Expertise
and insight
for the future

Muhammad Hassan

Public Cloud-Based Private Python Package Serving Platform

Metropolia University of Applied Sciences

Master of Engineering

Information Technology

Master's Thesis

April 7, 2021

PREFACE

This thesis includes a study and implementation of cost efficient, scalable, serverless and secure environment development of Python package repository which is typically hosted on a virtual server. I was assigned to this project as a result of internal product development for our Data Science team at our company CGI Suomi Oy. In order to meet the requirement of the product I had a chance to explore different serverless offerings of AWS public cloud followed by the integration of those services to design an end-to-end system. I have learned a lot of new ways of reusing the services that I have already been using previously in many projects.

I would like to show my gratitude to the Lord of earths and heavens, Allah the Almighty, who created and given us the purpose to understand Him, His creations by exploring, experimentation and expeditions. I would like to thank my family members for their prayers and motivation that helped me in pursuing my educational as well as professional goals throughout my life. I am also very thankful to Ville Jääskeläinen for supervising this thesis.

Let’s add value for the fellow humans by helping one another by using science and technology for good and hope our technological advancements would bring joy and ease to the people around us.

Helsinki, Wednesday, July 3, 2019
Muhammad Hassan

Author Title Number of Pages Date	Muhammad Hassan Public Cloud-Based Private Python Package Serving Platform 26 pages + 1 appendices 7 April 2021
Degree	Master of Engineering
Degree Programme	Information Technology
Instructor(s)	Ville Jääskeläinen

This thesis provides an overview of a solution that gives the ability to privately host a Python package distribution solution. A public Python package hosting can be implemented by a PyPi server and if there is a need for a private Python package distribution, the same PyPi server is hosted in company's own datacenter or on public cloud with restricted access within the company's network. To run the private package hosting on bigger scales and to meet high requests the deployment must be highly available and scalable.

This type of environment is very hard to manage and maintain and needs continuous operations to keep the virtual server up to date with patching and other necessary maintenance tasks. Some companies have high security requirements and need fine grain access controls on how one can download those privately hosted packages with request throttling and request limits. Also, those virtual machines hosting solutions need to run 27/4 even if there are no requests.

In the context of this thesis, a working, production ready serverless solution was implemented to host and serve Python packages privately on AWS public cloud using managed services in order to minimize the operations and running costs. Moreover, the solution was implemented with security features enabling package installation protected by access keys. The solution also implemented an end-to-end continuous integration and continuous deployment pipeline for upgrading packages and documentation. A deployment of the whole infrastructure as code is beyond the scope of this thesis and could be a future enhancement of this implementation.

The result of thesis shows that the serverless implementation is cost efficient and easy to scale to support millions of requests in a secure manner by using managed services provided by AWS public cloud as compared to self-hosted PyPi server on virtual machines.

Keywords	AWS, PyPi server, virtual machines, serverless, cloud computing, AWS Lambda, AWS S3, AWS Api Gateway, high availability, scalable, Python, package hosting

“Make everything as simple as possible, but not simpler.”

Albert Einstein

Table of Contents

List of Abbreviations

AAS	Advanced Analytics Solutions
API	Application programming interface. A software implementation that is used to regulate communication and data exchange between many software implementations.
AWS	Amazon Web Services
AWS EC2	Amazon Web Services Elastic Compute Service (Services that is used to provisioned bare metal virtual machines)
AWS RDS	Amazon Web Services Relational Database service
CD	Continuous delivery
CGI	Canadian global information technology consulting
CI	Continuous Integration
FaaS	Function as a Service
IAM	Identity and Access Management (AWS)
IaC	Infrastructure as Code
IaaS	Infrastructure as a Service
JSON	JavaScript Object Notation
POC	Proof of concept
PaaS	Platform as a Service
PyPi	The Python Package Index
RDBMS	Relational Database Management System
S3	Simple Storage Service (AWS)
SaaS	Software as a Service
Serverless	A managed service where all IT related services like provisioning, scaling and other operational tasks related to server management and security patching are done by the data centre service provider.
USD	United States Dollar (Currency)

1 Introduction

This thesis is about research and implementation of a service-less cloud computing project that is used host Python packages as a private repository. In the introduction the motivation and background information are shared behind the overall paradigm of cloud computing and to be more serverless computing. The aim of the study and thesis structure is also explained at the end of this section.

1.1 Background and Motivation

Technological advancement has huge effects on human race and on all fields of sciences. With the development and advancement of transistors and data storage mediums now companies can enjoy huge processing power at their disposal. With the further development of cloud computing based data centres, the upfront cost in infrastructure has practically dropped to zero.

Before the start of 2000 companies with huge budget were only able to start very big projects, having their own data centres but most of the actual usage of the investment was very minimal. This was because resources in their own data centres were always provisioned to be able to handle the unknow spikes in the application usage, but most of the times those hardware resources were unused. As the resources even within the company were not shared between the projects hence wasting a lot if processing power and storage space.

Going throw the same dilemma The AWS platform was launched in July 2002. AWS is the cloud computing platform this is hosted by Amazon. Primarily, the platform was used for hosting servers and related infrastructure for amazon.com. Amazon soon realized the waste of processing power and storage that was extra to their own usage, hence they started selling that extra compute and storage capacities over the internet to other companies.

Figure 1. Magic Quadrant for Cloud Infrastructure as a Service, Worldwide

Source: Gartner (July 2019)

Figure 1 Gartner competitors in the cloud computing IaaS [1]

This idea has revolutionized the IT industry, on premises data centres and every other business related to that. All that happened just because of sharing the extra wasteful resources with others over the internet, and this became the beginning of the new era of public cloud computing. The paradigm was then followed by many other big giants in IT like Google, Microsoft Azure and last but not the least Alibaba. There are now many competitors in the cloud computing infrastructure-as-a-service department, but majority of the market is led by those names as Figure 1 illustrates.

The paradigm of acquiring the compute capacity from a shared pool of resources when you need those resources and then releasing back when you have no further need had a huge impact on the cost savings, meeting large processing demands during the peak hours of application usage hosted on public cloud computing platform. All that happened by introducing the concept of pay as you use, billed by number of seconds and minutes use of the processing and amount of data storage over a period of time, no upfront costs, no investments.

Companies like Amazon, Google and Azure not only started selling compute and storage capacity but also started selling managed services for application hosting, data storage, data processing and many of other services for handling many different use cases in almost all of the industries, in a very secure public cloud environment. The security, scalability and high availability of the services deployed on those platforms is achieved in many different ways and will be explained later in this thesis.

1.1 CGI Company Use Case

CGI, is a Canadian global information technology consulting, systems integration, outsourcing, and solutions company headquartered in Montreal, Quebec, Canada.

CGI Finland has data science team who is providing Artificial intelligence and machine learning related consultancy to the clients here in Finland. Our data science team has developed a custom Python package this is integrated with client's machine learning workflows to build models. The challenge of hosting private package hosting repository for ScienceOps package deployment. The main aim of the deployment is to have the package available round the clock to be downloaded and integrated inside the machine learning computations and model building. The existing solution was deployed by hosting PyPi artifactory on an AWS EC2 virtual machine. The solution previously used to deploy the package building hosting repository was mainly based on IaaS based model and therefore it was decided to start moving to more managed services. The benefits of using serverless architecture for this kind of workflows has clear benefits as the package hosting need not to run on the web server all the times, but when the request comes to download the package only. This model will save us operational and deployment costs, reduced server management overhead, easy scalability and out of the box security. With traditional self manged server, patching security updates and updating all the packages and utilities installed on the server is a hectic task and could be avoided by switching to serverless offering of the cloud provider. The benefits of serverless over own managed servers is discuss in the next section of this thesis.

1.2 The Aim of Thesis

This thesis evaluates the managed services provided by AWS to make the Python package hosting easier to manage, deploy and scale. As the previously deployed solution was based on IaaS model the other aspect was gaining more knowledge about the pros

and cons of IaaS vs PaaS implementations, identifying the problems with the previously designed solutions, its scalability, cost and high availability and then sharing this information internally with the colleagues hence increasing the team knowledge.

The thesis also aims to introduce manged services provided by Amazon Web Services Cloud platform, best practices for using those services and then deploying and manging highly scalable and available solutions in the cloud platform. This thesis has only improved the package hosting part of the Python package part of the previously deployed solution. As there are three parts of this deployed the first one is Python package building, package hosting and package serving securely.

This thesis has left the first part of building packages on the previously deployed solution however improved the latter two parts of the old deployment hosting and serving. The solution developed as part of this thesis work is deployed using online web browser-based point and click method to make the proof-of-concept deployment as quick as possible. Different services from AWS managed pool were used to deploy the overall solution. The package building phase is run on Jenkins machine hosted in our AWS environment, this package and related documentation is then uploaded to AWS Simple storage service. Once the package is uploaded automatic serverless compute service AWS Lambda is triggered to cache the new paths of the package and update the AWS DynamoDB a NoSQL database service. The requests for new packages and authentication are handled by AWS managed service API Gateway and package is served on successful authentication. The detailed architecture and implementation details are in the section 3 of this thesis.

