Magomedbashir Kushtov

Serverless CI/CD pipeline based on Google Cloud Platform

Metropolia University of Applied Sciences

Bachelor of Engineering

Degree Programme in Information Technology

Bachelor’s Thesis

24 April 2022

Abstract

Author:	Magomedbashir Kushtov
Title:	Serverless CI/CD pipeline based on Google Cloud Platform
Number of Pages:	43 pages + 6 appendices
Date:	24 April 2022
Degree:	Bachelor of Engineering
Degree Programme:	Degree Programme in Information Technology
Professional Major:	IoT and Cloud Computing
Supervisors:	Tapio Wikström, Senior Lecturer

This thesis looks at the serverless CI/CD pipeline based on the Google Cloud Platform. The primary aim of this thesis was to design the serverless CI/CD pipeline based on the Google Cloud Platform. To meet the project objective, related project tasks were fulfilled.

The thesis consists of a theory part and a project part. The theory part includes the DevOps principles, Virtualization, Cloud Computing, Continuous Integration and Continuous Delivery processes (CI/CD). The project part focuses on implementing the design of a serverless CI/CD pipeline based on the Google Cloud Platform. First, the tests of Cloud Run and Cloud Build services were done in the Google Cloud Platform to understand their workflow logic. Second, GitHub was connected to the Google Cloud Platform to automate the Continuous Integration and Continuous Delivery process. Third, the Docker image was created to emulate the development part of the DevOps process. This Docker image was proceeded by the Google Cloud Platform and as a result, the serverless deployment of a web application was achieved.

The product of this thesis is the serverless solution for web application development and deployment using the Google Cloud Platform services. Thus, it is possible to avoid unnecessary work efforts and save time and costs compared to implementing and deploying both physical and virtual servers for web applications.
The project uses the module design, therefore, it can be reconfigured and reused according to developers’ needs.

The goal of creating the serverless CI/CD pipeline based on the Google Cloud platform was successfully achieved. This solution will benefit the author of the thesis project, students who are interested in DevOps and Cloud Computing area, as well as DevOps specialists.

Keywords: Google Cloud Platform, GCP, Serverless CI/CD pipeline, GitHub, Docker, DevOps, Artifact Registry, Google Cloud Run, Google Cloud Build

Contents

List of Abbreviations

DevOps:	An abbreviation of two parts: development and operations
IT:	Information Technology
GCP:	Google Cloud Platform, is a cloud computing service provided by Google Inc.
YAML:	A data serialisation language with a simple syntax that stores complex data in a compact and readable format. YAML stands for “Yet Another Markup Language”
CI/CD:	This abbreviates continuous integration (CI) and either continuous delivery or deployment (CD)
VCS:	Version Control System
VM:	Virtual Machine
OS:	Operating System
IaaS:	Infrastructure as a Service Platform
PaaS:	Platform as a Service
SaaS:	Software as a Service
ID:	An abbreviation of the word “Identifier”
KPI:	Key performance indicators.

SHA	Secure Hash Algorithm is a mathematical algorithm that can transform a random range of data into a fixed-length string consisting of letters and numbers.
MiB	Unit of measurement of the amount of information. One mebibyte is equal to 1,048,576 bytes.
IAM	Identity and Access Management

1 Introduction

This is the project type of bachelor’s thesis for the Degree Programme in Information Technology specializing in IoT and Cloud Computing at Metropolia University of Applied Sciences.

Presently, most cloud providers offer serverless solutions for their customers [6, p. 24]. This choice is justified by cost savings, reliability, better fault tolerance, and ease of maintenance. These advantages have been appreciated by DevOps professionals. The combination of cloud technologies and other platforms and tools provides solutions for complicated challenges.

The thesis work aims to design and implement a serverless CI/CD pipeline solution by combining Google Cloud, GitHub, and Docker services. The programming code and infrastructure of the pipeline can be reconfigured and used in the future. The ease of changing Docker images will allow using such a design for various purposes in development. The code for the pipeline will be comprehensive. Therefore, changes in the code will change the behaviour of the design. For example, the database image from the shared google library of Docker images can be added to the infrastructure and connected to the web application to store the data.

The CI/CD pipeline uses deployment servers as endpoints in integration, testing, or delivery systems. The developers upload the source code to the CI/CD repository, and deployment servers fetch the source code from it [8, p. 13]. If a pair or several branches are used during application development, then deployment servers will be located at the ends of these branches as well. Such servers can be either physical servers or virtual ones hosted in the cloud.
This deployment server can be a bottleneck of the whole pipeline since it can be overloaded, hacked, rebooted, or in maintenance. This can seriously disrupt the operation of the entire CI/CD pipeline. Therefore, the more reliable and functional solution will be to change deployment servers to the serverless version of the CI/CD pipeline based on microservices. Thus, the cloud provider

allocates resources for the application code automatically. The customer is no longer concerned about the technical characteristics of the server and its maintenance. This solution will benefit DevOps professionals, the author of this thesis, and other IT students who are interested in the DevOps path.

2 Fundamentals of DevOps

Previously, IT jobs were strictly divided into areas. The IT department of companies had to have specific professionals for each area. The developers were writing software, while system engineers were responsible for IT infrastructure and network engineers were working in the networking area. The rapid development of the latest trends in the information technology industry has demonstrated the need for new specializations [3, p. 9]. With the progress of cloud technologies, demand for specialists in cloud areas has arisen. Noticeably, the increasing number of employees raised the expenditures of the companies and slowed down the entire work process [3, p.11].

An original approach was required to get rid of the cumbersome and clumsy typical development process. Automation is a great solution to these issues. Applying the new method increased the ability of organizations to provide high-quality solutions in the shortest possible time. There was a need for new professionals, who had skills in the *development* and *operations* areas. Such specialists got their name from a combination of these words, namely DevOps engineers [6, p.2].

Using the DevOps principle does not apply to banking or energy companies and companies that process their customers' personal data. These companies need stable and highly secure products. However, the principle of DevOps works well for companies, for example, who develop websites or sell software products. The DevOps principle is also well suited to large corporations that develop software for their own use.

In “typical” development, also known as a “Waterfall” model, the stages go one after another. Meaning, that the team in the “Build” stage does not start building the application before receiving the technical requirements. As well as the “Deployment” stage does not start before it receives the complete product. And lastly, the “Feedback” stage does not send any information to the “Plan” team before a full analysis of received feedback. This development principle is too cumbersome and takes time to coordinate each step. However, the “Waterfall” model can give excellent results. Usually in this model, the order of technical requirements is clear and precise. Development management is quite easy, and there are strictly defined dates of deadlines, thus giving the possibility of a more accurate planning time [3, p. 13].

As an alternative to the “typical” approach, DevOps offers a more flexible method. All stages work in parallel. Due to this, high development speed and cost-saving are achieved.

With all the advantages of DevOps, it has disadvantages as well. The incompleteness of the testing cycle by skipping manual tests may cause critical errors in production. Additionally, the requirements for DevOps engineers are quite high, and often require high qualifications in many areas. An elevated level of professionalism also applies to the management team.

Currently, the DevOps engineer participates in software development and supports the entire process from development to implementation. To provide high-level quality applications, DevOps engineers' required skills include programming, scripting, and expertise in cloud technologies, version control systems, virtualization, operating systems, and network technologies [1, p.27,29].

To serve the innovative approach in the DevOps continuous integration and continuous delivery environment, new tools appeared to automate processes. These tools cover system build configuration management tests, version control systems, application deployment and delivery, as well as monitoring tools. These tools are increasing each day, providing more opportunities to automate DevOps processes [12, p. 31].

This chapter covers some of the technologies used in the DevOps environment. Such as integration and delivery systems, virtualization, and cloud computing. The DevOps process will also be considered in more detail.

2.1 DevOps Lifecycle

The continuous manner of DevOps leads developers to use the infinite loop to show the relationship between the stages of the DevOps lifecycle. Regardless of the external consistency of the cycle, it represents the need for endless cooperation and repetitive processes during the life cycle [2, p. 17].

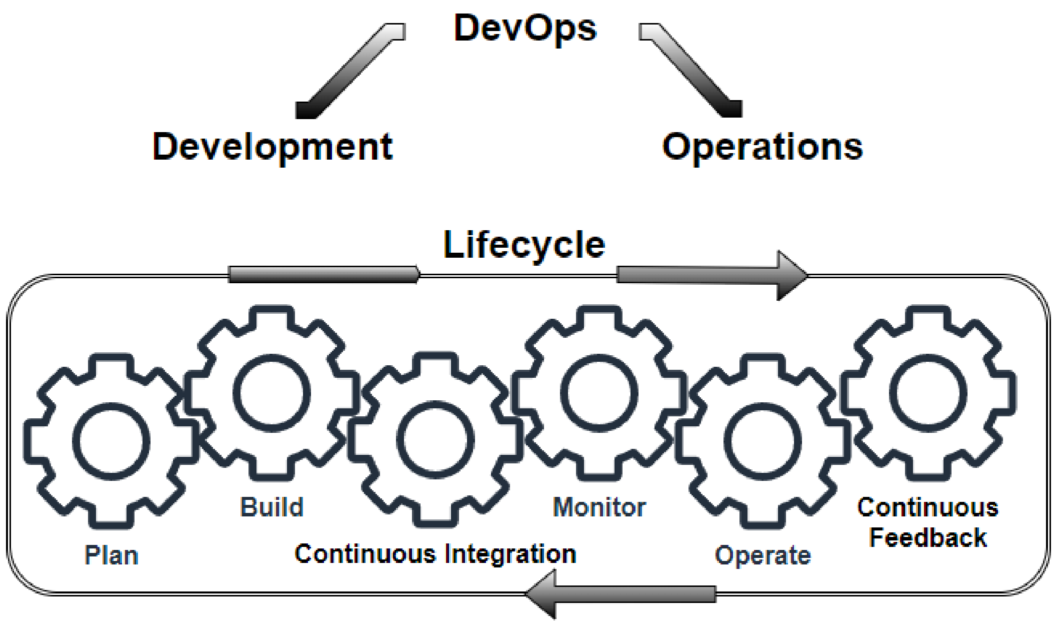


Figure 1. A conceptual model of the DevOps lifecycle (Adapted from [2, p. 18]).

The DevOps life cycle in Figure 1 shows six stages: Plan, Build, Continuous Integration and Deployment, Monitor, Operate, and Continuous Feedback. The main idea of the life cycle of DevOps is the infinite loop of all processes. These six stages are divided into two parts. The Plan, Build, Continuous Integration and Deployment belong to the “Development” part, while Monitor, Operate, and Continuous Feedback go to the “Operations” part [2, p. 18].

These stages should be described in more detail. Each of these stages has features, requirements, and rules. The responsibilities of each stage are strictly defined, and the successful completion of the work of the entire cycle depends on the quality of work of each of them.

2.1.1 Plan

The **“Plan”** stage is continuous planning based on lean principles, by understanding own resources and outcome results, continually adapting, evaluating progress, verifying customer requirements, and adjusting the path as needed to provide elasticity. The main goal of planning is to minimize costs and time to find the most profitable option, while constantly improving the quality of the product.

After implementing the DevOps life cycle, the “Continuous feedback” stage provides information to the “Plan” stage to increase excellence. The “Plan” stage is the start point of the DevOps lifecycle. All ideas for a future project are born at this stage. The customer requirements and the developers' possibilities are also discussed here [2, p. 19]. The direction of development is chosen based on the conclusions of the “Plan” stage. However, edits and changes can be constantly added and transferred to the development stage, thereby confirming the principle of the DevOps approach, the continuous work of all stages. This stage is one of the most important stages because it sets the pace and quality of the entire project. [2, p. 16]

2.1.2 Build

The **“Build”** stage starts next. The working process goes according to the “Plan” stage results. Previously selected tools and techniques for the implementation of the project are used in this stage.

The **“Build”** stage is the continuous process of collaborative development. The collaborative development process allows development, management, and testing teams to continuously deliver high-quality software. This consists of multi-platform development, programming language support, and lifecycle management. The outcome of the **“Build”** stage must be a usable product to an expected objective [2, p. 19]. However, the coding process does not stop even after obtaining a working version of the product. Attempts to improve the product are constantly ongoing. The constant work of the planning stage may require a change in the operation of the application. Even cardinal changes are possible, such as a change in the programming language, introduction of using cloud technologies or changes in which operating system the future version of the product will work on.

The following **“Continuous Integration and Deployment”** stage will begin its work after uploading a completed product to the Continuous Integration and continuous deployment platform with repositories and version control system. Such a platform is an essential part of the DevOps life cycle.

2.1.3 Continuous Integration and Continuous Deployment

The **“Continuous Integration and Deployment”** (CI/CD) stage provides a continuous pipeline that automates key processes. This reduces the resource wait times, and demands of rewriting code, and enables more releases. Automation is an important part of guaranteeing a steady and consistent software release. The goal of automation is to get rid of manual processes as much as possible. The coding, deployment, and delivery steps are automated in DevOps by applying the version control system [2, p. 190]. The version control system or VCS allows to roll back changes in the application code, by releasing one of the previous versions. Thus, making it possible to correct errors, if they have been in the released version of the application.

Besides, the VCS provides an opportunity to conduct an automated test of the code before the deployment of the application. The ability to split development branches into at least two branches, such as Staging and Production, minimizes the occurrence of critical errors in the release version of the product. If the application is complex, the version control system additionally provides an excellent opportunity to check and test all dependencies and libraries of the application.

The CI/CD platforms offer online repositories and a version control system. A useful advantage of using VCS with online repositories is the possibility of round-the-clock work on the project if the development department employees are in different time zones all around the world [2, p.7].

2.1.4 Monitor

The **“Monitor”** stage opens the **“Operations”** part. This stage use product observing tools that capture metrics and key performance indicators (KPIs) in real-time [2, p. 48]. All received information is observed at an early stage, providing that automated testing tracks the characteristics of the application. The condition metrics must be studied and evaluated when an application is examined and deployed. The monitoring system will identify problems related to the operation and quality of the application detected at this stage.

All tests necessary for correct operation are carried out. After that, the monitoring system will generate reports in a clear and accessible format and notify all interested groups. Previously, monitoring tools were able to monitor the application performance or network traffic only. While the current progress of application monitoring tools allows supervising the performance of the server, pipeline condition, the status of containers and database, and end-users experience of using applications. The complete tracking tools grant to obtain extensive information and see a fairly accurate picture of the application's performance [2, p. 31].

