Resolving Strategies for A Serverless Computing Service Timeout Limitation

Lijuan Ye

Delft University of Technology

Resolving Strategies for A Serverless Computing Service Timeout Limitation

Master's Thesis in Computer Science
EIT Digital Cloud Computing and Services

Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Lijuan Ye

23rd August 2021

Author
Lijuan Ye

Title
Resolving Strategies for A Serverless Computing Service Timeout Limitation

MSc presentation
27th August 2021

Graduation Committee

Prof. dr. ir. D. H. J. EPEMA	Delft University of Technology
Michael Holopainen	CGI Finland
Dr. J. S. Rellermeyer	Delft University of Technology
Dr. A. Katsifodimos	Delft University of Technology

Abstract

With the fast maturing of cloud-based technologies, a variety of cloud services proposed by numerous cloud service vendors to help businesses in a range of scales move to cloud solutions easily. In the cloud computing market, the services are categorized with different service models. Each service model has its own distinct characteristics, advantages, and disadvantages. Serverless or Function as a Service (FaaS) is a newborn service model type and it is leading a trend now.

Amazon Web Service(AWS)is one of the most advanced and eminent cloud service providers, which has produced plenty of powerful cloud services and always advances with the times. AWS Lambda is a pioneer of serverless services announced by AWS in 2014 and it has become one of the most popular services in the serverless field. However, the quotas, for example, function timeout and deployment package size, originally settled by AWS Lambda gradually become barriers during the phase when Lambda is employed in various application scenarios. A similar problem exists in other serverless service providers as well. This work will only focus on AWS Lambda time limitation study and other services studies or detailed comparison in future work.

This work consists of two main goals, one is to survey the state-of-the-art critical constraints of AWS Lambda service, including the possible reasons why they were limited in that way, the impact they may cause, and the feasible solutions pointing to different occasions. The other one is to present a resolving strategy for applications like our use case, which meet the timeout limitation of AWS Lambda for long-running tasks and require to decouple the continuous delivery pipeline from the application.

We firstly show the research result about serverless computing and AWS main limitations, such as deployment package size and function timeout. Then it is followed by describing our resolving strategy for the timeout limitation. We assess our strategy by experimenting with one use case scenario. The resolving strategy design, experiment implementation process, experiment result analysis, and considerable alternatives of the design will also be elucidated. We end with a summary of this work and an assumption of what can be explored in the future work.

Preface

I am a student studying in the EIT Digital MSc program Cloud Computing and Service and this major interests me a lot as it not only focuses on the technical knowledge about ”Cloud Computing” but also points out the importance of ”Service”. There is no doubt that the development of ”Cloud Computing” technology provides a lot of benefits, however, ”Service” plays a more important role to let more people get benefits from that and that is why SaaS(Software as a Service), PaaS(Platform as a Service) and IaaS(Infrastructure as a Service) are becoming more and more popular. Amazon is the pioneer in the field of providing service of cloud computing and it has created a variety of cloud computing services and solutions to meet the different needs of its users. Therefore, when my supervisor in CGI Finland in Helsinki, the company I did my internship, mentioned that there were some limitations in Amazon Lambda functions(a serverless compute service), which we could try to study and find a resolving strategy for that based on our scenario, I was so excited about working on this topic.

Last year is a special year for not only me but also the whole world. People are facing a huge threat from Coronavirus (COVID-19) and we were asked to work remotely at home. Long-time nervousness and too much stress I put on myself made me lost in depression. Thus, I would like to express my sincerest gratitude to my professor D.H.J Epema in the Delft University of Technology and my supervisor Michael Holopainen in CGI to give me great support in both my work and life. I cannot finish this work without their guidance and help. I also want to thank my coordinators, Inge Grootjansat the Delft University of Technology and Aino Roms at Aalto University. They tried their best to offer all their help and show me their understandings and encouragement.
Last but not the least, I would like to thank my friends Xuyang Zhang, Ling Sun, Paras Kumar, Oleg Vlasovets, Andrea Corsini, Pei Zhang, Jing Yan, and my parents who accompanied me to survive those tough days.

Lijuan Ye

Delft, The Netherlands
23rd August 2021

Contents

Preface v

1 Introduction 1
1.1 Problem Statement 2
1.2 Research Approach 3
1.3 Thesis Outline and Contributions 4

2 Background and Concepts 5
2.1 Cloud Computing Service Models 5
2.1.1 State-of-the-art Service Models 5
2.1.2 Cloud Service Vendors 15
2.2 AWS Cloud Services 16
2.2.1 AWS Lambda 17
2.2.2 Amazon Elastic Compute Cloud (EC2) 19
2.2.3 AWS Step Functions 20
2.2.4 AWS Elastic Beanstalk 22
2.2.5 AWS CodeCommit 22
2.2.6 AWS CodePipeline 23
2.2.7 Amazon EventBridge 24
2.3 Related Work 25
2.3.1 Cloud Computing Security Problems 25
2.3.2 Serverless Computing Analysis 26

3 Analysis of the AWS Lambda Quotas Effect 29
3.1 Serverless Platform Constraints 29
3.2 AWS Lambda Main Quotas Analysis 32
3.3 Strategies In Contemporary Work 36

4 Strategies for AWS Lambda Timeout Limit 39
4.1 Solution Design 39
4.1.1 Solution for Time Limitation of Long-Running Applications 40
4.1.2 Decoupling the Pipeline and Application Execution Design 42
4.2 AWSMA Illustration 44
4.2.1 Application Description 46

4.2.2 Problem Identification 46
4.3 Experimental Setup 47
4.4 Result Evaluation 50
4.4.1 Timeout Problem Resolving 52
4.4.2 Cost Efficiency 52
4.4.3 Insights from Monitoring Metrics 52
4.5 Alternative Solutions 53

5 Conclusion and Future Work **59**
5.1 Conclusions 59
5.2 Future Work 61

A AWS Lambda Quotas **69**

Chapter 1

Introduction

Twenty years ago, the word ”cloud” in the IT field was still an ambiguous concept and people would like to understand it by referring to the word ”Internet” in most cases. It was mainly used to describe networks of computing equipment at that moment, like the original ARPANET (Advanced Research Projects Agency Network) and the CSNET (Computer Science Network). Some big companies have also used it occasionally in the description of their technologies, like Apple. However, when talking about ”cloud” in recent years, more and more people will firstly associate it with cloud-based technologies like cloud computing and cloud storage. Cloud computing is one of the most critical concepts in cloud technologies and it is one of the hottest technologies in the past few years.

The popularity of ”cloud computing” should own to the development of different service models provided by different cloud computing providers. The three well-known standard cloud computing service models are Software as a Service (Saas), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). In addition, many new service models appear according to the demands of the industry market and the evolution of the cloud computing technologies, such as Serverless Computing, which is also known as Function as a Service (FaaS), Mobile ”backend” as a Service (MBaaS), Network as a Service (NaaS), Storage as a Service (StaaS) and Monitoring as a Service (MaaS).

Serverless Computing is one of the most popular cloud services in recent years as it releases developers from worrying about the arrangement of the resources for their applications. The applications can run in an event-driven manner and the users only pay for what they use. This service model brings people a new way to organize their application workflow but maybe also some concerns. Amazon Web Service (AWS) Lambda is a kind of Serverless Computing service or Function as a Service (FaaS). The main goal of this solution is to simplify the management processes of the applications that the customers hope to deploy on the cloud. Although the first release of the AWS Lambda was in 2014 and the applications of

Serverless Computing service model, which was even earlier first implemented by Zimki in 2006, the Serverless Computing technologies are still not ripened. There are still many challenges for those serverless services. In this work, we will only pay our attention to the study of a few AWS Lambda main limitations, involving **Deployment Package Size**, **Function Memory Maximum**, **Function Timeout**, and **Temporary Storage**. Meanwhile, we will propose a resolving strategy for the timeout limitation and show the experiment based on our use case scenario.

1.1 Problem Statement

Many serverless vendors provide Serverless Computing services, for example, Google App Engine, which was released by Google in 2008, PiCloud, which was released in 2010, and the more popular ones in the last few years, Amazon Web Service (AWS) Lambda, Google Cloud Platform, and Microsoft Azure. The different platforms may meet different challenges, in this thesis work, we choose the AWS Lambda platform to start the study since it provides all the typical features of Serverless Computing.

Although we limit our study scope only in AWS Lambda limitations, there is still a problem about which limitation we should give a higher weight and go into an in-depth analysis. In general, all the limitations have their effect in different aspects, however, they could also have some inner relations with each other. Thus, choosing the main limitations of AWS Lambda to study and how to analyze them becomes the first challenge of this thesis work.

The following challenge we would like to solve is to propose a kind of resolving strategy for the Lambda timeout limitation. We meet this limitation in our use case when we try to move our application onto the AWS cloud platform. There are many existing guides for dealing with the function timeout problem. However, we notice that the solution is missing for applications with one or more inseparable long-running tasks that will exceed the 900 seconds quota. The inseparable long-running task denotes the unit function that cannot be divided into smaller functions anymore. We implement our solution into practice as an experimental approach after considering the feasibility and reliability.

In this thesis work, we will try to find out the answers to the following three research questions:

1. **RQ1:** What are the most pivotal limitations of AWS Lambda and the effect that they may cause in practice? What could be the possible reasons and solutions?

2. **RQ2:** How can we resolve the AWS Lambda timeout bottleneck when de-

ploying long-running applications with serverless or other AWS cloud services?

3. **RQ3:** From the DevOps perspective, how can we decouple the continuous delivery (CD) pipeline from the application execution workflow with the help of AWS Lambda despite its timeout constrain?

All of the above research questions were inspired by real scenarios. Amazon has listed all their quotas of the Lambda service but we hope to dig out the effect that those most decisive quotas would cause on the applications in practice and try to analyze the possible reasons which lead to the limitations and the possible solutions. RQ1 will put more effort into analyzing the main effect caused by every single main quota we concentrate on but will also discuss a little bit about their relations to some extent when needed. RQ2 is only addressed on resolving the timeout limitation for applications that own indivisible long-running tasks like our example application. We hope this solution strategy can be inspiring for more realistic situations. RQ3 is an extension of RQ2. It is a kind of strategy to help with DevOps by using AWS Lambda despite its timeout limitation.

1.2 Research Approach

To obtain the satisfying results for all of those research questions mentioned above, different research approaches are needed for each of the research processes.

The purpose of the first research question is to provide developers or other users who want to select the AWS Lambda platform to deploy their applications a better view of how the "quotas" of Lambda service will limit or affect the deployment or performance of their projects. As the "quotas" were set on different computing resources, we cannot just use one standard method to analyze them but we are trying to show their influence by researching the real-world practices and the research would also take the possible cause and solutions into consideration.

The second and the third research questions will be answered based on the experimental experience of our specific solution for the timeout limitation of the AWS Lambda service. Hence, first of all, we need to design the solution. Then, implementing it into practice with AWS Lambda and the other AWS services we need. Next, testing and gathering the useful information for the next analyzing and evaluation step. Finally, giving the evaluated result and the conclusion for the research question. The main factors we would like to consider for the evaluation are feasibility, reliability, resource utilization, and cost-efficiency.

1.3 Thesis Outline and Contributions

This thesis is composed of five chapters which are described as following: In Chapter 2, the background information about Cloud Computing service models, cloud service providers, concepts related to AWS Lambda service, and the other AWS services that we need for our solution implementation and some other related work. In Chapter 3, we will provide our study result of the AWS Lambda main quotas that we found out including their possible effect, settled reason, and potential solutions. Then, Chapter 4 presents a resolving solution we propose for reducing the negative effect caused by the AWS Lambda function timeout limitation for applications with inextricable long-running functions. Finally, we summarize the consequence we find out through this research work and offer our view about the future work in Chapter 5.

Our main contributions could be summed up as follows:

- We bring an overview of the state-of-the-art service models and their vendors. (Chapter 2)
- We provide a comprehensive and in-depth analysis about the result could be lead to by the limitations of AWS Lambda (Chapter 3)
- We offer the potential resolving ideas or directions for some limitations (Chapter 3).
- We propose an inspiring strategy to decrease the unsatisfactory influence caused by the Lambda timeout quota while processing a long-running task. (Chapter 4).
- We show an approach to separate the continuous delivery (CD) pipeline and the application execution process despite the AWS Lambda timeout limit (Chapter 4).
- We evaluate our strategy from both the technical side and service side and bring a conclusion at the end (Chapter 4).

Chapter 2

Background and Concepts

This chapter introduces the background knowledge and concepts which are related to this thesis work. In Section 2.1, we provide a general introduction of Cloud Computing and more details about its service models including both the frequently applied ones and those just emerged in recent years. This section also contains a discussion about the vendors who provide different kinds of Cloud Computing services. Then, Section 2.2 describes the definitions, applications and main concepts of the AWS cloud services that participate in our solution explained in Chapter 4 . Finally, Section 2.3 discusses related work in the scope of the intersection of Cloud Computing and AWS Lambda service, such as computing performance analysis.

2.1 Cloud Computing Service Models

Cloud Computing is defined as *"A model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."* by the National Institute of Standards and Technology (NIST). The NIST also states this could model consists of five essential characteristics, three service models, and four deployment models.[43] In this section, we only focus on the description of the service models. Several state-of-the-art service models are introduced in Section 2.1.1 and the vendors who provide cloud services by those service models are presented and compared in Section 2.1.2.

2.1.1 State-of-the-art Service Models

The National Institute of Standards and Technology (NIST) determines three basic service models of Cloud Computing, which are Software as a Service (SaaS), Platform as a Service (Paas), and Infrastructure as a Service (IaaS). With the development of Cloud Computing technologies, some advanced service models are proposed and become more and more popular in the last few years, for example,

Backend as a Service (BaaS) and Serverless Computing.

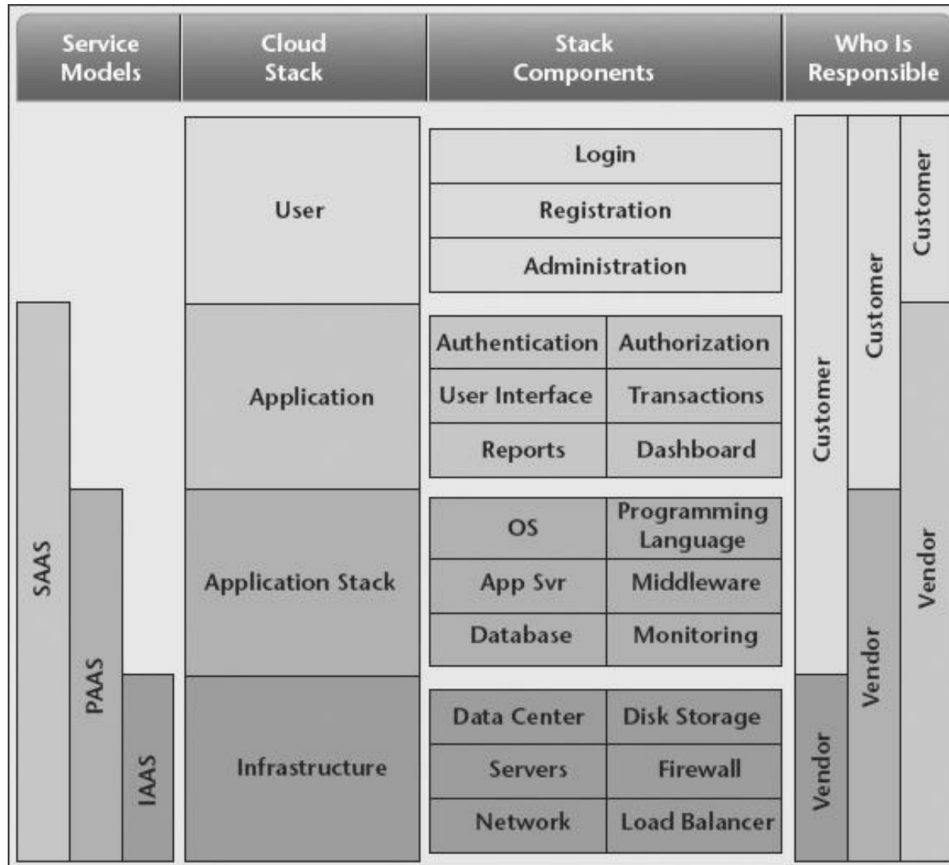


Figure 2.1: Cloud Stack and Cloud Computing Service Models.[35]

The three service models, Software as a Service (SaaS), Platform as a Service (Paas), and Infrastructure as a Service (IaaS), are still the most commonly recognized service models in the Cloud Computing field so far. Each of them offers the matched abstraction and automation services for the tasks of different levels in the cloud stack.