This thesis does not include the deployment of the whole solution of whole solution using infrastructure as a code deployment paradigm because of time, budget and other resource constraints. The solution deployed can be access via command line for installing the Python packages securely protected by access keys and the documentation for each version of the Python package is hosted in AWS Simple Storage Service S3 and protected by username and password.

1.3 Structure of Study

This thesis is divided into following four parts. The first part explains the motivation and background of the thesis, the aim of the study and structure of the thesis.

Second part of this document explains the history of Cloud computing, open source model and difference and benefits of serverless based application over own deployed solution on top of plain vanilla computing infrastructure i.e. AWS Elastic Computer service.

Third part of this document summarizes the old implementation of the project that is deployed with IaaS components. This section also has details regarding newly deployed service using AWS Serverless components. This section contains system architecture, implementation and deployment details for the proposed solution. Results of the implementation are also presented explaining the cost, scalability and security wise difference between the new and old implementation.

Fourth part and the final part of this document explains the conclusions and further research. This part also contains suggestions about the possible deployment paradigms of the implemented solution so that the whole solution could be deployed using infrastructure as a code concept. There are two main frameworks that could be employed for this purpose that are explained in more details in this section. Furthermore, suggestions for using AWS managed services for building the package building part of the pipeline are also documented in this part of the thesis.

2 Open-Source Model and Cloud Computing

It would be interesting also to discuss the role of open-source model and its impact on adopting public cloud platforms. The using open-source software and application is free to use for personal and commercial purposes, a vast majorizes of companies are moving towards adapting open-source frameworks and software. Inspired by the adoption of the open-source paradigm; cloud computing vendors have also made possibility of using that software and provided them as managed services.

There is a specific terminology used called open-source cloud; a cloud either public, private of hybrid cloud that is built and developed using open-source technologies. However, today many of the managed services provided by cloud vendors are proprietary and more advanced and improved version of those open-source software and frameworks. Companies has spent millions of dollars developing those products and improving the pain points in open-source software or frameworks. For example, Amazon provides MySQL, an open-source RDBMS (Relational Database Management System) as a managed service in one of its AWS RDS (Relational Database Service) offerings. In addition to that Amazon also provided, its own ground up built cloud native relational database system called AWS Aurora that has performance and high availability like a commercial grade RDBMS but at 1/10th of the cost.

2.1 Open-Source Model

Open-source model is way of creating software that emphasis on open collaboration between the peers working on the product. The development of the software under open-source model is usually decentralized. The source code is contributed by different peers and is publicly available to anyone for personal as well as commercial use depending on the term of the license under which the software project is released. In addition to that documentations, reference usage architectures, blueprints are also freely available.

The code that adds the functionality or that fixes any bug is mutually reviewed by many peers before accepted into the main software distribution. As any open-source project get famous and more and more people starts using it, a community of experts is then there to help you solved different problems that you might encounter while using that

project. Figure 2 shows characters of open-source projects, as those are more reliable, cost effective, secure and flexible.

Figure 2. Characters of open source projects. [2]

2.2 Cloud Computing

Cloud computing is a very fancy and buzz word these days. Cloud computing is a word given to this whole concept of sharing compute, processing and data storage resources over the internet. Companies provide different ways in which third parties could acquire those shared resources over the internet, usage them and pay as long as they are using those resources and then return back those resources to the shared pool of resources when they no longer want to use them, hence they will stop inquiring any charges.

An analogy of the paradigm could be easy to find in power and electricity industry. Nobody would like to run and invest in building a powerhouse just for the sake of providing electricity to their own home as the investment is really high and you are not certain with the usage. Sometimes the electricity use will be high, let's say in summer when you are running air conditions while it could get very low in winters. The same concepts lie in the any industry for the procuring IT resources. The usage and fluctuation in the users make it very unpredictable to invest in IT infrastructure to host software applications. Hence now instead of investing in their own data centres, companies tend to use cloud computing based infrastructure, scale up and down as per their needs and save huge amounts of money both in initial investment and running costs.

Hence, cloud computing is basically a practice of using internet connected remotely deployed and manged compute servers, network devices and storage layers for storing, analysing, processing and managing data over the internet instead of local servers as shown in figure 3.

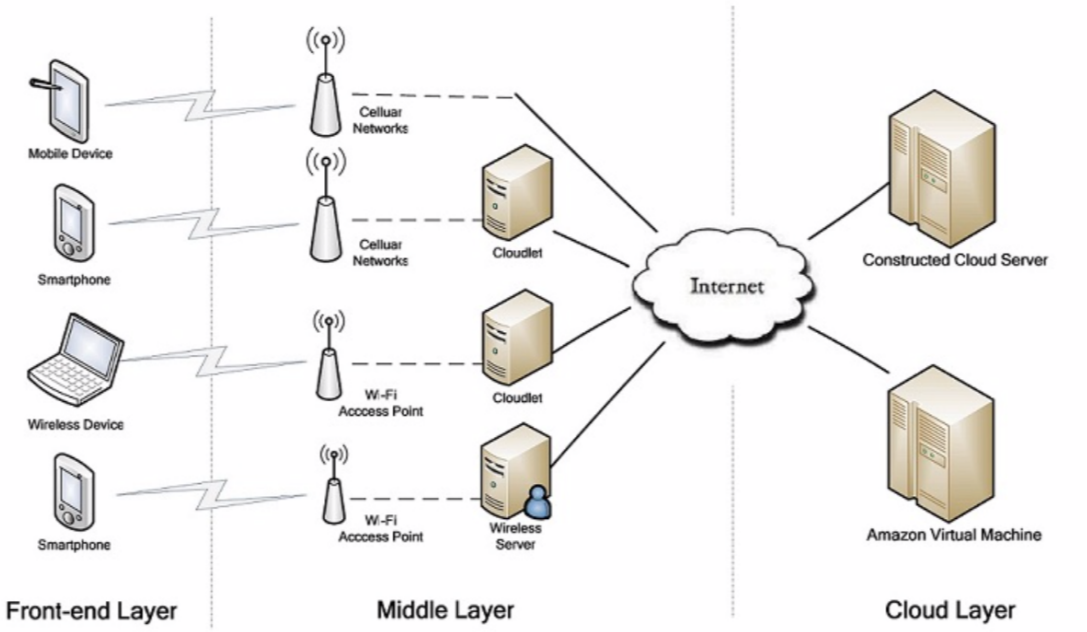

Figure 3: Cloud Computing architecture [3]

On the other side companies providing cloud computing services have designed different ways to rent out their computing, processing and storage resources in order to make the usage of their environment easier and more adoptable. Those different level of services mainly includes IaaS (Infrastructure as a service), PaaS (Platform as a service) and last but not the least SaaS (Software as a service).

Those three layers of services provided by cloud computing companies have different characters those could be accessed before choosing which is best for deploying applications. IaaS model offers very low level of access to the underlaying processing, compute, storage and network connectivity services. Customers are free to mix and match different types of core services to design their own platforms and solutions.

However, if comes with different scope of responsibilities on the customer's end. Like upgrading and patching the underlaying operating systems, installing firewalls, access

and control management, keeping installed software up to date etc. Platform as services gives a little bit more freedom to the customers. The underlaying infrastructure is already deployed and manged by the cloud provider on the behalf of the customers and the customers have only need to deploy the apps on top of that infrastructure. For example, AWS has ECS elastic container service as PaaS platform where AWS manages the virtual machines and networking and infrastructure for hosting and running container images like Docker. Software as a service is on the top and customers requires least effort to use those services. Those services have software installed, managed and provisioned and user have to sign up for the service and use it. Best example for SaaS is Google Docs, where users just sign up for the services, create docs and they do not have to worry about the underlaying applications and infrastructure, and everything is managed by the cloud provider like storage of those documents, high availability, access and management.

Based on the different level of expertise, time constraints and budgets, customers have all options to choose from different levels of services to make their work as quick and as easier as they want. If companies have more resources and experts, data security requirements they prefer to manage everything on their own, deploy their own apps and have fine grained control of all aspects of the applications, on the other hand companies how want to do less management they prefer to choose the managed services provided by the cloud provider.

2.3 Serverless Computing and Managed Services

Serverless computing service is sometimes confused with a service that is running without any servers. This can never be true as for a service to run there must be some underlying web servers or application servers running in order to fulfil user's requests, perform some actions, stores or retrieve data and return information or data to the requester. With serverless architecture paradigm all operational responsibilities are shifted to the cloud data center provider.