2.1.5 Operate

The **“Operate”** stage includes the application's maintenance, direct use, and troubleshooting. At this stage, the application is ready for use by the end-user and represents a workable product. The quality of the product clearly illustrates the quality of the organization of the entire life cycle of DevOps. Product reliability and fast troubleshooting capability are essential parts of this stage. The development team must use reliable deployment solutions on client servers or in cloud storage, to identify problems earlier than they affect the customer side. At this stage, customers of the product may notice imperfections and defects in the application. In the future, all this information will be analyzed, and modifications will be made [2, p. 20]. Based on the above, it is important to have rapid product technical support, to remove the errors as fast, as possible and reduce the downtime of the application. Constantly repeating the cycle in DevOps helps to correct the application errors whenever they happen. If a critical error occurs, then one of the processes in the previous stages was skipped or poorly produced.

2.1.6 Continuous Feedback

The last stage of the cycle is **“Continuous Feedback”**, after which the entire process will start again. All information related to the application usage is gathered at this stage; the customers’ feedbacks are a major part of it [2, p. 4,44]. The customers’ wishes and suggestions to improve the usability of the application are collected and provided to the “Plan” stage. Furthermore, the problems and errors of an application are collected by DevOps teams to be fixed in the next stages [2, p. 8].

The key principle of DevOps is **“Continuous Everything”**, each stage is repeated constantly. For example, there is no need to wait until the **“Plan”** stage receives the collected data from **“Continuous Feedback”** to keep its work. Instead, it may continue the process of increasing the quality of the current version and provide suggestions for the **“Build”** stage [2, p. 18]. Thus, there are constant attempts to improve the already existing version of the product at all stages simultaneously. The whole DevOps process can be compared with the work of the conveyor, the correct and uninterrupted movement of which is guaranteed by the small rollers of its mechanism.

2.2 Virtualization, Cloud Computing, Continuous Integration and Continuous Delivery Platform

The modern development of computer technologies allows the creation of infrastructures that are quite complex from a constructive point of view. Lots of solutions that were created as separate independent products found their purpose as part of one entire process [11, p. 19].

Cloud services have long ceased to be ordinary Internet storage [4, p. 15]. The constant combination of new and existing software products into cloud services has become a frequent practice for cloud providers. For example, virtualization technologies have merged with cloud service technologies, and have already become an integral part of them [11, p. 113].

2.2.1 Virtualization

Virtualization was invented as a technology that allows emulating the functions of servers. According to virtualization principles, one physical server can contain many virtual servers isolated from each other and running at the same time. All physical hardware of the real server is shared with virtualized servers. Such a server is also called a Host Server. Therefore, a single real server can have the ability to play roles such as a Database, Web, or File server.

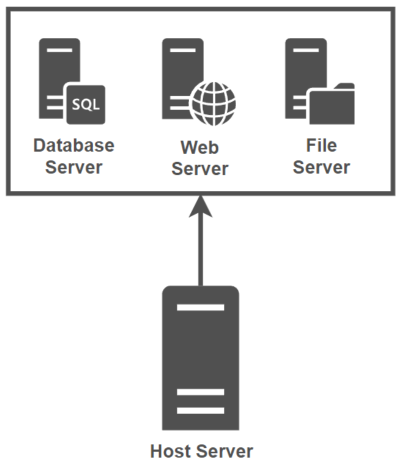


Figure 2. An example of consolidated servers (Adapted from [4, p. 10]).

In this kind of application, the server is also called a *consolidation* [4, p. 9,11]. Still, the evolution of virtualization technology has evolved to the ability to emulate the work of physical devices as well. For example, emulate the operation of network devices, mobile devices, and storage drivers [4, p. 2].

The virtualization technology uses various terms such as Hypervisor, Virtual Machine, Guest OS, Host OS, and Containers. To understand how virtualization works, each of these concepts must be considered and explained.

A ***hypervisor*** is a fundamental part of virtualization technology, which allows for managing multiple operating systems on a single computer. The hypervisor plays the role of an intermediary between the physical server and the virtual machines, providing isolated hardware resources of the physical server for each virtual machine. All interaction between physical servers and virtual machines goes through the hypervisor [5, p. 23]. Figure 3 shows the logic of the working process of virtualization.

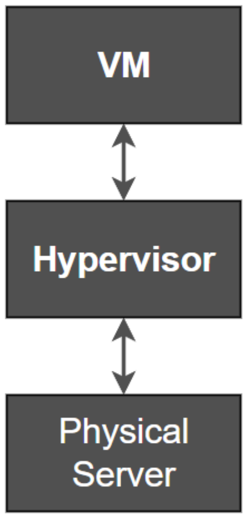


Figure 3. The virtualization logic diagram (Adapted from [4, p. 23])

At the bottom layer is the physical server also known as the Host. At the middle layer is the hypervisor, and at the top layer is the virtual machine also known as Guest.

There are two types of hypervisors. The ***hypervisor type 1*** operates directly on the server. The virtualization software is installed on the server hardware without a preinstalled operating system. This form of installation is called *bare-metal implementation*. The type 1 hypervisor is more effective in utilizing servers' resources since it can communicate with them directly without an extra layer representing an operating system. The less consumption of resources makes the type 1 hypervisor economically preferable to type 2 since there is no need to spend expensive server resources on installing and maintaining the underlying operating system [4, p. 23,26]. Also, hypervisor type 1 provides better security and availability [5, p. 23].

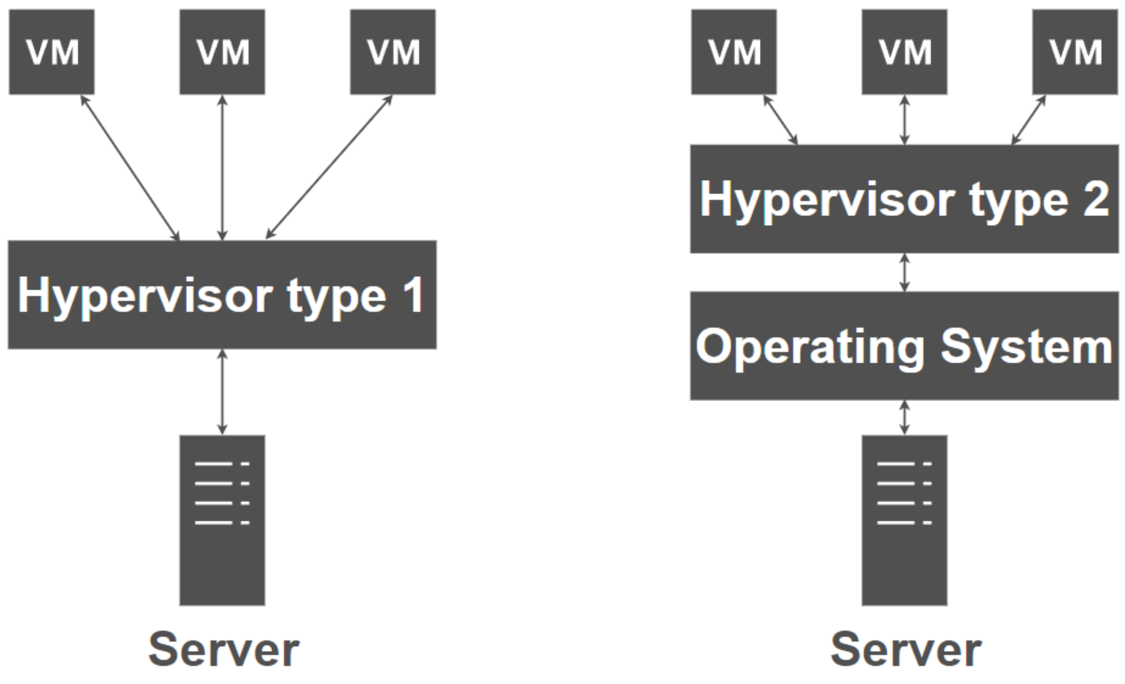


Figure 4. Types of Hypervisors (Adapted from [4, p. 24,25])

The type 2 hypervisor is installed on top of an operating system, referred to as *hosted virtualization*. In the case of the realization of type 2 virtualization, the hypervisor is installed as an application. The hypervisor can operate with resources that the host server can share with it. The operating system installed on the Server is called ***Host OS***, while the operating system installed on the VM is called ***Guest OS***.

The main difference between type 1 and type 2 hypervisors is that type 2 has an extra layer between VM and server where the Host OS is allocated [5, p. 24]. The Host OS makes the type 2 hypervisor less reliable. For example, any updating patch for Host OS that requires a reboot will force reboots of all VMs [4, p. 25].

A Virtual Machine also called a *VM*, is an emulated end–device, which has an operating system and a set of applications that emulates the performance of a real server. As mentioned before the hypervisor provides hardware support to the VM. Therefore, from the point of view of the operating system installed on the virtual machine, its “hardware” is real. The VM can be utilized as a typical server by installing software and applications [4, p. 38].

A physical server can contain many virtual machines, running on top of it. Each of the virtual machines is isolated from other VMs and can have different operating systems and separate roles of servers. In case of VM failure, the isolation also protects other machines from errors that occurred on failed VM [4, p. 37].

Containers are one of the next steps in the evolution of virtualization technology. While a virtual machine emulates the operation of a computer with an installed operating system, containers allow emulating the functions of an operating system. The container does not contain a complete copy of the operating system and shares the kernel with the Host OS [6, p. 63]. Further, an application can be placed in the container, and it can be a database or a web application [7, p. 117]. This structure greatly reduces the size of the container. Instead of using a virtual machine whose files can be several gigabytes in size, the container size can be quite small, a hundred, tens, or even a few megabytes [6, p. 63].

To communicate with the outer world, the containers use ports. For example, the classic MySQL database uses port 3306 for connections. Therefore, if the

MySQL database was placed in the container, then port 3306 must be exposed [8, p. 229].

Another excellent value of containers is that they can be used with the consolidation method described above. Instead of installing separate virtual machines for each server role, the containers with similar server roles can be used [10, p. 24].

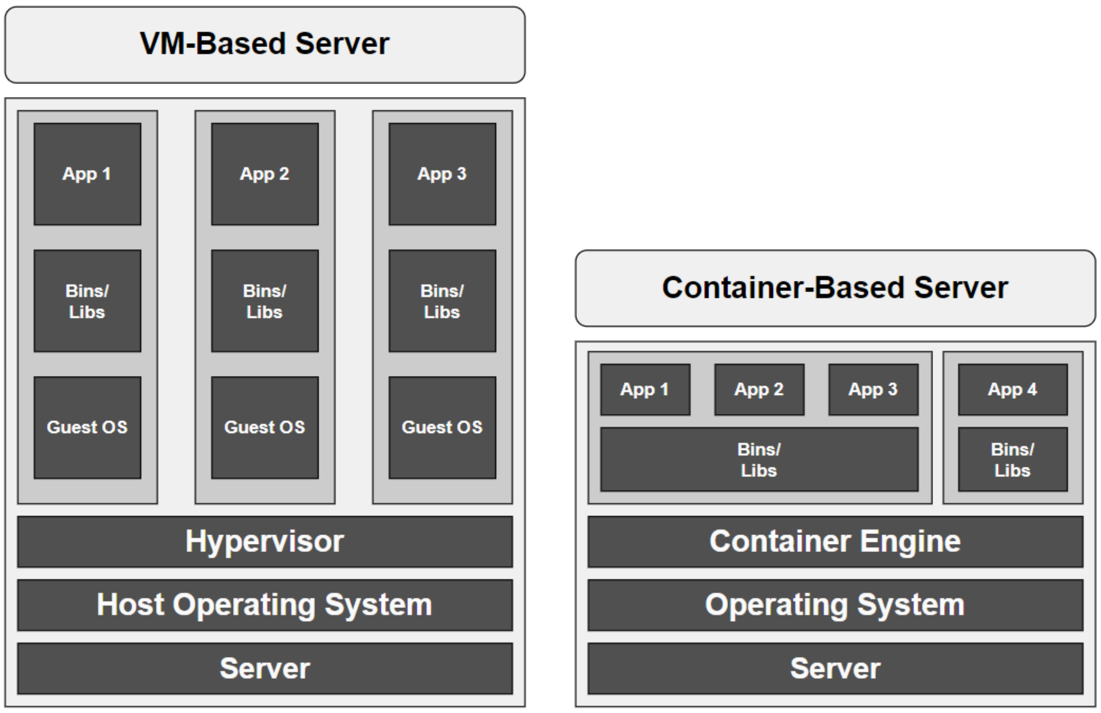


Figure 5. The models of VM-Based and Container-Based Servers (Adapted from [9, p. 12]).

For example, Figure 5 shows the difference between the working process of a VM-Based server and a Container-Based server. A container-based server frees up server resources that can be used to run more containers [10, p. 24]. The container engine is used for management. It allows to create, run, manage, and orchestrate the containers [4, p. 271].

In fundamental, the containers are isolated mini-virtual machines. This kind of advantage makes them a perfect choice of application delivery tools. For example, to run a certain application on the server, the presence of some additional environments or a certain version of programming languages is necessary. In the case of using a container, everything needed can be placed inside. [4, p. 271]. This feature of the container makes it an indispensable tool for software developers [7, p. 117].

2.2.2 Cloud Computing

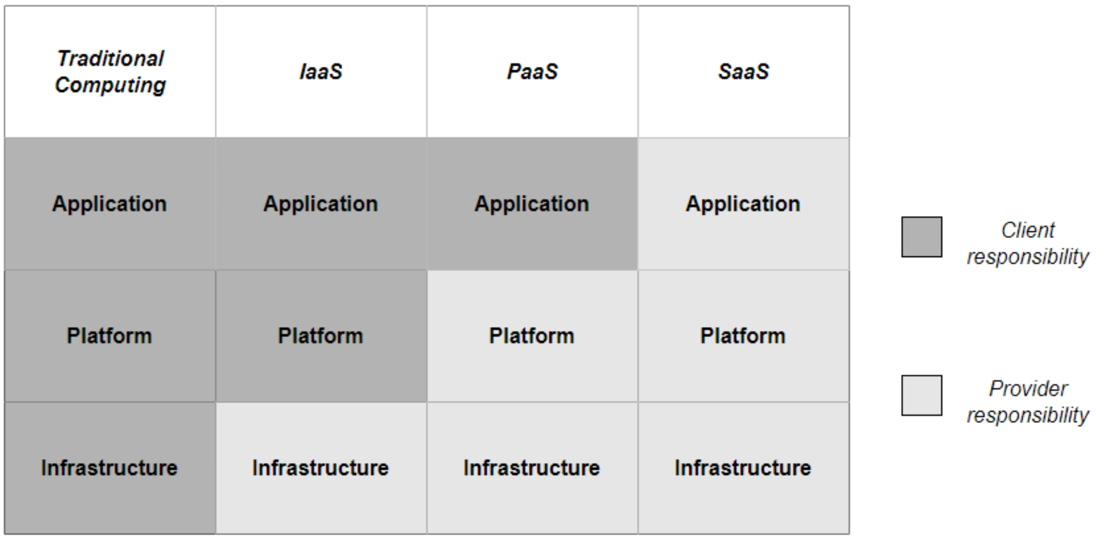
Cloud Computing is a set of virtual services offered by cloud providers to customers, over the network, including the provision of virtual servers, online storage, virtual networks, and software applications. The common name for all these services is “IT as a Service”. [5, p. 431]

The service delivery model of cloud computing has been adapted from the traditional computing model, which is divided into three levels: infrastructure, platform, and application. **Infrastructure** is the level of all physical devices such as servers and their components, network devices, and peripheral devices. The list of devices can be supplemented with power supplies and cooling systems. The ***platform*** is the level that lies on the infrastructure level. It includes servers’ operating systems, firmware and drivers for networks and peripheral devices such as printers, scanners, routers, and switches. The function of this layer is to ensure the correct operation of the infrastructure by providing the necessary software. The ***application software*** level is the level at which the end-user interacts with the installed software. At this level, the comfortable work of the programs necessary for the user is guaranteed. An example would be using a text editor, or even playing a computer game. That is the use of any application that was installed additionally and is not a required part of the server operating system [11, p. 6].