1. **Infrastructure as a Service (IaaS)**: The definition of this service model given by The National Institute of Standards and Technology (NIST) is "The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with

the possible exception of limited user-specific application configuration settings." The cloud infrastructure usually refers to the bottom layer in the cloud stack, which is constituted by hardware and software components for building a cloud. The hardware components could also be viewed as a significant physical layer typically including server, storage, and network components to support the upper cloud services. The software components that are deployed across the physical layer could be seen as an abstraction layer, which abstracts the resources and exports their interface to users. The abstraction layer is normally considered sitting above the physical layer. The infrastructure as a Service (IaaS) model makes it available for customers to use the IT infrastructures without managing and maintaining them by themselves. The service could be offered on-demand and can be reached by calling an application program interface (API), a web-based management console, or a graphical interface. The cost of the service is calculated by the amount of the required resources.[43]

In conclusion, the Infrastructure as a Service model (IaaS) involves the following characteristics and components:

- Computer hardware.
- Computer network.
- Internet connectivity.
- Desktop virtualization.
- Service level agreements.

The benefits offered by IaaS are (1) Efficient IT services with a prepared environment and customized demand. (2) Managing tasks automatically. (3) Maintenance could be promptly reachable through the internet. (4) Dynamic scaling. (5) Multiple virtual instances can be launched per command. (6) Reduce the cost of obtaining and maintaining the hardware computing resources. However, the security issues of IaaS need to be concerned at the same time. Virtualization manager security and the reliability of data stored in the hardware of the IaaS providers can be viewed as two main factors causing security holes. Deployment models of cloud computing also show their impact on this aspect to some extent. Since different deployment models decide the level of sharing the cloud infrastructure, which possibly affects the risk rates of incurring damages. For example, the public cloud is supposed to be more riskier than the private cloud.[46, 18, 58]

The technical challenges when dealing with the security issues are operational trust modes, resource sharing, new attack strategies, and digital forensics. The IaaS providers help to maintain the infrastructure layer of the customers' systems, which means all the operations made by the consumers are nearly

transparent to the providers. The operational levels of trust granted to the providers have an influence on obtaining the trust in the resource providers and risk assessment. Data leakage is proved to be possible to occur with resource sharing and some other risks could also result from this challenge. Some new attack strategies appear targeted to the cloud model and they are also very important when preparing the security plan. Digital forensics is also more challenging in the cloud environment compared to the traditional approaches of building the systems. The main issues comprise: (1)The ephemeral nature of cloud resources. (2)Seizing a "system" for examination. [33]

2. **Platform as a Service (PaaS)**: NIST characterized this service model as "The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment." From the perspective of the cloud stack, the application is the upper level of the infrastructure, and Platform as a Service (PaaS) abstracts most of the application-stack-level utilities as service in the cloud environment. This service model is supposed to deliver a computing platform for developing and provisioning cloud applications which are normally made up of hardware and a certain amount of software required for the complete life cycle of the application building and delivering. Hence, the user group of this service model is developers looking for setting up and run cloud applications for a specific platform. With the support from the service vendors, the developers are released from a huge number of codes to handle the scalability and elasticity of the application, for example, database scaling, application caching, and asynchronous messaging. In this case, consumers of PaaS have little-to-no control over the cloud infrastructure but only a few vital configuration settings. They access all the services and tools through APIs offered by the vendors. PaaS providers normally will charge for the service on the basis of per-user or time.[43]

   Basically, the main traits of the Platform as a Service model (Paas) are:

   - Auto-provisioning of the underlying infrastructure.
   - Scalability and elasticity.
   - A suite of tools for simplifying and speed the application development and management.
   - Interface to integrate with other infrastructure modules like LDAP(Lightweight Directory Access Protocol), database, and web service.

- Accessing API provided.
- Providing log, analysis report, and code instrumentation.
- Allowing many concurrent users to use the platform service.
- Security and redundancy.

By establishing on the top of the IaaS, PaaS gained plenty of similar advantages, for instance, utility computing, hardware virtualization, and dynamic resource allocation. In addition, PaaS has a better performance on cost efficiency and saving time and energy, which helps to speed up to the market. With the ability of easy-to-access a huge third-party software solution resource pool, product builders could offer to fail over, high service level agreements (SLAs). Considering the perspective of the developers, PaaS comes up with more options for them to build their development stack. It also increases the convenience and decreases the restriction for developers by supporting multiple programming languages. Nevertheless, PaaS also has its own challenges and security issues to be examined for the service model users.[30, 35]

There are two main challenges that could be in the way for the users of choosing PaaS for their development stack. One challenge is lacking a standard guideline of features, development tools, database types, software, APIs, middleware, or languages for all Paas providers to refer to. Hence, the developers may be struggling with choosing the most suitable one or how to switch from one to the other when it is needed. The other challenge is the rapid growth of this service model and the stability of the providers. As it is uncertain for the developers that paying effort on learning to build their products on top of a new specific platform is worthwhile.[30]

Those dares might attenuate overtimes with the progress on both providers' and users' sides. The security issues exist in PaaS as well. Since the Service-oriented Architecture (SOA) model is the underlying of PaaS, the security issues live in the SOA realm, like Man-in-the-middle attacks, Dictionary attacks, XML-related attacks, DOS attacks, and Input validation related attacks, are also inherited by PaaS. Moreover, Web Service (WS) Security, mutual authentication, and authorization security issues become more complex as it is a shared responsibility among cloud providers, service providers, and consumers. API Security is another concern in PaaS security issues. As mentioned above, API is an important feature of PaaS so that it is crucial to equip the APIs with security controls and standards.[16]

3. **Software as a Service (SaaS)**: Software as a service (SaaS) stands on the top level of the stack. The description from NIST is " The capability provided

to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings." The service SaaS-delivered to its consumer is a complete software or application which is hosted and maintained by the service provider. The end-users only have limited permits for accessing and authority of configurations. The distribution of this service model is always through a thin client interface like a web browser or a program interface. SaaS gains its popularity in recent years owing to the increasing universality of the underlying technologies which support its provision and mature and new development methods of Service-oriented Architecture (SOA), such as Ajax. SaaS is usually associated with the Application Service Provider (ASP) and on-demand computing delivery models. International Data Corporation (IDC) determines two marginally different delivery models for SaaS as the hosted application model and the software development model. Some prevalent SaaS applications are Customer Relationship Management (CRM), Enterprise Resource Planning (ERP), payroll software, and accounting software.[43]

To sum up, SaaS has the following main characteristics:

- Application delivery.
- Accessing through a thin client or program interface.
- Limited authority for configuration.
- Unaware of the underlying infrastructures

The benefits of manipulating the SaaS service model are obvious in optimizing the operation efficiency and reducing the budget. According to the main characteristics of SaaS, the administration, scalability, availability, and maintenance becomes much easier and it improves the speed of the development process and the production performance. Since the provider will deliver the same version of the software to all its users, it declines the difficulty compatibility and collaboration. The benefit of initial investment cost saving on software, hardware, and the staff was emphasized by Microsoft Corporation. Up to 64% saving over 4 years could be achieved by SaaS solutions for a comparable on proposition solution.[30, 15]

The challenging part for SaaS mainly focuses on the security problems, although it mentioned that SaaS provides also security benefits. Compared to the positive side that SaaS brings into the security domain, the obstructive

influence may need more concern when enterprises consider choosing it. As stated in [29], customer or tenant can have greater security control over more resources as one move from SaaS to PaaS and again from PaaS to IaaS service model according to the NIST cloud model. This is also the reason why most enterprises still feel uncomfortable with the SaaS model. Web application vulnerability security issues must be faced by SaaS, besides that SaaS model also inherits the security concerns discussed in the above two models because of its development stack position. Another factor could have effects on data security of SaaS model is IDM, which helps the enterprises to build enforce control over their own data processed in the cloud. But there are also security issues existing in the Security Assertion Markup Language (SAML), which is the industry standard for IDM.[30, 58, 56, 41]

With the wild adoption of cloud computing, security problems should attract more attention, which not only exits in the service models mentioned above but also the whole cloud environment, showing in Figure 2.2. We will not discuss this in the depth of this topic in this paper.

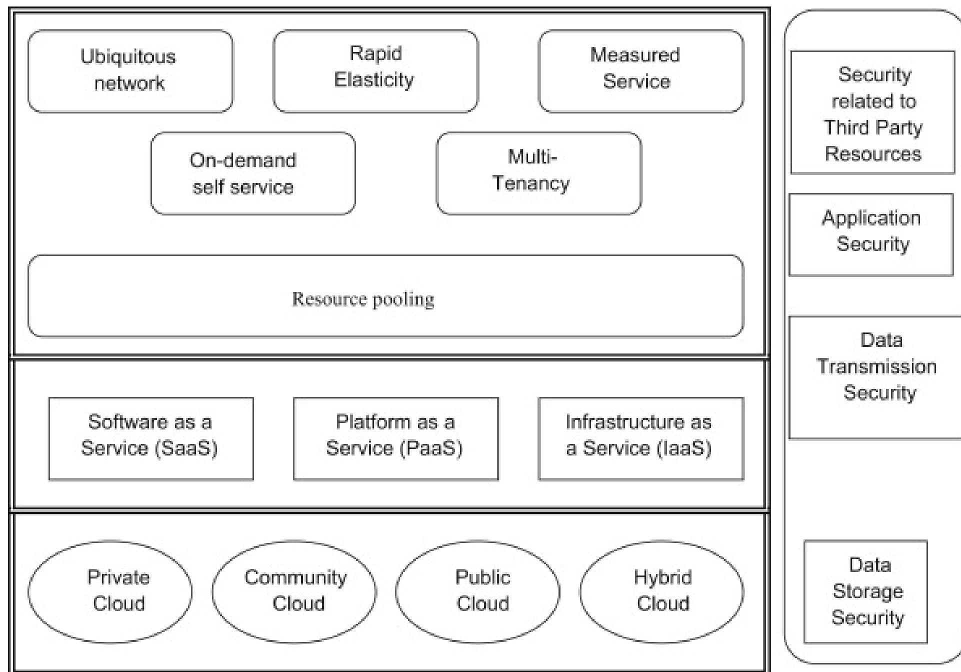


Figure 2.2: Security Issues in Cloud Environment.[58]

This is a cloud age, which means the requirement for new types of service models always exists. We list some of the new service models in the market below.

1. **Serverless Computing**: After Amazon Web Service (AWS) first launched the "Lambda" service at the end of 2014, "serverless" starts to spread like

a marketing term to represent a new model for the deployment of cloud applications. It briskly becomes a new trend and so far all the big cloud service vendors have come up with their own similar services, for example, Google Cloud Functions, Azure Functions, and IBM OpenWhisk. The enterprises have benefited a lot by building their applications on top of the cloud services before the appearance of Serverless Computing. However, the pressure for the developers was still exiting when dealing with scaling requests as it could not be perfectly estimated what resources (CPU time and memory) and how much they should pay for it with a pay-as-you-go utility pricing model. The research [59] gives a reason for this issue from the point of view of the NIST definition of architectures (models) for cloud computing services, which shows there is a large gap between PaaS and SaaS where is quite suitable to place Serverless architecture.[14, 20]

It is quite common to see Function as a Service (FaaS) together with Serverless. Even sometimes people use them to refer to the same thing as there was no clearly defined terminology for them. From the description in the book [51] and most of the studies, it can be generally accepted that Serverless Computing is a more comprehensive concept that includes FaaS and other service models, like MBaaS (Another new service model we will introduce below). Based on the contribution has done in [59], a definition of Serverless Computing is stated by [60] as:

> *Serverless computing is a form of cloud computing that allows users to run event-driven and granularly billed applications, without having to address the operational logic.*

Also the definition of Function as a Service (FaaS) as:

> *Function as a service (FaaS) is a form of serverless computing in which the cloud provider manages the resources, lifecycle, and event-driven execution of user-provided functions.*

In short, Serverless Computing is usually described as a high-level abstraction of cloud software development. The three main features of Serverless Computing is summarized as [59] :

- Granular billing: the user of a serverless model is charged only when the application is actually executing;
- (Almost) no operational logic: operational logic, such as resource management and autoscaling, is delegated to the infrastructure, making those concerns of the infrastructure operator;
- Event-Driven: interactions with serverless applications are designed to be short-lived, allowing the infrastructure to deploy serverless applications to respond to events when needed.

As either Serverless Computing or FaaS is still a new notion in cloud computing territory and it has obtained the focus from both industry and academic sides, there are plenty of related researches have been done and even more, are still on their way. We will also introduce some of the related work in Section 2.3 to help to build a better understanding of this service, which is the groundwork of this paper.

2. **Mobile Backend as a Service (MBaaS)**: Mobile Backend as a Service, also generally known as Backend as a Service (BaaS) is a paradigm created mainly for mobile application developers to simply manage and maintain their application backends in the area of Mobile Cloud Computing (MCC). The goal of this model is planned to be realized by the provider with delivering essence like cloud storage, messaging access, and pushing notifications. Web APIs are designed for developers to easily get access to the data and resources. Compared with the FaaS, MbaaS is more provider-specific and concentrates more on operational logic to reach its goal. The most popular MBaaS example is Firebase, which has been applied to build over a million mobile applications in a period of no more than 1 year. With the increasing amount of mobile applications, MBaas may also be part of the mainstream of the cloud computing development future trend.[26, 59]

3. **Network as a Service (NaaS)**: Network as a Service (Naas) is proposed as "a framework that integrates current cloud computing offerings with direct, yet secure, tenant access to the network infrastructure" by [27]. According to the definitions of the three service models given by NIST, we have known that the network is placed at the bottom level in the architecture and is totally isolated from the end application. On the one hand, it simplified the work for the developers to deploy their application and save their energy to focus more on the business logic of the application. But on the other hand, it limits the visibility of the network on the application logic, which may lead to a waste of network bandwidth in large-scale data centers. Therefore, the need for a network-aware functionality grows, which accentuates the NaaS concept.

   NaaS makes it possible for the end-users to custom networking and implement arbitrary processing within the network, which could be integrated into the switches and routers. The benefits that the customers could gain from the NaaS including bandwidth-on-demand, custom routing, flexible and extended Virtual Private Network (VPN), multicast protocols, content monitoring, and filtering, security firewall, intrusion detection and prevention, Wide Area Network (WAN), antivirus and so on. Nonetheless, the widespread of NaaS is still facing plenty of research challenges, such as scalability, performance

isolation programmability, and unclear pricing model.[17, 27]

4. **Storage as a Service (StaaS)**: Storage as a Service (SaaS or StaaS) is explicated as "a business model in which a company leases or rents its storage infrastructure to another company or individuals to store data" [37]. With StaaS service model, the customers can use the data storage and access services provided by the vendors, which is guaranteed to meet the service level agreements (SLAs) so that they do not have to worry about the scalability and performance of the deployment of their applications. Furthermore, the pay-as-you-go cost model is also allowed to be used in StaaS, making it to be more cost-effective.

   Although IaaS also includes providing storage part, StaaS is a different concept from getting external storage for the IaaS VM. StaaS is able to be sold and applied for any utilization scenarios without considering any other dependencies. StaaS is usually presumed to be only built on top of the cloud, in fact, it offers on-premises delivery as an option as well. StaaS provides an economically friendly way with high-quality storage services, which let it soon become favored by the small or mid-sized businesses that lack the budget for establishing their own storage infrastructure and IT staff. Most StaaS providers support the data storage forms including Object-based storage, block storage, and file storage. StaaS brings the bonus in the aspects of cost, scalability, accessibility, and disaster recovery, while some extra price will also be paid like giving up control more or less.

5. **Monitoring as a Service (MaaS)**: Monitoring as a service is another cloud computing service model under anything as a service (XaaS) we would like to give a brief introduction here. MaaS is seen as a kind of framework that enables the monitoring capability for a range of cloud applications and services. The most well-known implementation of MaaS is state monitoring, which is also a fundamental component of cloud services. State monitoring helps with keeping tracing some specific states of any adaptable cloud elements, such as storage, networks, and application instances.

   In the complicated cloud environment, the importance and value of MaaS are being revealed for improving cloud environment development. The benefits that MaaS offers can be summarised as (1) Reducing the cost of ownership. (2) Offering pay-as-you-go profit model for state monitoring. (3) Motivating the cloud service vendors to continue investigating the monitoring technology so that the quality and performance of the monitored service is able to always get improved.[44]

2.1.2 Cloud Service Vendors

No matter how big the enterprise is, there could always be a type of cloud service model that satisfies its definite requirement. We have displayed most of the cloud service models in fashion beforehand. The next significant questions would be who provides those services and how to choose the suitable provider from a wealth of options.