For the implementation of the proposed project a similar paradigm is used. Amazon web services provides many serversless services where users do not have to provision any servers, neither one has to configure the networking and monitor the storage. With serverless and manages services compute, storage and network bandwidth scales up and down with the increase or decrease in requests respectively. In order to use the serverless services users just had to specify the memory, CPU and sometimes initial storage

and rest of the provisioning, network configuration, access management, patching latest security updates, backup and restore, ensuring high availability and scaling of the service is handled by the service provider. Figures 4 shows high level benefits of using serverless architecture for deploying applications.

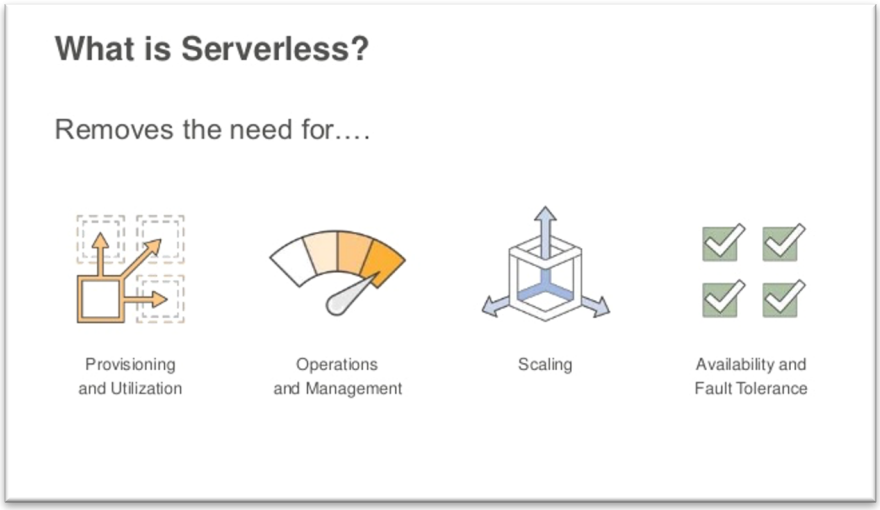

Figure 4: Benefits of serverless services [5]

On a very high level serverless architecture provides several benefits that are described in following subsections.

2.3.1 Provisioning and Utilisation

With serverless deployment mode resource provisioning and over or under utilisation is no more blockage for adoption of certain technology. Previously in order to use particular software services companies have to hire experts to confirm and provision the software solutions to make them usable for a company. Now with serverless architecture everything is preconfigured and parameterised and before using a service initial capacity is chosen that can be scaled automatically. For example Apache Kafka is an open source software that is used to ingest very large quantity of streaming events from Web servers, mobile phone, click stream data etc and in order to deploy and make this service highly available a lot of expertise is needed before making it useable in production environments.

However, with similar but serverless offerings Amazon launched AWS Kinesis a highly available readily scalable serverless services that can be used to ingest millions of data events per second and required very little provisioning for companies and behind the

scene everything is managed by the cloud service provider. Utilisation of serverless and managed service is also easy and requires only few clicks to deploy those services, and users can use SDKs in many different languages to push the streaming data to the service.

2.3.2 Operations and Management

When companies have to deploy open source or commercials tools they have to do all the heavy lifting: provisioning servers, running security patch management, installing tools and dependencies, configuring networks, data back and restore, disaster recovery and all such activities that comes with deploying infrastructure on bare metal virtual servers. This requires a lot of operational cost and firefighting. Serverless solution have the benefit of delegating all those operational and management activities to the cloud vendor and companies only need to focus business logic.

2.3.3 Scaling

Serverless solutions have the elasticity of the cloud built into the offering. Elasticity is the capability of the service to handle as many requests as it gets by scaling the underlying infrastructure to thousands of servers. The companies using those services only pay for the requests or for the duration those services were used and only cost for the storage or networking bandwidth consumed. Companies do not have to worry about the scaling of the services during peak usage of the company's products as those are deployed on top of serverless architecture.

2.3.4 Availability and Fault Tolerance

High availability of the application is highly desirable as the business revenue is at the stake if an application or service is not accessible. Most of the companies providing software-based solutions have some sort of Service Level Agreement (SLA), that states how often the service of application could be inaccessible. As serverless based application deployment is highly scalable and available because the underlying infrastructure is deployed in multiple data center with redundancy of the servers hosting the serverless services. If for some reason one data center is down and cannot host certain serverless application, then another backup server is already up and serving the requests.

Hence the business application becomes highly available and scalable managed infrastructure in the form of serverless service where they can deploy their applications. There

are many serverless services provided by AWS and can be configured with few clicks. For example, AWS S3 simple storage service is a serverless and managed object storage service. Designed for durability of 99.999999999% of objects across multiple Availability Zones and 99.9% availability over a given year [7].

AWS S3 was also used in the implementation of this thesis. Python package that was generated by Jenkins servers was first uploaded to AWS S3 bucket and from there it was made available and served to end users by using AWS API Gateway.

	S3 Standard	S3 Standard-IA	S3 One Zone-IA	Amazon Glacier
Designed for Durability	99.999999999%	99.999999999%	99.999999999%	99.999999999%
Designed for Availability	99.99%	99.9%	99.5%	N/A
Availability SLA	99.9%	99%	99%	N/A
Availability Zones	≥3	≥3	1	≥3
Minimum Capacity Charge per Object	N/A	128KB*	128KB*	N/A
Minimum Storage Duration Charge	N/A	30 days	30 days	90 days
Retrieval Fee	N/A	per GB retrieved	per GB retrieved	per GB retrieved**
First Byte Latency	milliseconds	milliseconds	milliseconds	select minutes or hours***
Storage Type	Object	Object	Object	Object
Lifecycle Transitions	Yes	Yes	Yes	Yes

Figure 5: AWS S3 storage classes [7]

AWS S3 has many storage classes that could be used to minimize the storage cost based on the use case. Figure 5 shows different storage classes with service level information. In this project standard storage class is used. These services will be explained mores in the implementation details.

3 Project Specifications

The goal of this project was to design, develop and launch a serverless solution for Python package hosting based on AWS public cloud. Based on the internal stakeholder's surveys and interviews, functional and non-functional requirements and criteria for technology selection was made. This section describes all those specifications that influenced the selection of particular technology and services offered by AWS public cloud. This section contains both non-functional requirements and functional requirements

3.1 Non-functional Requirements

Following non-functional requirements were agreed with the internal stakeholders. These requirements consist of both non-runtime and runtime qualities.

Scalability

Scalability of the system was very important. The system should be able to function as expected despite the increase in the traffic load and size of the package serving. The system should add more nodes or backend infrastructure and should distribute the load across those processes to meet the user needs. As this application will be shared with many clients so each client and its developer should be able to access the system

Maintainability

The deployed system should be highly maintainable. The system should be modularized, and each component must perform only one action. It should be very easy to add new features, enhancements and should be easy to debug errors and perform bug fixes. Each module should send logs separately for easy debugging. The components should be parameterized where possible. The system should be able to serve Python packages regardless of where the package is built, and it should be portable. The system should also make new code releases to the Python packages easily and the automatic pipelines should push the new changes to the package.

High availability

The package serving service must be highly available during the weekdays and business working hours at least. The system should be designed in such a way

that even in case of data centre outages the system should serve the users requests. The system should also be resilient to the component failure and disaster scenarios. The system should also distribute the packages and documentation using content delivery networks where applicable.

Audit logging

Auditing of the system is really important with respect to security, operations and performance. Each component of the system should be capable of logging audit entries to the central logging place and can be transformed into reports when needed. System should log all the requests to the system and API keys used by the clients to download packages.

Data management

The Python packages served by the system must be stored correctly and data for the previous versions of the package along with documentation must be maintained by the storage layer irrespective of the service or component failure. Data life cycle policies for the previous version of the packages should be enforced. Data backup and restore processes should also be defined.

Security

As the system is only intended to service the Python packages to the private clients, proper authentication and authorization measures must be in place for both documentation access and for downloading the packages. Users must login to access the package version documentation. Package serving must also be protected by API access keys and package download only be authorized for valid API keys from the client's machines. Each client company should be provided by a valid API access key that must be used in order to download the Python packages.

Performance

The required throughput of the system should be parameterized and configured based on the contract with the client. The mechanism to throttle the request should be in place and performance of the system should be configured based on the number of downloads allowed by each client. The throughput requirements for each client's API access key should be enforced separately and system

should be capable of administering a given number of packages download transactions with a given unit of time i.e. number of downloads in a day or in a month.

Legacy integration

As the previous system is building the Python packages using old system, the package serving system must be able to integrate with a legacy package building server. The Python packages should sync from the old system to the new package serving system without any major updates to the previous running solution.

Interoperability

The system must be modularized, and each module should perform its own tasks, with possibility to interest with each other and without causing an interference. Each module should have its own resources, CPU, Memory and should not be limited to scale and be dependent on other components in term of resources. Each module should be deployed separately as a microservice.

