# SECURITY AS A SERVICE FOR DOCKER CONTAINERS BY LEVERAGING SERVERLESS ARCHITECTURE

By

**Shahzaib Khan**

**Fall 2019-MS(IS) - 000003520958**

Supervisor

**Dr. Hasan Tahir**

**Department of Computing**

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science in Information Security (MSIS)

In

School of Electrical Engineering and Computer Science,

National University of Sciences and Technology (NUST),

Islamabad, Pakistan.

(December 2021)

# THESIS ACCEPTANCE CERTIFICATE

Certified that final copy of MS/MPhil thesis entitled "Security as a Service for Docker Containers by Leveraging Serverless Architecture" written by SHAHZAIB KHAN, (Registration No 00000320958), of SEECS has been vetted by the undersigned, found complete in all respects as per NUST Statutes/Regulations, is free of plagiarism, errors and mistakes and is accepted as partial fulfillment for award of MS/M Phil degree. It is further certified that necessary amendments as pointed out by GEC members of the scholar have also been incorporated in the said thesis.

Signature: [illegible]

Name of Advisor: Dr. Hasan Tahir

Date: 24-Dec-2021

Signature (HOD): __________

Date: __________

Signature (Dean/Principal): __________

Date: __________

# APPROVAL

It is certified that the contents and form of the thesis entitled "Security as a Service for Docker Containers by Leveraging Serverless Architecture" submitted by SHAHZAIB KHAN have been found satisfactory for the requirement of the degree

Advisor: Dr. Hasan Tahir

Signature: [illegible]

Date: 24-Dec-2021

Committee Member 1: Dr. Mehdi Hussain

Signature: [illegible]

Date: 27-Dec-2021

Committee Member 2: Dr. Qaiser Riaz

Signature: [illegible]

Date: 27-Dec-2021

Signature: ______________________

Date: ______________________

# DEDICATION

I dedicate this dissertation to my parents, colleagues and honorable teachers for their love and affection.

# CERTIFICATE OF ORIGINALITY

I hereby declare that this submission titled "Security as a Service for Docker Containers by Leveraging Serverless Architecture" is my own work. To the best of my knowledge, it contains no materials previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any degree or diploma at NUST SEECS or at any other educational institute, except where due acknowledgement has been made in the thesis. Any contribution made to the research by others, with whom I have worked at NUST SEECS or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except for the assistance from others in the project's design and conception or in style, presentation, and linguistics, which has been acknowledged. I also verified the originality of contents through plagiarism software.

Author Name: SHAHZAIB KHAN

Student Signature: [illegible]

# ACKNOWLEDGEMENT

I am thankful to my beloved parents and my siblings for their moral and spiritual support. They have always encouraged me to overcome any challenge. I am profoundly grateful to my supervisor Dr. Hasan Tahir for his continued interest in planning, execution, and successful completion of the project. It is only because of his consistent encouragement, inspiring guidance, dynamic supervision, and sympathetic attitude that enabled me to prepare this manuscript. He will remain a great source of inspiration and kindness for me. Heartiest thanks and gratitude are also extended to my other two respected committee members Dr. Qaiser Riaz and Dr. Mehdi Hussain for their scholarly contribution, valuable suggestions, and constructive criticism toward the successful completion of the thesis. I am obliged to all my respectable teachers for sparing their valuable time and sharing the knowledge. I believe that this work would not have been possible without their cooperation and support.

I will also express my gratitude to my colleagues, who encouraged and supported me to do MS along with my regular job. Despite all the assistance provided by the supervisor, committee members and others, I take the responsibility for any errors and omissions which may unwittingly remain.

# Table of Contents

# LIST OF FIGURES

# LIST OF TABLES

# ABSTRACT

The shift of the software development industry towards a more Agile and DevOps centered approach to have smaller and faster release cycles has led to the rapid adaptability of Docker. Docker presents itself as a lightweight solution to package applications into images with all the required libraries and environments. Such images can present some serious security vulnerabilities due to their dependability on the host operating system and distribution mechanism of public registries. A dedicated security vulnerability service can detect these threats by scanning the images periodically and isolating them from the production environment. Such an event-driven approach is best suited for a serverless architecture that is not only automated but more cost-effective and scalable than the conventional approach. In this thesis, the design and implementation of a dedicated Docker scanning service have been presented that is based on the serverless architecture using Amazon cloud services as the underlying infrastructure. The comparative analysis of the proposed design in comparison to a conventional security deployment model around four major factors including performance, cost, privileges, and scalability has shown promising results and highlights the benefits of shifting towards Serverless in the form of statistical data.

# Chapter 1

# INTRODUCTION

This introductory chapter includes fundamental concepts about the research work conducted in this thesis to develop a basic understanding. The technical terms and methodologies are discussed in a simplified way to develop an understanding for both novice and expert audiences. The problem statement is discussed along with the research question and the motivation behind this study.

## 1.1 Background

Docker containers have grown in popularity in recent years because of cloud-based production environments which led to the widespread of Docker virtualization adoption [1]. Specifically, Linux based containers are helping in this rapid shift by providing lightweight packaging and a simplistic approach to deployment. Linux kernel components like namespaces and cgroups [2] are the main technologies behind the Docker sandbox environment, they remove the virtualization layer and provide fast start-up times. Docker containers are based on very lightweight Docker images that include mainly the files and libraries to support the application, unlike virtual machines.

While the removal of the virtualization layer improves overall performance, it exposes the application to security threats because of the kernel resources being shared between multiple containers of the same host operating system thereby leading to container security becoming the main concern. The mode of distribution for Docker images is also a concerning matter, as images are publicly available having executables with known security vulnerabilities. Such security flaws can be discovered using a security scanner tool that can scan stored images in the registry or at build time. Security vulnerability databases are updated periodically with the new security threats information and require regular scanning to isolate any image with a newly discovered flaw.

The cloud computing seems very complicated, but in comparison to the on-site infrastructure it has way fewer issues. The biggest concern when hosting your own infrastructure and applications is reliability which in case of cloud computing is handled by a third-party vendor whose main job is to provide reliable infrastructure so you can focus on the application part. Cloud computing mainly depends on large size virtualized servers that are used to deploy different operating systems and application environments. The problem with virtual machines is that they are very hard to manage when they start growing in number rapidly. The most essentials requirements of modern application, scalability and performance are very hard to configure when dealing with virtual machines.

Serverless computing is a computing term for ephemeral resources that are created to perform an operation and destroyed immediately after the completion as compared to always-on virtual machine-based servers. Docker containers are the key source for this implementation due to their fast boot-up times and lightweight environment impact. A well-known implementation of the serverless architecture is the AWS Lambda service. Function as a Service (FaaS) [3][4] term is used for this kind of implementation because of the stateless nature of their programs, that triggers in response to an event.

## 1.2 Attack Model

Docker containers are made up of different components, starting from the Docker client which is used to connect with the system service of Docker daemon. The communication is established using APIs between the client and daemon for various operations like pull, push, start, and stop. Application that are packaged with Linux containers are exposed to more security threats as compared to the applications that are deployed on the bare metal virtual machines because of the architecture of Docker containers which can leak privileges to the entire host in case of a compromise. As Docker containers shares the kernel with the host operating system, the kernel vulnerabilities are wildly exploited in the form of malwares. Docker images available on the Docker Hub registry is a major concern, as anyone on the internet can upload an image and most of the images available are missing critical security patches that can lead to system compromise if used unchecked.

The requirement of Docker scanning security system is very critical and integrating it within the SDLC can automate this process as well as resolve the vulnerabilities at a very early stage.

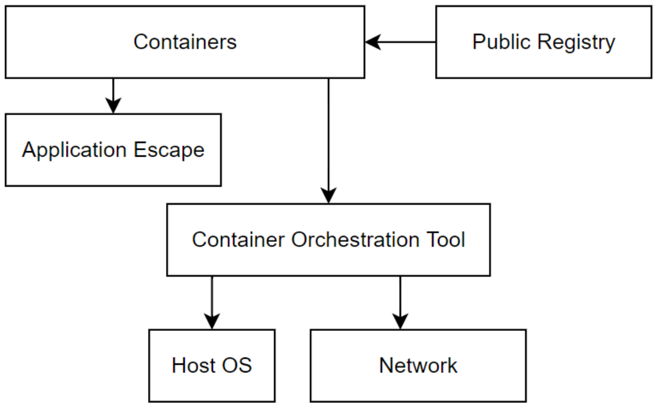


Figure 1.1: Docker Attack Model

## 1.3 Problem Statement

The vulnerabilities in the Docker containers [5] running in a production environment can be fatal, as live patches cannot be applied to running containers because of their stateless nature. The manual effort of scanning the Docker images and applying patches may take a long time thus providing an ample opportunity for the exploitation of these vulnerabilities. So, the scanning phase has to be implemented at the start of the software development lifecycle (SDLC) [6][7] to isolate any vulnerable image before it goes to production. A dedicated service is required to handle this workload, which is cost-effective, highly scalable, least privileged and doesn't come with any maintenance overhead. Based on these speculations the following problem is formulated:

**"Design a dedicated security service to analyze Docker vulnerabilities which is Cost effective, Highly Scalable, Based on Least Privileged Model and has no Management Overhead."**

## 1.4 Goals and Objectives

To design a security scanning system for Docker images that utilizes the serverless architecture to run on-Demand scans in response to a trigger. This trigger is an API request [8][9], which takes input about the target Docker image that needs to be scanned. The scanner will run via automated pipelines triggered by a serverless function whenever there's an event through API as opposed to continuously running day and night.

This implementation reduces the overall hardware consumption, thus introducing a green computing design to reduce environmental impact. It will also be cost-

effective as it's based on the pay only for the data processed pricing model. A dedicated scanning service that is isolated from the rest of the production environment in the same cloud will eliminate the risk of damaging the critical servers as well. Due to the nature of serverless architecture, this implementation is highly scalable and require little to no configurations. The whole environment is volatile which means it will be destroyed after the scan completion which eliminates the maintenance and management overhead of servers and tools.
The main functions of the proposed implementation are:

- Sending an HTTP request to API Gateway with a target image parameter that needs to be scanned.
- The API event acts as a trigger for the serverless Lambda function to perform processing.
- The serverless function can trigger a specific pipeline on CodeBuild using parameters received from the API, and a security scan is conducted on the targeted Docker image.
- Final output in JSON format report that comes out and gets stored in S3 storage service as an artefact.

## 1.5 Thesis Motivation

Security is always seen as a secondary feature when it comes to software development in many organizations. A security testing step is incorporated at the end of the development lifecycle in the form of a Pentest managed by a separate team. This seemed manageable when the software releases were limited to once or twice a year but with the rapid advancements in the DevOps automation technologies, the development lifecycle has been cut shorter to weeks or even days and the conventional security testing has become a bottleneck or simply inadequate.

The requirement now is to automate the security scanning as well and integrate it into the development lifecycle. A dedicated scanning service that addresses security issues as they emerge is the need of the hour because at this stage these vulnerabilities are easier, cost-effective, and faster to fix rather than at the end of the release cycle. The Docker containers have become a very popular choice for packaging application executables, but their base design and distribution mechanism leads to some serious security flaws.

The motivation for this project comes from the fact that organizations want to move towards automation, and Docker security scanning is an integral part of this process, but the conventional approach requires you to manage a dedicated appliance that runs 24 hours and require maintenance and patching on regular basis. This adds another layer of management complexity when considering the need to shift towards automation. The proposed implementation in this thesis tackles this problem specifically by introducing a serverless architecture that is based on volatile infrastructure that gets deployed and destroyed on every iteration thus offering extreme agility and no server overhead for maintenance with the added benefit of cost reduction.