For the first question, we demonstrate some of the prominent vendors for each service model we mentioned in the previous subsection in Table 2.1. As for how to make a selection from so many options, we suggest always keep two points in mind. First of all, *Acknowledge your demand clearly*. A few questions might be helpful for guiding:

- What is the obstacle in the way of your business development process which asks you to go for the cloud?
- Which layer of the cloud stack can provide the best solution for your obstacle?
- Which service model fits your requirement best?

Secondly, *Select an appropriate vendor rather than a favorite one*. People usually prefer to choose the staff they are familiar with, which will lead to a mistake when it happens on the companies selecting the vendors. Maybe you have been working with the service provided by one cloud provider before, however, this reason is still not strong enough to let you directly select it for your next application. Some examples are given by [35]

Cloud Service Model	Service Vendors
Infrastructure as a Service (IaaS)	- Amazon Elastic Compute Cloud (EC2) - Google Compute Engine - IBM Cloud - Microsoft Azure - Digital Ocean - Alibaba Elastic Compute Service
Platform as a Service (PaaS)	- Amazon Web Services (AWS) Elastic Beanstalk - Google App Engine - Microsoft Azure - Oracle Cloud Platform (OCP) - Salesforce ‘application Platform as a Service’ (aPaaS) - Red Hat OpenShift PaaS

Software as a Service (SaaS)	- Amazon Web Services SaaS - Salesforce - Microsoft - Google G Suite - Adobe Creative Cloud - Box
Serverless Computing (FaaS)	- Amazon Web Services (AWS) Lambda - Google Cloud Functions - IBM Cloud Functions - Cloudflare Workers - Microsoft Azure Functions - Oracle Functions
Mobile Backend as a Service (MBaaS)	- Amazon Web Services (AWS) Amplify - Firebase - Backendless - Progress Kinvey - Back4App
Network as a Service (NaaS)	- Aryaka SmartServices - Perimeter 81 - Cloudflare Magic WAN - Cisco Plus NaaS - Amdocs NaaS
Storage as a Service (StaaS)	- Dell EMC - Hewlett Packard Enterprise (HPE) - NetApp - Pure Storage
Monitoring as a Service (MaaS)	- Amazon CloudWatch - Monitis - Zenoss - Altnix

Table 2.1: Service Vendors for different Cloud Service Models.

2.2 AWS Cloud Services

We introduce a basic knowledge background about the AWS cloud services we utilized in our solution in this section, which includes AWS Lambda, Amazon Elastic Compute Cloud (EC2), AWS Step Functions, AWS Elastic Beanstalk, AWS CodeCommit, AWS CodePipeline, Amazon EventBridge.

2.2.1 AWS Lambda

In this subsection, we are trying to draw a basic understandable picture for AWS Lambda by answering these three questions: (1) What is AWS Lambda? (2) What are the main features of AWS Lambda and what are they used for? (3) The concepts need to know when making use of AWS Lambda.

1. What is AWS Lambda?
   AWS Lambda as we have announced before is a serverless service provider which helps you to get rid of struggling with configuring or managing underlying infrastructures for scaling, such as servers or networks. Moreover, it frees you from maintaining the integration of events and controlling runtimes. Lambda offers a high-availability compute infrastructure to its users to run their codes totally on behalf of them, which means it will take over all the administration work. Lambda allows being used for running codes for virtually all kinds of applications or backend services with its supported languages.

   AWS Lambda is quite cost-effective since it only charges for the compute time and it runs the Lambda function where the users deploy their codes only when needed and scale automatically. Lambda functions can also be invoked by integrated with other AWS services on the basis of the defined events, for example, API Gateway, DynamoDB, and Kinesis, Amazon Simple Storage Service (Amazon S3), Amazon Simple Queue Service (Amazon SQS), Amazon Simple Notification Service (Amazon SNS) and AWS Step Functions.[1]

2. AWS Lambda features. The features listed below empower the Lambda applications to be scalable, secure, and easily extensible.

   - Concurrency and scaling controls. Grant developers powdery control over the scaling and responsiveness of their applications.
   - Functions defined as container images. A powerful tool that is for developers to simplify their developing process when using Lambda functions.
   - Code signing. A verification way that only allows the unaltered code which approved developers has published to be deployed in the specific Lambda functions.
   - Lambda extensions. This feature makes it easier for developers who want to integrate Lambda with the monitoring, secure or other kinds of tools they would like to use.
   - Function blueprints. Guide the developers with sample code on how to combine Lambda with other third-party applications.

- Database access. A proxy function that maintains a database connection pool to achieve a high concurrency level.
- File systems access. An approach to let the developers get access to their local resources safely and at high concurrency by working together with Amazon Elastic File System (Amazon EFS).

3. AWS Lambda Concepts. To understand better how Lambda works, we need to understand its own concepts. We will also use some of them in 4 to describe our solution.

   - Function: A resource that handles the event passed to it and can be invoked to execute the code deployed in Lambda.
   - Trigger: A resource or configuration to invoke a Lambda function, which could be other AWS services, applications, and event source mappings. An event source mapping also refers to a kind of resource in Lambda which could invoke a function by reading items from a stream or queue.
   - Event: A JSON-formatted document including data to be processed for a Lambda function. It is defined when the function is invoked and is converted to an object by the runtime to pass to the function code.
   - Execution environment: A secure and isolated runtime environment for Lambda function. It controls all the resources and processes needed for the function.
   - Deployment package: All the Lambda function code need to be deployed as a deployment package, which only supports two types of archive, a .zip file, and a container image.
   - Runtime: Creating a language-specific environment for running in an execution environment.
   - Layer: Layer is a .zip file package that could include additional code or other content like libraries, custom runtime, or configuration files. Layers are helpful with deploying code and iterating faster, however, only up to five layers per function can be included according to the standard Lambda deployment size quotas.
   - Extension: It could be the tools that are provided by some existing services or developers could create their own Lambda extensions.
   - Concurrency: The number of requests that the function could manage when the function is invoked. It is affected by quotas at AWS Region level.
   - Qualifier: The qualifier is used to specify a version or alias when the developers want to use a stable interface to invoke or view a function.

2.2.2 Amazon Elastic Compute Cloud (EC2)

Amazon Elastic Compute Cloud (EC2) is one of the most essential components of the AWS cloud computing services. It is a web-based service and used to offer to compute capacity with security and resizability in the cloud. Developers could easily obtain and control their compute capacity through the provided web service interfaces. Amazon EC2 gives its users complete control of their computing resources and ample flexibility of the operating system choice for launching their instances with a true virtual computing environment. An adaptable choice of processors, networking, storage, and purchase model is allowed on the EC2 platform. It also announces that the platform is equipped with the fastest processors, the fastest ethernet network transmission speed which is 400 Gbps, the most powerful GPU, and the lowest cost-per-inference instance in the cloud.

An amount of useful features are suggested by Amazon EC2 to improve application's scalability, failure resilience, and performance. We choose the part of them to list below:

- A variety of instance types for some specific optimizing requirements. For instance, bare metal instances, GPU compute instances, GPU graphics instances, high I/O instances, and dense HDD storage instances.

- Pause and resume the instances. The instances are allowed to be hibernated and resumed again later. This feature would benefit the applications that have a long bootstrap procedure and hold state into memory (RAM).

- Optimized CPU configurations. Developers are available to customize the number of vCPUs according to their needs and disable multithreading if the single-threaded CPUs are more suitable.

- Flexible storage options. To fit different storage requirements of diverse EC2 workloads, Amazon Elastic Block Store (Amazon EBS) and Amazon Elastic File System (Amazon EFS) is recommended to integrate when the storage is not built in the instance.

- Auto scaling. This feature offers the ability of to scale up or down the number of instances according to the defined condition by the users.

- High-performance computing (HPC) Clusters. High compute and network performance can also be achieved even though the customer computational workloads are quite complicated with the benefit of this feature.

- Enhanced networking. The notable higher packet per second (PPS) performance, lower network jitter, and lower latency is the profit brought by this feature.

Amazon EC2 prescribes five payments strategies for Amazon EC2 instances, including On-Demand, Saving Plans, Reserved Instances, Spot Instances, and Dedicated Hosts. The customers could choose the way fitted to their occasions with the optimization advice recommended by AWS cost optimization services and tools.

2.2.3 AWS Step Functions

AWS Step Functions is a service that is designed to make it easier for its users to orchestrate various AWS services to complete a mission together, automate their business processes, or build a serverless application. The visual workflow editor lets the users focus more on their workflow logic with less code. AWS Step Functions performs as a coordinator who helps to arrange a series of steps of a workflow following a determined flow logic and keeps tracking the inputs and outputs of all the steps. The most significant functionality of AWS Step Functions is maintaining the application state during its execution period, which incorporates track which steps the application is in at any moment and storing the data passing between the steps of the flow as well as the event log of the data.

The value of the Step Functions in the serverless ecosystem is that it propounds an approach to orchestrate multiple decoupled services, which is challenged by sharing the different parts of state needed access by a number of small services. Splitting the application logic into small pieces could help with the scaling and development efficiency of the application, where the AWS Step Functions play an integral role.

The underlying mechanism of the AWS Step Functions is a state machine. The Step Functions calls its primary abstractions to be its states and the configuration of a Step Functions tells it all involved steps and their transitions. AWS Step Functions uses a kind of JSON-based and Amazon proprietary language which is called Amazon States Language to define all the states and the transitions.

The other favorable features of AWS Step Functions could be:

- Built-in service primitives. AWS Step Functions has already created a set of so-called states ready-made steps for replacing the logic for basic service primitives in the users' applications.
- AWS service integrations. A lot of AWS services support the integration with the AWS Step Functions by using the Step Functions service tasks. The supported services and integration patterns for standard workflow can be found in Figure 2.3.
- Coordination of distributed components. AWS Step Functions activity tasks can help the collaboration with any application through HTTPS connection

no matter where it is hosted, like on mobile devices or Amazon EC2 instances.

- Built-in error handling. The built-in try/catch and retry methods in AWS Step Functions will handle errors and exceptions automatically and respond in a good manner by retreating to appointed cleanup and recovery code.

- History of each execution. With the combination of Amazon CloudWatch and AWS CloudTrail, the real-time diagnostics and dashboards, and logs of each execution can be delivered by AWS Step Functions, which is extremely helpful for monitoring and understanding the whole application workflow.

- Automatic scaling. The operations and underlying compute for the application steps will be automatically scaled while the workloads are changing. No extra configuration required for the users is needed.

- Pay peruse. The users are only charged by state transitions which means no cost during the idle time even if one state could last for one year.

Supported Service Integrations

Service	Request Response	Run a Job (.sync)	Wait for Callback (.waitForTaskToken)
Lambda	✓		✓
AWS Batch	✓	✓	
DynamoDB	✓		
Amazon ECS/AWS Fargate	✓	✓	✓
Amazon SNS	✓		✓
Amazon SQS	✓		✓
AWS Glue	✓	✓	
SageMaker	✓	✓	
Amazon EMR	✓	✓	
Amazon EMR on EKS	✓	✓	
CodeBuild	✓	✓	
Athena	✓	✓	
Amazon EKS	✓	✓	
API Gateway	✓		✓
AWS Glue DataBrew	✓	✓	
Amazon EventBridge	✓		✓
AWS Step Functions	✓	✓	✓

Figure 2.3: AWS Step Function Supported Service Integration (Standard Workflow) [2]

2.2.4 AWS Elastic Beanstalk

AWS Elastic Beanstalk is an easy-to-reach service for developers hoping to establish a powerful and easily managed web application in the fastest way. As a kind of PaaS service, AWS Elastic Beanstalk allow the customers to host their applications without being aware of the fundamental infrastructure. It could take care of all the rest deployment details once the application code gets uploaded, for example, provisioning the required resource, balancing the load, scaling automatically, and monitoring the application status. Even if AWS Elastic Beanstalk can take charge of the software stack and infrastructure management, the users still own full control of each practice of the whole deployment process.

AWS Elastic Beanstalk is formed on top of plenty of existing valuable AWS services, such as Amazon EC2, Elastic Load Balancing, Amazon CloudWatch, and Auto Scaling. Therefore, building with AWS Elastic Beanstalk can take full advantage of those services and it is economical as the billing is only asked for the AWS compute or storage resources like AWS EC2 instances or AWS S3 buckets and none for AWS Beanstalk.

Most of the popular application platforms, as well as programming languages and frameworks, are supported by AWS Elastic Beanstalk such as Java, PHP, Python, Node.js, .NET, Ruby, Go, and Docker. Moreover, there are slight or even no mandatory code changes while migrating to the cloud from the local development machines.

2.2.5 AWS CodeCommit

Developers need teamwork coding for the same project where a source version control system is desired. AWS CodeCommit is a service that maintains private Git repositories on a high security and scalability level. It takes full responsibility for managing the source control servers, including hosting, maintaining, backing up, and scaling. The infrastructure of the service will be automatically scaled regarding the growth of the project. AWS CodeCommit declares that anything can be stored on it from code to binaries. Additionally, developers could continue to use their preferred Git-based tools as the common Git functionality works also well with AWS CodeCommit.

Other typical features of AWS CodeCommit:

- Access control. Users are granted the right to control the accessibility of the project hosted on AWS CodeCommit with the assistance of AWS Identity and Access Management. The access history is also available for checking. The repositories condition can be monitored via AWS CloudWatch and AWS CloudTrail.

- Encryption. HTTPS and SSH are used for documents transfer with AWS CodeCommit. Encryption for the repositories is automatically applied with customized keys through AWS Key Management Service (AWS KMS).

- High availability and durability. Amazon S3 and Amazon DynamoDB are the places where the repositories are stored. A redundant storage strategy is utilized across multiple facilities to improve the availability and durability of the repository data.

- Unlimited repositories. There is no limit for the number of repositories that users can create but need to request more if exceeding the default quota, which is 1000.

- Notifications and custom scripts. Notifications will be sent to the users for the impactive changes happening on the repositories. Furthermore, AWS CodeCommit repository triggers can be configured to send notifications or invoke AWS Lambda functions in respect to the trigger event.

2.2.6 AWS CodePipeline

Continuous delivery becomes more and more popular in modern application development practices, which is a way to automatically prepare the code changes ready for a production release version. AWS CodePipeline is a completely managed continuous integration and delivery service which benefits the development teams with fast and reliable application and infrastructure updates.[3]

AWS CodePipline offers a graphical user interface for their users to easily create, configure and manage their pipelines. Pipelines are set up for defining the release workflow and how to react to a new code change with the release process. Developers can specify a group of actions to be a stage of their pipelines, which could be code building or testing operations. A series of stages compose a pipeline heeding the designed workflow logic. Those actions in the stages can run parallel to speed the workflow.

Other significant features of AWS CodePipeline comprises:

- Integration with other AWS services. AWS CodePipeline allows its users to configure their pipelines combined with the other AWS services based on their needs, for example, pulling source code from AWS CodeCommit or triggering a Lambda function. Additionally, some third-party developer tools, such as GitHub, CloudBees, BlazeMeter, and XebiaLabs, as well as custom systems are also easy to be integrated with the pipeline.

- Custom plugin support. Custom actions can be registered to integrate the custom systems by means of incorporating the CodePipeline open-source

agent in the custom servers or making use of the CodePipeline Jenkins plugin.

- Declarative templates. Developers can designate the release workflow and its stages and actions in a declarative JSON file to specify the structure of their pipelines. The declarative files are also in the use for creating a new pipeline or updating the existing ones.

- Access control. AWS CodePipeline manages the access and control right of the release workflow with the help of AWS IAM. Users' access can be granted through IAM users, IAM roles, and SAML-integrated directories.

- Notifications. Notifications for events that may have an effect on the pipelines can be formed with the Amazon SNS notifications service.

2.2.7 Amazon EventBridge

Amazon EventBridge is a serverless service that is designed for helping to generate event-driven applications on large scale easier. It works as an event bus with the events published by the customer applications, joint Software-as-a-Service (SaaS) applications, and other AWS services like AWS Lambda. A stream of real-time data is delivered from the event source to the target services or applications. The event publisher and consumer can be totally independent of each other by configuring proper routing rules to dictate the data forwarding destination for building a real-time responding application architecture.

Part of the major features of Amazon EventBridge are:

- Decouple event creators from subscribers. The event creators and subscribers in the event-driven applications only need to publish or subscribe to the events via the event bus without being aware of each other.

- Scalable and fully managed. The service is a scalable and fully managed serverless service. No infrastructure or capacity needs to be provisioned.

- Reliable delivery. At-least-once delivery strategy with exponential backoff retry for up to 24 hours is used by Amazon EventBridge to promise its reliability. In addition, the events are persistently saved across many availability zones (AZs). The availability service level agreement (SLA) that Amazon EventBridge offers is up to 99.99%.