3.2 Features and Functional Requirements

Following table contains the functional requirements and key futures of the system. These features are within the scope of this project.

Functional Req#	Functional requirement description
FR 1	User should be able to download the Python package hosted on the platform using pip command line interface
FR 2	The system should validate the package downloading request with a valid secret key in the url
FR 3	The system should allow administrators to generate secret keys for downloading packages
FR 4	System should should allow to view the package documentation in the web browser after authentication
FR 5	System should allow to add users for viewing documentation using a username and password
FR 6	System should be able to host multiple versions of the Python packages, and latest version of the package should be available as soon as the package is built and published

FR 7	System should be allowing some mechanism to rollback the packages to previous version either manually or automatically
FR 8	System should be able to host and serve the package using HTTP/HTTPS endpoint
FR 9	System should send package building status and notifications to slack or any other team's communication channel.

The table above defines the functional aspect of the package building service. These identified functional use cases present the holistic picture of the solution that will be deployed using the public cloud service.

As Amazon web services public cloud was chosen for implementing the solution. The services offered by the cloud vendor were chosen so that they will achieve the target of implementing serverless architecture. In the next part of this thesis, project architecture is explained along with the implementation details on the usage of different managed services as technical stack of this project.

4 Implementation and Results

AWS has number of serverless offerings ranging from serverless compute, serverless SQL and NoSQL to serverless networking services. In the implementation of this project comprehensive study of those services was conducted and best services were selected in order to implement the architecture.

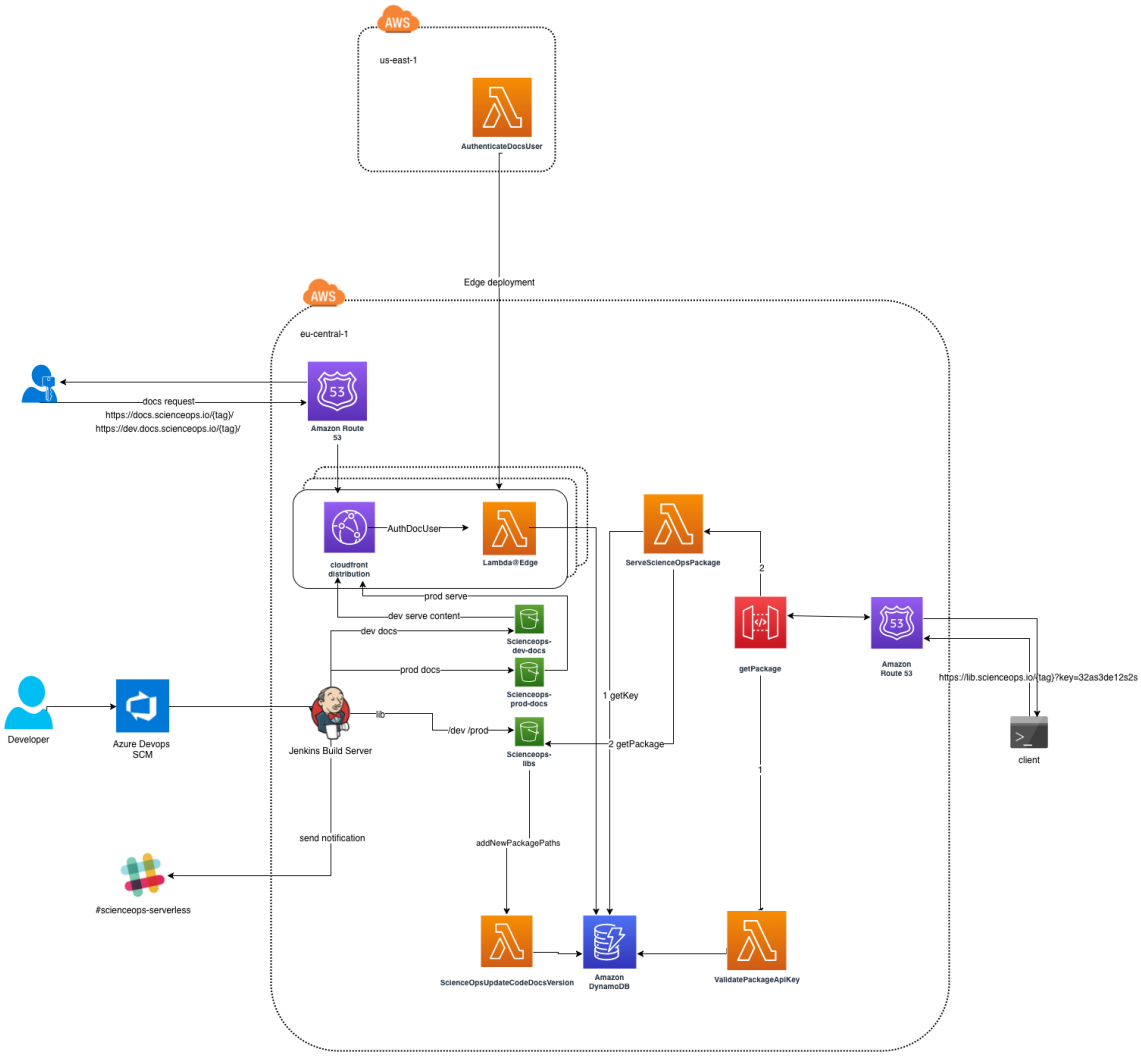

Figure 7: Project solution architecture using AWS Public Cloud Services

The implementation of project was divided into three main phases: artifact/package building and package hosting and versioning and finally package serving. The project scope included mainly the last two phases. The server for the first phases of the project was reused from the previous implementation of the package services happening through JFrog artifactory solution that is not using a serverless model but self-hosted framework.

The first phase of the project was chosen from the previous solution in order to speed up the implantation of the project and it was planned for the future improvements of the project to reimplement the package building using serverless computer as described in the section 5 of this thesis. However, it is important to explain briefly what services and frameworks were used in order to build the Python package. The actual Python package code was not allowed to share in the context of this thesis because of the copy rights issue, however this solution works for any Python package and architecture could be extended for any artifact building, hosting and serving over HTTP/HTTPS protocol. Following section contains the technical implementation and technology selection details and criteria for different phases of the project.

4.1 Package and Documentation Building

Package building was the first part of the full pipeline for end-to-end delivery of the package. The package building process start when a developer pushes the code to the Azure DevOps. The Azure DevOps was integrated with Jenkins. Jenkins is an open-source tool for building continues integration and continues development/delivery pipeline. Jenkins is installed on the AWS side using ec2 machine.

There are many ways to installed Jenkins server on virtual machines using single or multiple data centres [10]. This ensure the reliability of the Jenkin server in case of data-centre failure. Source control inside the Azure DevOps service was configured using web-hooks to trigger Jenkins pipeline to start the code fetching, building and publishing the code to the next stage of the process. In order to integrate Azure DevOps with Jenkins server “VS Team Services Continuous Development” plugin was installed. Figure 8 shows searching the plugin and installing it.

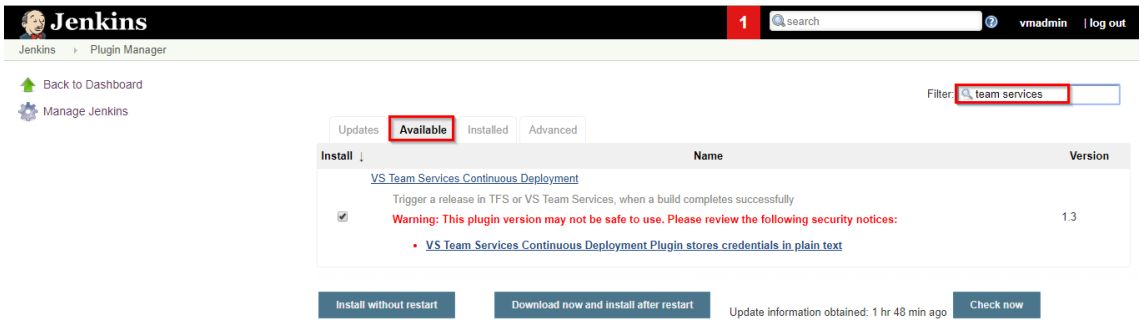

Figure 8: Installing integration plugin. [11]

Once the Jenkins server was installed along with the plugins and integration between Azure DevOps and Jenkins was established, we created a Jenkins groovy template file was created in order to build and push the Python package on to the second phase of package hosting.

Jenkins provide us with point-and-click method using a web browser-based interface to configure and run CI/CD pipelines as well as possibility to write groovy template file. The template file with name Jenkinsfile was saved in the default source control repository. Once the pipeline was triggered from the source changes on the Azure DevOps side Jenkinsfile was automatically parsed by Jenkins server and commands were executed as stated in the different build stages of the pipeline specified in the template file. Jenkinsfile has a proper syntax and template format that could be modified as per build process requirements [12].