## 1.6 Thesis Organization

The thesis is break down into different chapters to maintain a consistent flow of understanding and semantics.

Following chapters are included in this thesis:

- Chapter 1 provides background knowledge about the technical terms discussed and basic concepts about the problem.

- Chapter 2 explains the technologies used in this research and the recent work done by various scholars in the same domain.
- Chapter 3 explains the proposed design and architecture along with the methodology.
- Chapter 4 presents a comparative analysis between the proposed solution and the conventional approach.
- Chapter 5 concludes the research work and presents directions for the future work.

## 1.7 Summary

This chapter introduced the problem statement and core research topics that this thesis further discusses. In this chapter, the adoption of automation in the development lifecycle is explained and how the conventional security approach is becoming a bottleneck. The architecture has been presented which addresses most of the problems that exist in the conventional deployment and administration of a dedicated security scanning service. The goals are also explained that were to achieve with this specific research and the possible outcomes that can improve the conventional deployment strategy by taking advantage of serverless architecture.

Finally, the key discussion around motivation points behind this research. In the next chapter, the technical terms are explained along with related work conducted by researchers in the same domain.

# Chapter 2

# LITERATURE REVIEW

The previous chapter explored the problem domain in detail and presented the motivation of the work. This chapter presents the related work done by various researchers over the years, that contributed to the development of this thesis. The goal of this chapter is to provide recent and notable research related to the work done in this thesis.

## 2.1 Docker OS Level Virtualization

OS Level virtualization has become popular in recent times because it can operate consistently across different platforms and has the ability to be transferred between different environments. Docker is an open-source OS-level virtualization solution for applications that makes the process of development and distribution moderately easy. Docker-packaged applications have all of their supporting dependencies in a standard format known as a container. The container utilizes the kernel layer of the operating system in an abstract manner, to utilize the underlying host's resources. The containers require a docker engine to run, similar to a hypervisor when it comes to virtual machines. Containers can be run in a segregated fashion and still utilize the same host OS which can be Windows or Linux. All the required components

including libraries and binary files are shared across all the containers this making them very lightweight, few Mb in size [10].
When compared to virtualizing a whole hardware server, process separation and use of the container host's kernel is more efficient [11]. The container comprises most of an application's dependencies. Which enables seamless movement between various operational environments like development, staging, testing, and production. Docker's ability to provide consistent environments and patching flexibility has made it an excellent choice for companies looking to transition from waterfall to the contemporary DevOps approach towards software delivery.

### 2.1.1 Docker Architecture

Docker employs a client-server architecture, as illustrated in Figure 2.1. The communication between the Docker client and server is used to exchange commands and perform different actions like build, create, run, etc. Docker provides a RESTful API as well as a command-line executable to handle the communication process. Docker daemon services can coexist on the same host and communicate over the network. Various Docker objects like containers, volumes, images, and network are managed by Docker daemon which responds to requests coming from Docker API. With the help of Docker Clients, users can interact with Docker. Docker clients utilize a command-line interface (CLI) through which users can send run and stop application commands to a Docker daemon [12]. Docker Host provides a complete environment for program execution and operation. The service daemon, running Containers, stored Images, Networks, and Storage are all components of it.

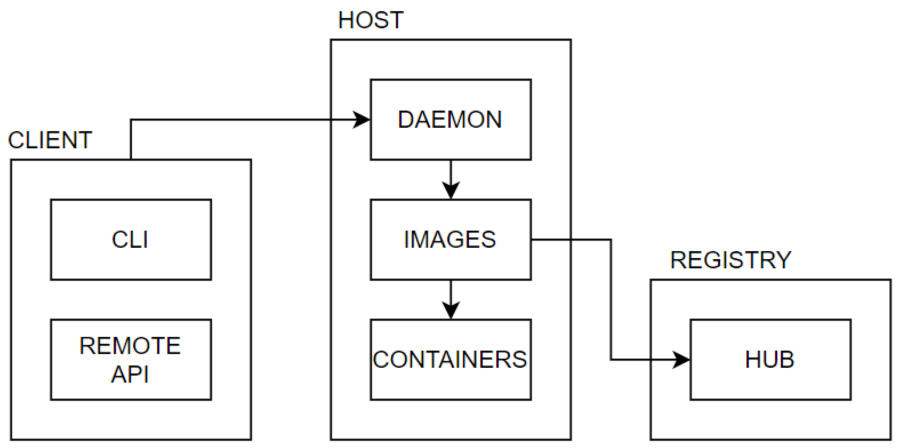


Figure 2.1: Docker Architecture

**1) Images**

Docker Images are binary templates that are read-only and used to create containers. Metadata on the container's unique functionalities and requirements can also be found in images. After building these Images, software applications can be stored or shipped to different places. An image can be used to build a container or can be modified to add new features to enhance the existing configuration. A private registry is required to share Container images within an organization or to publish these images publicly for the rest of the world, a public registry is required like Docker Hub. For collaboration between developers across different teams, Images acts as an essential component of the Docker experience [13][14]. Docker images can be built using a docker file by providing a set of instructions and running the “docker build” command from the bash terminal. A base image is required, which acts as a foundation and is usually an operating system image, such as Ubuntu 16.04 LTS, or CentOS 7.9.2009. The desired application or service can be added to this image as a change, and a new image needs to be built.

### 2) Containers

Containers are isolated environments in which applications can be executed. The image and any additional configuration parameters provided when launching the container, including but not confined to network access and storage variables, comprise the container. Unless extra access is provided when creating the image into a container, containers only have access to the resources defined in the image [15]. You may also build a new image depending on a container's existing state. Containers, which are significantly smaller than VMs, may be set up in seconds and result in substantially higher server efficiency.

Containers still require a full functioning kernel that is shared with all containers. Furthermore, the microservice design emphasizes the need for temporary state containers, in which any data persistence is transferred to another data repository or service. Containers are widely accepted as the conventional method for deploying microservices to the cloud.

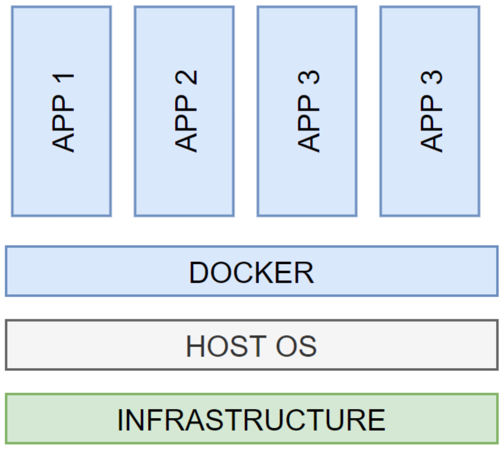


Figure 2.2: Containerized Applications

**3) Registries**

Similar to the source code repository, the Docker registry is required to host docker images. Images can be pushed or pulled from these registry and various versions can be maintained. Docker Hub is a public registry available for everyone to push or pull various images, but private registries can also be hosted for images containing classified data [12].
The beginning of comprehension around the growing popularity of Docker containers, DevOps adoption, and microservices has started. We can also observe how Docker makes underlying containers lighter, quicker, and more robust, which simplifies infrastructure administration. Docker also isolates the application layer from the infrastructure layer, providing somewhat mobility, cooperation, and governance over the software delivery cycle.

### 2.1.2 Docker Security Challenges

Following a study of several key research papers on Docker security from major academic journals and books, the most prominent threats to docker containers are Image vulnerabilities. Image misconfigurations, clear-text secrets, container runtime vulnerabilities, and application vulnerabilities [16][17]. Table 2.1 shows a list of top vulnerabilities and their impact reported around Docker containers.

| Vulnerability | Impact | CVE |
|---|---|---|
| **runC Remote Execution** | Give the attacker root access. | CVE-2019-5736 |
| **Docker Skeleton Runtime** | Allow attacker to replace user functions inside container | CVE-2018-11757 |
| **PHP Runtime** | Allow attacker to replace user functions inside container | CVE-2018–11756 |
| **Windows Host Compute** | Remote code execution on the host file system | CVE-2018-8115 |
| **util.c in runV** | Root access through numeric value in /etc/passwd | CVE-2018-9862 |

Table 2.1: Top 5 Docker Vulnerabilities

Delu Huang et al. discussed the common attacks on docker containers and also provided a review for the security features available currently for Docker containers and Linux kernel [18]. Docker security will be most jeopardized in the near future due to the following factors: network connectivity, image building, container running, registry storage, and kernel integration [19]. Theo et al. introduced the concept of Docker ecosystem security and suggest that third-party component security and Docker software life cycle security must be enhanced [20].

S. Sultan et al. discussed a few use cases to test out container security including application security, inter-container protection, host to the container, and container to host isolation [21]. Wenhao et al. discussed that Docker vulnerability analysis can be categorized into four aspects: file system isolation, network isolation, resource limitation, and image transmission. It also shared some views on the current security mechanism for docker security [22].

Docker containers can mount volumes to store persistent data, which can lead to exposing sensitive data to the host Operating system as well as other containers. Securing the Docker environment with respect to storage-related security issues is a challenge and many container orchestration tools like Docker swarm provide solutions [23].

## 2.2 Serverless

A cloud-based architecture is used in serverless computing which lets facilitates running relatively small code snippets without any management of underlying resources. It is somewhat misleading as the underlying resources do exist, but the operation overhead such as resource allocation, maintenance, monitoring, scalability, and fault tolerance is not managed by the customer due to the Event-Driven nature.

Serverless computing is emerging as a compelling technology for cloud-based deployment models, mainly because of the shift of enterprise applications towards the containers and microservices architecture. Figure 2.3 depicts the rising popularity of the keyword "serverless" as reported by Google over the previous ten years.

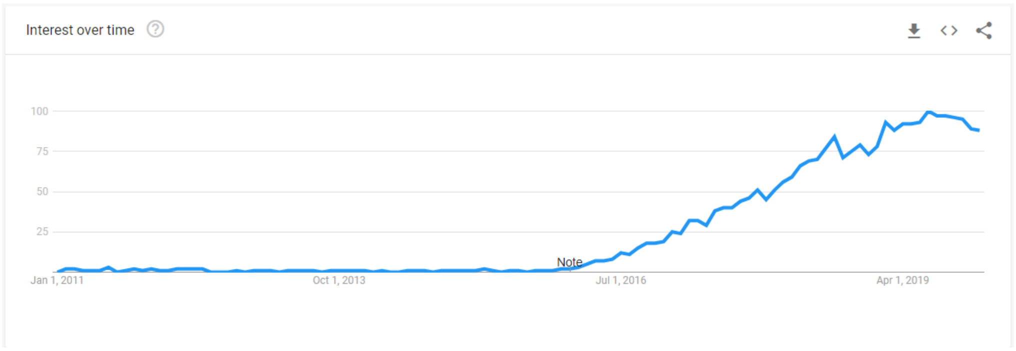


Figure 2.3: Trends for the keyword “serverless” reported by Google

This paradigm presents both an opportunity and a risk. If we look from an Infrastructure-as-a-Service (IaaS) customer's perspective, then deploying the application in a serverless platform is challenging due to the platform design and concerns regarding scaling, monitoring, and fault tolerance. On the other hand, it gives engineers a simpler development paradigm for cloud-based apps that abstracts away the majority of the operational problems. The cost is less due to the costing model of charging per execution instead of overall resource allocation.
It is designed to rapidly deploy cloud-native small code snippets that respond to various events, which traditionally require some kind of middleware application.
From the perspective of a cloud provider, serverless architecture lets them further optimize the development stack and reduce the cost of cloud resources. This platform is different from Platform-as-a-Service (PaaS), as it provides a function-based development model [24], which is stateless in nature. This model due to its explicit use of function as the deployment unit is also known as Function-as-a-Service (FaaS) [25].
The current academic literature around serverless performance and design implementation is somewbrhat lacking[26]. Lin et al. explored the serverless approach while taking into account many cloud aspects. They also presented a model-based solution forserverless unexpected performance and cost issues, which can estimate end-to-end response time and cost. [27]. Kim et al. investigated the performance of network resources in data-intensive serverless applications [28]. Elgamal et al analyzed the problems in serverless regarding cost optimization and execution times [29].
McGrath et al. discussed the current implementation of serverless as well as the deployment models available [30]. AWS Lambda is a popular serverless computing service offered by Amazon Web Services. Lambda is designed to offer a per function execution cost model which abstracts away the deployment, operational

configurations, and monitoring of web servers thus allowing developers to only worry about writing individual functions for each microservice [31].