- Monitoring and analyzing. Amazon CloudWatch metrics and logs can help to monitor, store and analyze the events dispatch. AWS CloudTrail can be used to monitor the Amazon EventBridge API calls.

- Schema Registry. The event schema is stored in a schema registry for the developers in the same team to easily find the events and their structure. The registry makes it possible to use an event as an object in the code with a code bindings generator for different programming languages, such as Python and Java. The event schemas are able to be automatically discovered by activating schema discovery for an event bus.
- Security control and acceptance. Amazon EventBridge provides access control with AWS Identity and Access Management (IAM) service. Virtual private cloud (VPC) endpoints are supported with TLS 1.2 encryption. Amazon EventBridge is HIPAA-eligible and acquiescent to GDPR, SOC, ISO, DoD CC SRG, and FedRamp.
- API destinations. This feature eliminates the limitation of integrating with the services out of AWS by making use of REST API calls.
- Events recording and replying. The users are enabled to recover processed events back to an event bus or a particular EventBridge rule according to this feature.
- Pay per event. Only the events that created by integrated Software as a Service (SaaS) applications or customer applications are invoiced.

2.3 Related Work

Some related topics are discussed in this section, which are security concerns about cloud computing and serverless maturing and utilization. We have shown some security concerns when introducing the cloud computing service models, in **??**, we extend the discussion scope to the whole cloud computing field. The aim of this section is only to provide a wilder rather than deeper vision of realizing the security problems in cloud computing. In 2.3.2, we evaluate the current performance, the constrictions, and the potential of serverless applications.

2.3.1 Cloud Computing Security Problems

In the early 2000s, large vendors gradually released their cloud products, such as Amazon Web Service (AWS), Simple Storage Service(S3), Elastic Compute Cloud(EC2), and Google App Engine. The moment most people were trying to explore the potential benefits that cloud computing can provide, there were also, some people starting to put attention to the threats and risks that this new technology may bring on privacy and security aspects. Hence, the Cloud Computing Security Workshop (CCSW) was founded and with the purpose *"to bring together researchers and practitioners in all security aspects of cloud-centric and outsourced*

computing." After the first workshop was held in 2008, a variety of consequential potential security risks have been discussed.[13]

The outsourced data security problem is one of the most traditional security topics that has been studied. Different strategies were proposed to resolve it in an efficient way and for different scenarios, such as [19, 63, 69]. Related to accessing the outsourced resources, the most regularly reviewed subjects are cryptographic protocols [31, 47, 68], authentication and authorization [22, 57, 67], and also verification and integrity [55, 21, 52].

Virtualization security concerns are also kept attraction from the researchers all the time as it is the fundamental technology for all cloud products. [64] describes the security threads which threaten administrators and users of a cloud image repository. Virtual machine converts channel communication risks were increasing with the exploration of the VM further utilization and L2 cache covert channels is a typical one.[66, 61]

Vulnerabilities menace is throughout the history of cloud computing development. Researches include incident handling, different attacks and defenses, and leakage issues. [32, 54, 52, 48]

New security matters are emerging with new technologies implementation, for instance, blockchain security and machine learning security. Security and privacy challenges grow together with new computing technologies. On the one hand, it obstructs the adoption of the new technologies, on the other hand, it could be used to evaluate their maturing.[53, 42]

2.3.2 Serverless Computing Analysis

We have introduced the benefits and challenges that serverless computing may bring. The approaches of evaluating serverless computing service providers and serverless applications are still unclear yet. Although there is still no such standard matrix for the assessment, we found some of the previous work that can be referred to.

For analyzing different serverless computing providers, [14] gives an example of investigating the economic and architectural impact of implementing applications on AWS Lambda, and [38] proposes an examination and benchmarking method for potential restriction and bottlenecks of Apache OpenWhisk detection. Moreover, thinking from designing the serverless platform may bring a new angle for analyzing.[39]

When considering to assess the performance of serverless applications, cost-efficiency, resource utilization, stability, and scalability are still highly concerning metrics. [28] introduces an approach to predict the cost of the serverless workflow and [70] displays a quantification study of video serverless processing functions. Furthermore, a more general analytics framework is suggested by [25] for serverless application and the study [36] hopes to conquer the defect of lacking the benchmark suites for the serverless computing and FaaS execution model by offering plenty of FaaS workloads that are ready to be executed on public cloud function execution services.

As a trendy cloud computing service model, we believe more and more measurement methodologies and benchmark suite will be come up with and investigated in the future.

Chapter 3

Analysis of the AWS Lambda Quotas Effect

This chapter aims at surveying the present-day quotas that Amazon Lambda settled and their influence on users' experience and serverless platform expansion. Section 3.1 starts with a review of the current quotas of AWS Lambda and a brief comparison with the constraints that other major serverless platform providers placed. Then, we focus our attention only on a few main quotas of AWS Lambda to probe the possible reason for them and the influence that may produce on serverless promotion in Section 3.2. Furthermore, in Section 3.3, we present several proposed solutions for answering or minimizing the negative effect caused by the serverless platform limitations.

3.1 Serverless Platform Constraints

In this section, a few main state-of-the-art quotas of AWS Lambda are exhibited in Table A.1 at the beginning of the introduction. A complete list of all the AWS Lambda quotas is in Appendix A. Afterward, we demonstrate common constraints comparison among three main serverless service platforms, which are AWS Lambda, Google Cloud Functions, and Microsoft Azure Functions.[4]

Quota name	Description	Default Quota Value	Adjustable
Burst concurrency	The maximum immediate increase in function concurrency that can occur when the functions scale in response to a burst of traffic.	3,000 instances	No
Deployment package size (console editor)	The maximum size of a deployment package or layer archive when uploading through the console editor.	3 megabytes	No
Deployment package size (direct upload)	The maximum size of a deployment package or layer archive when uploading directly to Lambda.	50 megabytes	No
Deployment package size (unzipped)	The maximum size of the contents of a deployment package or layer archive when it's unzipped.	250 megabytes	No
Function and layer storage	The amount of storage that is available for deployment packages and layer archives in the current Region.	75 gigabytes	Yes
Function layers	The maximum number of layers that can be added to the function.	5 layers	No

Function memory maximum	The maximum amount of memory that can be configured for a function.	10,240 megabytes	No
Function memory minimum	The minimum amount of memory that can be configured for a function.	128 megabytes	No
Function timeout	The maximum timeout that can be configured for a function.	900 seconds	No
Processes and threads	The maximum combined number of processes and threads that a function can have open.	1,024 processes and threads	No
Temporary storage	The amount of storage space that is available to a function in the /tmp directory.	512 megabytes	No

Table 3.1: AWS Lambda Main Quotas

Google Cloud Functions categorizes its quotas into three types: Resource Limits, Time Limits and Rate Limits[5]. The architecture of Microsoft Azure Functions is quite different from AWS Lambda which is based on the Azure App Service and App Service Plans. Therefore, its limits are also affected by the hosting plans.[6] We compare the most notable constraints of these three serverless platforms in Table 3.2.

From the comparison of the constraints, we could summarize that: (1) AWS Lambda and Microsoft Azure Functions have a better performance on scalability; (2) Google Cloud Functions and Microsoft Azure Functions are more suitable for deploying large packages and invocation payloads. Users can consider this constraints comparison when they are looking for one serverless platform provider to build their applications. Besides this comparison, we recommend also comparing the other factors, such as deployment difficulty, cost efficiency, real performance in practice, and so on.

Limit Name	AWS Lambda	Google Cloud Functions	Microsoft Azure Functions
Number of functions	Unlimited	1,000/per region	Depends on plan type
Concurrent executions	1000 (depends on region, maximum 3000)	1000 parallel executions (per function)	Unlimited
Maximum execution time	900s	540s	900s for consumption plan (other plans are unbounded)
Deployment package size	250 MB (unzipped)	100MB (compressed) 500MB (uncompressed)	Unlimited
Function memory maximum	10,240 MB	4096MB	min 1.5GB (depends on plan type)
Max API request rate	Unlimited	1000/s	Unlimited
Invocation payload (request and response)	6 MB (synchronous) 256 KB (asynchronous)	10MB (request) 10MB (response)	100MB (request) Unlimited (response)

Table 3.2: Service Limit Comparison among AWS Lambda, Google Cloud Functions and Microsoft Azure Functions.

3.2 AWS Lambda Main Quotas Analysis

We unveil all AWS Lambda quotas' basic information in the last subsection. Even though all those quotas may, to varying degrees, have an effect on the application deployment, some of them still show a higher weight, which is evaluated as the main limitations of working with AWS Lambda. They are **Deployment Package Size**, **Function Memory Maximum**, **Function Timeout**, and **Temporary Storage**. We try to analyze them deeper in two directions: (1) The impact it has on the process of applying AWS Lambda functions; (2) The reason that they are settled for.

- **Deployment Package Size**

  As it is displayed before, the quotas that Amazon Lambda set for the Deployment Package Size are 50 megabytes for the compressed file, which can

be directly uploaded to Lambda, and 250 megabytes for unzipped files when uploading the codes through Amazon S3 (Simple Storage Service).

The reason we include this limitation into the main constraints is that it is one of the limitations that get a notice from developers at an early stage and may still have an influence in the future. Most of the developers think this limitation is acceptable now by using a 50 MB zipped package on Lambda or a 250 MB uncompressed deployment package including layers (dependencies and libraries) on the S3 strategy. However, it is still possible to make the deployment impractical or leads to errors, for example, when using JAVA deployment artifacts [49] or any off-the-shelf ML libraries [62]. Hence, many project builders recently worry that the risk of reaching this limit is not caused by the core codes but by the dependencies.

We also review the researches and analysis about this limit to scrutinize its existing ground. The most persuading motivation to limit the package size is that it causes the biggest impact on AWS Lambda cold start time [7]. Cold start for serverless services represents the time needed for applications to be ready to handle the first request. This period is required for the serverless platforms to provision and scales the applications automatically. On AWS Lambda, once the Lambda function is invoked, the environment starts to be provisioning, and the code gets loaded and executed. Therefore, if the uploaded package size is large, it will take a long time to finish the initialization procedure. The other possible reason to have this limit is reducing the latency when supporting concurrency. AWS Lambda allows the creation of additional threads. But the resources assigned to the Lambda function will also be shared by all the threads and processes. Then, it can lead to the starving and long sleep of the thread. If the concurrent requests are coming to hit Lambda function, new containers will be initiated and the latency can be increased by large package sizes.

- **Function Memory Maximum**

  The Lambda function can be allocated with a maximum memory size of 10,240 megabytes (10 GB) as we presented above. This new quota was just adjusted by AWS Lambda in December 2020, which is more than three times increase compared to the previous restriction of 3,008 megabytes.

  To understand the impact of this limitation, we need to understand what is the role that memory plays in Lambda. When utilizing Lambda to establish the applications, the users are allowed to administer memory size but not CPU for their Lambda functions. As described by Amazon, Lambda will assign CPU power proportional to the amount of memory accoutred, which means the memory allocation decides not only the capability of the memory itself but also the computing. Thus, it is easy to realize that increasing the allocated memory can decrease the function execution time even if the maximum used memory is far less than the allotted size [8]. Moreover, there are also some

examinations showing the memory size can help to optimize the cold start time and the standard deviation in a roughly linear trend [9].

The CPU power depending on the memory size can also explain why the limit for the memory size is needed. As a service provider, Amazon has to balance the requirement of the users and its own resources. The newest adjustment for the memory size, which enables the users to access 6 vCPUs now, improves its memory-intensive operation scalability for workloads like extract, load (ETL) jobs, batch, transform, and media processing applications. In addition, it helps to optimize the performance of computing-intensive applications, such as modeling, machine learning, genomics, and high-performance computing (HPC) applications. The updated limitation meets most of the trendy requirements but it is hard to predict whether enlargement will be demanded again in the future.

- **Function Timeout**

  The quota of the maximum execution time (timeout) for one Lambda function has also been updated once in 2018, which can run up to 15 minutes now. The previous one was a maximum of 5 minutes and it choked the performance of Lambda functions when handling problems as big data analysis, bulk data transformation, batch event processing, and statistical computations using longer running functions. We consider this limit to be especially critical because the execution of customized Lambda functions will be immediately aborted once it is reached and this quota is one of the Lambda fixed limits. Furthermore, the time limit is a complicated topic existing in the whole process of all kinds of programming development.[10]

  Not all kinds of Lambda functions are affected by this maximum execution time limitation. The trouble results from this allocation mainly occur in long-running and large-scale workloads, which is more and more common in modern created applications. Normally, Lambda functions are expected to be short-lived, around 3 to 6 seconds. According to that expectation, 15 minutes is already a high-enough and hard-to-reach roof. Amazon suggests avoiding configuring the timeout for Lambda functions to be the maximum value as well. Since the expense for Lambda functions is calculated on basis of per 100 ms execution time. There are many scenarios that can save the budget by setting an appropriate timeout value. But when the expiration does happen to the function, its performance could be unforeseeable. Additionally, all the background processes, sub-processes, or asynchronous processes generated by the Lambda function are unreliable as all of them will be stopped at the same timeout occurs.

  This restriction for function execution time is not only announced by AWS Lambda, we could see it also in other serverless platforms as shown in Table 3.2. According to this point, an assumption about the existential reason for this limit could be it helps the providers to build the service with less com-

plexity and higher reliability. If we investigate the underlying reason even deeper, it could be the providers are trying to eliminate the negative effect caused by live migration for long-running VMs. The live migration is regularly used by the administrator to migrate the long-running VMs when they need to be synthesized or sustained in VM-based cloud computing.[71]

- **Temporary Storage**

  There is a 512 megabytes limitation for ephemeral disk space to use defined by Lambda. The other point of this space which would be better to know beforehand is that this space could be shared across multiple invocations of the same function. This feature makes it hard to organize when processing long-running functions because space will be shared when the containers are reused and that is something out of the control of the users. This limit normally is quite inconspicuous. But it actually acts as something which is easy to be ignored until it brings about a real problem.

  So, the impact that this limitation will cause for the users is none or fatal as it will sharply decelerate the speed of your work processes or directly blocks the completion of some determined tasks if you meet this constraint. For example, some files could not be totally processed by only one function and streamed directly from their original storage to their output destination. /tmp directory could be used at that moment but then it is easy to exceed the limit of 512 MB when the size of the files is large, such as the fully designed PDF files.

  In most cases, storing temporary things inside of Lambda for Lambda functions is the best option since the content could be dynamic and the access speed of the relative data from Lambda is the fastest. Nonetheless, we still have to be aware of that sharing and reuse rule for this storage directory. Also, the storage will die off once the function execution is completed or the underlying instance is terminated. The most possible reason we can assume why this restriction exists is the design principle of Lambda functions which should be stateless. Relying too much on the temporary storage capacity deviates from the stateless principle to some degree and Amazon may hope to remind its customers about this by this restraint.

To sum up, those limits of AWS Lambda may bring bothers for its existing or potential users, but there is the rationality behind them. The providers have to balance their resource management and users' experience. Besides, the design for the limits could be seen as a protection for keeping the users away from the unexpected behavior or guardrails for leading to the best practice.

3.3 Strategies In Contemporary Work

To overcome the barrier produced by the limitations, a variety of workarounds and strategies have been discussed and proposed in recent works. We introduce a few general hints to conquer those main limits we described above at first, followed by some complete solutions suggested by the previous research work.

- **Deployment Package Size and Temporary Storage**

  We discuss the workarounds for the limitations of deployment package size and temporary storage together as they can be mutually complementary on certain occasions.

  Advice for solving directly uploading compressed package limitation is to use the S3 bucket instead. The cold start problem as we mentioned before can be a side effect. Alternatively, using AWS Lambda Layers as a solution behaves better with a shorter deployment time. But the hard limit of 250 MB still cannot be escaped with both the S3 bucket and Lambda Layers solutions. Moreover, relying on fewer third-party dependencies and choosing frameworks or tools which are helpful with reducing the packaging size, such as Webpack for JavaScript codes. Another solution with the support of the temporary directory is loading large resources dynamically at runtime.

  The limit of temporary storage is not often hit and also depending on the ephemeral in Lambda file system is not recommended. However, there are always special cases, then the hints could be monitoring the left space of the disk or considering the option of getting support from the Amazon Elastic File System (Amazon EFS). The best practice of arranging the data storage needs more examinations according to different customer requirements. A comparison of different storage strategies offered by AWS can be referred to in Figure 3.1.