Appendix B contains the template file used in building phase of our project. Jenkins pipeline contains different stages to build the artifact, build automatic documentation of the artifact, uploading the documentation and the production artifact to the hosting platform of the solution. For hosting the package on the AWS, S3 (Simple Storage Service) was used. AWS S3 is a managed service provided by Amazon for object-based storage using web service interface [13]. AWS S3 is highly available, reliable, scalable and provides 99.999999999% (11 9's) of Durability. AWS S3 provides different kinds of storage classes and for the simplicity of this project standard S3 storage was selected for hosting the artifact.

All AWS services require authentication and authorization using AWS IAM Service (Identity and access management). As our virtual machine running Jenkins service was running inside AWS environment, all necessary permission to other services were provided using AWS IAM Instance profile [14].

Jenkins pipeline has built the package as well as the documentation for the package, both package and static documentation of the package is uploaded to s3. Jenkins has three different pipelines for building and uploading documentation and actual artifact to dev, test and prod buckets configured in the pipeline as environment variables. The documentation is hosted using AWS S3 public hosting feature. AWS S3 provided managed service to host websites and content and server them using the edge distribution network using AWS CloudFront service. Detailed step-by-step instructions for configuring and serving content from AWS S3 bucket documented on AWS documentation site [15]. For Python documentation generation Python pdoc package was used to generate static documentation for the modules and functions [16].

In AWS S3 bucket was created with the name scienceops-libs where all the packages from dev, test and prod Jenkins pipeline were uploading to /dev /test and /prod sub directories inside the root of the bucket.

Once the package was built and uploaded successfully to AWS S3 service the hosting pipeline was trigged for indexing the artifact and updating different links to the document and version of the system as explained in the second phase of the package serving.
AWS S3 provides triggering features that were used to trigger the hosting and indexing using AWS Lambda functions. AWS Lambda functions is a serverless computer service that can be triggered against the data upload events inside AWS S3. AWS Lambda is function-as-a-service and can be coded in different programming languages without provisioning any servers [17]. All the versioning and indexing information with tags and links were saved in a NoSQL database called AWS DynamoDB

Build job status was communicated back to the application team using Slack channel. SlackSend plugin was installed on Jenkins server and after every build, the build job status was communicated via message formats provided by Slack API to the developers [18]. Once the plugin was installed messages were sent using the following command within the groovy script.

```
    slackSend color: "danger", message: "Job: ${env.BUILD_URL} was
    failed"
```

After the Python package and documentation was uploaded to AWS S3 storage service, the package hosting pipeline was triggered to update the links.

4.2 Package and Documentation Hosting

Once the package and documentation were uploaded to AWS S3 Service, A lambda function got triggered in order to update the documentation and package version links inside the AWS DynamoDB. AWS S3 Service provides trigger feature [19] that was configured in such a way that when a package is uploaded in a particular path structure inside the AWS S3 bucket it ran AWS Lambda function appendix A 5 ScienceOpsUpdateCodeDocsVersion index.js. AWS DynamoDB table had two columns path (env + latest or version number), object_key (full path to the AWS S3 Python package object uploaded from Jenkins server) and upload_time (insertion request time). The lambda function updates two items inside the DynamoDB table first for the object_key for the latest path to the current triggered path and secondly it adds the current version and path mapping inside the table scienceops-lib-versions. Path was extracted from the S3 triggered event.

AWS Lambda function had permissions to update paths inside the DynamoDB Table [20]. Following JSON object shows DynamoDB table item structure.

```
{
    "path" : "(dev/test/prod)/(version/latest)/ ",
    "object_key": "(dev/test/prod)/(version)/scienceops-(version).tar.gz",
    "upload_time":"Tue Mar 09 2021 21:09:48 GMT+0500"
}
```

On every upload of the Python package from the server to S3 the latest object_key and path was updated and a new version for object_key and path was registered. This design was chosen to handle both the cases ; if a client requested the latest package so the serving module could quickly fetch the object_key for latest path or versioned path if requested. Package documentation was also uploaded to AWS S3 bucket Scienceops-(dev/prod)-docs from the Jenkins server. Two buckets were used in order to host static html documentation generated for the development and production packages separately.

4.3 Package and Documentation Serving

Package and documentation were served from AWS S3 storage service following two different paths. Package was served via REST API using AWS API Gateway [21] a managed service while documentation was served using AWS CloudFront content delivery network distribution [22]. A custom domain name was registered using AWS Route53 Service for serving the documentation [23].

One of the requirements for serving documentation using AWS S3 was implement authenticated and authorized access to the documentation. In order to meet the requirement AWS Lambda function feature Lambda@Edge was used. The feature was generally available in the US East (N. Virginia) Region (us-east-1). [24]. Username and password were used to generate basic auth generate string that that information was saved in AWS DynamoDB table with primary key as authString. Following code in NodeJS was used to generate that auth string. The original username and password were shared with the customer to use in the login process.

```
    const authString = 'Basic ' + new Buffer(authUser + ':' +
authPass).toString('base64');
```

Appendix A.1 LambdaAuthenticateDocsUser AWS Lambda function was deployed to be used as Lambda@Edge function [25]. The function intercepted all the requests made to the url http://docs.scienceops.io/{tag}. Tag was either latest or the version number of the documentation. Once the lambda@Edge was deployed and configured with the AWS S3 bucket used for the documentation, a popup dialog [figure 9] was shown to the user automatically requested username and password. The username and password were used to create the basic auth string automatically and that information was passed in authorization headers, and that basic auth string was compared with the value saved in the AWS DynamoDB table in order to validate the credentials and access permissions.

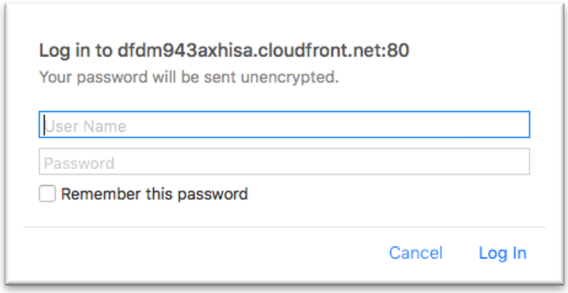

Figure 9 https://douglasduhaime.com/posts/s3-lambda-auth.html

Python package serving functionality contained two modules developed using AWS Lambda functions; one for package serving and other of authenticating the request made for package. Package fetch command was initiated using pip, pip is the package installer for Python. You can use pip to install packages from the PythonPackage Index and other indexes. It is installed automatically when Python is installed [26]. Following command was used to get the package on local development machines or on the production systems.

```
pip install https://lib.scienceops.io/{tag}?key={client-access-key}
```

Package serving part of the system was implemented using AWS API Gateway Service.

“ Amazon API Gateway is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. APIs act as the "front door" for applications to access data, business logic, or functionality from your backend services. Using API Gateway, you can create RESTful APIs and WebSocket APIs that enable real-time two-way communication applications. API Gateway supports containerized and serverless workloads, as well as web applications.” [27]

In order to serve package using AWS API Gateway the raw bytes are returned because the Accept header is set to a binary media type of application/octet-stream and binary support is enabled for the API Gateway as well as headers response of the lambda function used in fetching package AWS S3 Object and returning it to AWS API Gateway.

```
"Content-Type": "application/zip, application/octet-stream",
```

Once the client made the request the request was received at the AWS API Gateway. The request had an access key in the query string. The access key was extracted from the url query string using custom Lambda authorizer function [28] Appendix A 3 LambdaValidatePackageApiKey and was matched with keys in AWS DynamoDB table “sci-

enceops-access-keys” and on successful match the request from the AWS API Gateway is forwarded to the lambda function Appendix A 2 LambdaServePackage. The lambda function requested the information for the object path from the AWS DynamoDB table “scienceops-lib-versions” using path and stage. The stage value was fetched and passed on from the AWS API Gateway as the API was deployed in different stages like dev, test or prod [29]. Once the object was received inside the AWS Lambda function from AWS S3 the object was sent back to the AWS API Gateway and from there the client requesting the package using pip successfully installed the package. Custom domain name was also registered for the AWS API Gateway with a sub domain to access the Python packages [30].

The serverless deployed solution outperformed previously deployed virtual machine based deployed solution in cost, security and maintenance. The serverless package hosting solution has no need for administration of the deployed infrastructure. AWS Lambda function required minimum configurations and deployment boiler plate code as compared to AWS EC2 based server deployment. EC2 based solution required first provisioning of virtual machines and then installation of hosting solution was carried out that consumed a lot of time and required more experience.

The biggest benefit was around the cost saving of the serverless solution as compared to EC2 based virtual machines. The cost of running the machine for 24/7 on on-demand pricing was around 1200 USD per year however the consumption of CPU and memory resources was around 40%. As the requests were not served all the times and during the weekend so the deployed solution was not cost efficient. Figure 10 shows an estimate of the cost using AWS cost calculator.