## 2.3 AWS Cloud Services

Cloud computing is boosting the ability to use the internet more than ever, and AWS is leading the market by providing huge benefits like data protection, compliance regulation, flexibility, cost-effectiveness, auto-scaling, high-performance processing, and multiple storage options.
AWS provides many cloud-native services, that usually require setting up third-party tools, thus reducing the cost and time required to set up infrastructure [32]. Amazon has many services; But only the services that are used in this study will be discussed.

### 2.3.1 API Gateway

APIs are the endpoints, from which apps can communicate with backend services to access data and business logic as well as other functionalities. AWS API Gateway allows you to develop and use public RESTful API endpoints to a large variety of AWS services. Through a secure gateway, a customer can simply connect with a large variety of AWS services such as multiple databases, messaging apps, and Lambda functions. API Gateway allows you to develop Restful Web services and WebSocket endpoints for bi-directional communication between applications in real-time. Docker container-based and serverless computing tasks, as well as conventional internet-facing applications, are supported by API Gateway. Because of the ease of setup for endpoint generation, it allows for the continuation of fast growth.

API Gateway covers all of the responsibilities required in listening and responding to tens of thousands of concurrent API requests, including web traffic distribution, CORS compliance, access control, filtering, analytics, and version control. API Gateway offers no minimum fees or starting costs; you just pay for the API requests you receive, and the quantity of data sent out.

### 2.3.2 Lambda

Amazon Cloud's Lambda service [33] is a serverless solution that provides you with an interface to run code without having to deploy or manage any servers, developing workload-aware cluster scalability, managing event interfaces, or handling third-party tools. Back in 2014, Amazon was the first big cloud provider to provide serverless computing. Initially, Lambda was offered with only integration with Node.js runtime environment, but now it supports various technology stacks such as Python, Java 8, and C#.
To begin, just submit a binary/code as a compressed file or Docker image, and Lambda will automatically assign necessary processing resources and execute the code/binary depending on the inward requests or events, regardless of traffic magnitude. You may configure your code to run automatically from a variety of AWS services and SaaS apps, or you can call it directly from an endpoint. Lambda functions are independent of other AWS resources and have built-in versioning capabilities to create multiple versions of your function on different stages such as development, testing, and production [34].

### 2.3.3 CodeBuild

A pipeline [35][36] is a pool of automated processes and components connected in sequence, with the product of one stage becoming the following stage's input. This architecture of the pipeline helps developers achieve CI/CD (continuous integration/continuous delivery) lifecycle [37] in software development. Amazon offers continuous integration services called CodeBuild that are fully managed by them. It provides integration with GitHub and many other third-party services, that help developers automate the process of compilation, testing, and packaging of software code. The process of provisioning, managing, and scaling of the build servers doesn't need to be managed by Developers anymore.

CodeBuild grows constantly and executes many builds in parallel, ensuring that the queue for builds does not stack up. Developers can spin up instances directly by utilizing preconfigured build configurations, also they can develop customized build environments using custom build tools. The computational resources that you use are charged by the minute. [38].

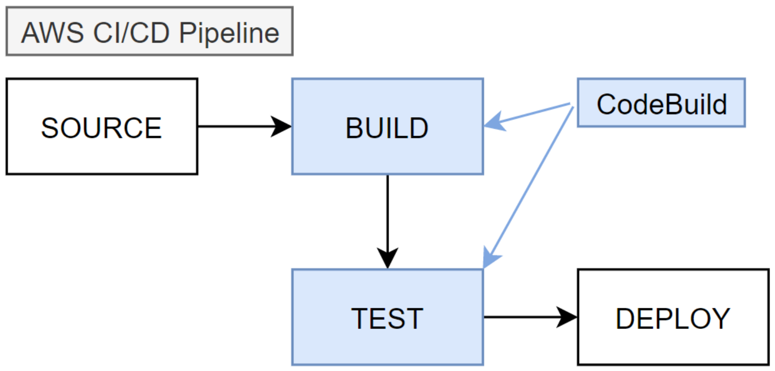


Figure 2.4: AWS CI/CD Pipeline

### 2.3.4 S3

Object storage services are getting popular these days and are being offered by many cloud service providers as opposed to other forms of storage such as block and file storage. Amazon offers an object storage service called S3 where every item is stored independently as an object and has its information (metadata) along with an ID number assigned to it. This storage technology is very different from conventional file or block storage, where a REST API can be called to access certain files.

Objects are stored in buckets, which is a native term for Amazon S3. These buckets do not allow public access by default, but their permission model can be changed by the administrator. The permission tab from the web interface can be used to manage the Read and Write access for buckets or single objects. Amazon S3 also offers a bucket policy mechanism that can be configured using a JSON (Javascript Object Notation) format configuration file which can be a powerful mechanism to manage individual rights over large storage resources, but it requires competent IT knowledge. The CORS (Cross-Origin Resource Sharing) policy [39] is also configurable using the provided editor, which lets you decide the actual websites and URLs that can access your objects.

A user can encrypt data before storing it in an S3 bucket, but it will require authentic credentials to fetch and view stored files, also individual user rights can also be defined [40]. Amazon S3 provides two types of storage i.e., S3 and Glacier. The cheapest option would be Amazon Glacier, but it is only meant to be used for long-term archival and file storage. Some of its lowest charges offering actually store data on tape drives that will need to be installed before it can be retrieved again by you which explains the occasionally sluggish access periods. The drawback is that it becomes more expensive if you access your data more frequently, like when you are pulling data from the web. Transferring data to the service, on the other hand, is free. In case of easy accessibility, the standard offering by Amazon (S3) is definitely the better option for anyone.

### 2.3.5 EC2

The classic standalone virtual machine offering from Amazon is Elastic Compute Cloud (EC2), and it is the most widely used service of AWS as well. Users can launch and administer servers of various specs at any time and for as long as they want. When launching a new machine, a user can choose from a plethora of available server specs configurations designed towards specific requirements like CPU intensive, Memory consumption, high network latency etc. Users have a selection of various operating systems to choose from, paid options are also available from various vendors that comes with preinstalled and configured products. In the next section, storage space has to be selected as well as the type including SSD based Fast storage or standard options. Network placement is configured using subnets and security groups to open certain ports for access and configurations, AWS also offers an assignment of Live IP address mapped directly to the virtual machine.

Auto-scaling groups [41][42] are configured to automatically scale up and down the number of instances required to complete an operation. From the pricing perspective, AWS offers a pay-as-you-go model where you can decide to pay in advance for a machine for a fixed amount of time (6 months) at a lower price, or by the hour for OnDemand machines.

## 2.4 Vulnerability Scanning

In a Docker-based environment, vulnerabilities that result in RCE (remote code execution) [43], escalation of privileges [44], or sensitive information leakage are disastrous. One exploited container in an environment has the ability to compromise other containers on the same host, and the use of multi-tenant infrastructure makes it severely concerning.

The vulnerability analysis on Docker containers comprises a thorough examination of security reports and available fixes to determine the scope of these issues over a certain time as well as sources, effects, and consequences. The vulnerabilities that affected Docker in the past are kept documented in a repository along with security reports and approved patches. The description includes the details about a certain flaw, the kind of operating system that is affected, and the resolution process [45]. Several Docker scanning tools take advantage of the vulnerabilities database to compare the versions present in the image. These tools can be used to get an overview of the vulnerabilities affecting a certain image before using it in the production environment. Zhao et al. performed an analysis with wide parameters on Docker images available from the Docker Hub registry. The results have identified that there are security concerns regarding the storage mechanism of Docker Images and there is room for optimization [46].

### 2.4.1 Clair

Clair is a program that parses picture material and reports vulnerabilities in the content. This is done through static analysis rather than during runtime. Clair may extract contents and assign security hotspots from a long list base operating system container:

- Ubuntu
- Debian
- RedHat
- OpenSuse
- Oracle
- Alpine
- Amazon Linux
- VMWare Photon
- Python

Clair's analysis is divided into three sections. The process of indexing begins with the submission of a configuration file to Clair, which is able to retrieve underlying layers, scan through the files, and deliver a base result file known as an Index upon reception.
Clair's depiction of a Docker image is an index file, and it makes use of the fact that OCI Index and Layers are processed together to save duplication. An IndexReport is an outcome when Manifests are indexed, and this report is saved for further use. Matching is the process of picking an Index and comparing security hotspots that impact the manifest that the initial report provided [47].

### 2.4.2 Trivy

A security vulnerability scanner, specific for Docker containers and some other artefacts. Trivy identifies vulnerabilities in base operating system containers (such as Alpine, RedHat, and CentOS) and application dependencies (yarn, composer, npm, CMake, bundler, etc.). Trivy is a simple application to use. For scanning, it is a very simple application in usability as all you need to do is input a target Docker image, and it will start scanning.

Trivy can scan Operating System packages from various flavours like Debian, RHEL, AWS Linux, Suse, Alpine, Photon OS, CentOS, Debian, Ubuntu, Oracle Linux, and Distroless.

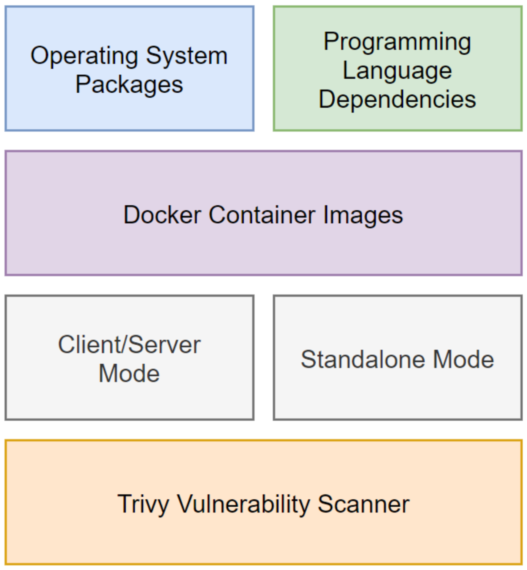

Operating System Packages
Programming Language Dependencies
Docker Container Images
Client/Server Mode
Standalone Mode
Trivy Vulnerability Scanner

Figure 2.5: Trivy Security Scanner Architecture

Trivy is stateless and doesn’t require any maintenance unlike most of the security scanners which require a short period of time (10 minutes) to retrieve CVE database (vulnerability and hotspot information) on their initial run and add another requirement to keep a long-term database of vulnerability information. [48].