- **Function Memory Maximum and Function Timeout**

  These two limits have an effect on each other when searching for the solutions. Firstly, current quotas for Lambda function memory and execution time are identified to be adequate for most of the applications. However, the CPU-intensive or logical calculation-based functions may lead to exceeding the time constrain. Allocating more memory is beneficial for relieving the situation but it needs to be careful with the amount assigned to a single-threaded process when taking into account the real performance and cost-efficiency of the application. The other suggestions for handling timeout errors are:

  1. Keep monitoring; CloudWatch and X-Ray are the native tools offered by AWS for helping with monitoring the logs.

2. Optimize the function; Sometimes the execution time is too long because of too many steps included in one function. Splitting the big function by AWS Step Functions into smaller ones works well in this circumstance.
3. Implement fallback methods; This is the last option if the timeout is inevitable. Then at least some cached data or data from other sources can be involved to implement the fallback methods for application improvement.

	Amazon S3	/tmp	Lambda Layers	Amazon EFS
Maximum size	Elastic	512 MB	50 MB (direct upload; larger if from S3).	Elastic
Persistence	Durable	Ephemeral	Durable	Durable
Content	Dynamic	Dynamic	Static	Dynamic
Storage type	Object	File system	Archive	File system
Lambda event source integration	Native	N/A	N/A	N/A
Operations supported	Atomic with versioning	Any file system operation	Immutable	Any file system operation
Object tagging	Y	N	N	N
Object metadata	Y	N	N	N
Pricing model	Storage + requests + data transfer	Included in Lambda	Included in Lambda	Storage + data transfer + throughput
Sharing/permissions model	IAM	Function-only	IAM	IAM + NFS
Source for AWS Glue	Y	N	N	N
Source for Amazon QuickSight	Y	N	N	N
Relative data access speed from Lambda	Fast	Fastest	Fastest	Very fast

Figure 3.1: Comparison of Different Data Storage Options. [11]

A collection of contributions have been done for improving the convenience of AWS Lambda usage in regard to the constraints. For instance, optimizing strategies introduced in [23, 24, 49], novel serverless architectures described in [45, 62], assistant frameworks and methodologies proposed by [50, 65, 71], and helpful cloud services like [40]. Even though lots of effort has been made, we still have a range of unsolved or even unknown challenges on the road of promoting serverless adoption.

Chapter 4

Strategies for AWS Lambda Timeout Limit

This chapter describes strategies that we design to solve the timeout bottleneck of the Lambda function applications. Even though the memory limitations of Lambda are also an interesting topic for us to investigate, the complication of the memory sharing strategy, for example, different Lambda functions sharing their memory size, makes us decide to leave it to future work.

We categorize the applications which are blocked by the timeout limit of AWS Lambda functions into two main types. One is separable long-running applications and the other one is inseparable long-running applications. In Section 4.1, we show our design of the solution for different applications encountering execution time limitations of AWS Lambda. Besides, the design separates the continuous delivery (CD) pipeline from the application workflow by using AWS Lambda without the timeout concern. Then, we give our specific application ”AWSMA” introduced as an experimental example for inseparable long-running applications and decoupling the CD pipeline from the application workflow. We explain the scenario where we need to involve the AWS Lambda function in ”AWSMA” and how the timeout problem influences it in Section 4.2. The experimental details of the solution for ”AWSMA” are expounded in Section 4.3. Next, Section 4.4 arrays the ”AWSMA” application experimental result and provides an analysis of it. And the final step of our exploration is to study the alternative solutions, which are narrated in Section 4.5.

4.1 Solution Design

This section first shows our design of strategy to getting rid of the time limitation of AWS Lambda for the two types of long-running applications in Section 4.1.1. The design for decoupling the CD pipeline and the application workflow execution is exhibited in Section 4.1.2

4.1.1 Solution for Time Limitation of Long-Running Applications

During the period of the whole design process, we have to achieve two main goals. First of all, the software quality must be assigned the top priority. Secondly, making full and rational use of different sorts of cloud services supplied by AWS regarding our requirements.

For the software quality, we need to consider some crucial attributes listed below for our applications, which is following the software product quality model defined by ISO/IEC 25010:2011 standard [34].

- Functional suitability.
- Reliability.
- Performance efficiency.
- Security.
- Maintainability.

To satisfy the second demand, we reviewed and assessed the scores of AWS cloud services. Afterward, we picked a few of them to be part of this solution design for inseparable long-running applications and their essential background knowledge is disposed of in Section 2.2.

1. **Solution Design for Separable Applications.**
   There are already some existing solutions for the separable long-running applications regarding the research result in Section 3.2. Here we introduce one solution which is built on AWS Step Functions. It contains two key steps. The first one is to partition the long-running application into several small tasks and the second one is to deploy the small tasks with Lambda and apply AWS Step Functions to organize all the divided tasks together based on the application workflow. Figure 4.1 shows the structure of the design deployed on AWS Step Functions. One drawback of this solution is that the users have to separate the computation manually and make sure the execution time of each divided function is within the Lambda time limitation.

2. **Solution Design for Inseparable Applications.**
   Depending on the solution for separable applications, we choose to use AWS Step Functions together with Amazon EC2 and AWS Lambda to compose the core unit of the solution for inseparable applications. We also add the AWS Elastic Beanstalk into the organization to promote application management. This design is not a completely FaaS strategy but an integration

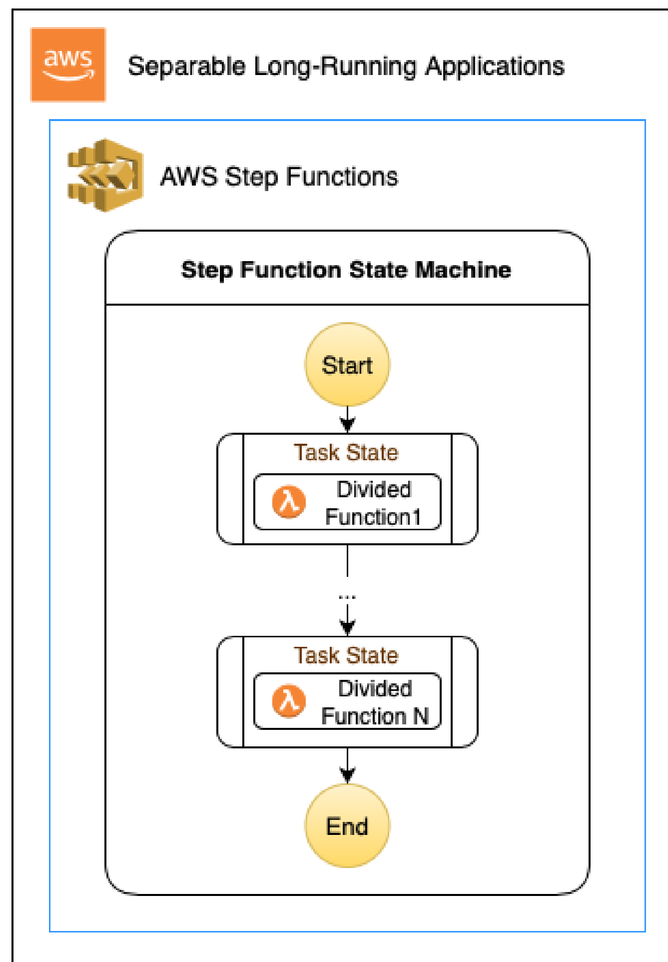


Figure 4.1: Separable Applications Solution Design

strategy with SaaS. The solution design is graphed in Figure 4.2

The central idea of this design is to detach the long-running procedure from the application which is planned to be deployed on the AWS Lambda function without breaking up the entire workflow. So we design the following:

(a) AWS Step Functions is the best option for us to orchestrate the workflow, which works as a state machine to maintain the input and output of each state and ensure the workflow proceeding as it is expected. AWS Step Functions allows its task states directly to coordinate an assortment of other AWS services, such as AWS Lambda, AWS Batch, and DynamoDB (see Figure 2.3). Moreover, the activity feature of AWS Step Functions creates a so-called ”activity worker” to perform the task work which could be hosted anywhere.

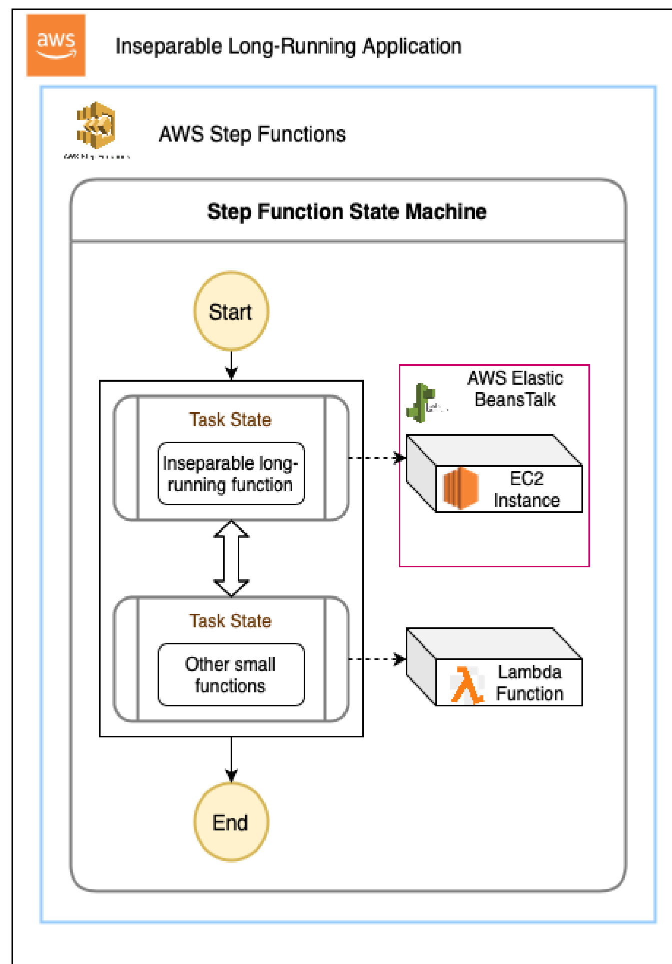


Figure 4.2: Inseparable Applications Solution Design

(b) Amazon EC2 is the place where we decided to host that long-running data synchronization task. Benefited from the Step Functions activity feature, there is no worry about how to connect to the Step Functions to continue the workflow.

(c) We still use the AWS Lambda function to run the rest part of the application and it is originally supported by Step Functions.

(d) To make resource management easier and deployment faster, we equip AWS Elastic Beanstalk to take care of the Amazon EC2 instances.

4.1.2 Decoupling the Pipeline and Application Execution Design

This division design aims to decouple the continuous delivery (CD) pipeline from the application processing workflow implemented with AWS Step Functions if the developers use AWS CodePipeline. For example, in the design that we presented in

the previous sub-section, AWS Elastic Beanstalk will provision the resources and handle the deployment automatically after the code is uploaded. This action will also start the first activity task state of the Step Functions state machine. But it is meaningless to always interrupt the long data synchronization process by irrelevant code changes and we try to avoid that situation. We wish the pipeline to prepare the code ready, which has been done in the pipeline source stage, but not directly deploy it to the Elastic Beanstalk until the previous state machine has ended.

To achieve the detachment, we need to observe the running status of the initiated state machine and trigger to start a new one if the last one has been completed and the pipeline is on an expected stage. We think AWS Lambda is suitable to take the responsibility for the surveillance and corresponding operations. Nevertheless, the timeout issue impedes Lambda functions to monitor that long-lasting state. Luckily, we found a possible solution with the continuous token inspired by an AWS DevOps blog [12]. So we come up with an adjusted design to be fitted to our purpose, which is revealed in Figure 4.3.

We define three types of stages in the pipeline, which are the source stage, deploy stage, and customized stage. The logic steps of building the CodePipeline are explained below:

- **Developers commit their codes on any source control service in the source stage.**
- **The first customized stage monitors the status of the last state machine through a Status-Tracking Lambda function.** The working processes marked in Figure 4.3 around the first Lambda function represents:

    1. The action configured in the first customized stage invokes this Status-Tracking Lambda function.

    2. The invoked Lambda function is going to detect the status of the state machine.

    3. As a result of the last step, the Lambda function obtains a returned status of the state machine, which could be RUNNING, SUCCEEDED, FAILURE, TIMEOUT, or ABORT.

    4. If the Lambda function receives a RUNNING returned status, it will send a continuous token which contains the unique identification - the state machine execution ARN to the pipeline action. In this case, the pipeline action will invoke the Lambda function again in several seconds until the condition meets the next step.

5. If the SUCCEEDED status is returned, the Lambda function will tell the pipeline that action is completed, or else, the Lambda function fails the pipeline action when any other status (FAILURE, TIMEOUT or ABORT) is returned.

- **The deploy stage action will deliver the code to the service where the application will be deployed.**
- **The second customized stage action triggers a new state machine and completes the pipeline via a New-State-Machine-Trigger Lambda function.** The procedures marked in Figure 4.3 around the second Lambda function indicates:

  1. The settled action in this stage invokes the New-State-Machine-Trigger Lambda function.

  2. The invoked Lambda function will start a new state machine execution.

  3. A continuous token is sent back to the pipeline action by the Lambda function for further operation.

  4. The pipeline action invokes the Lambda function again in a few seconds to check the execution state of the triggered state machine with the continuous token received in the previous step.

  5. The Lambda function asks the new state machine for its status following the last step.

  6. The Step Function returns the execution state of the new state machine.

  7. The steps (4, 5, 6) will be repeated if the Lambda function notifies the pipeline action that the state machine is still in a RUNNING state. The repetition will end with a none-RUNNING state received by the pipeline action. The pipeline action is recognized as complete if the state machine execution completes successfully or else the Lambda function fails the pipeline and stops its job.

We classify the roles that all AWS services apply in our strategy designs in Table 4.1

4.2 AWSMA Illustration

We only experiment with the design for inseparable long-running applications here as the separable long-running solution has already been applied in many applica-

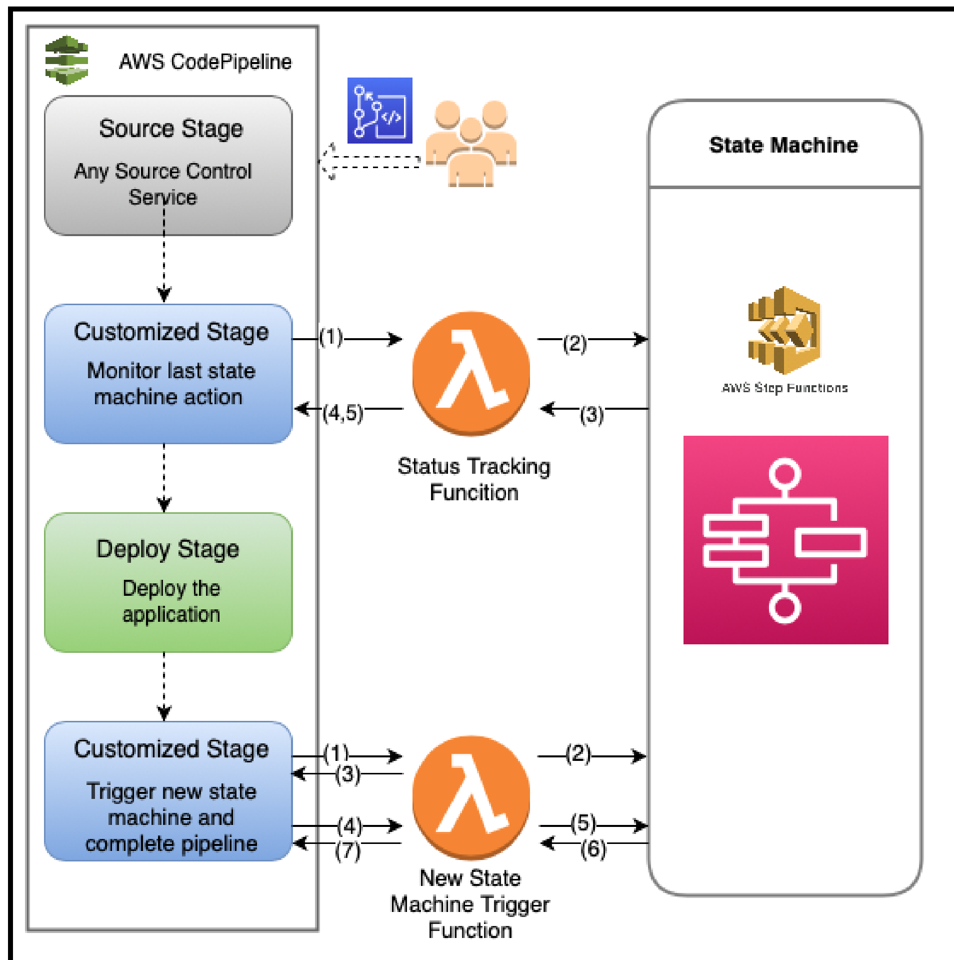


Figure 4.3: Decoupling Continuous Delivery (CD) Pipeline from Application Workflow Design

tions. The specific application that we implement the solution on is one migrating project to AWS cloud, we will call it in short as "AWSMA" in the following description. The AWSMA is a JAVA Spring Boot project which is currently deployed on our private servers and we plan to deploy it on the cloud with the AWS serverless services. We need to consider the following factors:

- All the functionality of the application will not be broken by the migration.
- Change the project structure or codes as little as possible. Since it is a large project, the architecture is very complicated and we try to keep away from those business logic traps. Besides, the project is likely to be refactored soon so that too much change is not worthwhile to be done at this stage.
- The solution should be cost-effective.