Cloud providers use the same structure to provide services to customers. The main difference is that the services are virtual and located in the provider's cloud. The IT as a Service principle changes the names of these levels from Infrastructure to Infrastructure as a Service (IaaS), from Platform to Platform as a Service (PaaS) and from Application to Software as a Service (SaaS) [11, p. 10].

The ***Infrastructure as a Service (IaaS)*** provides computing resources such as virtual servers, online storage, and virtual networks. For example, a cloud provider can offer a virtual server of a certain capacity, but without an installed operating system. In this case, the client is renting the virtual server and can install the operating system of his choice [13, p. 477].

When talking about infrastructure, the shared responsibility model should be mentioned. The shared responsibility model is the model that clarifies the responsibility of cloud resources between customers and cloud providers, where the cloud provider guarantees the efficient operation of the chosen level [11, p. 77]. Figure 6 shows the shared responsibility model.



Traditional Computing	*IaaS*	*PaaS*	*SaaS*
▨ Application	▨ Application	▨ Application	□ Application
▨ Platform	▨ Platform	□ Platform	□ Platform
▨ Infrastructure	□ Infrastructure	□ Infrastructure	□ Infrastructure

▨	*Client responsibility*
□	*Provider responsibility*

Figure 6. Shared responsibilities in traditional and cloud computing (Adapted from [11, p. 104]).

The ***Platform as a Service (PaaS)*** is the level that includes the infrastructure as a Service level and additionally provides the servers' operating systems. The customer can rent the virtual server with a certain capacity and operating system for his needs. For example, a developer team can rent a server to test their application [13, p. 477]. The cloud provider is responsible for the accurate work of the underlying infrastructure and the operating system software of the server as well [11, p. 79].

Software as a Service (SaaS) provides applications as a service. As the name of that service says, the cloud provider could provide online access to the software application. The cloud provider takes care of infrastructure and platform levels, therefore the customer is responsible only for the application use. An example of a SaaS solution could be a Google Docs service. Where a customer can use a browser to get the functionality of a text editor, without installing any additional software on his computer [11, p. 80].

2.2.3 Continuous Integration and Continuous Delivery Platform

Continuous Integration and Continuous Delivery Platform, also known as CI/CD platform is a system of standardized procedures and automation. In general, the CI/CD platform is the repository with a version control system. Software developers upload the application files or source code to the repository periodically. Each upload gets the ID number which makes it easier to control the versions of uploaded files [6, p. 37].

The source code repository can be split into branches; therefore, each development team can work with the source code in a branch, specially created for that team. Additionally, teams of developers can work with repositories, as CI/CD provides round-the-clock access [8, p. 144].

To understand the principles of operation of CI/CD platforms, GitHub, will be considered. GitHub uses terms such as **“commit”**, **“push”**, **“pull”**, or **“fetch”**. By applying them, the developers can upload, download, delete, or roll back the source code or its changes [8, p. 140]. Figure 7 shows the usual set of commands for the GitHub platform.

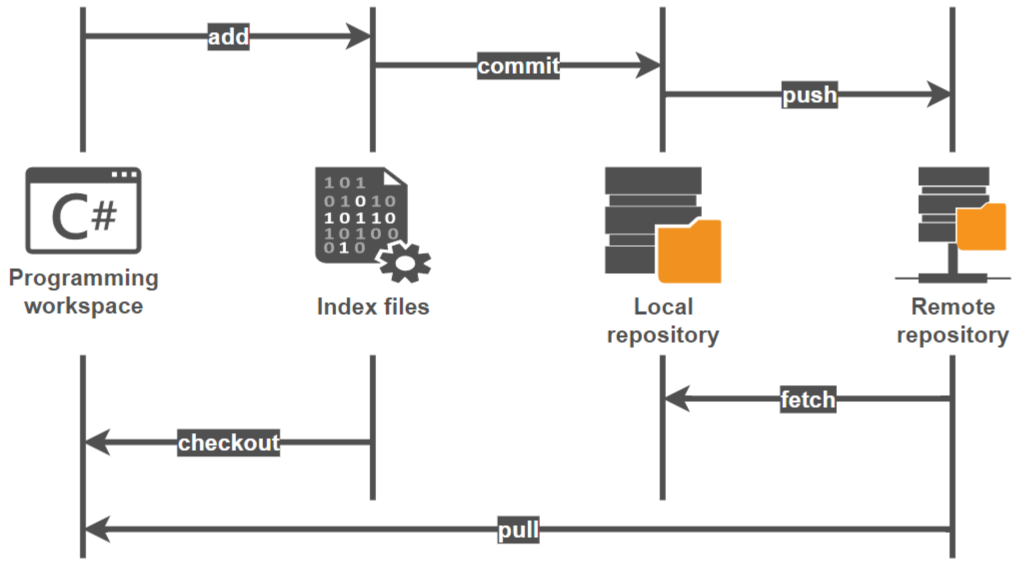


Figure 7. Commands of GitHub CI/CD.

The ***add*** command adds the file or list of files to the tracking system, for subsequent placement in the local repository. For example, the *git add config.py* command will index the config.py file for the tracking system. The developer indexes the config.py file, which in the future should be placed in the local repository. If changes have been made in the config.py file, then the commands must be re-executed, since the tracking system remembers the state of the file at the time when the *add* command was applied last [8, p. 143].

The ***commit*** takes the indexed file and copies it into the local repository. In the future, all files from the local repository will be copied to the remote repository. Every time a change has been made to the files; the *commit* command must be re-applied [8, p. 140].

The ***push*** command will push the latest commit to the remote server. Thus, the local and remote repositories will be synchronized. All changes made in the local repository will be placed in the remote repository. It is possible to send files also to a specific branch using attributes [8, p. 141]. For example, *git push origin main* will push “committed” files to the *main* branch of the *origin* repository.

The ***fetch*** command downloads the changes from the remote repository to the specific folder on the local computer. It does not change any files of the local repository unless the *merge* command was issued [8, p. 141].

The ***checkout*** command switches the branches, but another feature of the checkout command is that it can undo changes in the working directory. For example, *git checkout – filename* will revert the file's changes to the state it was before the commitment to the local repository. In other words, the changes to the file in the working directory of the project will be reset [8, p. 141].

The ***pull*** command downloads the changes from the remote repository that other developers have made. Those changes will appear in the local repository [8, p. 141].

3 Design of serverless CI/CD pipeline, Services and Tools

Commonly, servers at various stages of work are involved in the process of integration and delivery [8, p. 14]. These servers can be either real machines or virtual ones, and they also can be hosted in the clouds. It is important to understand that hosting servers in the cloud does not make it a serverless solution. Such servers still exist, albeit in the virtual space. Figure 8 shows a typical implementation of such a solution.

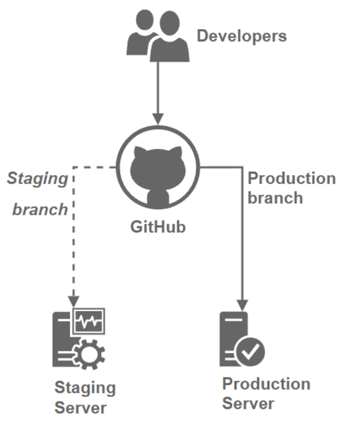


Figure 8. CI/CD pipeline based on physical or virtual servers.

The thesis aimed to design a CI/CD pipeline based on the Google Cloud Platform and get rid of any servers in the integration, delivery, and deployment process. Instead of using servers, Google Cloud Platform services are used.

Figure 9 demonstrates the complete realization of a serverless solution for CI/CD pipeline based on GCP, with the additional platforms and tools.

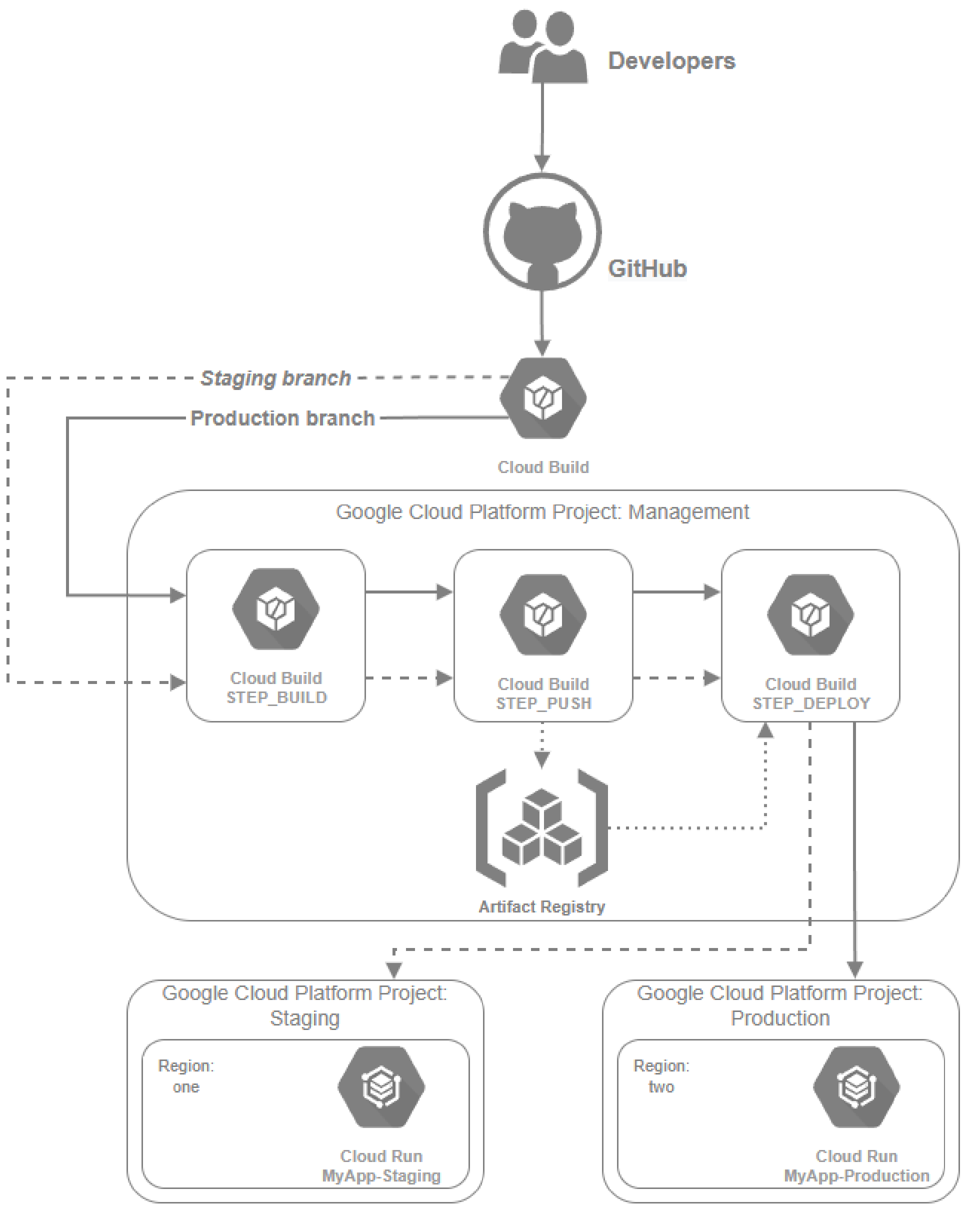


Figure 9. Serverless CI/CD pipeline based on GCP.

The next chapters describe setting up the serverless CI/CD pipeline process step-by-step, from scratch to the complete stage. The whole CI/CD procedure work will be emulated, and the code and all necessary files will be provided as well.

The GCP services used in this project are Cloud Run Service, Cloud Build and Artifact Registry. Another platform is GitHub, which is connected to the GCP. The source code is pushed from GitHub to the GCP, and the trigger on GCP launches the build, test and deploy actions. According to the branch name, the source code of the web application is provided to the relevant Google Run service.

The GCP has 3 projects inside, Management, Staging and Production. The Management Project is a major project. It has common connections with GitHub and other projects. It also has all the necessary rights, and permissions to process the received code, and then send the results to other projects. The management project includes three Cloud Build services, representing the build, push and deploy stages. After receiving the source code from GitHub, each of these stages is turned one after another. In the first STEP_BUILD stage, the code is managed, and the build process begins. At this stage, the desired container image is downloaded from the Google Docker library. Based on this image, the required image is created with included web server files and web application code. The STEP_PUSH stage pushes the image to the Artifact Registry, where the image is stored as a copy, and then pushed to the STEP_DEPLOY stage. Finally, the stage STEP_DEPLOY sends the image to the Cloud Run service with instructions and attributes to create a web server with the corresponding code for the web application.

The Staging project contains the Cloud Run service. After receiving the image and its attributes from the Management project, it deploys a serverless web application according to the received information. The Production project does the same steps; the difference is only the project’s destination.

3.1 Google Cloud Services

The Google Cloud projects are applied to manage services, and control permissions and collaborators. For the implementation of serverless CI/CD pipeline, 3 projects are created. To create a new project, the GCP dashboard is used as seen in figure 10.

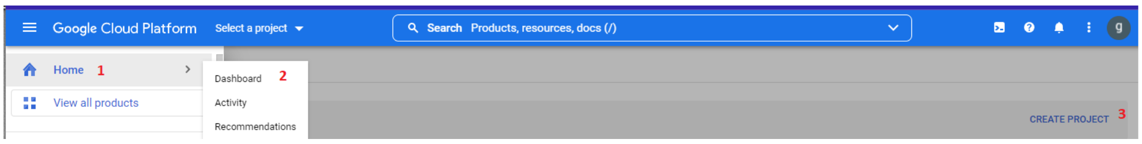


Figure 10. Creating a new project.