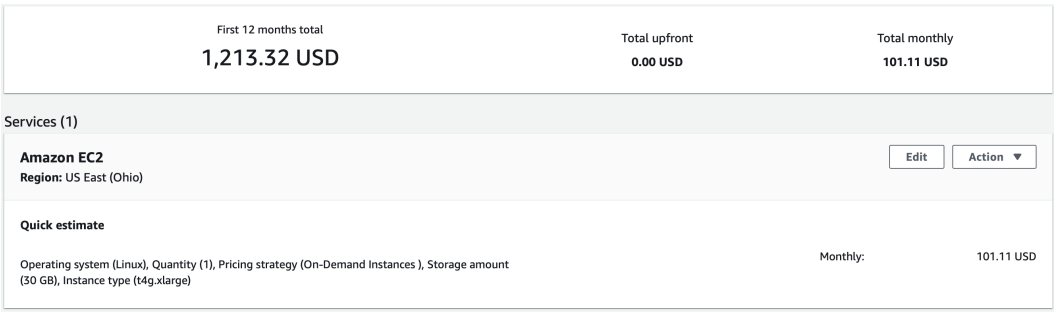

Figure 10 https://calculator.aws/#/estimate

On the other hand, serverless solution was running free of cost as there was monthly free resources available and as the consumption was low than the free quota, Hence

the solution was running completely free for most of the part. There were few costs associated with running the AWS R53 hosted zones and domain purchase, but the cost very significantly low. AWS provides a list of free tier services limits, some of services and resources expire after first year of new account creation while many of resources and serverless offerings have monthly free quota [31].

Serverless architecture removed the patching and manual OS update work. EC2 required security layer management on the OS level with regular pathing and updating. As serverless AWS lambda functions and API Gateway scaled automatically with the increased number of requests hence it provided good protection against DDoS attacks. Once the application was live as soon as the code was uploaded hence it the development time was relatively short as compared to EC2 based solution. The EC2 based solution was not scalable and was only deployed on one virtual machine so it was a bottleneck to scalability of the system and could not handle more requests at certain times.

5 Conclusions and Future Improvements

The thesis evaluated the managed services provided by AWS to make the Python package hosting easier to manage, deploy and scale. As the previously deployed solution was based on IaaS model the other aspect was gaining more knowledge about the pros and cons of IaaS vs PaaS implementations, identifying the problems with the previously designed solutions, its scalability, cost and high availability and then sharing this information internally with the colleagues hence increasing the team knowledge.

The Serverless solution provided a good understanding of the managed services offered by AWS on the enterprise level. A good experience was gained in order to deploy production grade software solution with high availability, scalability and reliability.

The solution was deployed as a part of POC so infrastructure as code (IaC) part was used for deployment. Because of limited time and man hours, many improvements mentioned below were out of scope for this project. All services were deployed using point and click method. There are various tools and framework that could be used in order to deploy all those components and services as a code. AWS provides CloudFormation to deploy cloud services and resources as a template either in JSON format or YAML. AWS also provides Cloud development Kit that could also be used to deploy the infrastructure in many high-level programming languages such as Nodejs, Python, java etc. Other frameworks such as Terraform could also be used. Terraform is an open-source infrastructure as code software tool created by HashiCorp. Users define and provide data center infrastructure using a declarative configuration language known as HashiCorp Configuration Language, or optionally JSON [32].

The CI/CD part of the package building was hosted on Jenkins server, which can be called a self-hosted solution. This part of package building could also be moved to AWS and native services provided by AWS such as AWS CodeBuild and AWS CodePipeline could also be used to make the serverless package building as part of the process [33].

Amazon web services provides many serverless solutions and resources that could be deployed in order to make scalable internet applications. As the cloud is still growing with cloud vendors innovating and upgrading their services, all cloud native solutions should

be reviewed against the best practices and cloud adoption frameworks in order to get most out of the cloud deployments [35].

References

1 Gartner July 2019 cloud computing quadrant <https://cloud.google.com/gartner-cloud-infrastructure-as-a-service>. Accessed 8 April 2021.

2 Characteristics of open source project <https://opensourceforu.com/2019/07/why-enterprises-should-opt-for-open-source-devops-tools/>. Accessed 8 April 2021.

3 Cloud computing architecture <http://www4.comp.polyu.edu.hk/~csbxiao/MCCLab/MCCLab_background.html>. Accessed 8 April 2021.

4 How To Install A Private PyPi Server On AWS <https://medium.com/swlh/how-to-install-a-private-PyPi-server-on-aws-76993e45c610>. Accessed 8 April 2021.

5 Why serverless?? <https://www.slideshare.net/AmazonWebServices/aws-may-2016-webinar-series-deep-dive-on-serverless-web-applications>. Accessed 8 April 2021.

6 AWS Managed Services <https://aws.amazon.com/blogs/aws/category/aws-managed-services/>. Accessed 8 April 2021.

7 AWS S3 Storage classes <https://aws.amazon.com/s3/storage-classes/>.Accessed 8 April 2021.

8 Designing Data-intensive Applications – Martin Kleppmann O'Reilly Media, Inc. March 2017

9 Integrating JFrog Artifactory with AWS CodePipeline <**https://aws.amazon.com/blogs/devops/integrating-jfrog-artifactory-with-aws-codepipeline/**>. Accessed 8 April 2021.

10 Installing highly available production ready Jenkins on AWS <https://www.udemy.com/course/learn-devops-deploy-highly-available-jenkins-on-aws/>.Accessed 8 April 2021.

11 Jenkins and Azure DevOps integration <https://azuredevopslabs.com/labs/vstsextend/Jenkins/>. Accessed 8 April 2021.

12 Jenkins Grovvy file template <https://www.jenkins.io/doc/book/pipeline/jenkinsfile/>. Accessed 8 April 2021.

13 Cloud Object Storage <https://aws.amazon.com/what-is-cloud-object-storage/>. Accessed 8 April 2021.

14 What is AWS Instance Profile? <https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_use_switch-role-ec2_instance-profiles.html>. Accessed 8 April 2021.

15 Custom domain registration with AWS R53 <https://docs.aws.amazon.com/AmazonS3/latest/userguide/website-hosting-custom-domain-walkthrough.html>. Accessed 8 April 2021.

16 Python automatic documentation generation package <https://pdoc3.github.io/pdoc/doc/pdoc/#gsc.tab=0>. Accessed 8 April 2021.

17 AWS Lambda , Function as a Service <https://docs.aws.amazon.com/lambda/latest/dg/welcome.html>. Accessed 8 April 2021.

18 Jenkins Slack plugin <https://www.jenkins.io/doc/pipeline/steps/slack/>. Accessed 8 April 2021.

19 AWS S3 and Lambda integration <https://docs.aws.amazon.com/lambda/latest/dg/with-s3-example.html>. Accessed 8 April 2021.

20 AWS Lambda and DynamoDB integration <https://aws.amazon.com/blogs/security/how-to-create-an-aws-iam-policy-to-grant-aws-lambda-access-to-an-amazon-dynamodb-table/>. Accessed 8 April 2021.

21 AWS Api Gateway example <https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-create-api-from-example.html>. Accessed 8 April 2021.

22 AWS S3 and Cloudfront integration for serving content <https://aws.amazon.com/premiumsupport/knowledge-center/cloudfront-serve-static-website/>. Accessed 8 April 2021.

23 AWS S3 hosting static website <https://docs.aws.amazon.com/AmazonS3/latest/userguide/website-hosting-custom-domain-walkthrough.html>. Accessed 8 April 2021.

24 AWS S3 static website protection with username and password <https://hackernoon.com/serverless-password-protecting-a-static-website-in-an-aws-s3-bucket-bfaaa01b8666>. Accessed 8 April 2021.

25 Custom Authentication with AWS Lambda <https://douglas-duhaime.com/posts/s3-lambda-auth.html>. Accessed 8 April 2021.

26 Pip Installation guide <https://pip.pypa.io/en/stable/installing/>. Accessed 8 April 2021.

27 AWS Api Gateway deep dive and examples <https://aws.amazon.com/api-gateway/>. Accessed 8 April 2021.

28 AWS Lambda authorizer with Api Gateway <https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-use-lambda-authorizer.html>. Accessed 8 April 2021.

29 Configuring stages in AWS Api Gateway <https://docs.aws.amazon.com/apigateway/latest/developerguide/stages.html>. Accessed 8 April 2021.

30 AWS Api gatway custom domain integation with AWS R53 <https://docs.aws.amazon.com/apigateway/latest/developerguide/how-to-custom-domains.html>. Accessed 8 April 2021.

31 AWS Free Tier services <https://aws.amazon.com/free/?all-free-tier.sort-by=item.additionalFields.SortRank&all-free-tier.sort-order=asc>. Accessed 8 April 2021.