AWS Service	Role in the Designs
AWS CodeCommit	Source Code Management.
AWS CodePipeline	Continuous Delivery from source code holder (AWS CodeCommit) to Deployment Platform (AWS Elastic Beanstalk).
AWS Step Functions	Orchestration for AWSMA workflow. Integrating all the AWS services needed.
AWS Elastic Compute Cloud (EC2)	Data synchronization procedure of AWSMA execution environment.
AWS Lambda	Computing the Rest part of AWSMA except the data synchronization. Assisting with decoupling the pipeline from the application work flow.
AWS Elastic Beanstalk	Application deployment platform. Previsioning the AWS EC2 and other resources for AWSMA automatically.

Table 4.1: AWS Services Roles in the Strategy Designs.

4.2.1 Application Description

A brief description of AWSMA is given here without irrelevant business logic details. AWSMA is an information management system that is composed of two major tasks: (1) Managing the system database operations and processing the data maintained in the database; (2) Generating proper REST API for the interaction between external requests and the web application for displaying the system information. The data source of AWSMA database is assembled with two parts, one part is uploaded by our users and the other part is synchronized from another database maintained outside of the AWSMA through asynchronous communications. Therefore, the AWSMA workflow can be simplified as it is manifested in Figure 4.4.

4.2.2 Problem Identification

The AWSMA is not a huge project in general even though it was created several years ago and has kept iterating during the years. Hence, we are supposed to deploy it with one Lambda function. However, we realize that the execution time of our application will exceed the quota of the Lambda function which is a hard limitation that we could not request more through the support service.

With several times tests, we noticed that the ”data synchronizing” process is the key to the timeout problem. As we need to retrieve data from an external database, many uncertain elements may affect the retrieving process, such as network and operations in that external database. According to the requirements we need to ponder on the way to serverless and the Lambda function timeout obstacle we

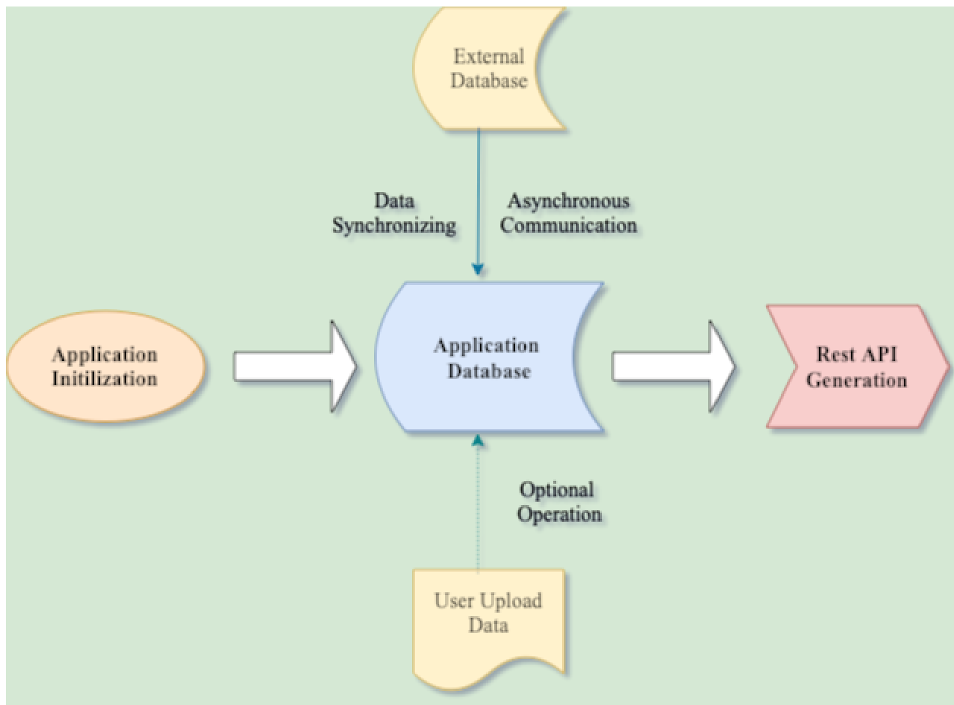


Figure 4.4: AWSMA Application Main Workflow.

were facing, we tended to choose the solution with the AWS Step Functions.
But we were still troubled by the timeout errors after we split the ”data synchronization” process into a separate Lambda function as a step in the Step Function state machine. Thus, we realize that we need a solution to deal with the inseparable long-running task like this ”data synchronization” procedure.

One more additional problem or requirement for AWSMA is decoupling the continuous delivery (CD) pipeline and the application execution for the DevOps chain. Because we do not want to interrupt the execution process by any new commit of our codes. We hope to trigger the new execution of the application when the previous application execution has finished and new code commits arrive.

4.3 Experimental Setup

According to the design for inseparable long-running applications and the description of our specific use case ”AWSMA”, we set up the experiment for evaluating the solution with AWSMA in this section. The AWSMA application implementation part is established firstly, which is followed by the settlement of the decoupling of the CD pipeline part. The experimental design for AWSMA is shown in Figure 4.5 and the design for the CD pipeline division is demonstrated in Figure 4.6 Before the experiment, we have prepared codes for two tasks of the state machine

ready, which are divided from the original project. Also, the preconditions and setup routine for applying AWS services is not elaborated here, for example, the IAM user creation and permission grant, AWS CLI configuration, and Git tools for cooperating with AWS CodeCommit repositories.

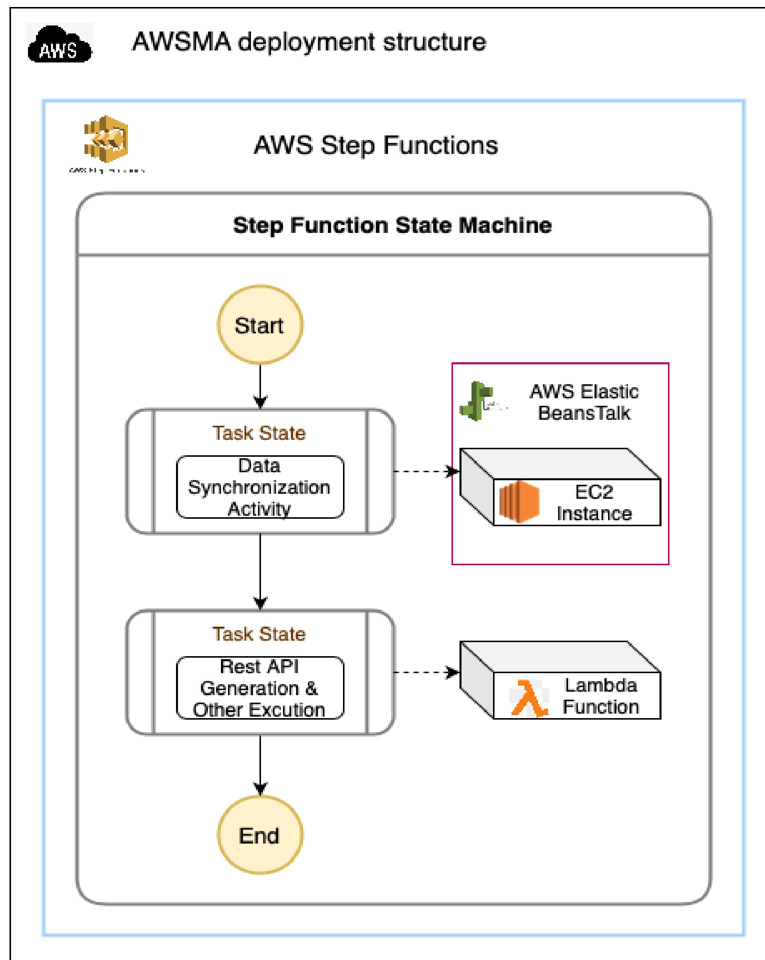


Figure 4.5: AWSMA Implementation Structure

Steps for setting up the AWSMA application workflow (A state machine demo is given to help with explanation in Figure 4.7a):

- Create an Activity on AWS Step Functions and acquire its Amazon Resource Names (ARN), like the state *"GetSynchronizedData"* in the demo figure, which is used to connect the state machine with the *"worker"*, the program we run on AWS Elastic Beanstalk.

- Create a Lambda Function on AWS Lambda service and acquire the unique

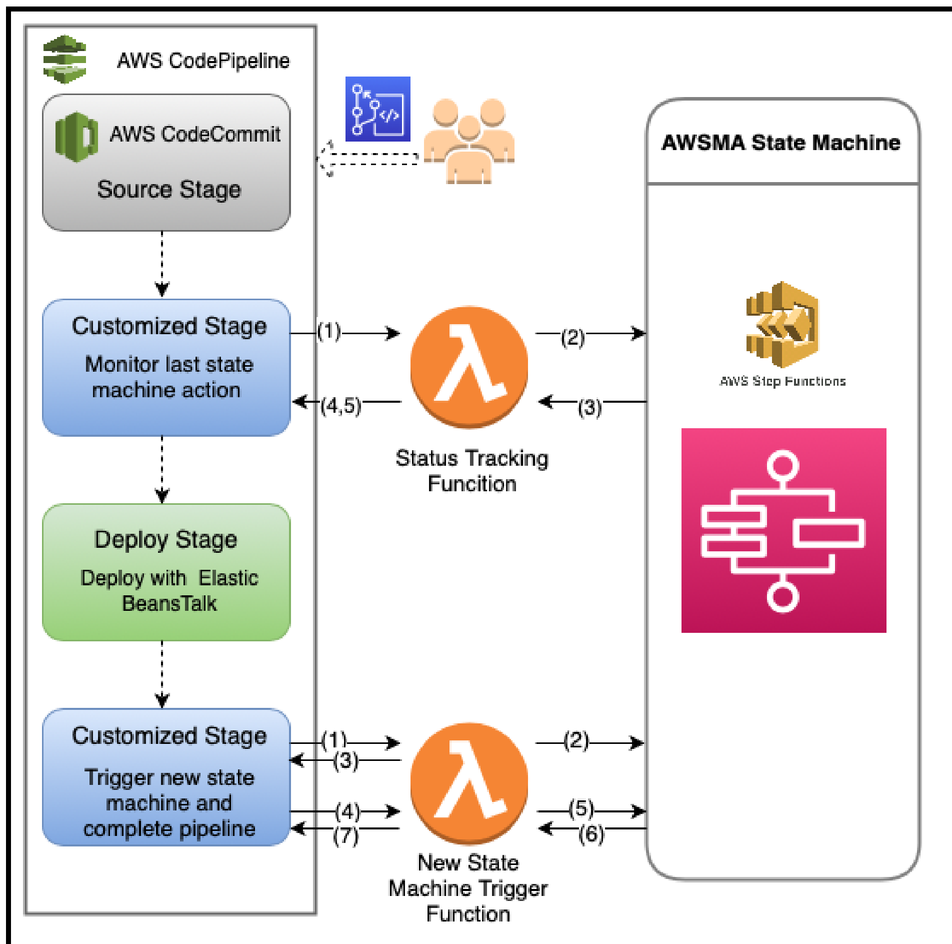


Figure 4.6: AWSMA Continuous Delivery (CD) Pipeline Division Design

ARN, like the state *"LambdaExecution"* in the demo figure, which finishes the rest computing assignment of our application.

- Create a State Machine on AWS Step Functions like the state machine shown in the demo figure, which arranges the workflow from invoking the activity *"GetSynchronizedData"* to the end of the Lambda function *"LambdaExecution"* by their ARNs.

- Create an application deployment environment on the AWS Elastic Beanstalk platform. We need to upload our bundled source code during the Elastic Beanstalk application generation process. But later we will configure our pipeline to be the source of the environment. As AWS EC2 default execution timeout is 3600 seconds, which is normally enough for our long-lasting data

synchronization activity, we just keep the default settings. This execution timeout value is adjustable, so it can be set higher than 3600 seconds if it is needed. After creating the application, AWS Elastic Beanstalk automatically creates an environment for that application. The environment configuration is editable and some main configuration data for our environment are listed in Table 4.2

Steps for setting up the continuous delivery pipeline, like a demo one in Figure 4.7b:

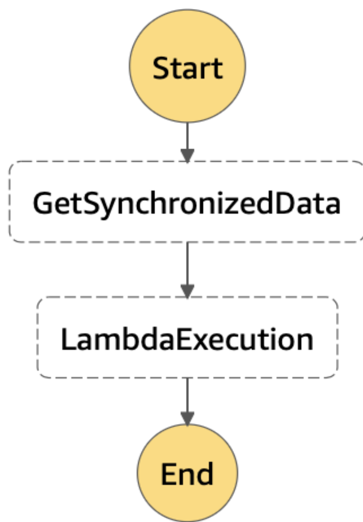
- Create a repository on AWS CodeCommit to manage the data synchronization process codes. We use Git tools locally and the SSH method to communicate with the remote repository.

- Create two Lambda functions. One is used to check whether the last state machine has completed its execution. The other one is to trigger a new state machine and track its status for giving corresponding feedback to the pipeline so that the pipeline could show the delivery status to us.

- Create a pipeline through AWS CodePipeline. To create the pipeline as we designed before, first, we need to create a normal two-stage pipeline, including the Source stage and the Deploy stage. We skip the Build stage and select the deployment provider to be AWS Elastic Beanstalk at the Deploy stage. Then, we could add two customized stages by editing the created pipeline. We add one "CheckStatus" stage between the Source stage and the Deploy stage. Add the created Lambda function which checks the status of the previous state machine to be the action in that stage. Finally, add one "Run" stage at the end of the pipeline and choose the Lambda function which triggers the new state machine and tracks its completion to be the action of this stage.

4.4 Result Evaluation

The main purpose of this section is to check the experimental result of the Lambda timeout solution for inseparable long-running applications and the CD pipeline decoupling solution with Lambda. We will focus our evaluation of the experimental result primarily on whether those two problems identified in AWSMA are solved by recording the two states' execution time of the state machine and the pipeline completion status, which also represents the "Functional suitability" and "Reliability" of our application. Next, we consider evaluating the cost efficiency as it is a commercial application. Furthermore, we will investigate the insight from the monitoring data provided by AWS monitoring metrics.



(a) State Machine Demo.

Source Succeeded
Pipeline execution ID: 83d2782c-ca83-4b90-ba16-37bc6886d7e0

Source
AWS CodeCommit
Succeeded - Just now
cc92a744

cc92a744 Source: add all

Disable transition

CheckStatus In progress
Pipeline execution ID: 83d2782c-ca83-4b90-ba16-37bc6886d7e0

CheckOldStateMachineStatus
AWS Lambda
In progress - Just now
Details

cc92a744 Source: add all

Disable transition

Deploy Didn't Run

Deploy
AWS Elastic Beanstalk
Didn't Run
No executions yet

Disable transition

Run Didn't Run

TriggerNewStateMachine
AWS Lambda
Didn't Run
No executions yet

(b) Pipeline Demo.

Figure 4.7: State Machine and Pipeline Demos

4.4.1 Timeout Problem Resolving

We pick ten times experiments data displayed in Table 4.3, involving execution time of the data synchronization state deployed on AWS EC2 instance, execution time of the Lambda function state, state machine completion status, and pipeline completion status. For the execution time, we use the rounded value in minutes.

Even though there is still one timeout record shown in the records table, we pick this record just to indicate the timeout problem could happen. It does not represent a 10% timeout possibility and in fact, it rarely happens. The reason caused the timeout usually is the network problem.

4.4.2 Cost Efficiency

We calculate the cost for the same 10 times experiments record in Table 4.4 according to the execution time of the EC2 instance and Lambda function. The other extra charge for monitoring or other services is not included. As the price for the AWS Step Function standard workflow is charged as $0.000025 per state transition thereafter and the price for the pipeline is $1.00 per active pipeline * per month, we will ignore the cost by them.