## 2.5 Summary

In conclusion, Docker virtualization being active in the industry for many years still hasn't been able to catch up to the Industry Security Standard. Docker images hosted on a public registry (like Docker Hub) have no security controls implementation or safeguard against vulnerabilities and malicious content. The requirement for In-house Docker image scanning is imminent and thus finding solutions with high performance for mass scanning and cost-effectiveness is a tedious task.

A discussion about the previous research around Docket security scanning and the limitations in that domain was explained. In this chapter, the related research on Docker architecture and the security concerns in the creation, distribution, and isolation of Docker images was discussed. We also discussed serverless computing and its benefits over conventional virtualization and application deployment architecture. Several offerings from AWS support this serverless architecture where the application can be deployed and managed with minimum effort at a fraction of cost as compared to virtual machines. Some of the Docker image scanning tools that provide a comprehensive analysis of the application and system libraries of a particular Docker image were also discussed.

In the next chapter, the discussion will be about the implementation of proposed approach which utilizes serverless computing to implement Docker image scanning and provide better performance, scalability, and cost.

# Chapter 3

# DESIGN AND IMPLEMENTATION OF SERVERLESS ARCHITECTURE

In this chapter, the discussion will be around the proposed architecture design, implementation of tools and configurations across all services. This chapter will go through the various application modules that have been developed to implement the proposed framework. The discussion will also include the overall flow and functionality of the framework in order to achieve desired goals and provide answers to the research problem.

## 3.1 Proposed Framework

To address the research question "Security service to analyze Docker vulnerabilities using serverless architecture, which is cost-effective, Highly Scalable and with No Management Overhead," the necessary set of AWS services and vulnerability scanning tools were identified, which are utilized as building blocks in the proposed framework. Fig 3.1 explains the architecture flow of the mechanism through all the services. The discussion around the technical implementation and configurations will be presented in this chapter.

The proposed framework consists of three major modules:

- API Gateway
- Lambda Serverless Function
- CodeBuild Scanning Pipeline

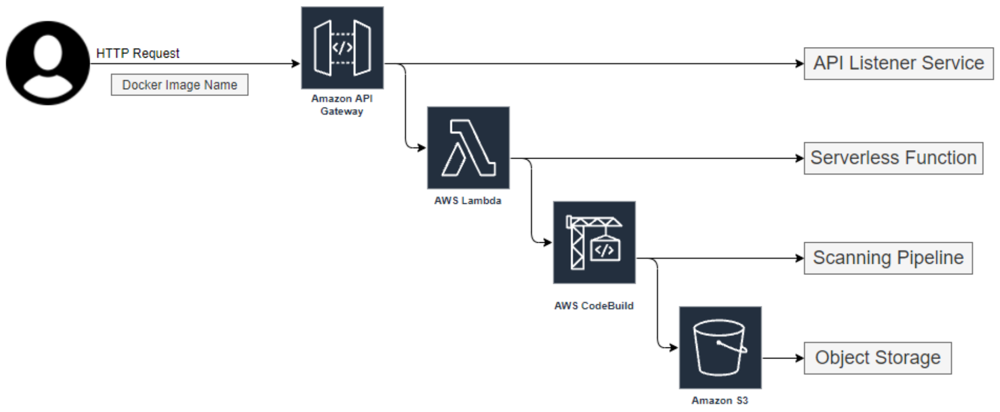


Figure 3.6: Proposed Architecture Flow

## 3.2 API Gateway

This service is used as an endpoint to retrieve input from the user, and the Docker Image name that needs to be scanned is the input in this case. AWS API Gateway service is used to implement this module, which can be set up easily, and supports auto-scaling as well as authentication. This module consists of several stages as described in Fig 3.2:

- Request Method
- Query Strings
- Integration Request
- Function Response

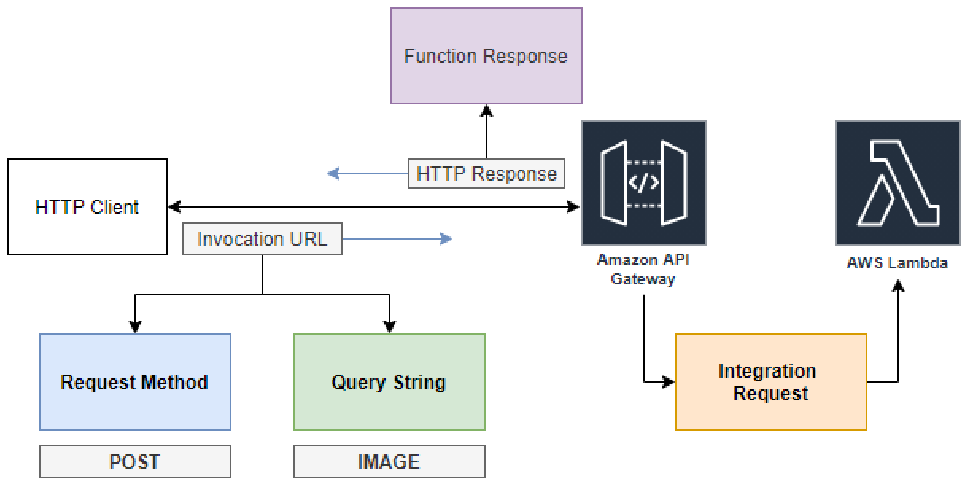


Figure 3.7: API Gateway Architecture

### 3.2.1 Request Method

This stage of the API Gateway module requires the implementation of the POST based HTTP request method, which can accept parameters as part of a request although AWS support all types of HTTP Methods [49] including GET, PUT, DELETE etc. These methods can be used to further interact with the system in future work and an HTTP client can be used to initiate this request from the user side.

### 3.2.2 Query String

This stage considers the most important part of the request that originates from the client-side, The parameters. A Docker image that needs to be scanned through the security service is provided to the system using the parameter named “image” and a value against it at this stage. A client application can provide the parameter as part of the request URL like this:
***invocation_url?image=ubuntu***

### 3.2.3 Integration Request

After receiving the request on the API Gateway Listener service, an integration request is sent to another AWS service called Lambda. This integration request holds the parameter as well as other environmental variables that are required for the Lambda function to process and trigger the scanning process

### 3.2.4 Function Response

A plain HTTP response with response code 200 (Success) is returned to the client-side if the processing of parameters as well as integration request is completed. This ensures the user, that the request sent was valid, the parameter syntax was correct, and the scan has been triggered. In the scenario, where any of the above stages fails, a respective error code is returned to the user with debug information.

## 3.3 Lambda Serverless Function

The main trigger module, which is responsible for triggering different services to scan the Docker image in response to some events. This stage composes mainly of the python function that utilizes the boto3 library to trigger the CodeBuild pipeline using parameters received from the previous module. The function is also responsible for creating buckets on the AWS S3 service for storing code files as well as configurations in the form of a CloudFormation template.
This module has the following components:

- Serverless Application
- Python Trigger Function
- CloudFormation Template

AWS Lambda service requires an initial configuration of the main application that will hold all the trigger functions. The configurations include the runtime environment, processing resources, function definitions and monitoring integration. Along with that the application also requires an IAM (Identity & Access Management) role which provides it with the access to procure necessary resources on related services and perform required actions based on the trigger functions that it holds.

For runtime environment, Python 3 is the selected language framework, and for access management, the built-in IAM role from AWS labelled as AWSCodeBuildAdminAccess has been attached which grants access to CodeBuild Pipeline. Along with that an IAM role to write logs in AWS Cloud Watch and storage in S3 is also attached. This access model restricts the underlying serverless functions of this application to only access these services instead of the whole infrastructure.

## 3.3.2 Python Trigger Function

The serverless function requires an event as a trigger along with environment variables and some general hardware configuration like runtime memory. Following are the set of components that needs to be configured:

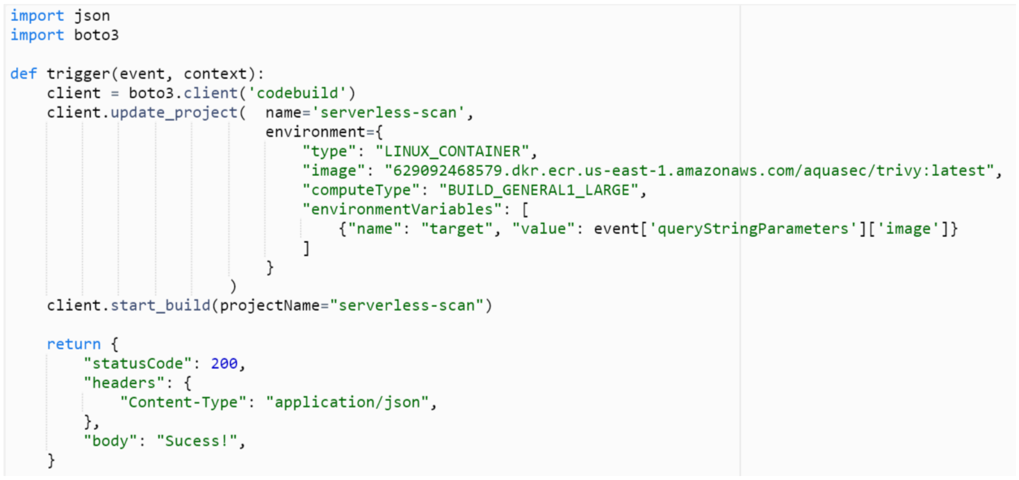
**Runtime Memory**: A runtime memory of 128 Megabytes have been assigned to the Lambda function because of the simple logic of the python program.

**Triggers**: In this case, the trigger event is the previously configured API Gateway, which receives the value of the Docker image name. Whenever the API is called from the client side, an event will be generated, which will act as a trigger for the Lambda function.

**Permissions**: The permission model designed in the previous stage is assigned here as well which grant the permissions to trigger the CodeBuild pipeline.

**Monitoring**: To monitor the activity of the functions, logs and metrics can be forwarded to Cloud Watch service or any third-party log monitoring solution. This step is also essential for debugging problems with the Lambda function.

Fig 3.3 explains the Trigger Function which is utilizing the boto3 [50] library to trigger a CodeBuild pipeline using some parameters like type, image, computeType and other environmental variables. One of the environmental variables is the image variable, which contains the name of the Docker image that was retrieved from the API Gateway as an event.



```
import json
import boto3

def trigger(event, context):
    client = boto3.client('codebuild')
    client.update_project(  name='serverless-scan',
                            environment={
                                "type": "LINUX_CONTAINER",
                                "image": "629092468579.dkr.ecr.us-east-1.amazonaws.com/aquasec/trivy:latest",
                                "computeType": "BUILD_GENERAL1_LARGE",
                                "environmentVariables": [
                                    {"name": "target", "value": event['queryStringParameters']['image']}
                                ]
                            }
                        )
    client.start_build(projectName="serverless-scan")

    return {
        "statusCode": 200,
        "headers": {
            "Content-Type": "application/json",
        },
        "body": "Sucess!",
    }
```

Figure 3.8: Lambda Function Code Snippet

The trigger function starts by initializing a client with the boto3 library for CodeBuild and then update project properties in a JSON format. The client.start_build statement initiates an API call to the CodeBuild services and waits for response code 200 which is an HTTP response for successful transmission.

## 3.3.3 CloudFormation Template

CloudFormation is an infrastructure as code service [51] by AWS, the benefit is to create a whole infrastructure based on various services using a YAML or JSON file. Lambda uses these CloudFormation templates to dynamically procure the required resources like Docker Container for the execution of trigger function, CloudWatch Logs and S3 bucket for storage. The process of migration becomes easier as well because you can use the same YAML/JSON in the new environment to have an absolute identical deployment.