The projects Production and Staging include the Cloud Run services. The Cloud Run service is a fully managed serverless solution for web applications based on containers. This service also allows converting any back-end code written in any supported programming language with dependencies into a service function.

To add Cloud Run service to the Staging project, the Staging project is selected and Cloud Run service is chosen from the left menu as shown in figure 11.

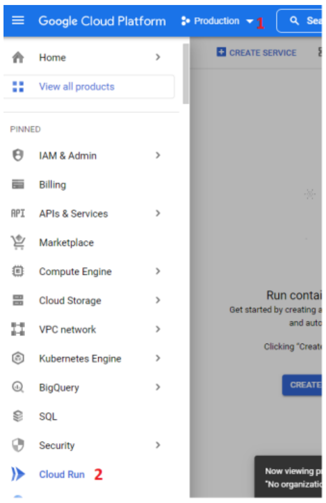
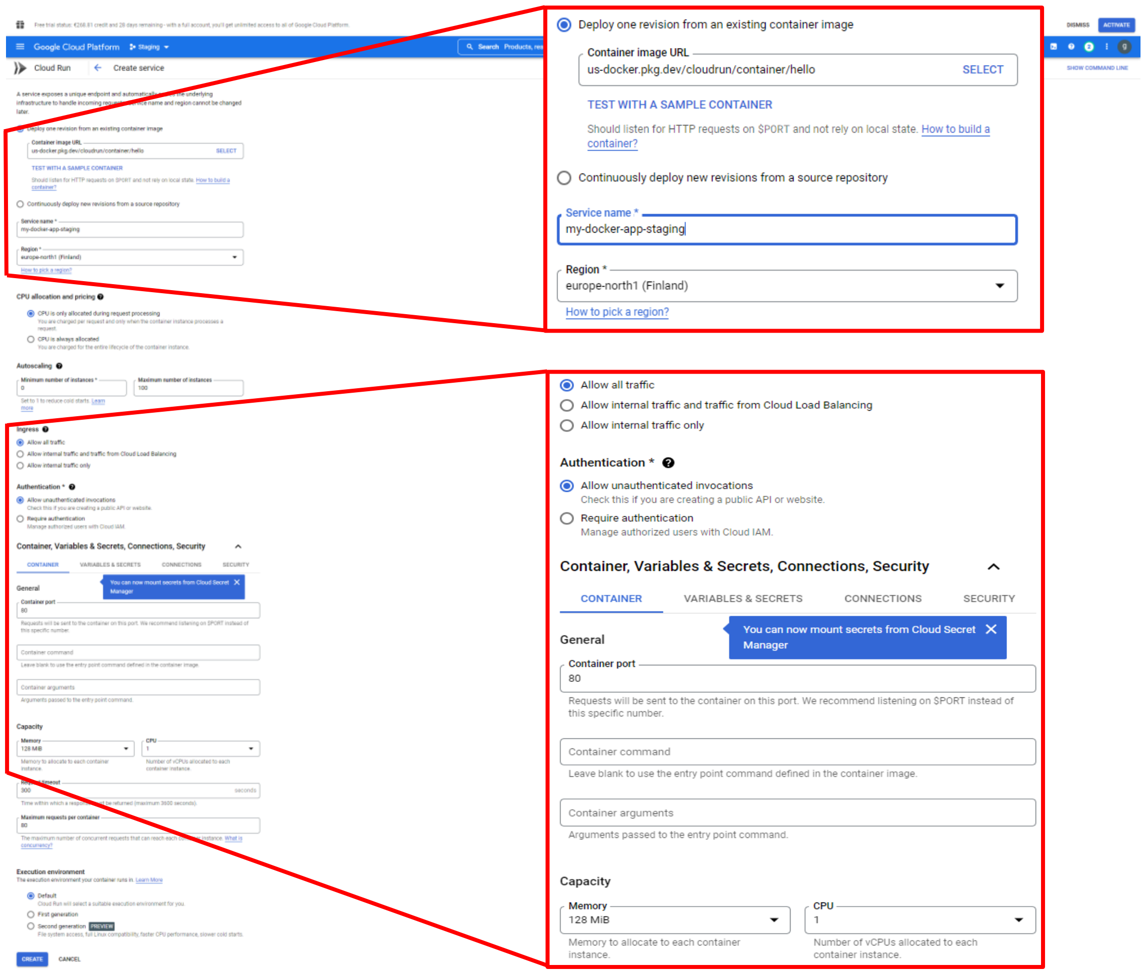


Figure 11. Adding Cloud Run service to the Production project.

By pressing the **create** button the Cloud Run service is started.

The next pages showed the settings for the Cloud Run service. Figure 12 demonstrates the setting changes of the Staging project according to requirements.


Free trial status: €268.81 credit and 28 days remaining - with a full account, you'll get unlimited access to all of Google Cloud Platform.

Google Cloud Platform · Staging

Cloud Run ← Create service

A service exposes a unique endpoint and automatically scales the underlying infrastructure to handle incoming requests. Service name and region cannot be changed later.

(●) Deploy one revision from an existing container image

Container image URL: us-docker.pkg.dev/cloudrun/container/hello — SELECT

TEST WITH A SAMPLE CONTAINER

Should listen for HTTP requests on $PORT and not rely on local state. How to build a container?

( ) Continuously deploy new revisions from a source repository

Service name *: my-docker-app-staging

Region *: europe-north1 (Finland)

How to pick a region?

CPU allocation and pricing

(●) CPU is only allocated during request processing
You are charged per request and only when the container instance processes a request.

( ) CPU is always allocated
You are charged for the entire lifecycle of the container instance.

Autoscaling

Minimum number of instances *: 0 — Set to 1 to reduce cold starts. Learn more

Maximum number of instances: 100

Ingress

(●) Allow all traffic

( ) Allow internal traffic and traffic from Cloud Load Balancing

( ) Allow internal traffic only

Authentication *

(●) Allow unauthenticated invocations
Check this if you are creating a public API or website.

( ) Require authentication
Manage authorized users with Cloud IAM.

Container, Variables & Secrets, Connections, Security

CONTAINER | VARIABLES & SECRETS | CONNECTIONS | SECURITY

You can now mount secrets from Cloud Secret Manager

General

Container port: 80
Requests will be sent to the container on this port. We recommend listening on $PORT instead of this specific number.

Container command
Leave blank to use the entry point command defined in the container image.

Container arguments
Arguments passed to the entry point command.

Capacity

Memory: 128 MiB — Memory to allocate to each container instance.

CPU: 1 — Number of vCPUs allocated to each container instance.

Request timeout: 300 seconds — Time within which a response must be returned (maximum 3600 seconds).

Maximum requests per container: 80 — The maximum number of concurrent requests that can reach each container instance. What is concurrency?

Execution environment
The execution environment your container runs in. Learn More

(●) Default
Cloud Run will select a suitable execution environment for you.

( ) First generation

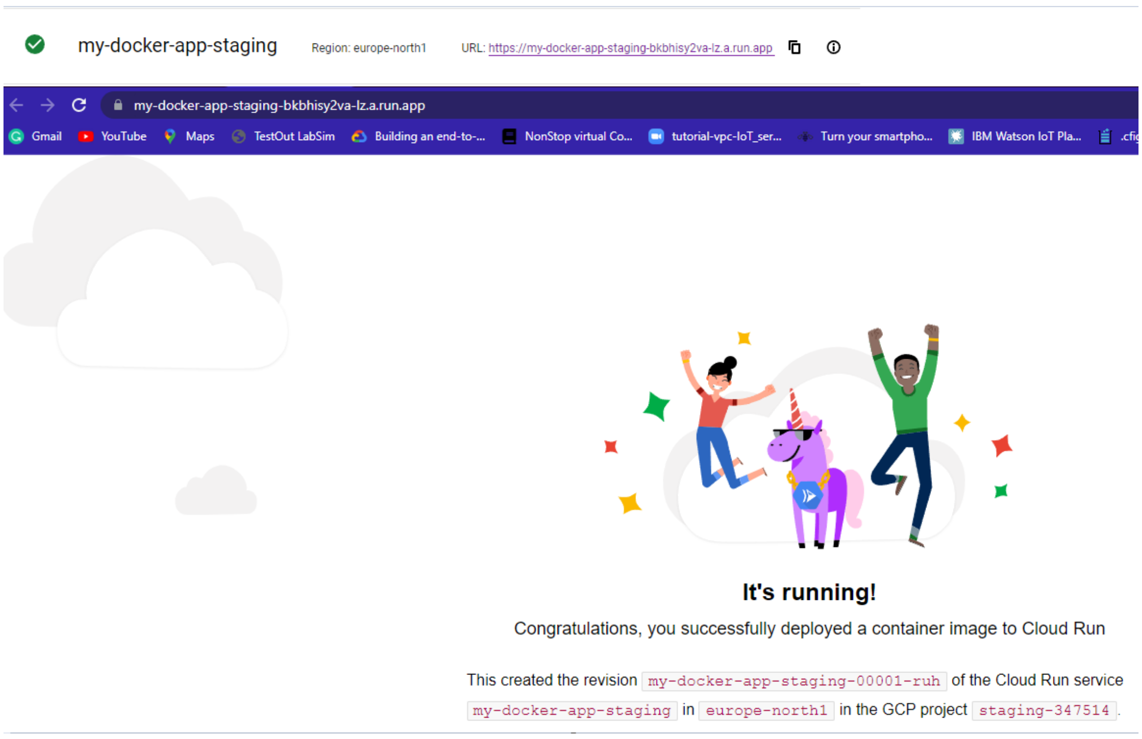
( ) Second generation PREVIEW
File system access, full Linux compatibility, faster CPU performance, slower cold starts.

CREATE   CANCEL

Figure 12. The setting of Cloud Run service of Staging project.

In the **Container image URL** settings, the **“TEST WITH A SAMPLE CONTAINER”** button was pressed. This adds the test “hello” container provided by Google as a sample. The **Service name** was changed to *my-docker-app-staging*. The **Region** setting was changed to *europe-notrh1 (Finland)*. In the **Ingress** setting, the ***“Allow all traffic”*** radio button is selected. In the **Authentication** setting, the ***“Allow unauthenticated invocations”*** radio button was selected. In **Container, Variables & Secrets, Connections,**

Security setting the *container port* was changed from 8080 to 80 port number. The **Capacity** setting was changed to the minimum memory allocated to each container, which is *128 MiB*. The same settings were applied to the Production project, with a changed **Service name** as *my-docker-app-production*, and **Region** as *europe-west1 (Belgium)*.



my-docker-app-staging Region: europe-north1 URL: https://my-docker-app-staging-bkbhisy2va-lz.a.run.app

my-docker-app-staging-bkbhisy2va-lz.a.run.app

It's running!

Congratulations, you successfully deployed a container image to Cloud Run

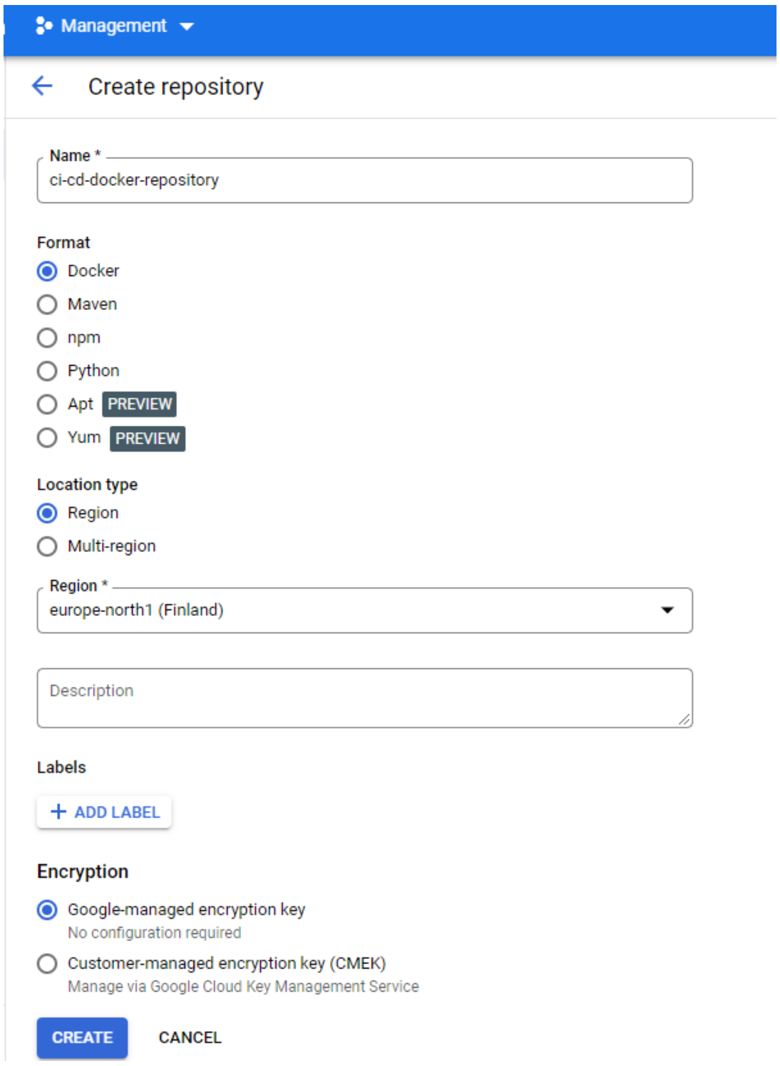
This created the revision `my-docker-app-staging-00001-ruh` of the Cloud Run service `my-docker-app-staging` in `europe-north1` in the GCP project `staging-347514`.

Figure 13. Successful deployment of the container.

Figure 13 demonstrates the successful deployment of the demo container for the Staging project. The same result of deployment was achieved on the Production project as well. The results screen in Figure 13 demonstrates the revision of the container image, the Cloud Run service name, the region, and the project name.

The project Management manages the main processes in this CI/CD pipeline. One of them is the Artifact Registry service. The Artifact Registry creates a repository with copies of all pushed images that pass through the Google Cloud infrastructure.

To create an Artifact Registry service, the **Management** project was selected, and from the left menu panel, the **Artifact registry** was selected. The GCP shows the warning that ***"Artifact Registry API must be enabled"*** since it is used for the very first time. The ***"enable"*** radio button was pressed. On the next page ***"create repository***" button was pressed. After that, the settings page of the Artifact Registry repository opens. Figure 14 shows the settings that are required to be set.



