which the framework is initialized. The initialization phase begins when the Java program is started, and ends if the entry point of the application, for example the *public static void main* method, is executed. The measurement from the startup time is split into different phases. The first phase is the initialization of AWS, where all Amazon services are started with the credential. In the code modification phase, the DTO classes are generated and dynamically loaded into the running application. With Maven a jar file with the main and the test module is built, which includes all dependencies defined in the pom.xml file. Next the jar file is uploaded to Amazon S3. Then the configuration is set for each endpoint. If an endpoint does not already exist, it is not immediately available. The time during which the framework is waiting until the endpoint is available is measured; the endpoint must be available during the runtime. After all endpoints are configured, the S3 bucket is removed. The rest of the time that is assigned to other small jobs is summarized under miscellaneous.

[3]http://ark.intel.com/products/65719/Intel-Core-i7-3770-Processor-8M-Cache-up-to-3_90-GHz/

The startup time depends on various factors. Among these, the number of serverless functions to be deployed to the cloud and the code size with all Maven dependencies are significant. If a serverless function is already in the cloud and does not require an update, the initialization phase is shorter.

In Figure 5.4 the startup time is shown as a function of the code size in MB. Some factors are independent of code size, such as the code modification or how long an endpoint requires until it is available. The build time of the jar file and the endpoint configuration slowly increases with the code size. The file upload is constant between 2.0 and 2.5 MB per second, so the upload time is linear dependent of the jar file size. In addition to the code from the developer, the jar file contains the jCloudScale Lambda and all third parties' frameworks, which require almost 15 MB of space. The code size in the plot is only the size from the written test project without the core, which is a Maven dependency.

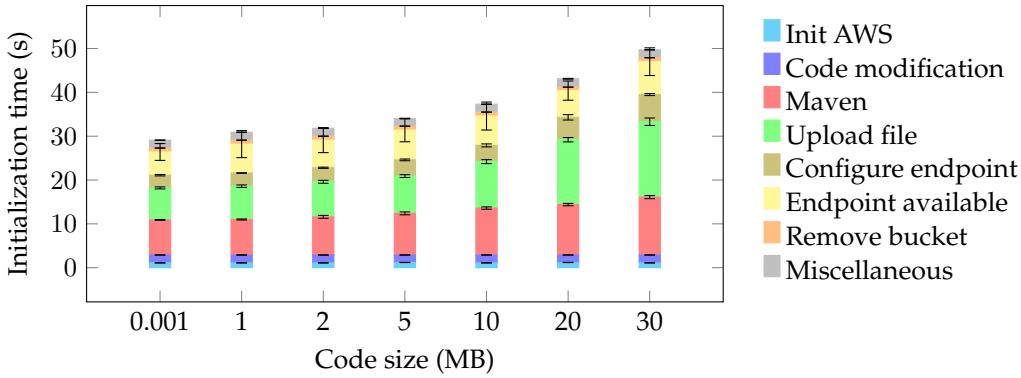


Figure 5.4: Startup time of an application depends on the code size

In the next measurement, the initialization time is shown as a function of the number of serverless functions. Figure 5.5 shows the result when the functions do not exist in AWS Lambda and must be created. The initialization of AWS, the time until the endpoint is available, and the remove bucket phase have no influence on the total initialization time in either measurement. Compared to the code size experiment the code modification time is not constant, because for each method with a *CloudMethod* annotation two DTO classes and one endpoint class are created. The Maven and upload phase are not constant, but the influence on the total time is small. The major difference is that the endpoint configuration takes much longer. Because the endpoint does not already exist, six requests are necessary to configure one single endpoint.

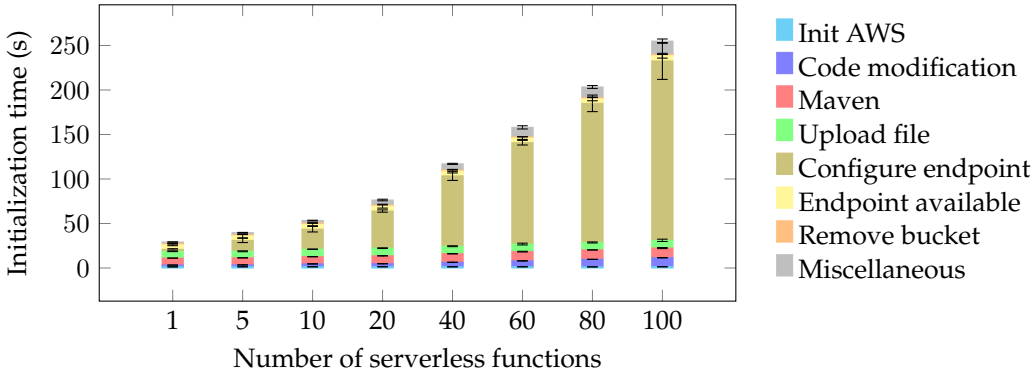


Figure 5.5: Startup time of an application depends on number of not already existing functions

If an endpoint already exists, only two requests are necessary to update the endpoint configuration and link the serverless function to the new uploaded jar file. The configuration for the AWS Gateway API is still the same and does not require an update. Figure 5.6 shows the result if an endpoint is only updated and not recreated. A difference to the creation of an endpoint is that the endpoint is already available, because the Gateway API configuration is not changed. If a new endpoint is published in Gateway API, the infrastructure from Amazon requires a few seconds (normally between three and eight) to distribute the information about a new endpoint. All other phases are unchanged.