32 Terraform, Infrastructure as Code framework <https://www.terraform.io>. Accessed 8 April 2021.

33 AWS CodeBuild pipelines <https://docs.aws.amazon.com/codebuild/latest/userguide/how-to-create-pipeline.html>. Accessed 8 April 2021.

34 AWS Cloud adoption Framework <https://aws.amazon.com/professional-services/CAF>. Accessed 8 April 2021.

AWS Lambda function files

This appendix shows the code files associated to the AWS Lambda functions used in the deployment of this thesis work.

1. LambdaAuthenticateDocsUser
    a. index.js

```
1. var AWS = require("aws-sdk");
2. var dynamodb = new AWS.DynamoDB({
3.   apiVersion: "2012-08-10",
4.   region: "eu-central-1",
5. });
6.
7. exports.handler = (event, context, callback) => {
8.   // Get the request and its headers
9.   const errorResponse = {
10.      status: "401",
11.      statusDescription: "Unauthorized",
12.      body: "Unauthorized",
13.      headers: {
14.        "www-authenticate": [{ key: "WWW-Authenticate", value:
   "Basic" }],
15.      },
16.    };
17.    const request = event.Records[0].cf.request;
18.    const headers = request.headers;
19.    console.log(JSON.stringify(request));
20.    console.log(JSON.stringify(headers));
21.
22.    if (headers.authorization == undefined || headers.authoriza-
   tion == null) {
23.      console.log("Auth headers missing");
24.      callback(null, errorResponse);
25.      return;
26.    }
27.    const authString = headers.authorization[0].value.split("
   ")[1];
28.
29.    var ENV = "prod";
30.    if (headers.host[0].value === "dev.docs.scienceops.io") {
31.      ENV = "dev";
32.    }
33.
34.    var params = {
35.      Key: {
36.        userId: {
37.          S: authString,
38.        },
39.      },
40.      TableName: "scienceops-users",
41.    };
42.    console.log("env request" + ENV);
43.    dynamodb.getItem(params, function (err, data) {
44.      if (err) {
45.        console.log(err, err.stack);
```

```
46.         callback(null, errorResponse);
47.       } else {
48.         console.log("data returned from dynamodb");
49.         if (
50.           data.Item == undefined ||
51.           typeof headers.authorization == "undefined" ||
52.           headers.authorization[0].value != "Basic " +
   data.Item.userId.S ||
53.           ENV !== data.Item.env.S
54.         ) {
55.           console.log("error validating user with username ");
56.           callback(null, errorResponse);
57.           return;
58.         } else {
59.           console.log("successfully validating user with username
   ");
60.           callback(null, request);
61.           return;
62.         }
63.       }
64.     });
65.   };
```

2. LambdaServePackage

   a. index.py

```
1. import os
2. from base64 import b64encode
3. import json
4. import boto3
5.
6. page_template = '<html><body><ul>{}</ul></body></html>'
7. list_template = '<li><a href="{path}">{name}</a>'
8.
9. FILE_TYPES = {'egg', 'whl', 'gz'}
10.
11.  s3       = boto3.resource('s3')
12.  dynamodb = boto3.client('dynamodb')
13.
14.  def is_allowed_type(path):
15.      return os.path.splitext(path)[1][1:] in FILE_TYPES
16.
17.  def get_objects(bucket_name):
18.      bucket = s3.Bucket(bucket_name)
19.      return {object.key: object for object in bucket.ob-
   jects.all()}
20.
21.  def parse_path(path):
22.      package, *filename = path.split('/')
23.      return package, filename[0]
24.
25.  def handler(event, context):
26.
27.      print(event)
28.      stage = event["requestContext"]["stage"]
29.      path = event["path"]
```

```
30.      print ("stage is "+stage)
31.      print ("path is "+path)
32.      response = dynamodb.get_item(
33.          TableName="scienceops-lib-versions",
34.          Key={
35.              'path': {
36.                  'S': stage+path+"/",
37.              }
38.          },
39.          AttributesToGet=[
40.              'object_key'
41.          ]
42.          )
43.      print(response)
44.      if 'Item' not in  response:
45.          print("item not found")
46.          raise Exception('Invalid file requested.')
47.      else:
48.          url = response['Item']['object_key']['S']
49.          print(url)
50.          object = s3.Object(event["stageVaria-
   bles"]["lib_s3_bucket_name"], url)
51.          package, filename = parse_path(url)
52.          local_path = f'/tmp/{filename}'
53.          object.download_file(local_path)
54.          with open(local_path, 'rb') as file:
55.              data = file.read()
56.          data = b64encode(data)
57.          return {
58.                      "isBase64Encoded": True,
59.                      "headers": {
60.                          "Content-Disposition": f"attachment;
   filename={filename}",
61.                          "Content-Type": "application/zip, ap-
   plication/octet-stream",
62.                      },
63.                      "body": data.decode()
64.                  }
65.
```

3. LambdaValidatePackageApiKey
    a. index.js

```
1. var AWS = require('aws-sdk')
2. var dynamodb = new AWS.DynamoDB({apiVersion: '2012-08-10', re-
   gion: 'eu-central-1'});
3.
4. exports.handler = function(event, context, callback) {
5.     console.log('Received event:', JSON.stringify(event, null,
   2));
6.
7.     // Retrieve request parameters from the Lambda function in-
   put:
8.     var headers = event.headers;
9.     var queryStringParameters = event.queryStringParameters;
```

```
10.      var pathParameters = event.pathParameters;
11.      var stageVariables = event.stageVariables;
12.      var requestContext = event.requestContext;
13.
14.      // Parse the input for the parameter values
15.      var tmp = event.methodArn.split(':');
16.      var apiGatewayArnTmp = tmp[5].split('/');
17.      var awsAccountId = tmp[4];
18.      var region = tmp[3];
19.      var restApiId = apiGatewayArnTmp[0];
20.      var stage = apiGatewayArnTmp[1];
21.      var method = apiGatewayArnTmp[2];
22.      var resource = '/'; // root resource
23.      if (apiGatewayArnTmp[3]) {
24.          resource += apiGatewayArnTmp[3];
25.      }
26.
27.      // Perform authorization to return the Allow policy for
   correct parameters and
28.      // the 'Unauthorized' error, otherwise.
29.      var authResponse = {};
30.      var condition = {};
31.      condition.IpAddress = {};
32.      if (queryStringParameters.key  == undefined ||
   queryStringParameters.key == "" || queryStringParameters.key ==
   null) {
33.          callback("Unauthorized");
34.          return
35.      }
36.      var params = {
37.      Key: {
38.       "keyId": {
39.         S: queryStringParameters.key
40.        }
41.      },
42.      TableName: "scienceops-access-keys"
43.     };
44.     dynamodb.getItem(params, function(err, data) {
45.       if (err) {
46.         console.log(err, err.stack);
47.         callback("Unauthorized");
48.         return
49.       }
50.       else {
51.         console.log("data returned from dynamodb");
52.         if (data.Item == undefined || queryStringParameters.key
   !== data.Item.keyId.S || data.Item.env.S!== stage) {
53.           console.log("error validating user with key ");
54.           callback("Unauthorized");
55.           return ;
56.        } else {
57.           console.log("successfully validating user with key");
58.           callback(null, generateAllow('me', event.methodArn));
59.           return ;
60.        }
61.       }
62.     });
63.  }
64.
```

```
65.  // Help function to generate an IAM policy
66.  var generatePolicy = function(principalId, effect, resource) {
67.      // Required output:
68.      var authResponse = {};
69.      authResponse.principalId = principalId;
70.      if (effect && resource) {
71.          var policyDocument = {};
72.          policyDocument.Version = '2012-10-17'; // default ver-
   sion
73.          policyDocument.Statement = [];
74.          var statementOne = {};
75.          statementOne.Action = 'execute-api:Invoke'; // default
   action
76.          statementOne.Effect = effect;
77.          statementOne.Resource = resource;
78.          policyDocument.Statement[0] = statementOne;
79.          authResponse.policyDocument = policyDocument;
80.      }
81.      // Optional output with custom properties of the String,
   Number or Boolean type.
82.      authResponse.context = {
83.          "stringKey": "stringval",
84.          "numberKey": 123,
85.          "booleanKey": true
86.      };
87.      return authResponse;
88.  }
89.
90.  var generateAllow = function(principalId, resource) {
91.      return generatePolicy(principalId, 'Allow', resource);
92.  }
93.
94.  var generateDeny = function(principalId, resource) {
95.      return generatePolicy(principalId, 'Deny', resource);
96.  }
```