## 3.4 CodeBuild Scanning Pipelines

The CodeBuild pipelines provide a ready-made build environment in the form of a Docker container where the different tasks can be executed in an automated manner. In this framework, the CodeBuild pipeline is responsible to run the container security scanning tool against a Docker image and provides us with a detailed report. The CodeBuild pipeline has the following components which require prior configurations.

- Environment Configuration
- Buildspec File
- Artefacts

### 3.4.1 Environment Configuration

The environment in which the Docker security scanning tool will be executed is a prebuilt container from Aqua Security, and available from the Docker Hub registry using the tag “aquasec/trivy:latest”. This container will be the base image that the CodeBuild pipelines will use to execute all the operations and thus creating an isolated temporary environment for the scanning which will be deleted after the execution.

CodeBuild requires limited permissions to procure resources and communicate with other services to pull Docker images and store artefacts. These permissions can be granted by attaching an IAM role with the required privilege level. Using these permissions, CodeBuild will not be destructive against our infrastructure while downloading and scanning unknown Docker images from the internet.

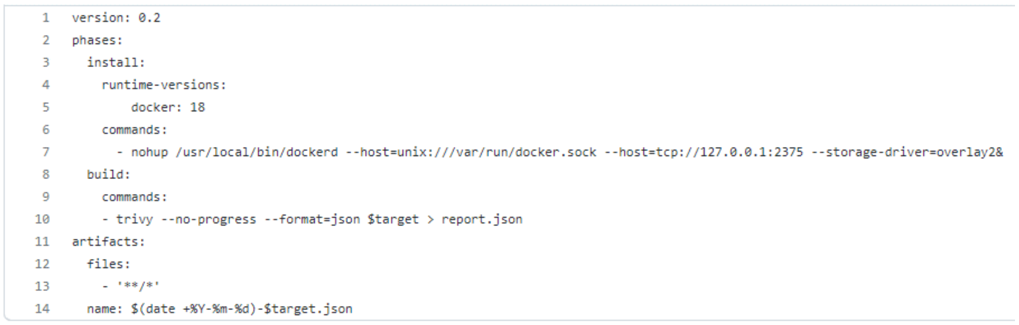
Aquasec Trivy is a very lightweight tool and doesn't require a huge number of hardware resources, A total number of 4 CPUs and 7 GB runtime memory (RAM) is allocated to the base security scanning container. The environment variables include the target image name that needs to be scanned which can be overridden using parameters at runtime.

Buildspec stands for Build specification and is provided in a YAML format for the CodeBuild pipeline. The Buildspec file contains the actual operational commands and actions that need to be executed on every trigger of CodeBuild pipeline. The Buildspec file consists of various phases as shown in Fig 3.4.

**Install Phase**: This phase mentions all the required packages that need to be installed before the execution, the actual command to run the provided Docker base image and the runtime environment. The base security scanning tool needs to scan the target Docker image and for that very purpose, it needs the privileges to run the target image container as well. To provide these privileges, the Docker API port 2375 is exposed while executing the Docker Daemon.

**Build Phase**: The actual set of commands that needs to be executed within our base container when it starts running are provided in this phase. Aquasec Trivy is executed with parameters to generate a report in JSON format and the name of the output file.

**Artifacts Phase**: The output (if any) generated after the execution of the Build Phase is called artifact and different operations can be performed on it. This phase includes the renaming of the security scan report file to a timestamped target-specific name from its original generic name. The discussion around the importance of the artifacts will be presented in the next stage.



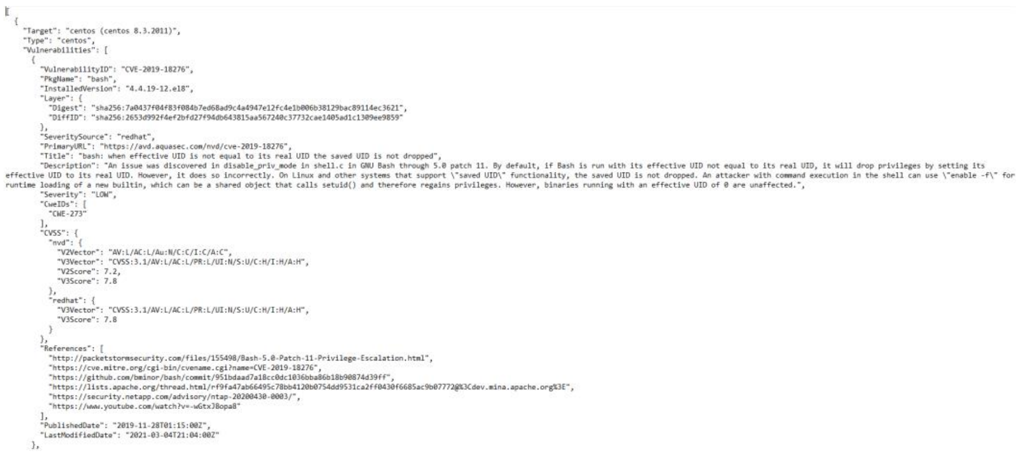
```
version: 0.2
phases:
  install:
    runtime-versions:
        docker: 18
    commands:
      - nohup /usr/local/bin/dockerd --host=unix:///var/run/docker.sock --host=tcp://127.0.0.1:2375 --storage-driver=overlay2&
  build:
    commands:
    - trivy --no-progress --format=json $target > report.json
artifacts:
  files:
    - '**/*'
  name: $(date +%Y-%m-%d)-$target.json
```

Figure 3.9: Buildspec Configuration File

### 3.4.3 Artefacts

Artefacts are the output at the end of CodeBuild pipeline execution [52], and several configurations are required to store, encrypt, compress, or simply process them for later use. The artefact generated in the previous stage is the security scan report out of Aquasec Trivy tool in JSON format and it gets stored in the S3 bucket according to the timestamp and target specific formatting that was initialized in the Buildspec file.

S3 can maintain versioning of the reports as well [53], in case a scan is conducted several times on the same Docker image. This provides the functionality to compare results of security scans across different time frames and measure the overall improvements. The report itself has different sections and starting from the target information and then traversing through all the vulnerabilities as shown in Fig 3.5.



```
[
  {
    "Target": "centos (centos 8.3.2011)",
    "Type": "centos",
    "Vulnerabilities": [
      {
        "VulnerabilityID": "CVE-2019-18276",
        "PkgName": "bash",
        "InstalledVersion": "4.4.19-12.el8",
        "Layer": {
          "Digest": "sha256:7a0437f04f83f084b7ed68ad9c4a4947e12fc4e1b006b38129bac89114ec3621",
          "DiffID": "sha256:2653d992f4ef2bfd27f94db643815aa567240c37732cae1405ad1c1309ee9859"
        },
        "SeveritySource": "redhat",
        "PrimaryURL": "https://avd.aquasec.com/nvd/cve-2019-18276",
        "Title": "bash: when effective UID is not equal to its real UID the saved UID is not dropped",
        "Description": "An issue was discovered in disable_priv_mode in shell.c in GNU Bash through 5.0 patch 11. By default, if Bash is run with its effective UID not equal to its real UID, it will drop privileges by setting its
effective UID to its real UID. However, it does so incorrectly. On Linux and other systems that support \"saved UID\" functionality, the saved UID is not dropped. An attacker with command execution in the shell can use \"enable -f\" for
runtime loading of a new builtin, which can be a shared object that calls setuid() and therefore regains privileges. However, binaries running with an effective UID of 0 are unaffected.",
        "Severity": "LOW",
        "CweIDs": [
          "CWE-273"
        ],
        "CVSS": {
          "nvd": {
            "V2Vector": "AV:L/AC:L/Au:N/C:C/I:C/A:C",
            "V3Vector": "CVSS:3.1/AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H",
            "V2Score": 7.2,
            "V3Score": 7.8
          },
          "redhat": {
            "V3Vector": "CVSS:3.1/AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H",
            "V3Score": 7.8
          }
        },
        "References": [
          "http://packetstormsecurity.com/files/155498/Bash-5.0-Patch-11-Privilege-Escalation.html",
          "https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-18276",
          "https://github.com/bminor/bash/commit/951bdaad7a18cc0dc1036bba86b18b90874d39ff",
          "https://lists.apache.org/thread.html/rf9fa47ab66495c78bb4120b0754dd9531ca2ff0430f6685ac9b07772@%3Cdev.mina.apache.org%3E",
          "https://security.netapp.com/advisory/ntap-20200430-0003/",
          "https://www.youtube.com/watch?v=-wGtxJ8opa8"
        ],
        "PublishedDate": "2019-11-28T01:15:00Z",
        "LastModifiedDate": "2021-03-04T21:04:00Z"
      },
```

Figure 3.10: Aquasec Trivy Security Scanning Report

The vulnerabilities section in the report has much important information along different sections, A comprehensive result against different vulnerabilities can be seen in Table 3.1 which was conducted against the base container image of CentOS 8.3.2011.

| CVE ID | Severity | Package | Title |
|---|---|---|---|
| CVE-2019-18276 | Low | bash 4.4.19.-12.el8 | when effective UID is not equal to its real UID the saved UID is not dropped |
| CVE-2020-8625 | High | bind-export-libs 32:9.11.20-5.el8 | Buffer overflow in the SPNEGO implementation affecting GSSAPI security policy negotiation |
| CVE-2021-25215 | High | bind-export-libs 32:9.11.20-5.el8 | An assertion check can fail while answering queries for DNAME records that require the DNAME to be processed to resolve itself |
| CVE-2018-1000876 | Medium | binutils 2.30-79.el8 | integer overflow leads to heap-based buffer overflow in objdump |
| CVE-2017-14166 | Low | libarchive | Heap-based buffer over-read in the atol8 function |

Table 3.2: Vulnerability Scanning for CentOS 8.3.2011

## 3.5 Summary

In conclusion, the proposed framework utilizes three major services of AWS cloud to implement a security scanner service for Docker containers. API Gateway is used to get input from the client side about the target Docker image that needs to be scanned and a trigger function in python is used to initiate the actual scan using the Lambda service. The scan itself takes place in an isolated and temporary environment, created by the CodeBuild pipeline for the period of the execution. As an output, the security vulnerabilities report is stored in the S3 storage service with versioning for later review.

In this chapter, the discussion was around the implementation of the proposed framework that utilizes serverless computing to implement Docker image scanning without using any conventional virtual server. HTTP post request is sent to Amazon API Gateway service, with a request parameter containing the name of the Docker image, which acts as an event to trigger the serverless function in Lambda service. Lambda function can take the parameter and trigger a CodeBuild pipeline to execute the instructions provided in the Buildspec configuration file. When all of this process is completed, a security vulnerabilities report is generated and stored in the S3 storage service in JSON format.

In the next chapter, the discussion will be around a comparative analysis of cost, privileges, performance and scalability between a conventional security tool deployment and the proposed framework of serverless architecture to validate the authenticity of the design.

# Chapter 4

# PERFORMANCE, COST AND SYSTEM EVALUATION

In this chapter, an analytical comparison between a conventional security system deployment and the proposed serverless architecture is presented. The major areas that will be covered for this comparative analysis include performance factors, access privileges, cost model and scalability which will prove the authenticity of the proposed architecture.