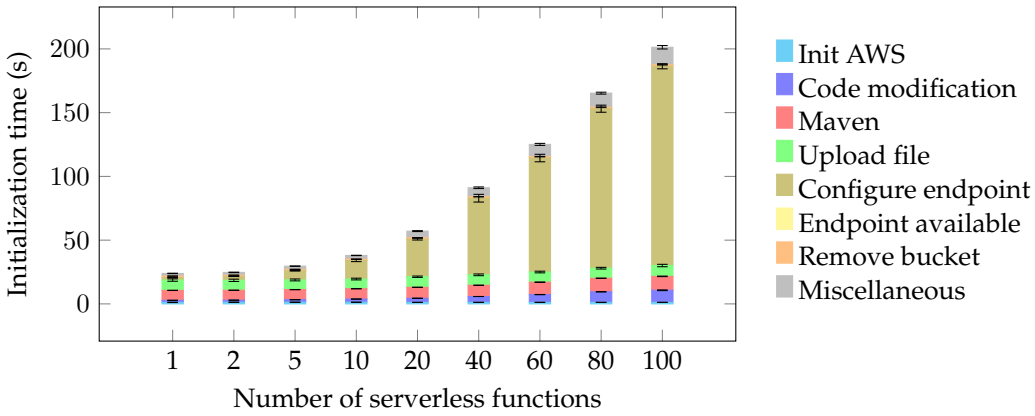
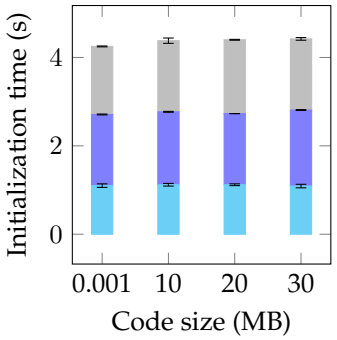
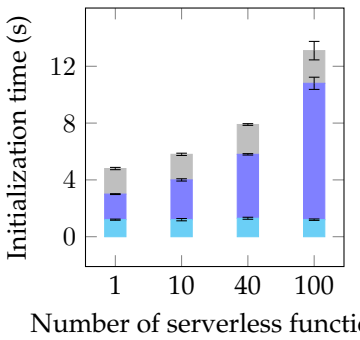


Figure 5.6: Startup time of an application depends on number of existing functions

A third option in the startup process is that a serverless function already exists and does not require an update. In this case nothing in the cloud has to be changed, and only the local applications must be initialized. For each file in the *src* folder, the framework checks the last modification date and compares the overall highest value from the folder with the value from the last startup process. Figure 5.7 shows that the startup process without modifying the serverless functions in the cloud is only influenced by three stages: the AWS initialization, code modification and miscellaneous. Figure 5.7a illustrates that the startup time is constant as a function of the code size. As Figure 5.7b shows, the number of functions has an influence on the initialization time. The AWS initialization and miscellaneous always use the same amount of time, but the code modification is influenced, because more serverless functions mean creating more temporary classes.



(a) Startup time depends on code size



(b) Startup time depends on number of functions

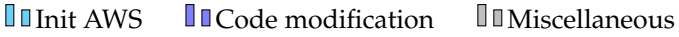


Figure 5.7: Startup time of an application without updating the cloud

Finally, the startup time is influenced by the code size and the number of serverless functions and whether something in the code was changed since the last startup. The code size has particular influence on the upload time and a minor one on the Maven build time. The number of serverless functions especially influences the endpoint configuration time and, to a minor extent, the Maven build time, the code modification and the miscellaneous part. Hence, the file upload and the endpoint configurations are the bottlenecks during the startup process.

5.2.3 Runtime Performance

The startup performance is one aspect of the framework, but another much more important aspect in a production environment is the runtime performance, because a method is invoked more frequently than an application starts. Spillner has shown that the first invocation of a function usually has a longer response time [Spillner16a]. Therefore the serverless functions were already invoked once for each measurement in this section, in order to obtain comparable results.

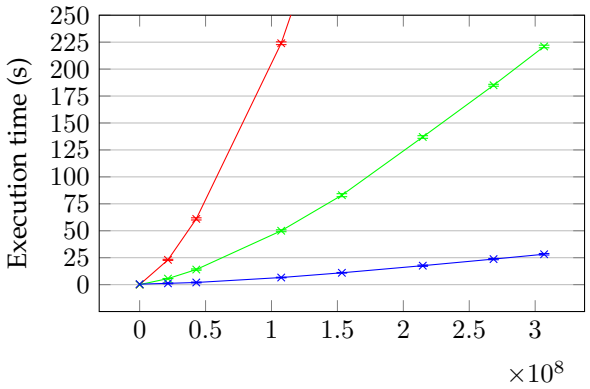


Figure 5.8: Runtime performance of a local application compared to application, which uses the jCloudScale Lambda framework

In Figure 5.8 the performance from the local execution is compared to a cloud-based execution. First, the application is run locally in one thread. Next, the application is run as a local multi-threaded program with a thread pool of 20 tasks. Last, the jCloudScale Lambda support is added and the application runs with 100 serverless functions executed in parallel. The multi-threaded variant uses 100% of the CPU, i.e. the maximum computing power of the local machine. The test case is the calculation of how many prime numbers exist between one and a defined number. The highest upper-bound in the test series is 306783378, where the cloud-based application is more than seven times faster.

The runtime performance can also split into different phases. First, using reflection, a new DTO request object is dynamically created and filled with values. This request object is serialized in the local application and sent to the cloud. In the cloud, the request object is deserialized and the computation is started. The result of the execution is now serialized and the response is sent to the local application. There the response is deserialized and all possible response issues are handled, such as the execution in the cloud times out or an exception is thrown. The sending processes are summarized under networking.

In Figure 5.9 the test case, the matrix multiplication from experiment 5.3, is illustrated with the different explained phases. The experiment has large method arguments as input and a large return value as output. As a result, the serialization and deserialization produce much overhead. The figure illustrates that networking, serialization, deserialization and execution of the methods take up most of the time. The networking is first constant, but with large objects, the speed is linearly dependent on the object size. The serializing and deserializing with GSON also depends on the object size, but in the cloud the process always takes longer than locally, because the serializer can use up to 16 GB RAM in the local application. The algorithm of the matrix multiplication is in the Big O notation O(n^3).