b. testevent.json

```
1. {
2.     "resource": "/{proxy+}",
3.     "path": "/dev/",
4.     "httpMethod": "GET",
5.     "headers": {
6.         "accept": "text/html,application/xhtml+xml,applica-
   tion/xml;q=0.9,image/webp,image/apng,*/*;q=0.8",
7.         "accept-encoding": "gzip, deflate, br",
8.         "accept-language": "en-GB,en-US;q=0.9,en;q=0.8",
9.         "cache-control": "max-age=0",
10.          "Host": "dev.lib.scienceops.io",
11.          "upgrade-insecure-requests": "1",
12.          "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X
   10_14_0) AppleWebKit/537.36 (KHTML, like Gecko)
   Chrome/71.0.3578.98 Safari/537.36",
13.          "X-Amzn-Trace-Id": "Root=1-5c5fe204-
   2fe050187a08fa18cbe54270",
14.          "X-Forwarded-For": "84.249.131.201",
15.          "X-Forwarded-Port": "443",
```

```
16.           "X-Forwarded-Proto": "https"
17.       },
18.       "multiValueHeaders": {
19.           "accept": [
20.               "text/html,application/xhtml+xml,applica-
   tion/xml;q=0.9,image/webp,image/apng,*/*;q=0.8"
21.           ],
22.           "accept-encoding": [
23.               "gzip, deflate, br"
24.           ],
25.           "accept-language": [
26.               "en-GB,en-US;q=0.9,en;q=0.8"
27.           ],
28.           "cache-control": [
29.               "max-age=0"
30.           ],
31.           "Host": [
32.               "dev.lib.scienceops.io"
33.           ],
34.           "upgrade-insecure-requests": [
35.               "1"
36.           ],
37.           "User-Agent": [
38.               "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_0)
   AppleWebKit/537.36 (KHTML, like Gecko) Chrome/71.0.3578.98 Sa-
   fari/537.36"
39.           ],
40.           "X-Amzn-Trace-Id": [
41.               "Root=1-5c5fe204-2fe050187a08fa18cbe54270"
42.           ],
43.           "X-Forwarded-For": [
44.               "84.249.131.201"
45.           ],
46.           "X-Forwarded-Port": [
47.               "443"
48.           ],
49.           "X-Forwarded-Proto": [
50.               "https"
51.           ]
52.       },
53.       "queryStringParameters": {
54.           "key": "hassan"
55.       },
56.       "multiValueQueryStringParameters": {
57.           "key": [
58.               "hassan"
59.           ]
60.       },
61.       "pathParameters": {
62.           "proxy": "dev"
63.       },
64.       "stageVariables": {
65.           "lib_s3_bucket_name": "scienceops-libs",
66.           "dynamodb_user_table_name": "scienceops-users",
67.           "dynamodb_keys_table_name": "scienceops-access-keys"
68.       },
69.       "requestContext": {
70.           "resourceId": "26cceb",
71.           "authorizer": {
```

```
72.             "numberKey": "123",
73.             "booleanKey": "true",
74.             "stringKey": "stringval",
75.             "principalId": "me"
76.         },
77.         "resourcePath": "/{proxy+}",
78.         "httpMethod": "GET",
79.         "extendedRequestId": "U4BAoEiZliAFdtA=",
80.         "requestTime": "10/Feb/2019:08:34:12 +0000",
81.         "path": "/dev/",
82.         "accountId": "218449960348",
83.         "protocol": "HTTP/1.1",
84.         "stage": "dev",
85.         "domainPrefix": "dev",
86.         "requestTimeEpoch": 1549787652023,
87.         "requestId": "a4441c98-2d0e-11e9-a89c-afd0992c6cfb",
88.         "identity": {
89.             "cognitoIdentityPoolId": None,
90.             "accountId": None,
91.             "cognitoIdentityId": None,
92.             "caller": None,
93.             "sourceIp": "84.249.131.201",
94.             "accessKey": None,
95.             "cognitoAuthenticationType": None,
96.             "cognitoAuthenticationProvider": None,
97.             "userArn": None,
98.             "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS
    X 10_14_0) AppleWebKit/537.36 (KHTML, like Gecko)
    Chrome/71.0.3578.98 Safari/537.36",
99.             "user": None
100.        },
101.        "domainName": "dev.lib.scienceops.io",
102.        "apiId": "laosux1vo1"
103.    },
104.    "body": None,
105.    "isBase64Encoded": False
106. }
```

4. ScienceOpsUpdateCodeDocsVersion
    a. index.js

```
1. var AWS = require("aws-sdk");
2. var docClient = new AWS.DynamoDB.DocumentClient()
3.
4. exports.handler = function(event, context, callback) {
5.     // TODO implement
6.     let package_url = event.Records[0].s3.object.key
7.     let url_split = package_url.split("/")
8.     let env = url_split[0]
9.     let version = url_split[1]
10.
11.      console.log(env, version, package_url)
12.
13.      var params = {
14.          TableName: "scienceops-lib-versions",
```

```
15.          Key: {
16.              "path": env + "/latest/"
17.          },
18.          UpdateExpression: "set object_key = :object_key, up-
   load_time=:ut",
19.          ExpressionAttributeValues: {
20.              ":object_key": package_url,
21.              ":ut": new Date()
22.          },
23.          ReturnValues: "UPDATED_NEW"
24.      };
25.
26.      console.log("Updating latest the item...");
27.      docClient.update(params, function(err, data) {
28.          if (err) {
29.              console.error("Unable to update item. Error JSON:",
   JSON.stringify(err, null, 2));
30.              callback(err, data)
31.          }
32.          else {
33.              console.log("UpdateItem latest succeeded:",
   JSON.stringify(data, null, 2));
34.              // add the version as separate entry as well
35.
36.              var params_version = {
37.                  TableName: "scienceops-lib-versions",
38.                  Key: {
39.                      "path": env + "/" + version + "/"
40.                  },
41.                  UpdateExpression: "set object_key = :ob-
   ject_key, upload_time=:ut",
42.                  ExpressionAttributeValues: {
43.                      ":object_key": package_url,
44.                      ":ut": new Date()
45.                  },
46.                  ReturnValues: "UPDATED_NEW"
47.              };
48.
49.              console.log("Updating " + version + "...");
50.              docClient.update(params_version, function(ver-
   sion_err, version_data) {
51.                  if (version_err) {
52.                      console.error("Unable to update item. Error
   JSON:", JSON.stringify(version_err, null, 2));
53.                      callback(version_err, version_data)
54.                  }
55.                  else {
56.                      console.log("UpdateItem version suc-
   ceeded:", JSON.stringify(version_data, null, 2));
57.
58.                      callback(null, version_data)
59.                  }
60.              });
61.
62.          }
63.      });
64.
65.  };
66.
```

```
67.
```

Appendix B Jenkins Groovy Pipeline Template

```
#!/usr/bin/env groovy

pipeline {
    agent {
        dockerfile {
            filename 'Dockerfile.build'
            args '-u root:sudo'
        }
    }
     parameters {
         string(
             defaultValue: 'dev',
             description: '',
             name: 'DEPLOYMENT')
     }

     environment {
         AWS_DEFAULT_REGION = 'eu-central-1'
         GOOGLE_APPLICATION_CREDENTIALS = credentials("dummy-datastore-sa-key")
     }

     stages {

         stage("Install scienceops from source") {

             steps {
                 sh 'python setup.py install'
             }

         }

         stage('Run Unit Tests') {
             steps {
                 sh 'python -m unittest test.test_scienceops'
             }
         }

         stage('Build Docs') {
             steps {
                 dir('docs/') {
                     sh 'rm source/*'
                     sh 'make build'
                 }

             }
         }

         stage('Upload Docs to S3') {
             steps {
                     sh "aws s3 sync docs/_build/ s3://science-ops-${params.DEPLOYMENT}-docs/latest/"
             }
         }

         stage('Build Source Distribution') {
```

```
55.             steps {
56.                 sh 'rm dist/*'
57.                 script {
58.                     if (params.DEPLOYMENT == 'dev') {
59.                         sh 'python setup.py egg_info --tag-
   build=dev -d sdist'
60.                     } else {
61.                         sh 'python setup.py sdist'
62.                     }
63.                 }
64.
65.             }
66.         }
67.
68.         stage('Upload Package to S3') {
69.             steps {
70.                 sh "aws s3 sync dist/ s3://scienceops-
   libs/${params.DEPLOYMENT}/"
71.             }
72.         }
73.     }
74.     post {
75.         always {
76.             script {
77.                 if ( currentBuild.currentResult == "SUCCESS" )
   {
78.                     slackSend color: "good", message: "Job:
   ${env.BUILD_URL} was successful"
79.                 }
80.                 else if( currentBuild.currentResult == "FAIL-
   URE" ) {
81.                     slackSend color: "danger", message: "Job:
   ${env.BUILD_URL} was failed"
82.                 }
83.                 else if( currentBuild.currentResult == "UNSTA-
   BLE" ) {
84.                     slackSend color: "warning", message: "Job:
   ${env.BUILD_URL} was unstable"
85.                 }
86.                 else {
87.                     slackSend color: "danger", message: "Job:
   ${env.BUILD_URL} its results was unclear"
88.                 }
89.             }
90.             cleanWs()
91.         }
92.     }
93. }
```