Name *
ci-cd-docker-repository

Format
[✓ Docker] Maven npm Python Apt PREVIEW Yum PREVIEW

Location type
[✓ Region] Multi-region

Region *
europe-north1 (Finland)

Description

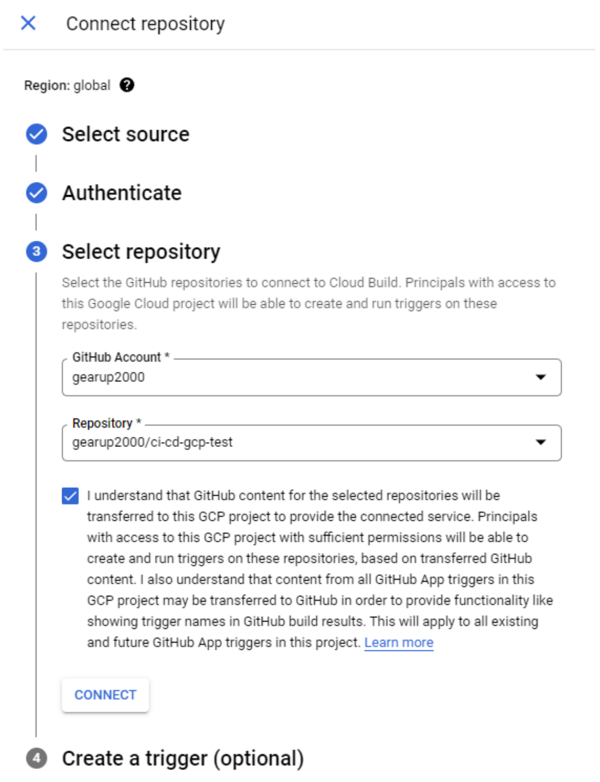
Labels

Encryption
[✓ Google-managed encryption key
No configuration required]
Customer-managed encryption key (CMEK)
Manage via Google Cloud Key Management Service

Figure 14. The Artifact Registry settings.

The name of the repository was changed to *ci-cd-docker-repository*, and the region was selected as e*urope-north1 (Finland)*.

The next step was to create triggers inside the Management project. From the left menu panel, the **Cloud Build** was selected, and after that the warning page that ***"API needs to be enabled"*** opened. After pressing the **enable API** button, the page of Cloud Build opens. On the left side, the ***Triggers*** button was pressed. The interconnection process of GitHub and GCP is started by pressing the ***manage repositories*** button on the top side. On the following page, the ***connect repository*** button was pressed. This opens the settings menu, with choices of **Sources**, **Authenticate**, **Repository**, and **triggers**. Since the author is using GitHub, the following Source was chosen.
Figure 15 demonstrates the *Connect repository* menu.



Connect repository

Region: global

Select source

Authenticate

3 Select repository

Select the GitHub repositories to connect to Cloud Build. Principals with access to this Google Cloud project will be able to create and run triggers on these repositories.

GitHub Account *: gearup2000

Repository *: gearup2000/ci-cd-gcp-test

✓ I understand that GitHub content for the selected repositories will be transferred to this GCP project to provide the connected service. Principals with access to this GCP project with sufficient permissions will be able to create and run triggers on these repositories, based on transferred GitHub content. I also understand that content from all GitHub App triggers in this GCP project may be transferred to GitHub in order to provide functionality like showing trigger names in GitHub build results. This will apply to all existing and future GitHub App triggers in this project. Learn more

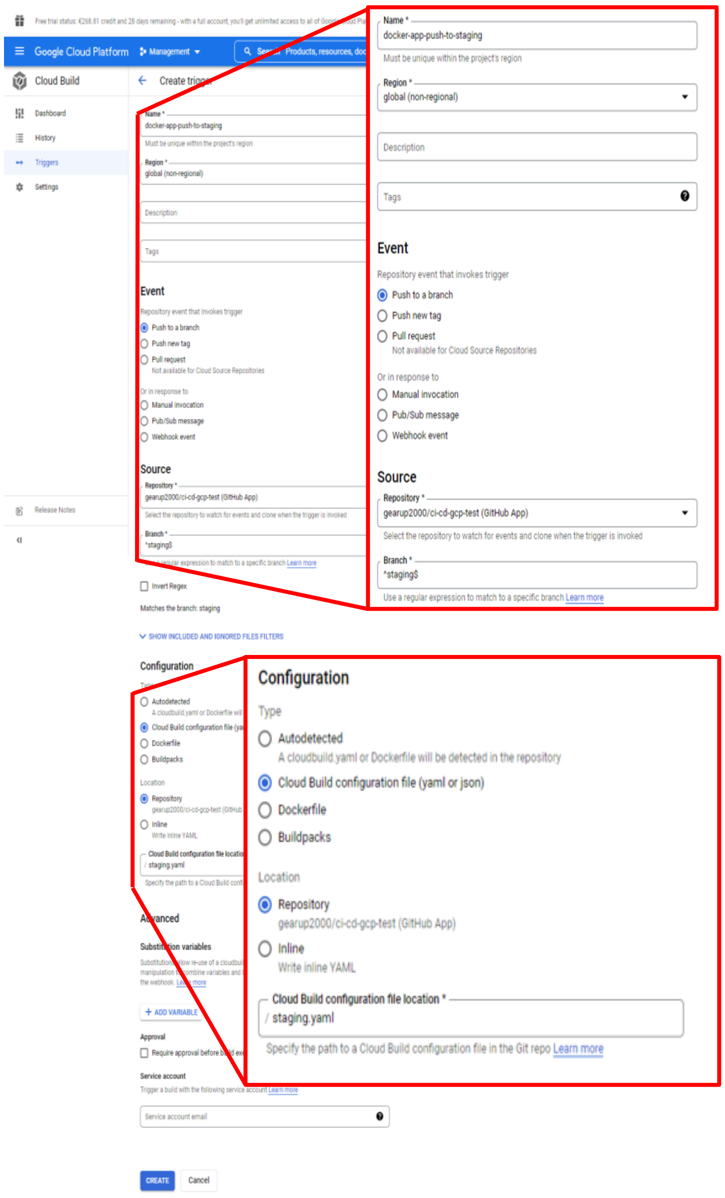
CONNECT

4 Create a trigger (optional)

Figure 15. Connecting to GitHub repository.

After choosing GitHub, the **Authenticate** menu asked to authenticate the user. Next, the GCP asks to install the Google Cloud Build app on the repositories that will be connected to the GCP. The Select repository step is required to select the account and repositories to connect to the Cloud Build service. The confirmation with conditions of GCP is also required to accept. The last step of ***creating a trigger*** was skipped by pressing the done button since the specific triggers will be created later.

The triggers are needed for the Production branch as well as for the Staging branch. They are added by pressing **three small dots** (view actions menu) right of the repository name and selecting ***add trigger*** choice from the drop menu. The next trigger setting page is required to be filled. Figure 16 shows the changed settings.



Name *
docker-app-push-to-staging
Must be unique within the project's region

Region *
global (non-regional)

Description

Tags

Event
Repository event that invokes trigger
(●) Push to a branch
( ) Push new tag
( ) Pull request
Not available for Cloud Source Repositories
Or in response to
( ) Manual invocation
( ) Pub/Sub message
( ) Webhook event

Source
Repository *
gearup2000/ci-cd-gcp-test (GitHub App)
Select the repository to watch for events and clone when the trigger is invoked
Branch *
^staging$
Use a regular expression to match to a specific branch Learn more
[ ] Invert Regex
Matches the branch: staging
SHOW INCLUDED AND IGNORED FILES FILTERS

Configuration
Type
( ) Autodetected
A cloudbuild.yaml or Dockerfile will be detected in the repository
(●) Cloud Build configuration file (yaml or json)
( ) Dockerfile
( ) Buildpacks
Location
(●) Repository
gearup2000/ci-cd-gcp-test (GitHub App)
( ) Inline
Write inline YAML
Cloud Build configuration file location *
/ staging.yaml
Specify the path to a Cloud Build configuration file in the Git repo Learn more

Advanced
Substitution variables
+ ADD VARIABLE
Approval
[ ] Require approval before build executes
Service account
Trigger a build with the following service account Learn more
Service account email

Figure 16. Trigger settings for the Staging branch.

The **Name** of the trigger was changed to *docker-app-push-to-staging,* the repository **Event** was changed to *Push to a branch* the **Source** was selected as *gearup2000/ci-cd-gcp-test (GitHub App),* the **Branch** was changed to ^*staging$*, the **Configuration** type was changed to *Cloud Build Configuration file (YAML or JSON),* the **Location** of the configuration file was changed to the *Repository* (since it is located in the GitHub repository) and the name of that file was changed to staging.yaml. The staging.yaml (Appendix 5) file includes the configuration for the trigger, and it is explained in the next chapters. At last, the *create* button was pressed. The same settings were applied to create a second trigger for the **Production** branch, which is the **main** branch in GitHub, with changes of the name to be *docker-app-push-to-production*, the **Branch** setting as ^*main$*, and the configurations yaml file as production.yaml (Appendix 1).

The last step was to give Service Account Permissions to the Management, Production and Staging projects to interact with each other. The Management project needs permission to deploy Docker images created by Cloud Build services on Cloud Run services of Production and Staging projects, and another permission to push the Docker images to the Artifact Registry service. Figure 17 shows the required permissions.

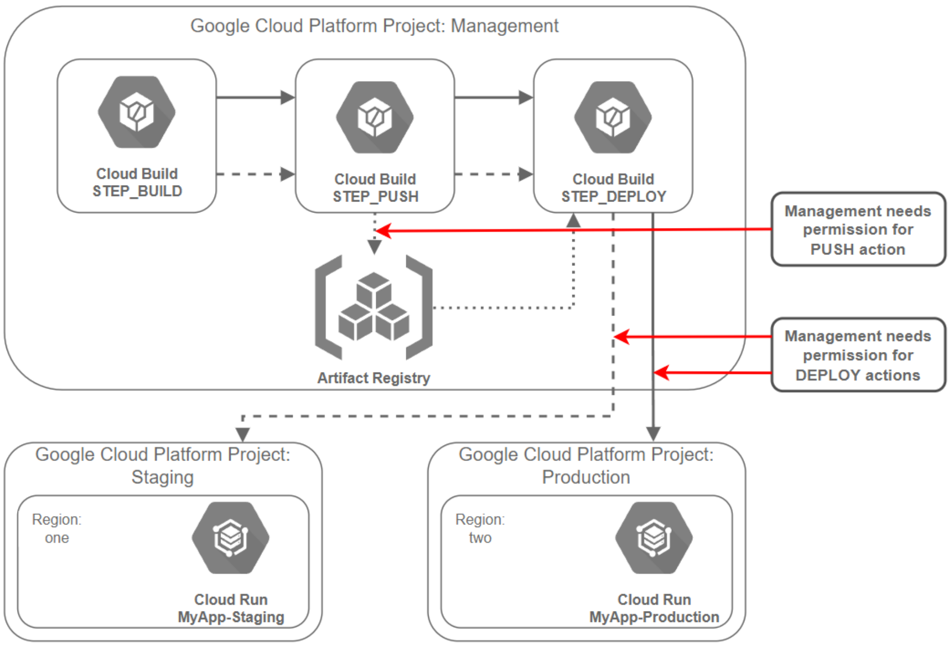
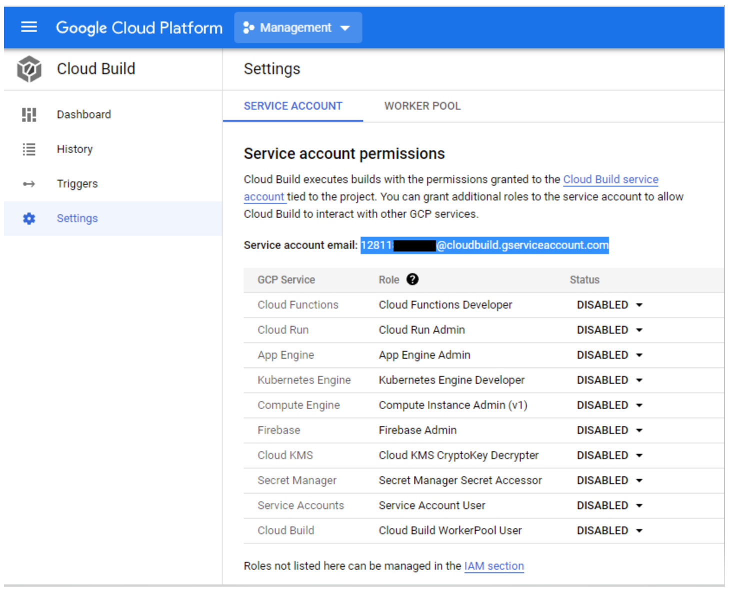

Google Cloud Platform Project: Management
Cloud Build STEP_BUILD
Cloud Build STEP_PUSH
Cloud Build STEP_DEPLOY
Artifact Registry
Management needs permission for PUSH action
Management needs permission for DEPLOY actions
Google Cloud Platform Project: Staging
Region: one
Cloud Run MyApp-Staging
Google Cloud Platform Project: Production
Region: two
Cloud Run MyApp-Production

Figure 17. Permissions for Management project.

To allow the Management project the deployment of the Docker images on Cloud Run services of Production and Staging projects, the **Settings** button on the left menu panel was pressed and the service account email was copied as shown in figure 18.



Settings

Service account permissions

Cloud Build executes builds with the permissions granted to the Cloud Build service account tied to the project. You can grant additional roles to the service account to allow Cloud Build to interact with other GCP services.

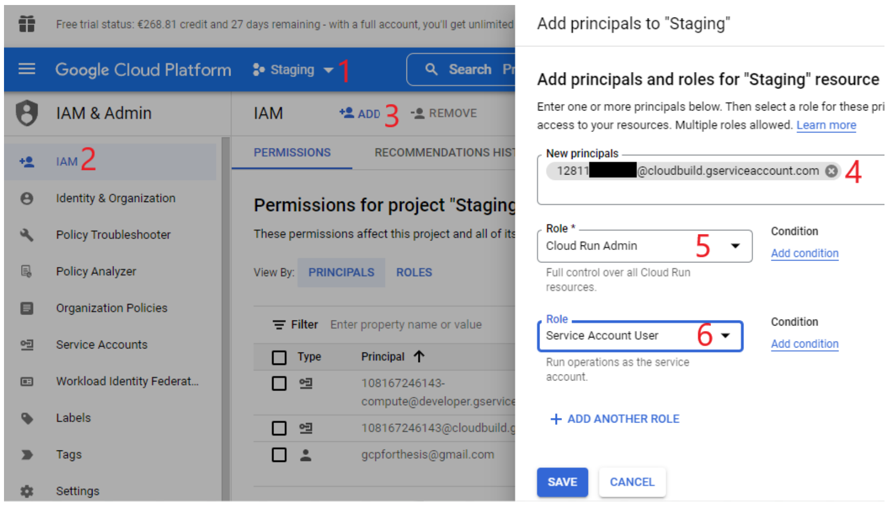
Service account email: 12811[redacted]@cloudbuild.gserviceaccount.com

GCP Service	Role	Status
Cloud Functions	Cloud Functions Developer	DISABLED
Cloud Run	Cloud Run Admin	DISABLED
App Engine	App Engine Admin	DISABLED
Kubernetes Engine	Kubernetes Engine Developer	DISABLED
Compute Engine	Compute Instance Admin (v1)	DISABLED
Firebase	Firebase Admin	DISABLED
Cloud KMS	Cloud KMS CryptoKey Decrypter	DISABLED
Secret Manager	Secret Manager Secret Accessor	DISABLED
Service Accounts	Service Account User	DISABLED
Cloud Build	Cloud Build WorkerPool User	DISABLED

Roles not listed here can be managed in the IAM section

Figure 18. Service account email of Management project.