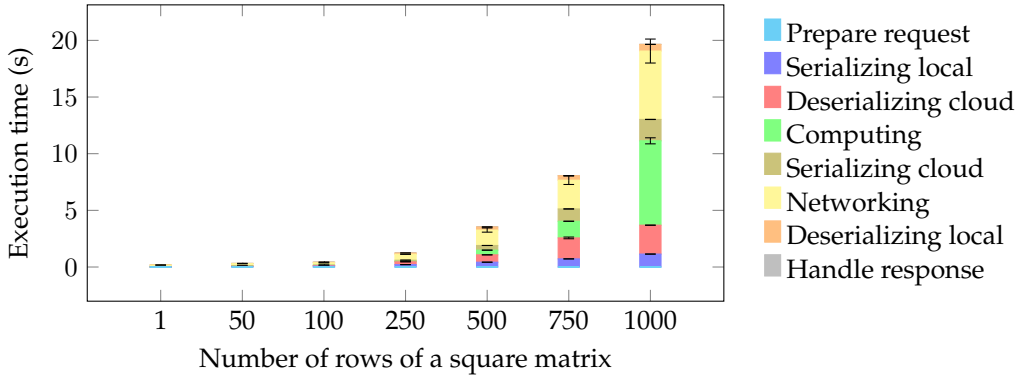
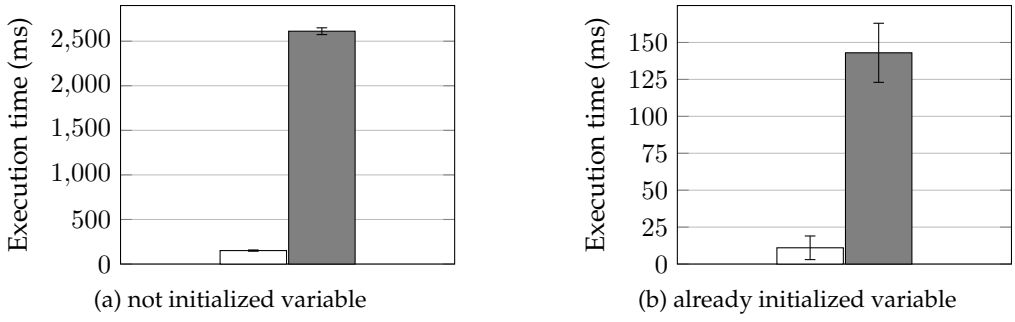


Figure 5.9: Runtime performance of jCloudScale Lambda

By-reference Types

The usage of by-reference types is sometime irreplaceable, but the performance of by-reference passed variable is worse than a by-value passing. If a developer uses by-value passing, the by-value instance variable value is sent as a copy if the execution is started. An application with by-reference variables must first request the current value or write a modified value. First the queue must be initialized that the cloud and local application can communicate. Figure 5.10a shows the performance of simple read access to an instance variable. The by-reference passing is 17 times slower because the value of a by-value passed variable is immediately available. Most of the time in by-reference passing is used for initializing the SQS object.



(a) not initialized variable

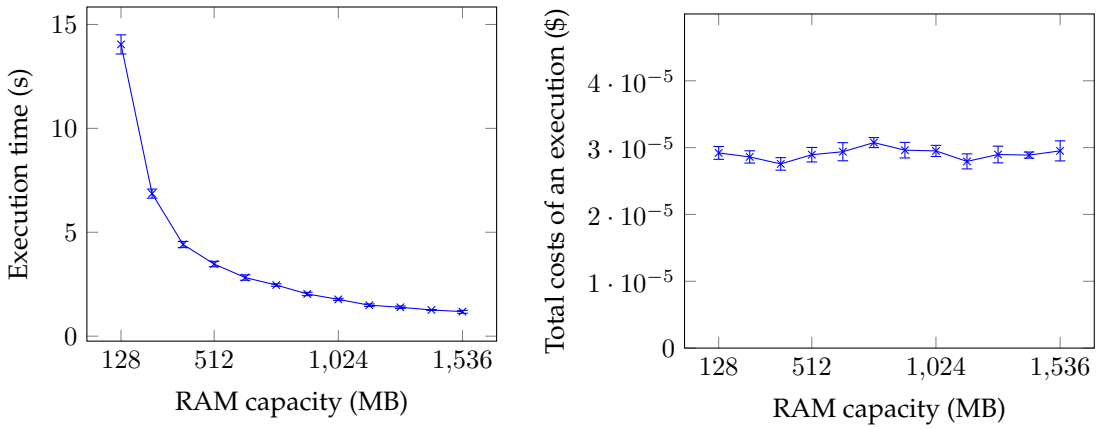
(b) already initialized variable

Figure 5.10: Performance of by-value and by-reference passed variables

If the message queue is already initialized, because of the container caching mechanism from Amazon, the function can immediately request the value. Figure 5.10b represents the performance of an already initialized queue. The cloud only sends a request with the required variable and waits until a response is received. By-reference passing is now only 13 times slower. If a message queue is not initialized, two seconds are necessary with the highest memory capacity until the cloud is connected to the queue. If a serverless function uses the queue multiple times or an Amazon container with an already initialized queue is used, the performance loss is tolerable. By-value passing is the preferred variant due to the better performance. However, sometimes a method cannot simply return the calculated value, then a by-reference passing is required.

Influence of the Memory on the Execution Time

The choice of how much RAM a serverless function should have is interesting. Maybe there is a trade-off between time and costs. From an economic viewpoint, a developer should try to find the configuration with the minimal total cost. Some applications are time critical and the costs are less important. In Figure 5.11a the execution time is illustrated as a function of the memory capacity. The test case is the same as in experiment 5.10. The result is surprising. With 512MB RAM the application takes twice as long as with 1024 MB RAM. The cost per tenth of a second depends on the chosen memory [LambdaPricing17]. The total cost is the multiplication of the needed time for the execution rounded up to the next tenth and the cost per tenth of a second.