The Docker containers are also referred to as lightweight virtual machines [54] but in reality, they are not virtual machines due to architectural change as described in Table 4.1.

| | **Virtual Machines** | **Docker Containers** |
|---|---|---|
| **Virtualization Layer** | Hardware | Operating System |
| **OS Layer** | Independent | Shared |
| **Boot Sequence** | Long | Short |
| **Resource Utilization** | High | Low |
| **Ready-to-go Images** | Difficult to find | Easily available |
| **Custom Images** | Difficult to build | Easy |
| **Size Consumption** | Huge including the OS | Small, shared host OS |
| **Mobility** | Easy to move | Recreation |

Table 4.3: Comparison between Virtual Machines and Docker Containers

## 4.1 Performance Evaluation

For the performance analysis of Docker containers compared to virtual machines, in our use case of security scanning tool, 5 different target Images that vary in size, number of layers, and underlying libraries were used. Ubuntu, CentOS, Nginx, Postgres, and MySQL images are scanned for the purpose of this analysis. In the proposed serverless architecture, CodeBuild is using the “general1.medium” compute instance type, which offers 4 virtual CPU computation power along with 7 Gigabytes of virtual memory. To simulate a conventional security scanner system, an EC2 virtual machine is used to host our security tool with the compute type “c4.xlarge” which offers similar compute power of 4 vCPU and 7.5 Gigabyte virtual memory.

For the series of this test, the following performance parameters were used in the benchmarking of the final results from the two systems.

### 4.1.1 CPU Execution

The computing performance of a system can be measured in two forms, the first being the number of operations a system can perform in a particular set of time and the second being the time consumed for the completion of a certain event [55]. This specific performance parameter depends on the number of cores that have been assigned to the base server. The scanning process can use parallel computation technology to execute multiple operations at the same time on each virtual core.

### 4.1.2 Memory Performance

The read and write operations of memory blocks on data, and the largest amount of memory available for caching purposes can affect the measurements of this performance parameter [56]. Copying, scaling, and adding commands are the most essential operations when it comes to memory performance. The copy command is used to transfer the data from one memory block to the other, while the modification of data after certain operations is handled by scale command and the add command is utilized when data is read from various locations in the memory for a certain operation.

### 4.1.3 Disk I/O Measurement

The input and output operations performed on a hard drive in the form of read and write commands are the most essential variables to measure the performance of this particular parameter. The record size and the file size itself can impact the read/write operations when it comes to the scanning of large files. For the purpose of this analysis, high-speed solid-state drives were used in AWS infrastructure to balance out the equation on both ends.

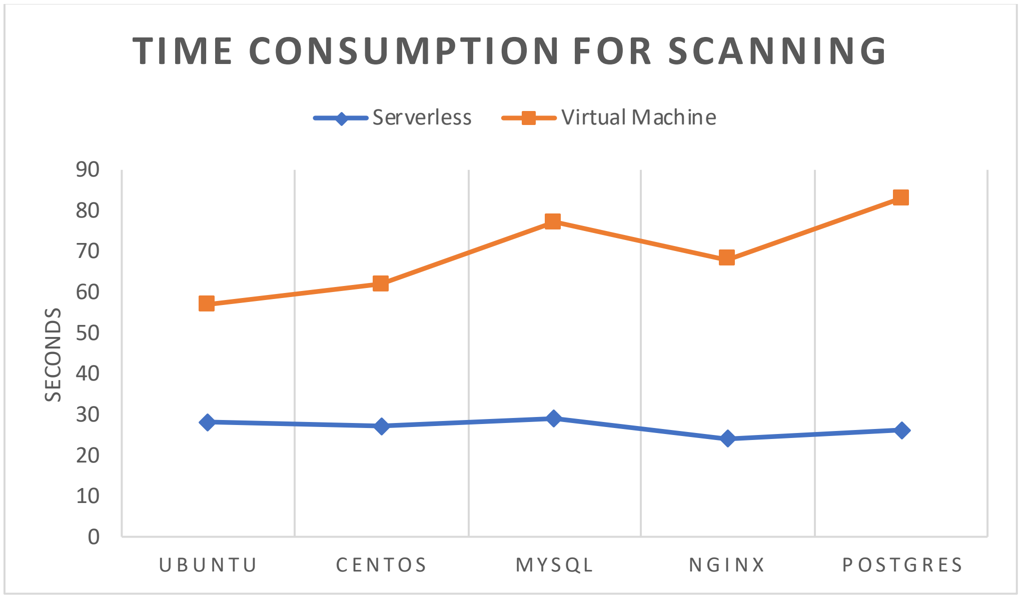


Figure 4.11: Performance Analysis Graph for Scanning

All 5 of the images were scanned using both security systems, serverless and conventional virtual machines. The analysis results showed that the security scanning tool running on the virtual machine took twice the time as the serverless architecture for the scanning of the images. Fig 4.1 shows a comparative chart between the two systems and the scanning time in seconds for each of the five selected Docker images.

## 4.2 Cost Evaluation

In the AWS EC2 platform, we can run standard virtual machines with various specs and can configure scalability and concurrency using EC2 autoscaling groups with custom policies for scale up and scale down. These custom policies require information for defining conditions for scaling like average threshold limits and instance add/delete actions. It can only be done by investing a lot of time in metric logs to populate these threshold values, as these values are very difficult to predict. In the case of serverless, we are given a built-in service for concurrency and scaling which require you to enter the maximum number of instances that will run concurrent with the limitation of default upper limit, 1000 in Lambda and 60 in CodeBuild but adjustable (can be increased).

For the series of these tests, Cost estimation was calculated using AWS cost estimator service. Cost estimator service can predict the estimated cost by taking input regarding the resources provision and the utilization frequency.

### 4.2.1 EC2 Cost Estimation

For this proposal, a c4.xlarge instance was used to host the security scanning tool. This instance provides a total number of 4 vCPU and 8Gigabyte of virtual memory. Base storage of 50Gigabyte was also procured for the operating system as well as the storage of docker containers that need to be scanned. Following cost estimation is calculated:

*0.199 USD On-Demand Hourly Rate*

*1 instances x 0.199 USD x 730 hours in a month = 145.27 USD (monthly)*

***Amazon EC2 On-Demand instances (monthly): 145.27 USD***

Using the above calculations, an estimated cost of 145.27 USD monthly has been calculated. This cost includes a continuously running instance of EC2 instance, regardless of being used or not.

### 4.2.2 Serverless Cost Estimation

For serverless implementation, two individual services are being utilized that's why the cost analysis will be independent as well. For the Lambda trigger function, the first 1 million requests in a month are not charged, which are more than enough for a security scanner. If over 100 scans are conducted each day, that sums up to about 3000 requests each month. The memory allocated to Lambda functions is 128 Megabytes and a single scan request takes about 576 milliseconds to complete on average. The following calculations don't include free-tier requests:

*Amount of memory allocated: 128 MB x 0.0009765625 GB = 0.125 GB*

*100 requests x 576 ms x 0.001 = 57.60 total compute (seconds)*

*0.125 GB x 57.60 seconds = 7.20 total compute (GB-s)*

*7.20 GB-s x 0.0000166667 USD = 0.00 USD (monthly compute charges)*

*100 requests x 0.0000002 USD = 0.00 USD (monthly request charges)*

***Lambda costs - Without Free Tier (monthly): 0.00 USD***

From the above estimation, it has been identified that due to the simplicity of the Lambda function, and low memory consumption, the cost is 0 USD.

On the other hand, the CodeBuild service utilizes more resources as it is responsible for creating a temporary infrastructure, running the scan, and publishing the results. The CodeBuild pipeline utilizes the "general1.medium" tier of hardware which allocates 4vCPU and 7.5 Gigabyte memory for each iteration. One scan takes about

20 to 30 seconds depending on the size of the Docker Image, the cost was estimated according to an average of 60 seconds, to be as flexible as possible.

*3,000 builds per month x 1 minutes = 3,000.00 billed minutes (monthly)*

*3,000.00 minutes x 0.01 USD = 30.00 USD*

***AWS CodeBuild cost (monthly): 30.00 USD***

This cost comparison shows a major difference between the conventional and serverless approach, the major factor being the continuous running of EC2 virtual machines even when it's not required, due to longer boot times it can't be turned off in the off-hours. Fig 4.2 shows a cost comparison according to different time frames.

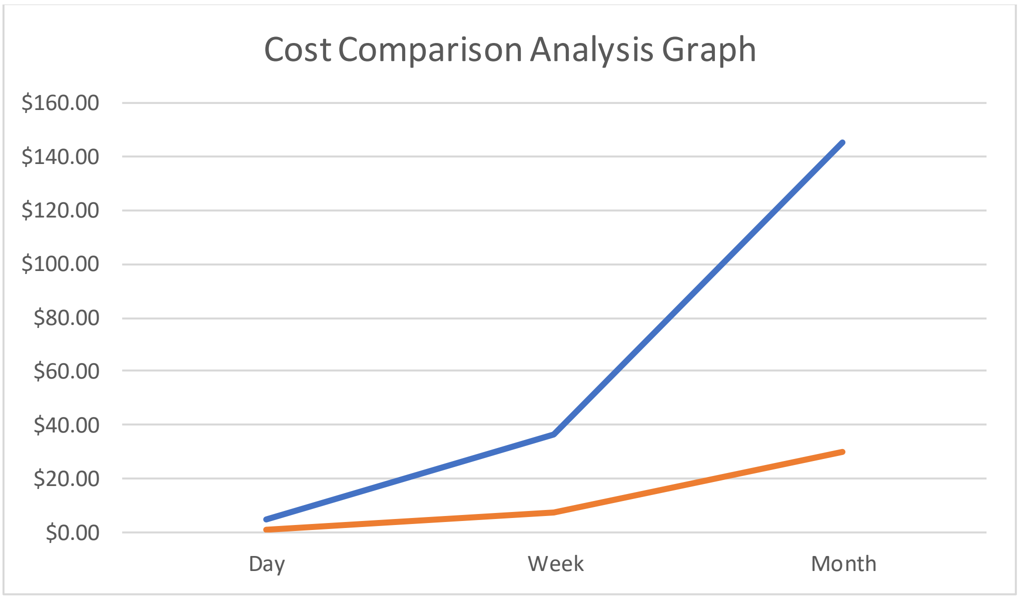


Figure 4.12: Cost Comparison Graph between Conventional and Serverless

| Time Frame | EC2 | Serverless |
|---|---|---|
| **Day** | $4.84 | $1 |
| **Week** | $36.31 | $7.50 |
| **Month** | $145.27 | $30 |

Table 4.4: Cost Comparison in Time Frames

Looking at the monthly cost, there's a significant difference of 384% between the proposed solution and the conventional security scanner tool. The main reason is the pay only consumption pricing model in the case of serverless.

## 4.3 Least Privilege Model Comparison

In this section, the analysis was conducted for the permission model used by both EC2 conventional virtual machines and services included in the serverless architecture. To be able to find the least privilege utilization model between the two, we first need to find the privileges it needs to execute the required operations.