This service account email was added as Principal with ***Cloud Run Admin*** and ***Service Account User*** roles in the Production and Staging project under the **IAM menus** of these projects. Figure 19 demonstrates the steps to add the principal to the Staging project.



Permissions for project "Staging

These permissions affect this project and all of its

108167246143-compute@developer.gservice

108167246143@cloudbuild.g

gcpforthesis@gmail.com

Add principals to "Staging"

Add principals and roles for "Staging" resource

Enter one or more principals below. Then select a role for these pri access to your resources. Multiple roles allowed. Learn more

New principals: 12811[redacted]@cloudbuild.gserviceaccount.com

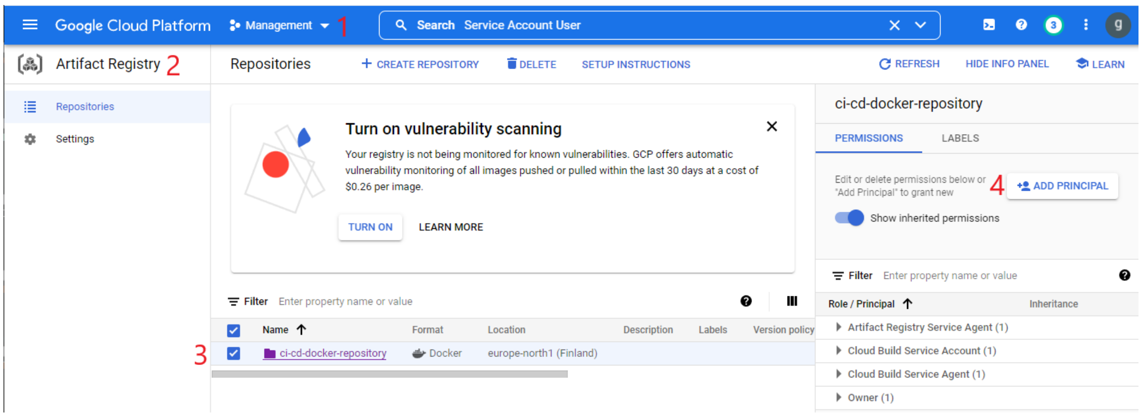
Role *: Cloud Run Admin — Full control over all Cloud Run resources. — Condition: Add condition

Role: Service Account User — Run operations as the service account. — Condition: Add condition

Figure 19. Adding Management service account email as principal to Staging project.

To add the Management's service account email as a principal to the Staging project, the Staging project was opened (1), then the **IAM menu** was opened (2) and the ***ADD*** button was pressed (3). This opens the setting to add the principal. As a new principal, the email of the service account of the Management project was added (4), and new roles were applied to that service account. The Cloud Run Admin (5) gives full control over Cloud Run services and the Service Account User (6) allows run operations as the service account. The same settings were applied in the Production project.

To permit the Management project to push the Docker images to the Artifact Registry following steps are done.



Your registry is not being monitored for known vulnerabilities. GCP offers automatic vulnerability monitoring of all images pushed or pulled within the last 30 days at a cost of $0.26 per image.

Edit or delete permissions below or "Add Principal" to grant new

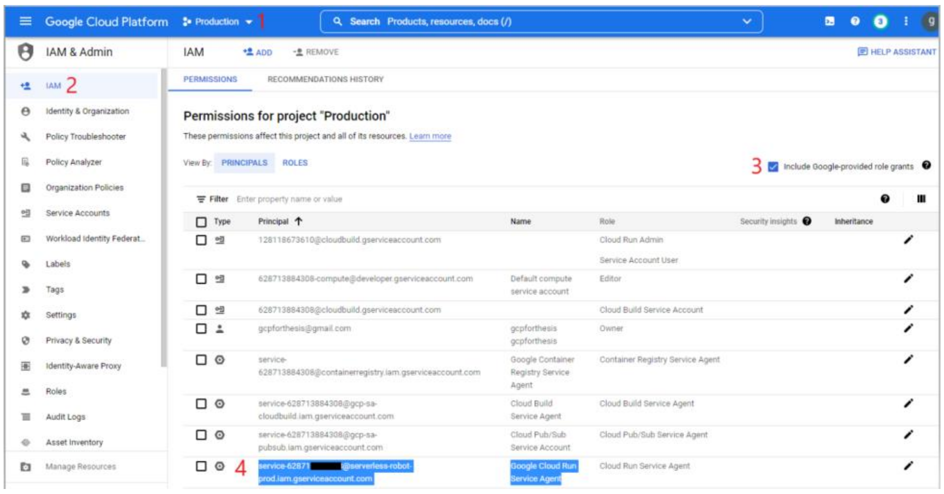
Name	Format	Location	Description	Labels	Version policy
ci-cd-docker-repository	Docker	europe-north1 (Finland)			

Role / Principal	Inheritance
Artifact Registry Service Agent (1)	
Cloud Build Service Account (1)	
Cloud Build Service Agent (1)	
Owner (1)	

Figure 20. Adding Management service account email as principal to Artifact Registry.

The **Artifact Registry** menu (2) under the Management project (1) was selected, then was selected the *ci-cd-docker repository* (3), and on the left menu, ***ADD PRINCIPAL*** button was pressed (4) as shown in figure 20. After that, the service account of the Management project was added with the Artifact Registry writer role, which gives access to read and write repository items.

The last permissions were applied to Production and Staging projects to have read access from Artifact Registry. This allows them to pull Docker images from Artifact Registry. Figure 21 shows the steps to be taken to find the list of accounts for the Production project.



Permissions for project "Production"

These permissions affect this project and all of its resources. Learn more

View By: PRINCIPALS ROLES

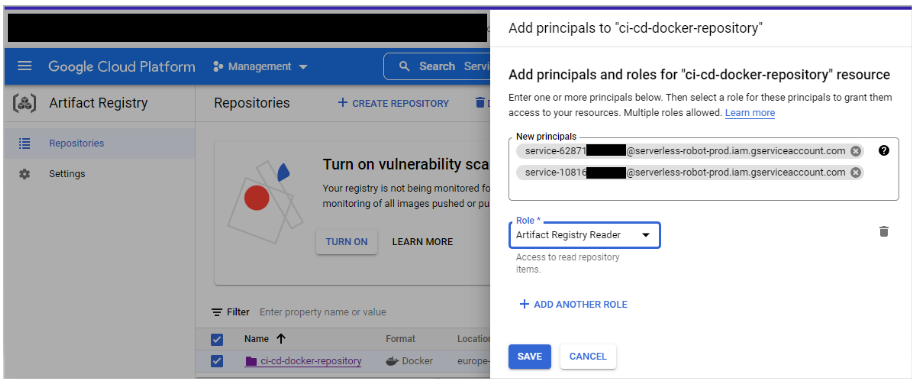
Include Google-provided role grants

Filter Enter property name or value

Type	Principal	Name	Role	Security insights	Inheritance
	128118673610@cloudbuild.gserviceaccount.com		Cloud Run Admin Service Account User		
	628713884308-compute@developer.gserviceaccount.com	Default compute service account	Editor		
	628713884308@cloudbuild.gserviceaccount.com		Cloud Build Service Account		
	gcpforthesis@gmail.com	gcpforthesis gcpforthesis	Owner		
	service-628713884308@containerregistry.iam.gserviceaccount.com	Google Container Registry Service Agent	Container Registry Service Agent		
	service-628713884308@gcp-sa-cloudbuild.iam.gserviceaccount.com	Cloud Build Service Agent	Cloud Build Service Agent		
	service-628713884308@gcp-sa-pubsub.iam.gserviceaccount.com	Cloud Pub/Sub Service Account	Cloud Pub/Sub Service Agent		
	service-62871[redacted]@serverless-robot-prod.iam.gserviceaccount.com	Google Cloud Run Service Agent	Cloud Run Service Agent		

Figure 21. The list of Production accounts.

To find the list of Production accounts, the **IAM** service (2) under Production project (1) was selected, to show *Google-provided role grants* the Include button was pressed (3), this showed the list of accounts, and the Google Cloud Run Service Agent email was copied (4). The same steps are done in the Staging project.



[redacted]

Add principals to "ci-cd-docker-repository"

Add principals and roles for "ci-cd-docker-repository" resource

Enter one or more principals below. Then select a role for these principals to grant them access to your resources. Multiple roles allowed. Learn more

New principals
service-62871[redacted]@serverless-robot-prod.iam.gserviceaccount.com
service-10816[redacted]@serverless-robot-prod.iam.gserviceaccount.com

Role *
Artifact Registry Reader
Access to read repository items.

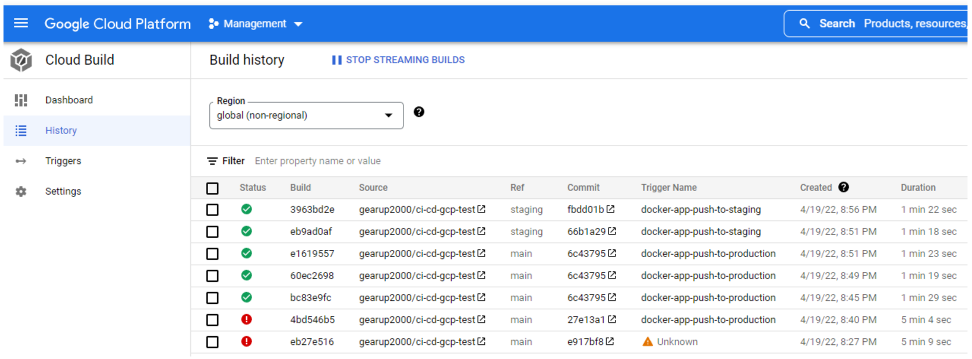
ADD ANOTHER ROLE

SAVE CANCEL

Figure 22. Adding Production and Staging Google Cloud Run Service Agent accounts as readers of Artifact Registry service.

Figure 22 shows how to add service account emails as the Artifact Registry Readers. This allows reading the images from the Artifact Registry.

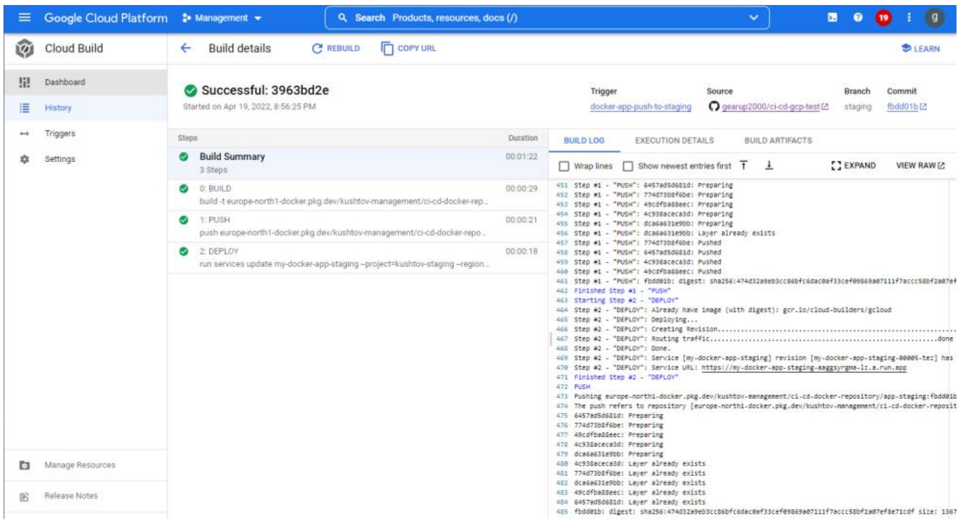
After applying for all required permissions, manual testing was done. To conduct tests, the **Cloud Build** menu item was selected in the Management project, after which the Triggers item was selected in the menu on the left, which opens a list of previously created triggers. Opposite the name of each trigger at the end of the line is a ***RUN*** button that initiates the launch of the trigger. Thus, a commit action from GitHub to the development branches was emulated. The test results were checked in the History menu item of the Cloud Build service, as figure 23 demonstrates.



	Status	Build	Source	Ref	Commit	Trigger Name	Created	Duration
☐	✓	3963bd2e	gearup2000/ci-cd-gcp-test	staging	fbdd01b	docker-app-push-to-staging	4/19/22, 8:56 PM	1 min 22 sec
☐	✓	eb9ad0af	gearup2000/ci-cd-gcp-test	staging	66b1a29	docker-app-push-to-staging	4/19/22, 8:51 PM	1 min 18 sec
☐	✓	e1619557	gearup2000/ci-cd-gcp-test	main	6c43795	docker-app-push-to-production	4/19/22, 8:51 PM	1 min 23 sec
☐	✓	60ec2698	gearup2000/ci-cd-gcp-test	main	6c43795	docker-app-push-to-production	4/19/22, 8:49 PM	1 min 19 sec
☐	✓	bc83e9fc	gearup2000/ci-cd-gcp-test	main	6c43795	docker-app-push-to-production	4/19/22, 8:45 PM	1 min 29 sec
☐	!	4bd546b5	gearup2000/ci-cd-gcp-test	main	27e13a1	docker-app-push-to-production	4/19/22, 8:40 PM	5 min 4 sec
☐	!	eb27e516	gearup2000/ci-cd-gcp-test	main	e917bf8	⚠ Unknown	4/19/22, 8:27 PM	5 min 9 sec

Figure 23. Results history of Cloud Build service.

The green color of the Status column indicates the successful results of pipeline work. By clicking on the build number, detailed information about each result can be checked. For example, for a detailed demonstration of the result, the last test was chosen, as can be seen in figure 24. All steps were completed, and the serverless version of the webserver was launched with a pre-installed HTML page.