(a) Execution time varies with memory capacity

(b) Total costs per execution

Figure 5.11: Influence of the memory capacity on the performance and occurring costs

The total costs are shown in Figure 5.11b. The total costs are always the same. Thus, in a system the maximum memory would always be the best, because the result is computed faster for the same costs. Amazon explains this behavior with its internal mechanism of CPU sharing [LambdaCPU14]. The amount of allocated RAM defines how much computing power a Lambda function has.

5.3 Discussion

The qualitative and quantitative part of this chapter evaluates the concept and implementation of the system in terms of usefulness and overhead. The finding of the qualitative evaluation is that an existing application can be refactored into a cloud-based one with jCloudScale Lambda. To take the benefits of the framework, the algorithm that would be deployed should be computation-heavy, memory-heavy or multi-threaded. A calculation that requires much synchronization or communication between multiple parallel executed instances is not recommended in combination with jCloudScale Lambda, because the framework does not have an efficient functionality for communication during the execution time with other instances executed in parallel. With jCloudScale Lambda, a developer can more focus on the business logic. The framework manages the interaction with the cloud in the background. In the qualitative part of the evaluation, some guidelines are provided that relate to the process a developer could apply when creating an application with jCloudScale Lambda.

In the quantitative part of the evaluation, first the overhead of automated transformation is analyzed. The slightly higher execution time is negligible, because the developer does not have to write each serverless function by hand. At startup time, the framework takes much longer compared to a native application. Possibly the code in the cloud is not up-to-date and the framework must deploy the modified code. If the newest version is already in the cloud, the Java bytecode of the local application must still be modified. The method body of each method with a *CloudMethod* annotation is replaced with a proxy, which invokes the cloud. In a production system, the code is not often modified and the initialization time of the framework without updating the serverless functions is only a few seconds. Therefore, the slowness of the framework during the startup process is not a major disadvantage.

The runtime performance is of more importance, because each method invocation of a serverless function has an influence on the performance of the whole application. A local application is limited to the RAM and CPU of the local machine. For larger calculations, a distributed solution is necessary. The jCloudScale Lambda framework can take advantage of its strengths as soon as the local calculation takes more than just a few seconds. If not, the overhead of the framework with serializing, deserializing and networking is too high.

The evaluation shows that the major disadvantage of the by-reference passing instance and class variable is the performance. A developer should bear in mind that by-reference passing sometimes solves problems that cannot be implemented with an *ReadOnly* annotated variable, but several problems can be solved by refactoring the code and returning the updated value with the return value.

The Amazon Lambda service is a black box. Thus, finding the reason of the bottleneck is hard. More runtime performance measurements with different use cases are necessary to find weak points in the current system design and to better understand the black box. The small number of 20 samples per data point in a plot limits the statistical expressiveness of the results. The current implementation is the first iteration of the solution.

5.3.1 Restrictions

The jCloudScale Lambda framework explained here is a research prototype in an early stage. There are still conceptual and technical issues. Some issues originate in the framework, and others are caused by Amazon. The jCloudScale Lambda framework does not support all features from JCloudScale. Most of the missing features are not yet implemented, because the time for the thesis is limited and the focus was on other essential features. For some of the restrictions, a possible solution is explained in the section 6.2.

Conceptual Restrictions

Every technical approach has some benefits and drawbacks. Some conceptual issues can be fixed by extending and modifying the framework. However, for fixing other issues another conceptual fundamental is necessary. Not every use case can be implemented with this framework, because the chosen application model defines some conceptual restrictions. The conceptual restrictions are listed here:

- Currently there is no way to pass method arguments by-reference.
- Inner classes are not completely supported. The use of inner classes is generally not a problem, but an inner class should not have any method with a *CloudMethod* annotation, because jCloudScale Lambda has no access to non-public classes.
- There is no possibility to create a synchronized variable or method. A variable cannot be blocked until the execution of the method is complete. Therefore, a serverless function can request the current value of a variable and calculate a new one from this value. After the new value is calculated, the value is sent over the message queue to the client, where the updated value is saved. During the calculation of the new value, another serverless method requests the variable. The second method receives the old value and maybe works with an wrong variable value. There is no possibility to not send the current value of the variable to the second method, until the first method has modified the value. A developer must pay attention to such situations.
- A serverless function is always started over a HTTP request. The local application waits until it gets the response object. If the execution takes several minutes, the connection between the cloud and local application remains open throughout. May be it would be better if a method is started over a message queue. After the calculation in the cloud is done, the return value is also sent over the queue.
- The jCloudScale Lambda does not support any file system operations; the cloud application cannot request a file from a specific path. If the cloud uses the content of a file, the content should be saved as a serializable object, for example as a string.

Framework-based Technical Restrictions

jCloudScale Lambda uses third-party frameworks such as AspectJ or GSON. An external framework is not designed for the specific needs of jCloudScale Lambda. For example, the GSON parser does not like objects created at runtime with a generic type. Sometimes there was a trade-off between performance and user friendliness. If a developer has to write one more line, for example for a no-argument constructor, the priority to auto-generate the missing no-argument constructor is not the highest. The framework currently has the following restrictions:

- Java automatically creates an empty no-argument constructor if the developer does not define any constructor. JCloudScale Lambda requires the no-argument constructor to initialize an object. It is recommended that the no-argument constructor is empty, because otherwise the content of this constructor is always executed from the framework.
- Generics such as an ArrayList are supported from jCloudScale Lambda with by-value passing. By-reference passing with generics is so far not possible, because the GSON parser has some problems during the deserialization process with types defined at runtime, which includes generics.