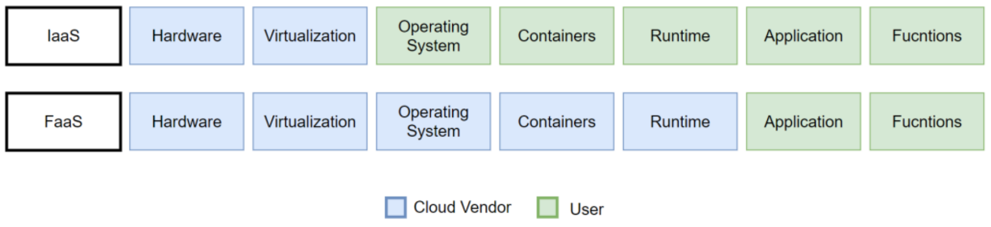


Figure 4.13: Responsibility Division Model between IaaS and FaaS

### 4.3.1 EC2 Privileges

EC2 is a user-managed service in the category of Infrastructure as a Service (IaaS), which means that the user is responsible for everything including the security of the instance. Fig 4.3 shows the division of responsibility between the Cloud vendor (AWS) and the user. From a security standpoint, you can configure different firewalls, security groups and network policies to secure your instance as well as control the flow of traffic. You also have the option to set up an antivirus product on your instance and use a patch manager to install OS updates and security patches on your virtual machine.

The security scanning tool needs to fetch Docker container images from the public repositories, which requires connectivity to the internet. Security scanner also requires permission to API Service to receive scanning requests from the user, which requires internet connectivity as well. For publishing reports on the bucket, the EC2 instance will require access to the S3 storage service as well.

All these accesses and being exposed to the internet makes the EC2 system very critical to security threats. In the case of many organizations, the production environment along with development and testing also resides on the same availability zone in cloud infrastructure, and this particular security scanning tool can present a threat to those environments by being in the same environment. A proper network placement in a DMZ and restricting network policies can secure this instance but requires a lot of manual configurations and experienced staff.

Docker Images that are downloaded from the internet can contain malicious files, and always present the possibility of bypassing the security measures that were implemented. Placing this security scanner in a very isolated environment, away and separated from the other environments is the only logical way to be sure but it will certainly present more issues regarding cost and maintainability.

### 4.3.2 Serverless Privileges

Lambda and CodeBuild both are based on the principle of volatile environments that are built on execution time and wiped out after. These temporary environments use CloudFormation service to quickly build infrastructure using specs from a file also known as Infrastructure as Code. The Docker-based environments can be spun up and destroyed in less than a second which makes it the ideal choice for an Infrastructure as Code based temporary environment.

Lambda service requires access to the CodeBuild service only because that's the main function it is performing in this case, triggering a CodeBuild pipeline that scans a Docker Image. As per exposing to the outside networks, Lambda is not exposed directly to the internet, it only receives a parameter from the API gateway and this parameter is passed on to the CodeBuild without any further processing.

CodeBuild requires access to the S3 bucket only, as it runs the pipeline and publishes reports. The CodeBuild service is isolated just like Lambda, and they have no network connectivity nor permissions to contact other environments like Development or Production. CodeBuild requires permission to CloudFormation to create a temporary infrastructure and S3 to publish reports, both are granted by using IAM roles.

If a malicious Docker image is downloaded during the scan time, the scope of the environment limits its spread to only that pipeline infrastructure which is temporary and will be destroyed after the scan finish. Only the scan report is published in the S3 bucket as an artefact.

## 4.4 Scalability Analysis

In this section, the discussion will be around the scalability options in both conventional and serverless approaches, also the amount of effort required for configurations. Generally, all AWS' managed services are scalable by design and provide you with an interface to set a few thresholds to customize the scalability while in the case of self-managed EC2 instances, manual effort is required which differs from application to application.

EC2 offers auto-scaling which lets you configure the availability of your services by adding one more or removing an instance. In the case of the security scanner, if more than 1 scan needs to be conducted in parallel then another instance of the same specs will have to be run. This will double the cost, as you are running two EC2 instances even though the second one will be turned off when the scan is complete. Time for the scanning will also be increased as creating a new EC2 instance and booting up will require significant duration.

In the case of serverless, Both Lambda and CodeBuild can spin up multiple instances at the same time because they are independent. The time required for spinning up a Docker container is much less than that of a virtual machine, so it doesn't affect the latency rate in the case of scalability. The serverless infrastructure is also volatile, which means that it will be discarded after the scan, so the problem with adding and removing instances won't happen here unlike EC2.

## 4.5 Recommendations

Recalling from Chapter 1, the primary goal is to help developers and organizations to perform an audit on the Docker Images that they utilize in a more efficient, cost-effective, and secure way. In many teams, security is often perceived as an add-on in the SDLC (Software Development LifeCycle). In this session, there is a mention of the approach which can in incorporated in the development lifecycle with a very little time investment. In the real world, securing and isolating the scanning environment from the production and development environment is easy but, in our experience, that's not the case always. Developers are encouraged to utilize the serverless approach and look into various aspects of AWS APIs invoked to further implement the least privileges model. The incorporation of security testing and scanning in the early stages of the development lifecycle improves the overall security stature of the organization, provide awareness for the developers, and saves the time that would have been spent in security improvement in a later stage of the development cycle.

Following are some recommendations based on the research and its outcome/observations:

- **Do not utilize built-in AWS policies** The built-in policies for several services in the IAM (Identity and Access Management) appears to be saving time, but mostly they are over-privileged. The developers need to analyze the required set of privileges and edit the policies before adopting them in their environment.
- **Identification of Least Privilege Set** This process can be very painful to find the right amount of privilege for the security scanner and can become a repetitive action in case of faulty or no documentation presence.
- **Provision of Individual Set of Roles** Every Lambda function and CodeBuild pipeline needs to have a very distinct role specifically designed for it. Usually, generic templates are created for this purpose but if your security scanning product has multiple stages incorporating multiple Lambda functions and CodeBuild pipelines then individual roles need to be created that only allows the required set of permissions.

- **AWS managed IAM policies** It is never a good idea to implement the built-in policies that are managed by AWS when assigning permissions to your roles. The user guide provided by Amazon indicates that AWS updates these built-in policies from time to time, the updates will affect all the entities that these are attached to, and such change can be very unexpected for the underlying service, So the best choice is to use self-managed policies.

## 4.6 Summary

Four various factors were presented that can be improved by implementing serverless architecture in place of conventional virtual machines for security scanning tools. Performance evaluation was conducted on 5 different targets Docker images on both serverless implementation and EC2 virtual machine. Results showed that besides having the same specs of the underlying infrastructure, serverless performed better and the scan times were nearly half then what was seen in the EC2 implementation. Cost evaluation was conducted to measure the daily, weekly, and monthly budgets for each type of implementation. A base scanning number of 100 scans a day was used, each scan consuming 60 seconds to complete. Results showed that the EC2 machine had to run continuously even when it’s not in use, that’s why the cost was about 384% more than what we incurred in the case of serverless. The serverless pay per usage cost model is effective when the application isn’t being continuously in use and the scanning needs to be conducted after every interval or so.

The implementation of the Least privileges model is much easier to implement in a serverless approach, as compared to the conventional EC2 service. The reason is an isolated and independent service as well as a volatile environment with restrictive scope of access and exposure. EC2 requires a hefty number of configurations including security groups, firewalls, and network policies to restrict access from other environments. The privileges in the case of serverless are very limited to certain services using IAM roles.

Serverless is a highly scalable approach with little to no configurations as compared to EC2 which requires auto-scaling groups to be implemented to add or remove an instance when required. EC2 increases the cost with every new instance it starts, also the time required to spin up a new virtual machine is significant and can affect the latency of scanning.

# Chapter 5

# CONCLUSION AND FUTURE WORK

The thesis has explored the design and implementation of a Docker security scanning service using the Serverless architecture on AWS infrastructure. This chapter summarizes the research work done and also identifies some of the open research problems that still need to be solved/explored further.

## 5.1 Conclusion

The implementation of a Docker image scanning tool using serverless architecture over a conventional approach of using virtual machines was addressed. The proposal is to provide a solution that is better in performance, cost-effective, highly scalable, low maintenance and least privileged. An analysis was also performed around both implementation and the results have shown that serverless implementation provides twice the performance, cost about 4 times less, is highly scalable out of the box, and implementation of the least privilege model is easier. The underlying cloud provider is AWS for the demonstration of both virtual machine-based implementation and serverless implementation including EC2, Lambda and CodeBuild services. The scanning tool Aquasec Trivy is used to perform testing on 5 different sets of Docker images to avoid biased results.

## 5.2 Future Work

There is tremendous scope for future work. The implementation presented in this thesis around serverless implementation is limited to several services and can be extended to various other services to increase the automation and user experience.

- Multiple Lambda functions can be utilized to further process the incoming requests with much more data than just image name, and a user-friendly interface can be achieved.
- Reporting can be improved by conversion of JSON data to Excel format files using a mediatory Lambda function for easier tracking.
- Much like all the services that were used in this thesis, other cloud offerings can be explored as well to further analyze the cost reduction and scalability options.
- Multiple Docker scanning tools can be incorporated to compare results of a target to minimize the occurrence of false positives and also improvement of overall results.

# REFERENCES

[1] D. Trihinas and G. Pallis, “DevOps as a Service : Pushing the Boundaries of Microservice Adoption Taking the Pulse of DevOps in the Cloud,” *IEEE Comput. Soc.*, no. June, pp. 65–71, 2018, [Online]. Available: www.computer.org/internet.

[2] J. Claassen, R. Koning, and P. Grosso, “Linux containers networking: Performance and scalability of kernel modules,” *Proc. NOMS 2016 - 2016 IEEE/IFIP Netw. Oper. Manag. Symp.*, no. Noms, pp. 713–717, 2016, doi: 10.1109/NOMS.2016.7502883.

[3] E. Van Eyk, A. Iosup, C. L. Abad, J. Grohmann, and S. Eismann, “A SPEC RG cloud group’s vision on the performance challenges of FaaS cloud architectures,” *ICPE 2018 - Companion 2018 ACM/SPEC Int. Conf. Perform. Eng.*, vol. 2018-Janua, pp. 21–24, 2018, doi: 10.1145/3185768.3186308.

[4] R. A. P. Rajan, “A review on serverless architectures-Function as a service (FaaS) in cloud computing,” *Telkomnika (Telecommunication Comput. Electron. Control.*, vol. 18, no. 1, pp. 530–537, 2020, doi: 10.12928/TELKOMNIKA.v18i1.12169.

[5] R. Shu, X. Gu, and W. Enck, “A study of security vulnerabilities on docker hub,” *CODASPY 2017 - Proc. 7th ACM Conf. Data Appl. Secur. Priv.*, pp. 269–280, 2017, doi: 10.1145/3029806.3029832.

[6] S. S, “A Study of Software Development Life Cycle Process Models,” *SSRN Electron. J.*, 2017, doi: 10.2139/ssrn.2988291.

[7] R. Kneuper, “Sixty years of software development life cycle models,” *IEEE Ann. Hist. Comput.*, vol. 39, no. 3, pp. 41–54, 2017, doi: 10.1109/MAHC.2017.3481346.

[8] L. Li, W. Chou, W. Zhou, and M. Luo, “Design Patterns and Extensibility of REST API for Networking Applications,” *IEEE Trans. Netw. Serv. Manag.*, vol. 13, no. 1, pp. 154–167, 2016, doi: 10.1109/TNSM.2016.2516946.

[9] V. Surwase, “REST API Modeling Languages -A Developer ’ s Perspective Related papers REST API Modeling Languages - A Developer ’ s Perspective,” *IJSTE - Int. J. Sci. Technol. Eng.*, vol. 2, no. 10, pp. 634–637, 2016, [Online]. Available: https://www.academia.edu/27064725/REST_API_Modeling_Languages_A_Developers_Perspective?bulkDownload=thisPaper-topRelated-sameAuthor-citingThis-citedByThis-secondOrderCitations&from=cover_page.

[10] A. K. Yadav, M. L. Garg, and Ritika, *Docker containers versus virtual machine-based virtualization*, vol. 814. Springer Singapore, 2019.