Successful: 3963bd2e
Started on Apr 19, 2022, 8:56:25 PM

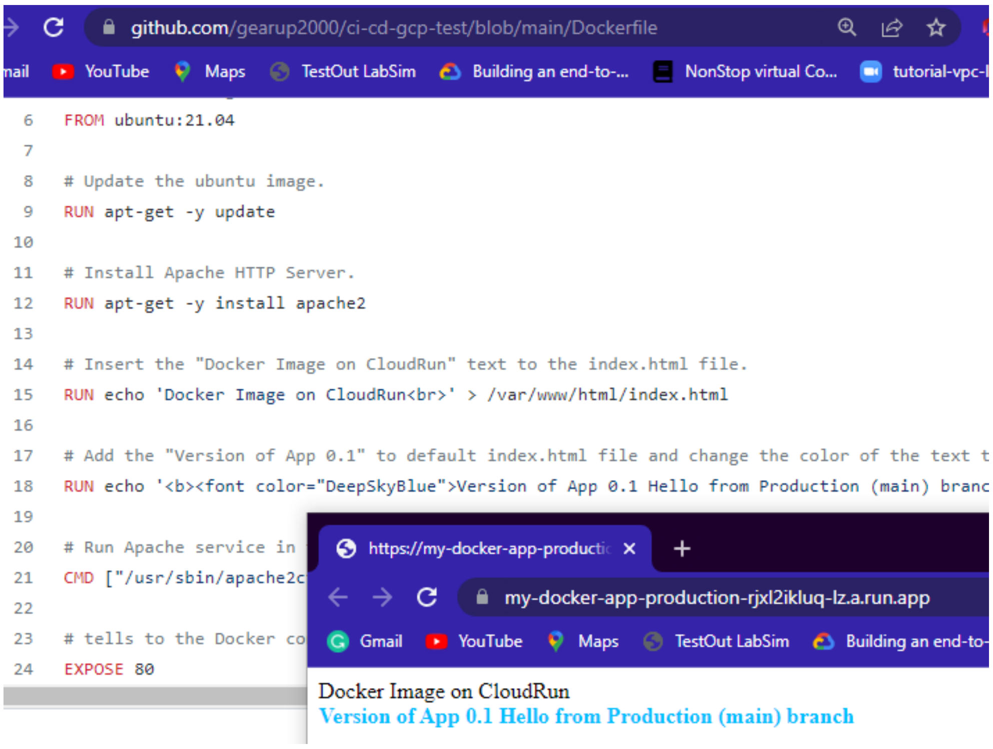
Trigger	Source	Branch	Commit
docker-app-push-to-staging	gearup2000/ci-cd-gcp-test	staging	fbdd01b

Steps	Duration
Build Summary 3 Steps	00:01:22
0: BUILD build -t europe-north1-docker.pkg.dev/kushtov-management/ci-cd-docker-rep...	00:00:29
1: PUSH push europe-north1-docker.pkg.dev/kushtov-management/ci-cd-docker-repo...	00:00:21
2: DEPLOY run services update my-docker-app-staging --project=kushtov-staging --region...	00:00:18

```
Step #1 - "PUSH": 6457ad5d681d: Preparing
Step #1 - "PUSH": 774d73b8f6be: Preparing
Step #1 - "PUSH": 49cdfba88eec: Preparing
Step #1 - "PUSH": 4c938aceca3d: Preparing
Step #1 - "PUSH": dca6a631e9bb: Preparing
Step #1 - "PUSH": dca6a631e9bb: Layer already exists
Step #1 - "PUSH": 774d73b8f6be: Pushed
Step #1 - "PUSH": 6457ad5d681d: Pushed
Step #1 - "PUSH": 4c938aceca3d: Pushed
Step #1 - "PUSH": 49cdfba88eec: Pushed
Step #1 - "PUSH": fbdd01b: digest: sha256:474d32a9eb3cc86bfc6dac0af33cef09869a07111f7accc58bf2a07ef
Finished Step #1 - "PUSH"
Starting Step #2 - "DEPLOY"
Step #2 - "DEPLOY": Already have image (with digest): gcr.io/cloud-builders/gcloud
Step #2 - "DEPLOY": Deploying...
Step #2 - "DEPLOY": Creating Revision..................................................................
Step #2 - "DEPLOY": Routing traffic..........................................................done
Step #2 - "DEPLOY": Done.
Step #2 - "DEPLOY": Service [my-docker-app-staging] revision [my-docker-app-staging-00005-tez] has
Step #2 - "DEPLOY": Service URL: https://my-docker-app-staging-aaggsyrgma-lz.a.run.app
Finished Step #2 - "DEPLOY"
PUSH
Pushing europe-north1-docker.pkg.dev/kushtov-management/ci-cd-docker-repository/app-staging:fbdd01b
The push refers to repository [europe-north1-docker.pkg.dev/kushtov-management/ci-cd-docker-reposit
6457ad5d681d: Preparing
774d73b8f6be: Preparing
49cdfba88eec: Preparing
4c938aceca3d: Preparing
dca6a631e9bb: Preparing
4c938aceca3d: Layer already exists
774d73b8f6be: Layer already exists
dca6a631e9bb: Layer already exists
49cdfba88eec: Layer already exists
6457ad5d681d: Layer already exists
fbdd01b: digest: sha256:474d32a9eb3cc86bfc6dac0af33cef09869a07111f7accc58bf2a07ef8e71cdf size: 1367
```

Figure 24. Detailed results of successful deployment.

The last tests were made using GitHub. Since two development branches are used, each of them contains a copy of the Dockerfile. The Dockerfile also stores instructions for building the HTML file and its contents. For a reliable test result, the text in the HTML part of Dockerfile in both branches was changed. In particular, the revision number of the web application. First, a commit was made from the Production (main) branch, and the second commit was made from the Staging(staging) branch. The last commit was made after merging two branches into the main (Production) branch, thereby emulating the work of developers, in the field of splitting and merging different branches of web application development. Figure 25 shows the results of the first commit from the Production (main) branch.


github.com/gearup2000/ci-cd-gcp-test/blob/main/Dockerfile
https://my-docker-app-productic
my-docker-app-production-rjxl2ikluq-lz.a.run.app

```
FROM ubuntu:21.04

# Update the ubuntu image.
RUN apt-get -y update

# Install Apache HTTP Server.
RUN apt-get -y install apache2

# Insert the "Docker Image on CloudRun" text to the index.html file.
RUN echo 'Docker Image on CloudRun<br>' > /var/www/html/index.html

# Add the "Version of App 0.1" to default index.html file and change the color of the text t
RUN echo '<b><font color="DeepSkyBlue">Version of App 0.1 Hello from Production (main) branc

# Run Apache service in
CMD ["/usr/sbin/apache2c

# tells to the Docker co
EXPOSE 80
```

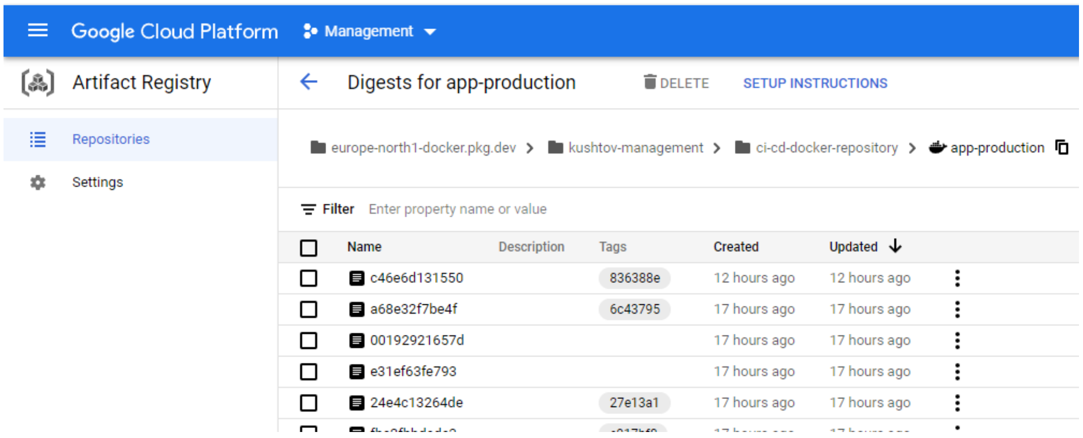
Docker Image on CloudRun
Version of App 0.1 Hello from Production (main) branch

Figure 25. Performance test of serverless CI/CD pipeline.

For clarity, a simple Dockerfile (Appendix 3) example was used, which is not an example of best practice, but sufficient to show the principle of operation of the serverless CI/CD pipeline. The files production.yaml (Appendix 1) and staging.yaml (Appendix 2) remains unchanged, as they are responsible for the operation of the entire pipeline and contain instructions for transferring the

Dockerfile of each branch from GitHub repositories to Google Cloud Platform and their further processing.

All built images were archived in the Artifact Registry. Figure 26 shows the saved copies of the used Docker images for building the web application in the Production (main) branch.


Google Cloud Platform
Management
Artifact Registry
Repositories
Settings
Digests for app-production
DELETE
SETUP INSTRUCTIONS
europe-north1-docker.pkg.dev > kushtov-management > ci-cd-docker-repository > app-production
Filter
Enter property name or value

Name	Description	Tags	Created	Updated
c46e6d131550		836388e	12 hours ago	12 hours ago
a68e32f7be4f		6c43795	17 hours ago	17 hours ago
00192921657d			17 hours ago	17 hours ago
e31ef63fe793			17 hours ago	17 hours ago
24e4c13264de		27e13a1	17 hours ago	17 hours ago

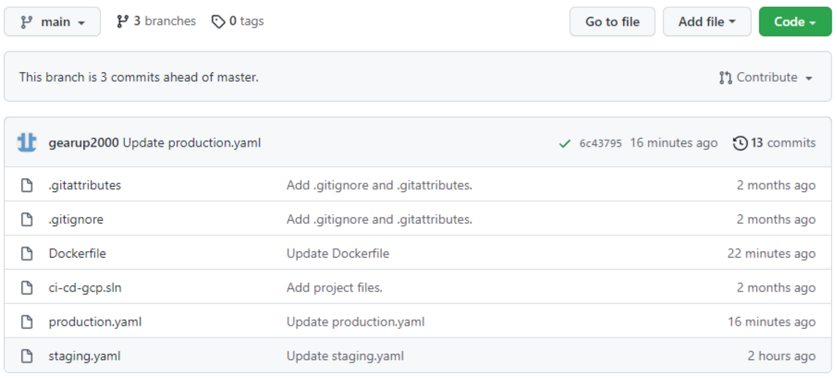
Figure 26. Docker images archive of Artifact Registry repository.

The Artifact Registry creates repositories for each branch separately and allows to get rid of confusion when deploying them in the pipeline. Any of these copies of images can be downloaded later to reuse or to be analyzed.

3.2 GitHub

GitHub provides a repository with a built-in version control system (VCS). The files can be stored in the repository and can be accessed via the Internet. A repository can be either public or private depending on the desires of the repository owner.

This chapter describes all files of the serverless CI/CD pipeline located on GitHub and their purposes. Figure 18 shows the content of the main (Production) branch of the GitHub repository.


main | 3 branches | 0 tags | Go to file | Add file | Code

This branch is 3 commits ahead of master. | Contribute

gearup2000 Update production.yaml ✓ 6c43795 16 minutes ago 13 commits

.gitattributes	Add .gitignore and .gitattributes.	2 months ago
.gitignore	Add .gitignore and .gitattributes.	2 months ago
Dockerfile	Update Dockerfile	22 minutes ago
ci-cd-gcp.sln	Add project files.	2 months ago
production.yaml	Update production.yaml	16 minutes ago
staging.yaml	Update staging.yaml	2 hours ago

Figure 27. Content of the main branch.

Some of these files are responsible for the correct operation of the pipeline, while the rest are part of GitHub and Microsoft Visual Studio.

.gitattributes – is the GitHub file that contains the attributes of each file in the relevant repository and does not affect serverless CI/CD pipeline operation.

.gitignore – is the GitHub file that contains a list of ignored files that should not be included in commits and does not affect serverless CI/CD pipeline operation.

Dockerfile – is a file (Appendix 3) that contains the instructions for building a Docker image for a serverless CI/CD pipeline. The instructions indicate which operating system to use and which services to install in the created image. This file is used in CI/CD pipeline. The content of the Dockerfile will be described in chapter 4.2.

cd-cd-gcp.sln - is the Microsoft Visual Studio file. It contains text information about the project environment and status, and stores project-specific settings. Does not affect serverless CI/CD pipeline operation.

production.yaml – is the file (Appendix 1) of instructions for the Cloud Build pipeline and Cloud Run service of the Production branch. It contains the instructions on steps for building, pushing, and deploying the Docker image file at the Production branch. To get the Docker image, *production.yaml* file refers to the *Dockerfile* (Appendix 3), which is the development file of the web application used in the Production (main) branch. The content of the *production.yaml* will be described in chapter 4.1.

staging.yaml – is the file (Appendix 2) of instructions for the Cloud Build pipeline and Cloud Run service of the Staging branch. It contains the instructions on steps for building, pushing, and deploying the Docker image file at the Staging branch.

To get the docker image, *staging.yaml* file refers to the *Dockerfile.* The content of the *staging.yaml* will be described in chapter 4.1.

As mentioned before the files Dockerfile, production.yaml and staging.yaml are used in serverless CI/CD pipeline project. Each of these files is responsible for their part of the pipeline performance and contains the code that is executed at the request of the developer or services such as a trigger.

3.3 Docker Image of Serverless CI/CD Pipeline Based on GCP

Docker is open-source software that grants the operating system to run processes in an isolated environment based on specially created images. The core principle behind Docker is application containerization. This type of virtualization allows the software to be packaged in isolated environments called containers. Each of these containers includes all the needed elements for the application to work properly.

The serverless CI/CD pipeline uses the Ubuntu docker image from the Container Registry library of images provided by Google Cloud Platform, which can be found at https://console.cloud.google.com/gcr/images/google.com:cloudsdktool. By changing the settings in the Dockerfile, the user can change the version of the operating system or completely change the operating system to another.

4 Configuration Files for Serverless CI/CD pipeline

The serverless CI/CD pipeline based on GCP uses three configuration files on each branch, that automate the deployment process. Three of them namely production.yaml (Appendix 1), staging.yaml (Appendix 2), Dockerfile (Appendix 3) are located at the main (production) branch and the other three namely production.yaml (Appendix 4), staging.yaml (Appendix 5), and Dockerfile (Appendix 6) are located at the staging branch. The production.yaml file contains the configuration and attributes for the Production branch. The staging.yaml file contains the configuration and attributes for the Staging branch. The Dockerfile contains the configuration Docker image. The production.yaml and staging.yaml files are almost identical. The difference is only in the variables for each development branch. All other configurations are similar. The YAML programming language was used to write these files. This language is a supported language of the Google Cloud Platform.