- The access to by-reference passed variables is restricted, because the AspectJ framework requires a *this* context to set or get a variable. Currently it is impossible to get the value of a static variable with a *ByReference* annotation without being in the *this* context of the variable. The method *setStatic* in Listing 5.2 is not working, because there is not a *this* context. However, the method *setNonStatic* is working, because their is a *this* context. There is a context, because the method is non-static and can be only invoked with a context.

```
public class TestObj{
    @ByReference
    private static int c;


    @CloudMethod
    public static void setStatic ( int newC ){
        c = newC;
    }

    @CloudMethod
    public void setNonStatic ( int newC ){
        c = newC;
    }
}



@StartUp
public static void main ( String [] args ){
    TestObj obj = new TestObj();

    // Case 1 - not working
    TestObj.setStatic( 2 );

    // Case 2 - working
    obj.setNonStatic( 2 );
}
```

Listing 5.2: By-Reference variable and the *this* context

- If a class exists in the default package, jCloudScale Lambda throws a runtime error because classes in the default package cannot be imported by classes in packages. The dynamically created endpoint class from jCloudScale Lambda need access to these classes. To prevent this problem, it is never permissible to use the default package from Java in combination with jCloudScale Lambda.
- Functionalities such as lambda expression or stream collection types, which were introduced with Java 8, work, but the development focus was not on these features.
- If the developer has multiple applications that use the jCloudScale Lambda framework, each should have an separated SQS queue. If an application is already running and a second application with the same message queue is started, the started application purges the message queue. Possibly the already running application loses some messages and will never terminate.
- If the queue is initialized, the queue is purged and all messages are deleted. However, Amazon allows only to purge a queue once a minute. If Amazon blocks the purging request, the queue is normally not very efficient, and in extreme cases the application fails.

Amazon-based Technical Restrictions

Amazon has defined some limits for AWS. Most of the limits are fundamental restrictions of the framework and a developer cannot avoid them. Some of the service limits are not fix and it is possible to request a higher limit over the AWS Support Center[4] [AmazonLimit17a]. In the past, Amazon increased some limits step-by-step. The limits exist to ensure that enough resources are available for all customers [AmazonLimit17c]. The most important Amazon limits that have an impact on the framework are as follows:

- The local application currently communicates with the serverless function over the Amazon API Gateway. A group of endpoints, called API from Amazon, is currently restricted to 300 endpoints. So there are two possibilities to provide more endpoints for a single application. The communication works over the SQS, or the framework uses multiple API Gateways [AmazonLimit17a].
- At the moment there is a concurrent Lambda function execution limit of 1000 parallel executions per region [AmazonLimit17b].
- The API Gateway has a throttle rate of 10000s request per second [AmazonLimit17a].
- The maximum execution duration of a Lambda function is 300 seconds. If the execution takes longer, the execution is stopped by Amazon [AmazonLimit17d].
- A HTTP request over the Gateway API is stopped after 30 seconds [AmazonLimit17a]. The user gets a 500 error message. The started Lambda function is not stopped, but the return value cannot be passed back to the invoker.
- The Lambda function deployment package size is limited to 50 MB [AmazonLimit17d]. From these 50 MB, 15 MB are used by the jCloudScale Lambda framework with all required dependencies.
- The ephemeral disk capacity ("/tmp" space) is limited to 512 MB per invocation [AmazonLimit17d].
- In one Amazon region, the total size of all Lambda functions is limited to 75 GB [AmazonLimit17d].
- The memory allocation capacity of a serverless function can be chosen between a minimum of 128 MB and a maximum of 1536 MB in 64 MB increments [AmazonLimit17d].
- The SQS has a limited message size of 256 KB [AmazonLimit17e]. Currently it is impossible to use by-reference variables, which are larger. An option would be to store large variables in a S3 bucket and only send the reference to the file.

[4] https://console.aws.amazon.com/support/

Chapter 6

Closing Remarks

This chapter summarizes the contribution of the thesis. Moreover, future work, both conceptual and technical improvements, are briefly explained.

6.1 Conclusion

This thesis presented a Java-based middleware framework called jCloudScale Lambda, which transforms a regular application into a cloud-based one. The goal was to create a framework that is powerful, but that is also understandable for a developer. First, relevant background information on AOP, cloud computing and FaaS were summarized in Chapter 2. Next, the concept of JCloudScale was explained, from where the core idea for this project was adapted. Next related projects were surveyed that are also middleware software in a FaaS environment. The architecture, functionality and important implementation details of the framework were explained in Chapter 4. As a part of the thesis a first prototype of the explained system was implemented.

In order to illustrate the capabilities of jCloudScale Lambda, Chapter 5 presented a case study that examines how a regular application can be rewritten into a cloud-based application, and along with quantitative measurements. The restrictions of the current implementation, both conceptual and technical, were explained in Chapter 5.3.1.

It can be concluded that the pioneered development model from JCloudScale can be adapted for a FaaS-based service model. A developer can simply configure the cloud behavior of the application with annotations. Initially, some extra configuration in the configuration file is necessary, but only once. Except for some missing and not completely matured features, the framework offers a versatile and usable functionality. Some nice-to-have features, which will increase the usability, are explained in the next section. A developer does not require specific knowledge about the chosen cloud service. He can more focus on the business logic and does not have to write scaling behavior or handle the cloud deployment.

The framework, the code of the evaluation projects and the raw data of the evaluation are available as an open-source project on GitHub[1].

6.2 Future Work

This section summarizes conceptual and technical improvements for possible future work. There are many directions for this work.

[1] https://github.com/stewue/jCloudScale-Lambda/

6.2.1 Conceptual Improvements

Conceptual improvements are open points in the current system architecture, which extend or improve the jCloudScale Lambda framework and provide a developer with a larger set of functionalities.