[11] R. R. Yadav, E. T. G. Sousa, and G. R. A. Callou, “Performance comparison between virtual machines and docker containers,” *IEEE Lat. Am. Trans.*, vol. 16, no. 8, pp. 2282–2288, 2018, doi: 10.1109/TLA.2018.8528247.

[12] B. Bashari Rad, H. John Bhatti, and M. Ahmadi, “An Introduction to Docker and Analysis of its Performance,” *IJCSNS Int. J. Comput. Sci. Netw. Secur.*, vol. 17, no. 3, pp. 228–235, 2017.

[13] Aquasec, “Docker - Architecture,” 2019. https://www.aquasec.com/cloud-

native-academy/docker-container/docker-architecture.

[14] Z. Lu, Y. Wu, J. Xu, and T. Wang, “An acceleration method for docker image update,” *Proc. - 2019 IEEE Int. Conf. Fog Comput. ICFC 2019*, pp. 15–23, 2019, doi: 10.1109/ICFC.2019.00010.

[15] C. Diekmann, J. Naab, A. Korsten, and G. Carle, “Agile Network Access Control in the Container Age,” *IEEE Trans. Netw. Serv. Manag.*, vol. 16, no. 1, pp. 41–55, 2019, doi: 10.1109/TNSM.2018.2889009.

[16] T. Yang, Z. Luo, Z. Shen, Y. Zhong, and X. Huang, “Docker’s security analysis of using control group to enhance container resistance to pressure,” *Proc. - 10th Int. Conf. Inf. Technol. Med. Educ. ITME 2019*, pp. 655–660, 2019, doi: 10.1109/ITME.2019.00151.

[17] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, “On the Relation between Outdated Docker Containers, Severity Vulnerabilities, and Bugs,” *SANER 2019 - Proc. 2019 IEEE 26th Int. Conf. Softw. Anal. Evol. Reengineering*, pp. 491–501, 2019, doi: 10.1109/SANER.2019.8668013.

[18] D. Huang, H. Cui, S. Wen, and C. Huang, “Security Analysis and Threats Detection Techniques on Docker Container,” *2019 IEEE 5th Int. Conf. Comput. Commun. ICCC 2019*, pp. 1214–1220, 2019, doi: 10.1109/ICCC47050.2019.9064441.

[19] A. A. Mohallel, J. M. Bass, and A. Dehghantaha, “Experimenting with docker: Linux container and baseos attack surfaces,” *Int. Conf. Inf. Soc. i-Society 2016*, pp. 17–21, 2017, doi: 10.1109/i-Society.2016.7854163.

[20] T. Combe, A. Martin, and R. Di Pietro, “To Docker or Not to Docker: A Security Perspective,” *IEEE Cloud Comput.*, vol. 3, no. 5, pp. 54–62, 2016,

doi: 10.1109/MCC.2016.100.

[21] S. Sultan, I. Ahmad, and T. Dimitriou, “Container security: Issues, challenges, and the road ahead,” *IEEE Access*, vol. 7, pp. 52976–52996, 2019, doi: 10.1109/ACCESS.2019.2911732.

[22] J. Wenhao and L. Zheng, “Vulnerability Analysis and Security Research of Docker Container,” *Proc. 2020 IEEE 3rd Int. Conf. Inf. Syst. Comput. Aided Educ. ICISCAE 2020*, pp. 354–357, 2020, doi: 10.1109/ICISCAE51034.2020.9236837.

[23] A. Krasnov, R. R. Maiti, and D. M. Wilborne, “Data Storage Security in Docker,” *Conf. Proc. - IEEE SOUTHEASTCON*, vol. 2020-March, p. 7281, 2020, doi: 10.1109/SoutheastCon44009.2020.9249757.

[24] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, “Serverless Programming (Function as a Service),” *Proc. - Int. Conf. Distrib. Comput. Syst.*, pp. 2658–2659, 2017, doi: 10.1109/ICDCS.2017.305.

[25] S. Chaudhary, G. Somani, and R. Buyya, “Research Advances in Cloud Computing,” *Res. Adv. Cloud Comput.*, pp. 1–465, 2017, doi: 10.1007/978-981-10-5026-8.

[26] D. Bardsley, L. Ryan, and J. Howard, “Serverless performance and optimization strategies,” *Proc. - 3rd IEEE Int. Conf. Smart Cloud, SmartCloud 2018*, pp. 19–26, 2018, doi: 10.1109/SmartCloud.2018.00012.

[27] C. Lin and H. Khazaei, “Modeling and Optimization of Performance and Cost of Serverless Applications,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 615–632, 2021, doi: 10.1109/TPDS.2020.3028841.

[28] J. Kim, J. Park, and K. Lee, “Network resource isolation in serverless cloud function service,” *Proc. - 2019 IEEE 4th Int. Work. Found. Appl. Self**

*Syst. FAS*W 2019*, pp. 182–187, 2019, doi: 10.1109/FAS-W.2019.00051.

[29] T. Elgamal, A. Sandur, K. Nahrstedt, and G. Agha, “Optimizing cost of serverless computing through function fusion and placement,” *Proc. - 2018 3rd ACM/IEEE Symp. Edge Comput. SEC 2018*, pp. 300–312, 2018, doi: 10.1109/SEC.2018.00029.

[30] G. McGrath and P. R. Brenner, “Serverless Computing: Design, Implementation, and Performance,” *Proc. - IEEE 37th Int. Conf. Distrib. Comput. Syst. Work. ICDCSW 2017*, pp. 405–410, 2017, doi: 10.1109/ICDCSW.2017.36.

[31] M. Villamizar *et al.*, “Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures,” *Serv. Oriented Comput. Appl.*, vol. 11, no. 2, pp. 233–247, 2017, doi: 10.1007/s11761-017-0208-y.

[32] L. N. Hyseni and A. Ibrahimi, “Comparison of the cloud computing platforms provided by Amazon and Google,” *Proc. Comput. Conf. 2017*, vol. 2018-Janua, no. July, pp. 236–243, 2018, doi: 10.1109/SAI.2017.8252109.

[33] Amazon Web Services, “AWS Lambda.” https://aws.amazon.com/lambda/.

[34] M. Stigler, *Beginning Serverless Computing*. 2018.

[35] M. O. Khan, “Fast Delivery, Continuously Build, Testing and Deployment with DevOps Pipeline Techniques on Cloud,” *Indian J. Sci. Technol.*, vol. 13, no. 5, pp. 552–575, 2020, doi: 10.17485/ijst/2020/v13i05/148983.

[36] G. B. Ghantous and A. Q. Gill, “DevOps: Concepts, practices, tools, benefits and challenges,” *Proc. ot 21st Pacific Asia Conf. Inf. Syst. “‘Societal Transform. Through IS/IT’”, PACIS 2017*, 2017.

[37] V. Ivanov and K. Smolander, *Implementation of a DevOps pipeline for serverless applications*, vol. 11271 LNCS. Springer International Publishing, 2018.

[38] P. Riti, “Cloud and DevOps,” *Pract. Scala DSLs*, pp. 209–220, 2018, doi: 10.1007/978-1-4842-3036-7_11.

[39] Y. Guo *et al.*, “Same-Origin Policy : Evaluation in Modern Browsers This paper is included in the Proceedings of the Same-Origin Policy : Evaluation in Modern Browsers,” *Nsdi*, vol. 40, no. 4, pp. 97–112, 2017, [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/sharma%0Ahttps://www.usenix.org/conference/nsdi17/technical-sessions/presentation/vanini%0Ahttps://www.usenix.org/conference/nsdi17/technical-sessions/presentation/kablan%5Cnhttps://.

[40] V. Persico, A. Montieri, and A. Pescape, “On the Network Performance of Amazon S3 Cloud-Storage Service,” *Proc. - 2016 5th IEEE Int. Conf. Cloud Networking, CloudNet 2016*, pp. 113–118, 2016, doi: 10.1109/CloudNet.2016.16.

[41] S. Gs, “An Auto-Scaling Approach to Load Balance Dynamic Workloads for Cloud Systems,” vol. 12, no. 11, pp. 515–531, 2021.

[42] C. Qu, R. N. Calheiros, and R. Buyya, “Auto-Scaling Web Applications in Clouds,” *ACM Comput. Surv.*, vol. 51, no. 4, pp. 1–33, 2018, doi: 10.1145/3148149.

[43] S. Biswas, M. Sohel, M. M. Sajal, T. Afrin, T. Bhuiyan, and M. M. Hassan, “A Study on Remote Code Execution Vulnerability in Web Applications,” *Int. Conf. Cyber Secur. Comput. Sci.*, no. October, pp. 1–8, 2018.

[44] T. Farah, R. Shelim, M. Zaman, M. M. Hassan, and D. Alam, “Study of race condition: A privilege escalation vulnerability,” *WMSCI 2017 - 21st World Multi-Conference Syst. Cybern. Informatics, Proc.*, vol. 2, no. 1, pp. 100–105, 2017.

[45] A. Duarte and N. Antunes, “An Empirical Study of Docker Vulnerabilities and of Static Code Analysis Applicability,” *Proc. - 8th Latin-American Symp. Dependable Comput. LADC 2018*, pp. 27–36, 2019, doi: 10.1109/LADC.2018.00013.

[46] N. Zhao *et al.*, “Large-Scale Analysis of Docker Images and Performance Implications for Container Storage Systems,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 4, pp. 918–930, 2021, doi: 10.1109/TPDS.2020.3034517.

[47] Quay, “Clair Documentation.” https://quay.github.io/clair/.

[48] Aquasecurity, “Trivy.” https://aquasecurity.github.io/trivy/v0.18.3/.

[49] J. Chen and W. Cheng, “Analysis of web traffic based on HTTP protocol,” *2016 24th Int. Conf. Software, Telecommun. Comput. Networks, SoftCOM 2016*, pp. 1–5, 2016, doi: 10.1109/SOFTCOM.2016.7772120.

[50] M. Zadka, *DevOps in Python*. 2019.

[51] J. Prassanna, A. R. Pawar, and V. Neelanarayanan, “A review of existing cloud automation tools,” *Asian J. Pharm. Clin. Res.*, vol. 10, no. September, pp. 471–473, 2017, doi: 10.22159/ajpcr.2017.v10s1.20519.

[52] J. Henkel, C. Bird, S. K. Lahiri, and T. Reps, “Learning from, understanding, and supporting devops artifacts for docker,” *Proc. - Int. Conf. Softw. Eng.*, pp. 38–49, 2020, doi: 10.1145/3377811.3380406.

[53] J. Nadon, *Website Hosting and Migration with Amazon Web Services*.

2017.

[54] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, “Containers and Virtual Machines at Scale,” *Proc. 17th Int. Middlew. Conf. - Middlew. ’16*, pp. 1–13, 2016, [Online]. Available: http://dl.acm.org/citation.cfm?doid=2988336.2988337.

[55] Y. C. Tay, K. Gaurav, and P. Karkun, “A Performance Comparison of Containers and Virtual Machines in Workload Migration Context,” *Proc. - IEEE 37th Int. Conf. Distrib. Comput. Syst. Work. ICDCSW 2017*, pp. 61–66, 2017, doi: 10.1109/ICDCSW.2017.44.

[56] M. S. Chae, H. M. Lee, and K. Lee, “A performance comparison of linux containers and virtual machines using Docker and KVM,” *Cluster Comput.*, vol. 22, pp. 1765–1775, 2019, doi: 10.1007/s10586-017-1511-2.