The calculation of the cost has also been rounded but it is still obvious that each execution for one completed state machine is only around $0.1. So if we have 20 execution per day, the monthly cost will be $60. However, this is only a rough calculation based on the execution time, AWS also offers many plans for saving the budget. Additionally, we also notice that we could adjust the EC2 capacity configuration to achieve better cost efficiency in the next evaluation part.

Another point that can be observed is that if the timeout happens, the cost would be three times the normal cost as the default timeout configuration of AWS EC2 is 3600 seconds, which is 60 minutes. The common execution time for the successful execution should be around 20 minutes, therefore, we can also save the cost by resetting the execution timeout with a smaller value, for instance, 1800 seconds.

4.4.3 Insights from Monitoring Metrics

AWS Elastic Beanstalk has already integrated AWS CloudWatch for monitoring the application and environment statistics. A variety of metrics are formulated and keep updating based on the demand of users. They are demonstrated on the Monitoring page of each environment created by Elastic Beanstalk on the AWS Elastic Beanstalk web console page. There are also a few other helpful detailed pages exhibiting relevant running information about the environment, such as:

- Health page: Detailed health status information about the instances that the environment has generated for user applications.
- Events page: A range of types of information or messages from the Elastic Beanstalk or related resource services.

The Environment Health metric is usually the first place where we are going to check the availability and the accessibility of the EC2 instance running our data synchronization codes. It is a direct sign to tell us the general health status of our application and we can continue to track the problem details according to the timestamp if a health issue happens. Figure 4.8 is an 8 hours tracking example, which shows our instance works as it is expected. Figure 4.9 shows the health check result when we encountered that timeout record mentioned before, which rarely happens.

The other insight we gain from the metrics is that we should adapt our instance capacity to achieve better CPU Utilization, which also affects the cost efficiency. In Figure 4.10, we notice the CPU utilization of our current capacity configuration, which is equipped with a *t2.xlarge* instance, is too low. We should change to a suitable instance type with lower capacity, such as t2.large or even t2.small, which still needs more experiments to verify.

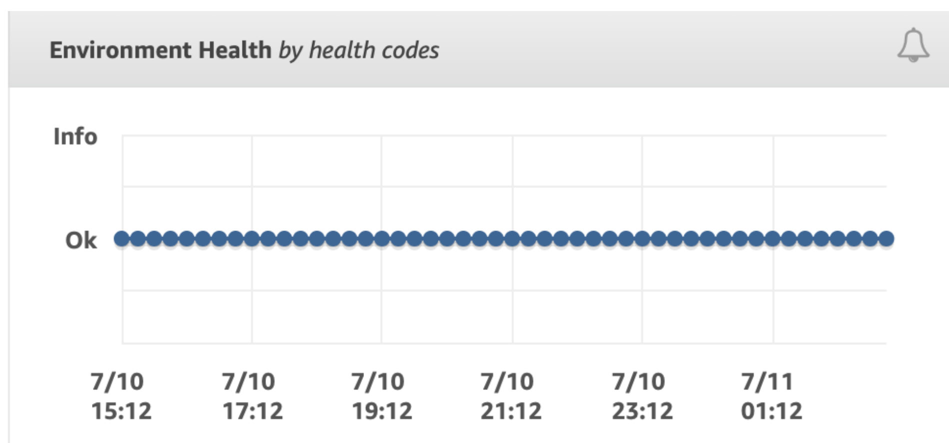


Figure 4.8: Environment Instance Health Check – Healthy Status

4.5 Alternative Solutions

After the implementation and analysis of our original solution design, we earn a better understanding of our application deployment requirement as well as multiple AWS cloud services. In addition to that, some of the AWS cloud services have also been upgraded when we started to search for probable alternative solutions. Therefore, we come up with three prospective alternative solutions including

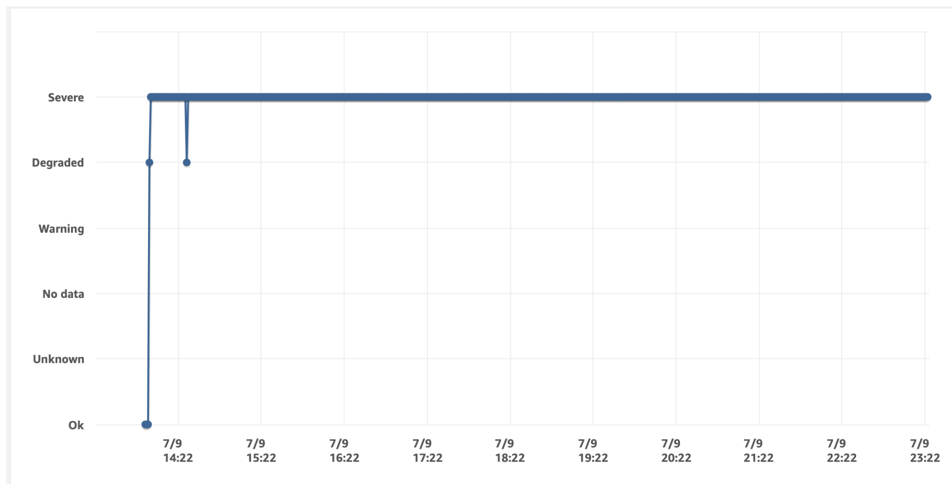


Figure 4.9: Environment Instance Health Check – Timeout Status

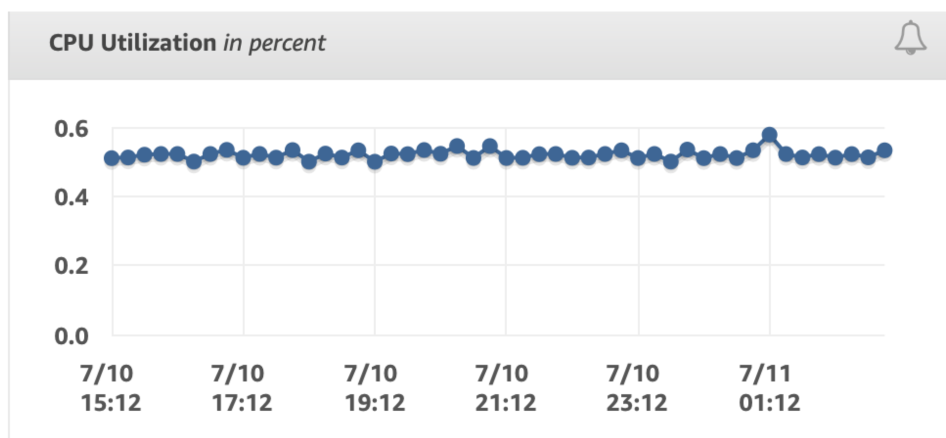


Figure 4.10: Environment Instance Health Check – CPU Utilization

both the application workflow part and pipeline part for our current solution design.

1. Even though we enjoy the convenience that AWS Elastic Beanstalk contributes to our development and assessment process, we surmise it can be removed from our current application workflow construction. AWS Elastic Beanstalk is a good guide when we are not familiar with the AWS infrastructure resources and not sure how to arrange the resources properly for our application at the beginning, but it becomes dispensable especially after we realize that the burstable resource is not indeed required in our scenario. The other reason is reaching the instance where our application is located in reality can be more undeviating and we can also monitor our application status directly from the AWS EC2 instances view.

2. One more proposal for the application structure is to deploy the whole application only with AWS EC2 instances. In this way, no more code-splitting, Lambda and Step Function implementation is needed. This idea is from the perspective of simplifying the application structure and eliminating the additional work for the code, yet the deployment work and the cost may increase.

3. There is also one alternative solution for the pipeline part. As AWS announces that AWS Step Function supports the integration with AWS EventBridge in May 2021, the events can be easily produced in a state machine workflow. Hence, we suppose that we could replace the first Lambda function with an event bus created by AWS EventBridge as manifested in Figure 4.11, which is responsible for tracking the status of the previous state machine. Step (1) indicates the Step Functions produces an event when its state machine state changes. Then configuring the specific event rule and assigning the customized pipeline stage to be the event target in step (2). The alternative solution can reduce the cost significantly as the AWS EventBridge will only charge for the event publishing and there is even no charge for all events generated by AWS services for their state change.

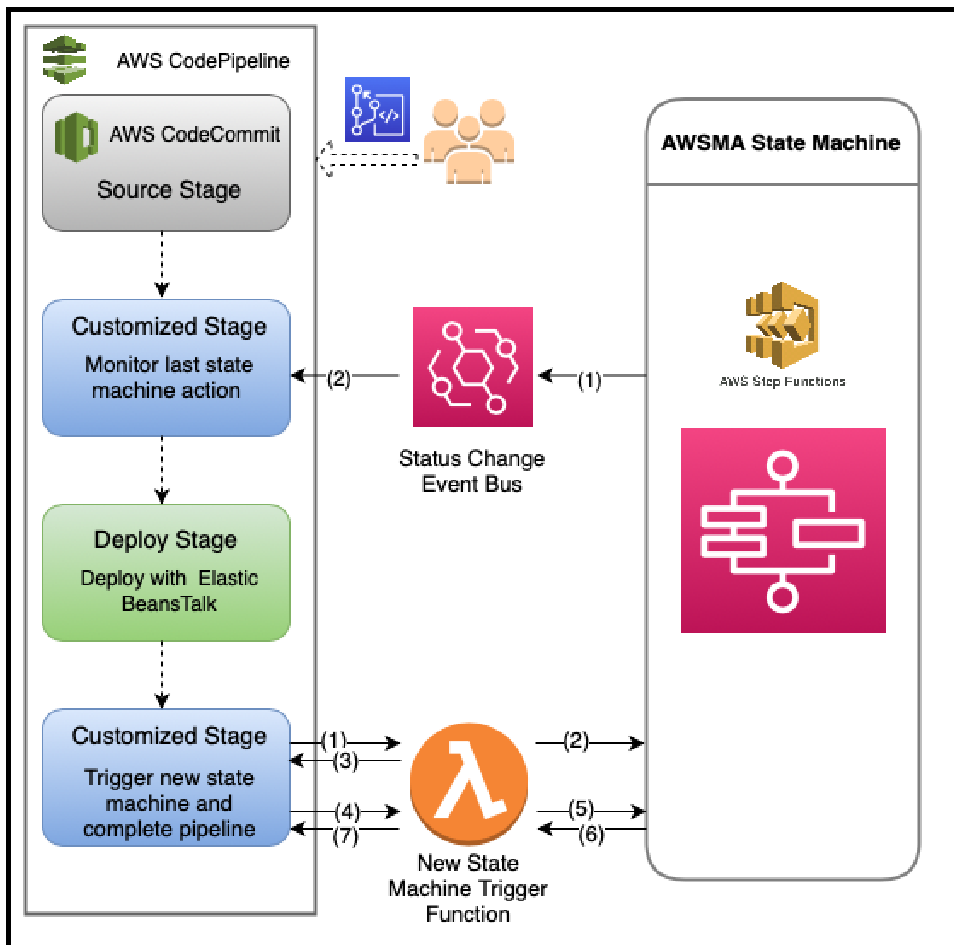


Figure 4.11: Alternative Solution with AWS EventBridge for Pipeline Decoupling

Category	Options
Software	Environment properties: GRADLE_HOME, JAVA_HOME, M2, M2_HOME, SERVER_PORT
Instances	Monitoring interval: 5 minute Root volume type: container default(None) Size: container default(None) Throughput: container default(None)
Capacities	Availability Zones: Any Breach duration: 5 (Min) Environment type: load balancing, auto scaling Instance type: t2.xlarge Lower threshold: 2000000 (Bytes) Max: 5 (EC2 instances) Metric: NetworkOut Min: 1 (EC2 instances) Period: 5 (Min) Scale down increment: -1 (EC2 instances) Scale up increment: 1 (EC2 instances) Scaling cooldown: 360 (seconds) Statistic: Average Upper threshold: 6000000(Bytes)
Load balancer	Listeners: 1 Load balancer type: application Processes: 1
Rolling updates and deployments	Batch size: 100% Command timeout: 3600 (seconds) Deployment policy: All at once Healthy threshold: Ok
Monitoring	CloudWatch Custom Metrics-Environment: CloudWatch Custom Metrics-Instance: Health event log streaming: enabled Ignore HTTP 4xx: enabled Ignore load balancer 4xx: enabled System: Enhanced Lifecycle: Keep logs after terminating environment Retention: 7 (Day)

Table 4.2: AWSMA Elastic Beanstalk Environment Configurations.

No.	Data Synchronization Execution Time (Min)	Lambda Function Execution Time (Min)	State Machine Completion Status	Pipeline Completion Status
1	20	3	SUCCEED	SUCCEED
2	25	2	SUCCEED	SUCCEED
3	22	2	SUCCEED	SUCCEED
4	18	2	SUCCEED	SUCCEED
5	28	3	SUCCEED	SUCCEED
6	23	2	SUCCEED	SUCCEED
7	timeout	-	TIMEOUT	FAILED
8	21	3	SUCCEED	SUCCEED
9	22	2	SUCCEED	SUCCEED
10	25	3	SUCCEED	SUCCEED

Table 4.3: AWSMA Experiments Execution Records.

No.	Data Synchronization EC2 On-demand Instance cost ($)	Lambda Function Cost ($)
1	0.1	0.0003
2	0.1	0.0002
3	0.1	0.0002
4	0.1	0.0002
5	0.1	0.0003
6	0.1	0.0002
7	0.3	-
8	0.1	0.0003
9	0.1	0.0002
10	0.1	0.0003

Table 4.4: Calculated Cost for the Experiments in Table 4.3.

Chapter 5

Conclusion and Future Work

Cloud-based technology and services have contributed a lot to improving resource sharing and utilization. The Serverless service model as one of the most popular emerging service models has attracted not only the service vendors but also many developers. Loads of serverless services have been offered by different multifarious service providers. AWS Lambda as the earliest one who appeared in the market becomes one of the most popular options for a majority of users. But then again, there is no perfect service. There is a list of AWS Lambda quotas that may cause some pain in the users' experience. We have experienced the trouble affected by the execution timeout limitation, therefore, we hope to understand more about the limits through a survey and share our approach to avoid the inconvenience.

In this chapter, we first outline our contributions in Section 5.1. Secondly, we discuss our next plan for the near future in Section 5.2.

5.1 Conclusions

We state our conclusions with our findings or resolving strategies for the research questions that we proposed before:

- **RQ1:** What are the most pivotal limitations of AWS Lambda and the effect that they may cause in practice? What could be the possible reasons and solutions?

  The most painful limitations which have been complained about by many developers are **Deployment Package Size**, **Function Memory Maximum**, **Function Timeout**, and **Temporary Storage**.

  There is no doubt that the most significant effect of those limitations is reaching any of the limitations will lead to a deployment failure or error occur-

rence. But we also realize that (1) Deployment package size has an influence on the Lambda function cold start time; (2)Function memory allocation is bound with the computing ability and also influences the cold start time; (3) Function execution time affects the cost for the consumers. Meanwhile, it affects the complexity and reliability of the service; (4)Temporary storage is dynamic and the fastest way of accessing the relative data from Lambda, but all the content will be erased at the same time the function execution is completed or the underlying instance is terminated. And these effects contribute to the reasons why they need to be limited.

We discuss a few hints and workarounds to overcome the trouble brought by the constraints. For deployment package size, S3 bucket and Lambda Layers can be helpful to some extent. But the key point is to reduce the third-party dependencies or minimized the package size by taking advantage of package management frameworks or tools. The temporary storage issue is avoidable by choosing a proper storage strategy according to the user's demands. Allocating more memory could help to relieve the timeout problem, but it could also bring the performance and cost-efficiency problem. Therefore, the core idea for controlling timeout is to optimize the logic and structure of the CPU-intensive or logical calculation-based functions that may lead to exceeding the time constrain.

- **RQ2:** How can we resolve the AWS Lambda timeout bottleneck when deploying the long-running applications with serverless or other AWS cloud services?

  We propose a resolving strategy for the long-running applications that wish to deploy on AWS Lambda by combining a list of AWS cloud services. The main application workflow will be orchestrated by the AWS Step Functions. If the long-running application can be split into several smaller functions, then they could be directly handled by Lambda functions and organized by the Step Functions. But if there is one part of the application that still meets the time limit after splitting like our specific use case, we suggest implementing that part with AWS Elastic Compute Cloud (EC2) or even AWS Elastic Beanstalk depending on the need. We provide an experiment as an example for the inseparable long-running applications and also the evaluation based on our use case requirements.

- **RQ3:** From the DevOps perspective, how can we decouple the continuous delivery (CD) pipeline from the application execution workflow with the help of AWS Lambda despite its timeout constrain?