- **Invocation of a serverless function:** Currently a serverless function is called with a HTTP request and the invoker waits until he gets the response. An alternative for longer executed serverless functions is necessary, where the invocation and response of the functions would use a message queue.
- **Debugging:** To debug a jCloudScale Lambda based application a developer requires more information about where the issue is. More research is required to understand how the debugging process can be improved in a serverless environment.
- **Data storage in the cloud:** Each by-reference variable value is saved in the local application. For some use cases it is useful that the variable value is saved in the cloud, for example in Amazon DynamoDB[2].
- **Isolation level and transaction for by-reference variable:** A by-reference variable has currently no mechanism for blocking a get or set request if a variable is used from another serverless instance. A transaction functionality would be helpful, to ensure that no one modifies the variable between a get and a set. Another feature could be that the developer can define the isolation level for each by-reference variable, like in Structured Query Language (SQL).

6.2.2 Technical Improvements

Technical improvements are possible extensions that enhance the technical implementation, or add nice-to-have features without changing the conceptual design. Most of the framework-based restrictions explained in section 5.3.1 can be implemented without great effort, and help the developers to write applications with jCloudScale Lambda more easily.

- **Serializer:** As experiment 5.9 has shown, the serializing and deserializing with GSON is a bottleneck in the current implementation. The performance could be improved by using another serializer such as Jackson[3] [Dreyfuss15].
- **Parallelize the startup process:** Currently the startup process is single-threaded. As shown in the startup time experiment series (Figure 5.5 and 5.6), the configuration part of the API Gateway and Lambda functions through the SDK is time-consuming. However, each single serverless function is independent, so multiple serverless functions can be configured at the same time.
- **Support of other cloud providers:** The framework currently supports only Amazon as cloud provider. The concept can be applied to other cloud providers such as Google or Microsoft.
- **Support of other programming languages:** An adaption of the concept to other programming languages that support AOP is possible.

[2]https://aws.amazon.com/dynamodb/
[3]https://github.com/FasterXML/jackson-core/

Appendix A

Acronyms

AOP Aspect Oriented Programming

API Application Programming Interface

AWS Amazon Web Service

CLI Command Line Interface

CPU Central Processing Unit

DTO Data Transfer Object

EC2 Elastic Compute Cloud

GUI Graphical User Interface

FaaS Function as a Service

HTTP Hypertext Transfer Protocol

IaaS Infrastructure as a Service

IDE Integrated Development Environment

JSON JavaScript Object Notation

JVM Java Virtual Machine

NIST National Institute of Standards and Technology

OOP Object Oriented Programming

PaaS Platform as a Service

RAM Random Access Memory

REST Representational State Transfer

S3 Simple Storage Service

SaaS Software as a Service

SDK Software Development Kit

SLA Service Level Agreement

SQL Structured Query Language

SQS Simple Queue Service

URL Uniform Resource Locator

VM Virtual Machine

XML Extensible Markup Language

Appendix B

Bibliography

All the following link and links in the footnotes were verified on 20 of July 2017.

[AmazonLimit17a] *Amazon Documentation: AWS Service Limits* https://docs.aws.amazon.com/general/latest/gr/ aws_service_limits.html

[AmazonLimit17b] *Amazon Documentation: Lambda Function Concurrent Executions* https://docs.aws.amazon.com/lambda/ latest/dg/concurrent-executions.html

[AmazonLimit17c] *Amazon News: AWS Lambda Raises Default Concurrent Execution Limit* https://aws.amazon.com/about-aws/whats-new/2017/05/aws-lambda-raises-default-concurrent-execution-limit/

[AmazonLimit17d] *Amazon Documentation: AWS Lambda Limits* https://docs.aws.amazon.com/lambda/latest/dg/limits.html

[AmazonLimit17e] *Amazon Documentation: Limits Related to Messages (SQS Limits)* https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/limits-messages.html

[AmazonSLA17] *Amazon S3 Service Level Agreement* https://aws.amazon.com/s3/sla/

[AmazonXRay17] *Amazon News: AWS X-Ray Now Supports Tracing for AWS Lambda (Preview)* https://aws.amazon.com/de/about-aws/whats-new/2017/04/aws-x-ray-now-supports-tracing-for-aws-lambda-preview/

[Armbrust09] Armbrust, Michael and Fox, Armando and Griffith, Rean and Joseph, Anthony D. and Katz, Randy H. and Konwinski, Andrew and Lee, Gunho and Patterson, David A. and Rabkin, Ariel and Stoica, Ion and Zaharia, Matei *Above the clouds: A Berkeley view of cloud computing.* Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS 28.13 (2009): 2009. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html

[Armbrust10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, Dav id Patterson, Ariel Rabkin, Ion Stoica and Matei Zaharia *A View of Cloud Computing* Communications of the ACM 53.4 (2010): 50-58. doi:10.1145/1721654.1721672

[AspectJ17] *AspectJ Join Points: Language Semantics* https://www.eclipse.org/aspectj/doc/released/progguide/semantics-joinPoints.html

[Avram16] Abel Avram *FaaS, PaaS, and the Benefits of the Serverless Architecture* 2016 https://www.infoq.com/news/2016/06/faas-serverless-architecture

[Bernstein14] Bernstein, David. *Containers and cloud: From lxc to docker to kubernetes.* IEEE Cloud Computing 1.3 (2014): 81-84. doi:10.1109/MCC.2014.51

[Bhardwaj10] Bhardwaj, Sushil, Leena Jain, and Sandeep Jain. *Cloud computing: A study of infrastructure as a service (IAAS).* International Journal of engineering and information Technology 2.1 (2010): 60-63.

[Brown16] Ryan S. Brown *Going Serverless Without The Serverless Framework* Wed, Sep 14, 2016 https://serverlesscode.com/post/serverless-without-the-framework/

[Bruneo14] Dario Bruneo, Thomas Fritz, Sharon Keidar-Barner, Philipp Leitner, Federica Longo, Clarissa Marquezan, Andreas Metzger, Klaus Pohl, Antonio Puliafito, Andrei Roth, Eliot Salant, Itai Segall, Massimo Villari, Yaron Wolfsthal, Chris Woods *Cloudwave: where adaptive cloud management meets devops.* In Computers and Communication (ISCC), 2014 IEEE Symposium on, pages 1–6. IEEE, 2014. doi:10.1109/ISCC.2014.6912638

[Buxmann08] P Buxmann, T Hess, S Lehmann *Software as a Service* 2008 doi:10.1007/s11576-008-0095-0

[Cito15] J. Cito, P. Leitner, T. Fritz, and H. C. Gall. *The making of cloud applications: An empirical study on software development for the cloud.* In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pages 393–403, New York, NY, USA, 2015. ACM. doi:10.1145/2786805.2786826

[Dillon10] Tharam Dillon, Chen Wu and Elizabeth Chang *Cloud Computing: Issues and Challenges* In 2010 24th IEEE International Conference on Advanced Information Networking and Applications doi:10.1109/AINA.2010.187

[Docker17] *What is Docker?* https://www.docker.com/what-docker

[Dragoni16] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, Larisa Safina *Microservices: yesterday, today, and tomorrow* 2016 arXiv:1606.04036v4

[Dragoni17] Nicola Dragoni, Ivan Lanese, Stephan Thordal Larsen, Manuel Mazzara, Ruslan Mustafin, Larisa Safina *Microservices: How To Make Your Application Scale* 2017 arXiv:1702.07149

[Dreyfuss15] Josh Dreyfuss *The Ultimate JSON Library: JSON.simple vs GSON vs Jackson vs JSONP* May 28, 2015 https://blog.takipi.com/the-ultimate-json-library-json-simple-vs-gson-vs-jackson-vs-json/

[Dua14] Dua, Rajdeep, A. Reddy Raja, and Dharmesh Kakadia. *Virtualization vs containerization to support paas.* Cloud Engineering (IC2E), 2014 IEEE International Conference on. IEEE, 2014. doi:10.1109/IC2E.2014.41

[Elrad01] Elrad, Tzilla, Robert E. Filman, and Atef Bader. *Aspect-oriented programming: Introduction.* Communications of the ACM 44.10 (2001): 29-32. doi:10.1145/383845.383853

[Foster08] Foster, I., Zhao, Y., Raicu, I., & Lu, S. (2008, November). *Cloud computing and grid computing 360-degree compared.* In Grid Computing Environments Workshop, 2008. GCE'08 (pp. 1-10). Ieee. doi:10.1109/GCE.2008.4738445

[Galante12] Galante, Guilherme, and Luis Carlos E. de Bona. *A survey on cloud computing elasticity.* Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on. IEEE, 2012. doi:10.1109/UCC.2012.30

[Geelan09] Geelan, Jeremy. *Twenty one experts define cloud computing.* Cloud Computing Journal 4 (2009): 1-5.

[Grobauer11] Grobauer, Bernd, Tobias Walloschek, and Elmar Stocker. *Understanding cloud computing vulnerabilities.* IEEE Security & Privacy 9.2 (2011): 50-57. doi:10.1109/MSP.2010.115

[Haeberlen12] Thomas Haeberlen, Lionel Dupré *Cloud Computing: Benefits, risks and recommendations for information security* enisa (European Network and Information Security Agency) https://resilience.enisa.europa.eu/cloud-security-and-resilience/publications/cloud-computing-benefits-risks-and-recommendations-for-information-security

[Hendrickson16] Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A. C., & Arpaci-Dusseau, R. H. (2016). *Serverless computation with OpenLambda.* Elastic, 60, 80. https://www.usenix.org/node/196323

[IBM13] Schouten, Edwin. *IBM® SmartCloud® Essentials.* Packt Publishing Ltd, 2013. ISBN-13: 978-1782170648

[Jackson17] Joab Jackson *With PyWren, AWS Lambda Finds an Unexpected Market in Scientific Computing* 16 Feb 2017 https://thenewstack.io/aws-lambda-finds-unexpected-market-scientific-computing/

[JCloudScale17] *JCloudScale - Documentation on GitHub* https://github.com/xLeitix/ jcloud-scale/blob/master/docs/Documentation.md

[Johnson05] Johnson, R., & Hoeller, J. (2005). *Expert One on One J2EE development without EJB.* John Wiley & Sons. ISBN-13: 978-0764558313

[Jonas16] Eric Jonas *Microservices and Teraflops* 2016, blog post on the PyWren website https://pywren.io/pywren.html

[Jonas17] Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht *Occupy the Cloud: Distributed Computing for the 99%* 2017 preprint arXiv:1702.04024

[Kiczales01] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., & Griswold, W. G. (2001, June). *An overview of AspectJ.* In European Conference on Object-Oriented Programming (pp. 327-354). Springer Berlin Heidelberg. doi:10.1007/3-540-45337-7_18

[Kiczales97] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. M., & Irwin, J. (1997). *Aspect-oriented programming.* ECOOP'97—Object-oriented programming, 220-242. doi:10.1007/BFb0053381

[Laddad03] Laddad, R. (2003). *Aspect-oriented programming will improve quality.* IEEE software, 20(6), 90-91. doi:10.1109/MS.2003.1241372

[Lambda14] *Amazon release notes on 2014-11-13* https://aws.amazon.com/releasenotes/AWS-Lambda/8269001345899110

[LambdaCPU14] *Amazon Forum: Lambda CPU relative to which instance type?* https://forums.aws.amazon.com/ message.jspa?messageID=614558

[LambdaPricing17] *Lambda Pricing Details* https://aws.amazon.com/lambda/pricing/