Some applications have to decouple the continuous delivery (CD) pipelines from their application execution workflows, for instance, controlling the delivery pipeline stage corresponding to the application workflow running status. We offer a solution by monitoring the application workflow with Lambda functions. Moreover, we apply a ”continuous token” together with Lambda functions to avoid touching its timeout constraint. We also demonstrate an example implementation for the solution and bring forward a probable alternative design with the AWS EventBridge.

5.2 Future Work

Base on the research findings and outcomes we gain from our evaluation and alternative solutions analysis, we classify a list of actions and study direction for the near future to improve our current implementations and contribute more value for serverless service adaption:

- The experimental result analysis reminds us that we need to adjust the capacity configuration of our current implementation with AWS Elastic Beanstalk to realize better resource utilization. The resource requirement needs to be more clarified and more possible patterns should be designed and tested.

- The experiments for examining the alternative solutions should be settled and the experiment result involving a comparison with our current experiment should be reported.

- As reported in our experiment result evaluation, our application is not that suitable to run with the serverless service model. However, there is still no such clear guide with precise metrics telling users when they should go serverless. A study and observation about the conditions of serverless-suitable applications will be beneficial for people who are considering turning to serverless.

Bibliography

[1] https://docs.aws.amazon.com/lambda/latest/dg/welcome.html#related-services. Accessed: 2021-02-20.

[2] https://docs.aws.amazon.com/step-functions/latest/dg/concepts-service-integrations.html. Accessed: 2021-06-13.

[3] https://aws.amazon.com/devops/continuous-delivery/. Accessed: 2021-06-13.

[4] https://console.aws.amazon.com/servicequotas/home/services/lambda/quotas. Accessed: 2021-05-12.

[5] https://cloud.google.com/functions/quotas. Accessed: 2021-05-12.

[6] https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale#service-limits. Accessed: 2021-05-12.

[7] https://lumigo.io/aws-lambda-performance-optimization/how-to-improve-aws-lambda-cold-start-performance/. Accessed: 2021-05-13.

[8] https://towardsdatascience.com/optimize-aws-lambda-memor-more-memory-doesnt-mean-more-costs-51ba566fecc7. Accessed: 2021-05-13.

[9] https://acloudguru.com/blog/engineering/does-coding-language-memory-or-package-size-affect-cold-starts-of-aws-lambda/. Accessed: 2021-05-13.

[10] https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes/. Accessed: 2021-05-13.

[11] https://aws.amazon.com/blogs/compute/choosing-between-aws-lambda-data-storage-options-in-web-apps/. Accessed: 2021-05-13.

[12] https://aws.amazon.com/blogs/devops/using-aws-step-functions-state-machines-to-handle-workflow-driven-aws-codepipeline-actions/. Accessed: 2021-06-13.

[13] *CCSW '09: Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, 2009.

[14] Gojko Adzic and Robert Chatley. Serverless computing: Economic and architectural impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 884–889, 2017.

[15] S Agarwal and Laurie McCabe. The tco advantages of saas-based budgeting, forecasting & reporting. *A Hurwitz white paper, Available: www. hurwitz. com*, 2010.

[16] Mohamed Almorsy, John Grundy, and Ingo Müller. An analysis of the cloud computing security problem, 2016.

[17] Hamada Alshaer. An overview of network virtualization and cloud network as a service. *Netw.*, 25(1):1–30, Jan. 2015.

[18] Sushil Bhardwaj, Leena Jain, and Sandeep Jain. Cloud computing: A study of infrastructure as a service (IAAS). *International Journal of engineering and information Technology*, 2(1):60–63, 2010.

[19] Kevin D. Bowers, Ari Juels, and Alina Oprea. Proofs of retrievability: Theory and implementation. CCSW '09, page 43–54, 2009.

[20] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Commun. ACM*, 62(12):44–54, Nov. 2019.

[21] Bo Chen, Reza Curtmola, Giuseppe Ateniese, and Randal Burns. Remote data checking for network coding-based distributed storage systems. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop*, CCSW '10, page 31–42, 2010.

[22] Richard Chow, Markus Jakobsson, Ryusuke Masuoka, Jesus Molina, Yuan Niu, Elaine Shi, and Zhexuan Song. Authentication in the clouds: A framework and its application to mobile users. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop*, CCSW '10, page 1–6, 2010.

[23] Angelos Christidis, Roy Davies, and Sotiris Moschoyiannis. Serving machine learning workloads in resource constrained environments: a serverless deployment example. In *2019 IEEE 12th Conference on Service-Oriented Computing and Applications (SOCA)*, pages 55–63, 2019.

[24] Angelos Christidis, Sotiris Moschoyiannis, Ching-Hsien Hsu, and Roy Davies. Enabling serverless deployment of large-scale ai workloads. *IEEE Access*, 8:70150–70161, 2020.

[25] Robert Cordingly, Hanfei Yu, Varik Hoang, Zohreh Sadeghi, David Foster, David Perez, Rashad Hatchett, and Wes Lloyd. The serverless application analytics framework: Enabling design trade-off evaluation for serverless software. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, WoSC'20, page 67–72, 2020.

[26] Igor Costa, Jean Araujo, Jamilson Dantas, Eliomar Campos, Francisco Airton Silva, and Paulo Maciel. Availability evaluation and sensitivity analysis of a mobile backend-as-a-service platform. *Quality and Reliability Engineering International*, 32(7):2191–2205, 2016.

[27] Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L Wolf. Naas: Network-as-a-service in the cloud. In *2nd {USENIX} Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 12)*, 2012.

[28] Simon Eismann, Johannes Grohmann, Erwin van Eyk, Nikolas Herbst, and Samuel Kounev. Predicting the costs of serverless workflows. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '20, page 265–276, 2020.

[29] Tina Francis and S. Vadivel. Cloud computing security: Concerns, strategies and best practices. In *2012 International Conference on Cloud Computing Technologies, Applications and Management (ICCCTAM)*, pages 205–207, 2012.

[30] J. Gibson, R. Rondeau, D. Eveleigh, and Q. Tan. Benefits and challenges of three cloud computing service models. In *2012 Fourth International Conference on Computational Aspects of Social Networks (CASoN)*, pages 198–205, 2012.

[31] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, page 95–100, 2011.

[32] Bernd Grobauer and Thomas Schreck. Towards incident handling in the cloud: Challenges and approaches. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop*, CCSW '10, page 77–86, 2010.

[33] Brian Hay, Kara Nance, and Matt Bishop. Storm clouds rising: Security challenges for iaas cloud computing. In *2011 44th Hawaii International Conference on System Sciences*, pages 1–7, 2011.

[34] Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. Standard ISO/IEC 25010:2011, International Organization for Standardization, Geneva, CH, 2011.

[35] Michael J Kavis. *Architecting the cloud: design decisions for cloud computing service models (SaaS, PaaS, and IaaS)*. John Wiley & Sons, 2014.

[36] Jeongchul Kim and Kyungyong Lee. Practical cloud workloads for serverless faas. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 477, 2019.

[37] Gurudatt Kulkarni, Ramesh Sutar, Jayant Gambhir, In Lecturer, Mitra Marathwada, Mandal Polytechnic, Thergoan, and Pune. Cloud computing-storage as service. *International Journal of Engineering Research and Applications*, Vol. 2:pp.945–950, 02 2012.

[38] Aleksandr Kuntsevich, Pezhman Nasirifard, and Hans-Arno Jacobsen. A distributed analysis and benchmarking framework for apache openwhisk serverless platform. In *Proceedings of the 19th International Middleware Conference (Posters)*, Middleware '18, page 3–4, 2018.

[39] Junfeng Li, Sameer G. Kulkarni, K. K. Ramakrishnan, and Dan Li. Understanding open source serverless platforms: Design considerations and performance. In *Proceedings of the 5th International Workshop on Serverless Computing*, WOSC '19, page 37–42, 2019.

[40] Wei-Tsung Lin, Chandra Krintz, Rich Wolski, Michael Zhang, Xiaogang Cai, Tongjun Li, and Weijin Xu. Tracking causal order in aws lambda applications. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 50–60, 2018.

[41] Christian Mainka, Vladislav Mladenov, Florian Feldmann, Julian Krautwald, and Jörg Schwenk. Your software at my service: Security analysis of saas single sign-on solutions in the cloud. In *Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security*, CCSW '14, page 93–104, 2014.

[42] Kalikinkar Mandal and Guang Gong. Privfl: Practical privacy-preserving federated regressions on high-dimensional data over mobile networks. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, CCSW'19, page 57–68, 2019.

[43] Peter Mell, Tim Grance, et al. The NIST definition of cloud computing. Technical Report 800-145, National Institute of Standards and Technology (NIST), Gaithersburg, MD, September 2011.

[44] Shicong Meng and Ling Liu. Monitoring-as-a-service in the cloud: Spec phd award (invited abstract). In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, page 373–374, 2013.

[45] Ajdin Mujezinović and Vedran Ljubović. Serverless architecture for workflow scheduling with unconstrained execution environment. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 242–246, 2019.

[46] Chinthagunta Mukundha and K Vidyamadhuri. Cloud computing models: a survey. *Adv. Comput. Sci. Technol.*, 10(5):747–761, 2017.

[47] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, page 113–124, 2011.

[48] Christian Priebe, Divya Muthukumaran, Dan O' Keeffe, David Eyers, Brian Shand, Ruediger Kapitza, and Peter Pietzuch. Cloudsafetynet: Detecting data leakage between cloud tenants. In *Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security*, CCSW '14, page 117–128, 2014.

[49] Hussachai Puripunpinyo and M.H. Samadzadeh. Effect of optimizing java deployment artifacts on aws lambda. In *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 438–443, 2017.

[50] Alfonso Pérez, Germán Moltó, Miguel Caballer, and Amanda Calatrava. Serverless computing for container-based architectures. *Future Generation Computer Systems*, 83:50–59, 2018.

[51] Peter Sbarski and Sam Kroonenburg. *Serverless architectures on Aws: with examples using Aws Lambda*. Manning Publications Company Shelter Island, 2017.

[52] Joshua Schiffman, Thomas Moyer, Hayawardh Vijayakumar, Trent Jaeger, and Patrick McDaniel. Seeding clouds with trust anchors. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop*, CCSW '10, page 43–46, 2010.

[53] Hossein Shafagh, Lukas Burkhalter, Anwar Hithnawi, and Simon Duquennoy. Towards blockchain-based auditable storage and sharing of iot data. In *Proceedings of the 2017 on Cloud Computing Security Workshop*, CCSW '17, page 45–50, 2017.

[54] Yuquan Shan, George Kesidis, and Daniel Fleck. Cloud-side shuffling defenses against ddos attacks on proxied multiserver systems. In *Proceedings of the 2017 on Cloud Computing Security Workshop*, CCSW '17, page 1–10, 2017.

[55] Alexander Shraer, Christian Cachin, Asaf Cidon, Idit Keidar, Yan Michalevsky, and Dani Shaket. Venus: Verification for untrusted cloud storage. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop*, CCSW '10, page 19–30, 2010.

[56] Duygu Sinanc and Seref Sagiroglu. A review on cloud security. In *Proceedings of the 6th International Conference on Security of Information and Networks*, SIN '13, page 321–325, 2013.

[57] Juraj Somorovsky, Mario Heiderich, Meiko Jensen, Jörg Schwenk, Nils Gruschka, and Luigi Lo Iacono. All your clouds are belong to us: security analysis of cloud management interfaces. In Christian Cachin and Thomas Ristenpart, editors, *Proceedings of the 3rd ACM Cloud Computing Security Workshop, CCSW 2011, Chicago, IL, USA, October 21, 2011*, pages 3–14. ACM, 2011.

[58] S. Subashini and V. Kavitha. A survey on security issues in service delivery models of cloud computing. *Journal of Network and Computer Applications*, 34(1):1–11, 2011.

[59] Erwin van Eyk, Alexandru Iosup, Simon Seif, and Markus Thömmes. The spec cloud group's research vision on faas and serverless architectures. In *Proceedings of the 2nd International Workshop on Serverless Computing*, WoSC '17, page 1–4, 2017.

[60] Erwin van Eyk, Lucian Toader, Sacheendra Talluri, Laurens Versluis, Alexandru Uță, and Alexandru Iosup. Serverless is more: From paas to present cloud computing. *IEEE Internet Computing*, 22(5):8–17, 2018.

[61] Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in xen. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, page 41–46, 2011.

[62] Hao Wang, Di Niu, and Baochun Li. Distributed machine learning with a serverless architecture. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 1288–1296, 2019.

[63] Weichao Wang, Zhiwei Li, Rodney Owens, and Bharat Bhargava. Secure and efficient access to outsourced data. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, page 55–66, 2009.

[64] Jinpeng Wei, Xiaolan Zhang, Glenn Ammons, Vasanth Bala, and Peng Ning. Managing security of virtual machine images in a cloud environment. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, page 91–96, 2009.

[65] Fei Xu, Yiling Qin, Li Chen, Zhi Zhou, and Fangming Liu. λdnn: Achieving predictable distributed dnn training with serverless architectures. *IEEE Transactions on Computers*, pages 1–1, 2021.

[66] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. An exploration of l2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, page 29–40, 2011.

[67] Hsin-Jung Yang, Victor Costan, Nickolai Zeldovich, and Srinivas Devadas. Authenticated storage using small trusted hardware. In *Proceedings of the 2013 ACM Workshop on Cloud Computing Security Workshop*, CCSW '13, page 35–46, 2013.

[68] Masaya Yasuda, Takeshi Shimoyama, Jun Kogure, Kazuhiro Yokoyama, and Takeshi Koshiba. Secure pattern matching using somewhat homomorphic encryption. In *Proceedings of the 2013 ACM Workshop on Cloud Computing Security Workshop*, CCSW '13, page 65–76, 2013.

[69] Aaram Yun, Chunhui Shi, and Yongdae Kim. On protecting integrity and confidentiality of cryptographic file system for outsourced storage. CCSW '09, page 67–76, 2009.

[70] Miao Zhang, Yifei Zhu, Cong Zhang, and Jiangchuan Liu. Video processing with serverless computing: A measurement study. In *Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '19, page 61–66, 2019.

[71] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. Kappa: A programming framework for serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 328–343, 2020.

Appendix A

AWS Lambda Quotas

The full list of AWS Lambda quotas is in the Table A.1.

Quota name	Description	Default Quota Value	Adjustable
Asynchronous payload	The maximum size of an incoming asynchronous invocation request.	256 kilobytes	No
Burst concurrency	The maximum immediate increase in function concurrency that can occur when the functions scale in response to a burst of traffic.	3,000 instances	No
Concurrent executions	The maximum number of events that functions can process simultaneously in the current Region.	1,000 events	Yes
Deployment package size (console editor)	The maximum size of a deployment package or layer archive when uploading through the console editor.	3 megabytes	No

Deployment package size (direct upload)	The maximum size of a deployment package or layer archive when uploading directly to Lambda.	50 megabytes	No
Deployment package size (unzipped)	The maximum size of the contents of a deployment package or layer archive when it's unzipped.	250 megabytes	No
Elastic network interfaces per VPC	The maximum number of network interfaces that Lambda creates for a VPC with functions attached.	250 network interfaces	Yes
Environment variable size	The maximum combined size of the environment variables that are configured on a function.	4 kilobytes	No
File descriptors	The maximum number of file descriptors that a function can have open.	1,024 file descriptors	No
Function and layer storage	The amount of storage that is available for deployment packages and layer archives in the current Region.	75 gigabytes	Yes
Function layers	The maximum number of layers that can be added to the function.	5 layers	No

Function memory maximum	The maximum amount of memory that can be configured for a function.	10,240 megabytes	No
Function memory minimum	The minimum amount of memory that can be configured for a function.	128 megabytes	No
Function resource-based policy	The maximum combined size of resource-based policies that are configured on a function.	20 kilobytes	No
Function timeout	The maximum timeout that can be configured for a function.	900 seconds	No
Processes and threads	The maximum combined number of processes and threads that a function can have open.	1,024 processes and threads	No
Rate of control plane API requests (excludes invocation, GetFunction, and GetPolicy requests)	The maximum number of API requests per second (excluding invocation, GetFunction, and GetPolicy requests).	15 API requests per second	No
Rate of GetFunction API requests	The maximum number of GetFunction API requests per second.	100 API requests per second	No
Rate of GetPolicy API requests	The maximum number of GetPolicy API requests per second.	15 API requests per second	No

Synchronous payload	The maximum size of an incoming synchronous invocation request or outgoing response.	6 megabytes	No
Temporary storage	The amount of storage space that is available to a function in the /tmp directory.	512 megabytes	No
Test events (console editor)	The maximum amount of test events for a function through the console editor.	10 events	No

Table A.1: AWS Lambda Quotas Full List