[Lawton08] Lawton, George. *Developing software online with platform-as-a-service technology.* Computer 41.6 (2008). doi:10.1109/MC.2008.185

[Leitner12] Philipp Leitner, Benjamin Satzger, Waldemar Hummer, Christian Inzinger, and Schahram Dustdar. 2012. *CloudScale: A novel middleware for building transparently scaling cloud applications.* In 27th ACM Symposium on Applied Computing (SAC'12). 434–440. doi:10.1145/2245276.2245360

[Lippert00] Lippert, M., & Lopes, C. V. (2000, June). *A study on exception detection and handling using aspect-oriented programming.* In Proceedings of the 22nd international conference on Software engineering (pp. 418-427). ACM. doi:10.1145/337180.337229

[Malawski16] Malawski, M. *Towards Serverless Execution of Scientific Workflows–HyperFlow Case Study.* In 11th Workshop on Workflows in Support of Large-Scale Science (WORKS@ SC), volume CEUR-WS 1800 of CEUR Workshop Proceedings (pp. 25-33). https://www.researchgate.net/publication/314950511_Towards_Serverless_Execution_of_Scientific_Workflows_-_HyperFlow_Case_Study

[Mell11] Peter Mell, Timothy Grance *The NIST Definition of Cloud Computing* National Institute of Standards and Technology (NIST) doi:10.6028/NIST.SP.800-145

[Microsoft17] *Solution Development Fundamentals patterns & practices Developer Center: Crosscutting Concerns* https://msdn.microsoft.com/en-us/library/ee658105.aspx

[Miller15] Ron Miller *AWS Lambda Makes Serverless Applications A Reality* TechCrunch 2015 https://techcrunch.com/2015/11/24/aws-lamda-makes-serverless-applications-a-reality/

[Ramgovind10] Ramgovind S, Eloff MM, Smith E *The Management of Security in Cloud Computing* published in Information Security for South Africa (ISSA), 2010 doi:10.1109/ISSA.2010.5588290

[Rimal09] Rimal, Bhaskar Prasad, Eunmi Choi, and Ian Lumb *A taxonomy and survey of cloud computing systems.* 2009 Fifth International Joint Conference on INC, IMS and IDC doi:10.1109/NCM.2009.218

[Roberts16] Mike Roberts, Martin Fowler *Serverless Architectures* 2016 https://martinfowler.com/articles/serverless.html

[Sareen13] Sareen, Pankaj *Cloud computing: types, architecture, applications, concerns, virtualization and role of it governance in cloud.* International Journal of Advanced Research in Computer Science and Software Engineering, 2013, 3. Jg., Nr. 3. ISSN: 2277 128X

[Serverless17a] *What is the Serverless Framework?* https://serverless.com/framework/

[Serverless17b] *Serverless Framework on Github - Documentation* https://github.com/serverless/serverless

[Serverless17c] *Serverless Framework - AWS Documentation* https://serverless.com/framework/docs/providers/aws/

[Sotomayor09] B Sotomayor, RS Montero, IM Llorente, I Foster *Virtual Infrastructure Management in Private and Hybrid Clouds* published in IEEE Internet Computing (Volume: 13, Issue: 5, Sept.-Oct. 2009) doi:10.1109/MIC.2009.119

[Spillner16a] Josef Spillner, Serhii Dorodko *FaaS: Function hosting services and their technical characteristics* https://blog.zhaw.ch/icclab/faas-function-hosting-services-and-their-technical-characteristics/

[Spillner16b] Josef Spillner, Serhii Dorodko *Introducing Podilizer: Automated Java code translator for AWS Lambda* https://blog.zhaw.ch/icclab/introducing-podilizer-automated-java-code-translator-for-aws-lambda/

[Spillner17a] Josef Spillner, Serhii Dorodko *Java Code Analysis and Transformation into AWS Lambda Functions* 2017 arXiv:1702.05510

[Spillner17b] Josef Spillner *Snafu: Function-as-a-Service (FaaS) Runtime Design and Implementation* 2017 arXiv:1703.07562

[Vaquero08] Vaquero, L. M., Rodero-Merino, L., Caceres, J., & Lindner, M. *A break in the clouds: towards a cloud definition.* ACM SIGCOMM Computer Communication Review 39.1 (2008): 50-55. doi:10.1145/1496091.1496100

[Villamizar16] Villamizar, Mario and Garces, Oscar and Ochoa, Lina and Castro, Harold and Salamanca, Lorena and Verano, Mauricio and Casallas, Rubby and Gil, Santiago and Valencia, Carlos and Zambrano, Angee & Mery Lang *Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures.* Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on. IEEE, 2016. doi:10.1109/CCGrid.2016.37

[Yan16] Mengting Yan, Paul Castro, Perry Cheng, Vatche Ishakian *Building a Chatbot with Serverless Computing* 2016 doi:10.1145/3007203.3007217

[Zabolotnyi15] Rostyslav Zabolotnyi, Philipp Leitner,Waldemar Hummer, and Schahram Dustdar. 2015. *JCloudScale: Closing the gap between IaaS and PaaS.* ACM Trans. Internet Technol. 15, 3, Article 10 (July 2015), 20 pages. doi:10.1145/2792980

[Zimki06] Brady Forrest *Zimki, hosted JavaScript environment* http://radar.oreilly.com/2006/09/zimki-hosted-javascript-enviro.html

[Zhang10] Zhang, Qi, Lu Cheng, and Raouf Boutaba. *Cloud computing: state-of-the-art and research challenges.* Journal of Internet services and applications 1.1 (2010): 7-18. doi:10.1007/s13174-010-0007-6

Appendix C

CD-ROM Content

- **bachelor_thesis.pdf**
  Bachelor Thesis as PDF
- **abstract.txt**
  Abstract in English
- **zusfsg.txt**
  Abstract in German
- **git/**
  Copy of th git repository, which contains the source code, evaluation data etc.