The Dockerfile contains the service commands for building the docker image and installing the webserver. Additionally, a simple HTML page is created and hosted on a web server.

4.1 Configuration Files for development branches

The file production.yaml and staging.yaml were used for the serverless CI/CD based on GCP. Each of them is responsible for its branches. The production.yaml file serves the main branch, which is the Production branch of the CI/CD pipeline. The staging.yaml serves the staging branch of the CI/CD pipeline. They contain the instructions and commands to use in the pipeline. The contents of the files are almost identical, with a small exception in the difference in the names of the development branches. The production.yaml (Appendix 1) file is used to explain the code. The ***steps*** part of the production.yaml file instructs GCP to create, push, and deploy Docker image. Each of these steps should be considered to describe in more detail, for this purpose the production.yaml file of the Production branch will be taken. The process of building a Docker image begins after the *steps* line.

Build step

```
steps:
  - name: gcr.io/cloud-builders/docker
    id  : BUILD
    args: ['build', '-t', '$_SERVICE_IMAGE', '.', '-f', 'Dockerfile']
```

Listing 1. The build step of production.yaml file.

The first **BUILD** step (listing 1) creates the Docker image from the provided source and applies the given parameters to it. The ***name*** field identifies the pre-

built Docker image which is stored in the Google Container Registry at *gcr.io/cloud-builders/docker.*
The ***id*** field specifies the ID which will be assigned to the built image. Examples of IDs, such as BUILD, PUSH and DEPLOY can be seen in the Cloud Build process shown in figure 24. The ***args*** field specifies the arguments to use. The ***'build'*** argument starts the build process of a Docker image from the Dockerfile and assembly context. An assembly context is a set of files located at a specific path or URL. The ***'-t'*** argument specifies the tag and repository to store the image on a successful build. The next ***'$_SERVICE_IMAGE'*** argument is the address where the built image must be saved. The ***'.'*** argument specifies the current working directory to build the image. The context for building the Docker image will be taken from the current directory only. The ***'-f'*** argument specifies the location of the Dockerfile. The ***'Dockerfile'*** argument specifies the location and name of the Dockerfile. In this example, the Dockerfile is in the current directory.

PUSH step

```
- name: gcr.io/cloud-builders/docker
   id  : PUSH
  args: ['build', '-t', '$_SERVICE_IMAGE', '.', '-f', 'Dockerfile']
```

Listing 2. The push step of production.yaml file.

The second **PUSH** step (listing 2) has ***id*** as ***PUSH*** and argument ***'push'*** which instructs the GCP to push the built image to the ***'$_SERVICE_IMAGE'*** location*.*

DEPLOY step

```
- name: gcr.io/cloud-builders/gcloud
  id  : DEPLOY
  args:
    - run
    - services
    - update
    - $_SERVICE_NAME
    - --project=$_SERVICE_PROJECT
    - --region=$_SERVICE_REGION
    - --image=$_SERVICE_IMAGE
```

Listing 3. The deploy step of production.yaml file.

The **DEPLOY** step (listing 3) has ***id*** as ***DEPLOY***. The name argument ***gcr.io/cloud-builders/gcloud*** is the repository with the latest version of the Docker image maintained by the Cloud Build team. The arguments ***–run, –services, –update*** can be represented in one row, such as ***run services update $_SERVICE_NAME,*** which instructs the GCP to run services update of Cloud Run environment variables and other configuration settings.

```
Images:
- $_SERVICE_IMAGE
```

Listing 4. The part of code to display the images in the build results.

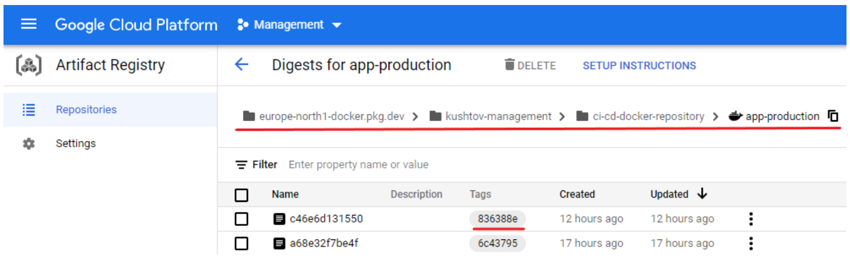
The lines with code ***Images:*** and **–** ***$_SERVICE_IMAGE*** (listing 4) display the created images in the build results.

The last part provides substitutions and parameters to build, run, and deploy processes (listing 5).

```
substitutions:
    _SERVICE_IMAGE:europe-north1-
docker.pkg.dev/${PROJECT_ID}/${_DOCKER_REGISTRY}/${_DOCKER_IMAGENAME}:${SHORT_
SHA}
    _SERVICE_REGION    : europe-north1
    _SERVICE_PROJECT   : kushtov-production
    _SERVICE_NAME      : docker-app-push-to-production
    _DOCKER_REGISTRY   : ci-cd-docker-repository
    _DOCKER_IMAGENAME  : app-production
```

Listing 5. The substitutions are part of the production.yaml file.

The line ***_SERVICE_IMAGE:europe-north1-docker.pkg.dev/${PROJECT_ID}/${_DOCKER_REGISTRY}/${_DOCKER_IMAGENAME}:${SHORT_SHA}*** is a link to a docker image located in the project repository and can be found in the Artifact Registry repository of the relevant project as shown in figure 28.



Digests for app-production

europe-north1-docker.pkg.dev > kushtov-management > ci-cd-docker-repository > app-production

Name	Description	Tags	Created	Updated
c46e6d131550		836388e	12 hours ago	12 hours ago
a68e32f7be4f		6c43795	17 hours ago	17 hours ago

Figure 28. Artifact Registry repository of the Production project.

As seen in figure 28, the ***PROJECT_ID*** is **kushtov-management**, ***_DOCKER_REGISTRY*** is ***ci-cd-docker-repository***, ***_DOCKER_IMAGENAME*** is **app-production**, and ***SHORT_SHA*** adds the SHA tag to the built image.

The ***_SERVICE_REGION*** defines the region to use. The ***_SERVICE_PROJECT*** defines the project name. The ***_SERVICE_NAME*** defines the trigger name. The ***_DOCKER_REGISTRY*** defines the repository name. The ***_DOCKER_IMAGENAME*** defines the name of the Docker image.

The staging.yaml (Appendix 2) file has the same setting as the production.yaml (Appendix 1) with changes in the names of the project, trigger, and Docker images for proper functioning in the Staging branch.

4.2 The Dockerfile Configuration

A Dockerfile (Appendix 3) is essentially a manual for building a docker image. The commands in the Dockerfile specify which services should be installed, which files should be created, or which firewall rules should be applied. Changes to this file change the behavior of the web application because it contains the code for it. The Dockerfile builds an image of a web server based on the Ubuntu operating system, then creates an index.html file and fills its contents with simple HTML code. To access a web page, the docker container opens port 80. There are two Dockerfiles in each branch. To describe their configuration of them, the Dockerfile of the Production branch will be used.

```
FROM ubuntu:21.04
RUN apt-get -y update
RUN apt-get -y install apache2
RUN echo 'Docker Image on Cloud Run<br>' > /var/www/html/index.html
RUN echo '<b><font color="DeepSkyBlue">Version of App 0.1 Hello from Production
(main) branch</font></b>' >> /var/www/html/index.html
CMD ["/usr/sbin/apache2ctl", "-D","FOREGROUND"]
EXPOSE 80
```

Listing 6. Content of Dockerfile of Production branch.

The line ***FROM ubuntu:21.04*** defines the image to use. This case instructs the docker engine to use the Ubuntu version 21.4 image for building the Docker image.

The line ***RUN apt-get -y update*** initiates the process to check for available updates of OS, libraries, and tools. If the updates are available, it will install them without asking the user, since the -y key is used, which stands for yes answer in case the user approval is needed.

The line ***RUN apt-get -y install apache2*** initiates the process of installing the Apache webserver.

The line ***RUN echo 'Docker Image on Cloud Run<br>' > /var/www/html/index.html*** creates the index.html and inserts the “Docker Image on Cloud RUN” text to the body of the index.html file.

The line ***RUN echo '<b><font color="DeepSkyBlue">Version of App 0.1 Hello from Production (main) branch</font></b>' >> /var/www/html/index.html*** insert the “Hello from Production (main) branch” text to the body of index.html file.

The line ***CMD ["/usr/sbin/apache2ctl", "-D","FOREGROUND"]*** initiates foreground run process of Apache web server.

The line ***EXPOSE 80*** instructs the Docker image to expose and listen the port 80. This port is used by web servers.

The Dockerfile file of the Staging branch has the same setting with a small difference in the body text, where it is indicated that this build of web application came from the Staging branch.

5 Conclusion

This thesis project aimed to design the serverless CI/CD pipeline based on the Google Cloud Platform. With the use of the information received and the combination of different technologies, the goal was achieved. In the beginning, the documentation of cloud services provided by Google Cloud Platform was studied. Based on the received information, the tests were carried out. This gave an understanding of the principles of operation of these services and their potential opportunities. By combining the GitHub and Google Cloud platforms, the basis for the future pipeline was obtained. Files for more fine-tuning configurations helped to successfully launch the entire pipeline. Additionally, with their help, it became possible to control the entire process of developing and deploying a web application. The Docker container technology was used to build the web application. A simple Dockerfile was used to test the operation of the pipeline since the goal of the entire project was not to develop a web application, but to create a serverless CI/CD pipeline. The Dockerfile was responsible for both the front end and the backend tasks. As a result of the symbiosis of all these technologies and tools, a successful result was achieved by hosting a web application on a serverless platform.

Replacing the functionality of the webserver with serverless Google Cloud Platform services will help free up resources spent on developing applications. This solution is quite relevant and significant in the field of DevOps. The modular design of the project leaves the possibility for its reconfiguration and reuse in other projects as well.

The study and creation of such a project helped the author of the thesis to cover a wide range of technologies and tools. This project will be beneficial for DevOps professionals, as well as for IT students. The application of the knowledge gained in the process will assist the author of the project in the future.

In the author’s opinion, the further development of serverless solutions is a very promising direction and its development should be continued.

References

1 Davis, Jennifer; Daniels, Katherine. 2016. Effective DevOps: Building a Culture of Collaboration, Affinity, and Tooling at Scale.

2 Mulder, Jeroen; 2021. Enterprise DevOps for Architects.

3 Metish, Soni; 2016. DevOps for Web Development.

4 Portnoy, Matthew; 2016. Virtualization Essentials 2nd Edition.

5 Tulloch, Mitch; 2010. Understanding Microsoft Virtualization Solutions (Second Edition).

6 Riti, Pierluigi; 2018. Pro DevOps with Google Cloud Platform: With Docker, Jenkins, and Kubernetes.

7 Diagboya, Ewere; 2021. Infrastructure Monitoring with Amazon CloudWatch: Effectively Monitor Your AWS Infrastructure to Optimize Resource Allocation, Detect Anomalies, and Set Automated Actions.

8 Krief, Mikael; 2019. Learning DevOps: The Complete Guide to Accelerate Collaboration with Jenkins, Kubernetes, Terraform and Azure DevOps.

9 McKendrick, Russ; Gallagher, Scott. 2018. Mastering Docker: Unlock New Opportunities Using Docker's Most Advanced Features, 3rd Edition.

10 Farcic, Viktor. 2016. The DevOps 2.0 Toolkit: Automating the Continuous Deployment Pipeline with Containerized Microservices.

11 Bhowmik, Sandeep; 2017. Cloud Computing.

12 Reed, Mark; 2020. DevOps The Ultimate Beginners Guide to Learn DevOps Step-by-Step.

13 Fox, Richard; Hao, Wei; 2017. Internet Infrastructure: Networking, Web Services, and Cloud Computing.

The content of the production.yaml file of Production (main) branch

```
#----------------------------------------
# Cloud Build Pipeline for Production Cloud Run
#----------------------------------------
steps:
# Docker Build Image request, the arguments are:
# 'build' - is the entry point to the Docker cloud builder,
# '-t' - is the Docker flag,
# '$_SERVICE_IMAGE' - is the name of the image to be built in Artifact
Registry.
# '.' - is the location of the source code, which indicates the source code is
in the current working directory.
# '-f' - indicates to use a file as a source,
# 'Dockerfile' - indicates the name of a Dockerfile to use. In this case, the
name is "Dockerfile".

  - name: gcr.io/cloud-builders/docker
    id  : BUILD
    args: ['build', '-t', '$_SERVICE_IMAGE', '.', '-f', 'Dockerfile']

# Docker Push Image to Artifact Registry Service, the arguments are:
# 'push' - push the image to Artifact Registry Service.
# '$_SERVICE_IMAGE' - is the name of the image to be pushed to Artifact
Registry.
  - name: gcr.io/cloud-builders/docker
    id  : PUSH
    args: ['push', '$_SERVICE_IMAGE']

# Docker Deploy the image to the Cloud Run
# the arguments are: Update Cloud Run environment variables and other
configuration settings.
  - name: gcr.io/cloud-builders/gcloud
    id  : DEPLOY
    args:
      - run
      - services
      - update
      - $_SERVICE_NAME
      - --project=$_SERVICE_PROJECT
      - --region=$_SERVICE_REGION
      - --image=$_SERVICE_IMAGE

images: # Display the image in the build results.
- $_SERVICE_IMAGE

substitutions:
    _SERVICE_IMAGE    : europe-north1-
docker.pkg.dev/${PROJECT_ID}/${_DOCKER_REGISTRY}/${_DOCKER_IMAGENAME}:${SHORT_
SHA}
    _SERVICE_REGION   : europe-north1
    _SERVICE_PROJECT  : kushtov-production
    _SERVICE_NAME     : my-docker-app-production
    _DOCKER_REGISTRY  : ci-cd-docker-repository
    _DOCKER_IMAGENAME : app-production
```

The content of staging.yaml file of the Production branch

```
#-----------------------------------------
# Cloud Build Pipeline for Staging Cloud Run
#-----------------------------------------
steps:
# Docker Build Image request, the arguments are:
# 'build' - is the entry point to the Docker cloud builder,
# '-t' - is the Docker flag,
# '$_SERVICE_IMAGE' - is the name of the image to be built in Artifact
Registry.
# '.' - is the location of the source code, which indicates the source code is
in the current working directory.
# '-f' - indicates to use a file as a source,
# 'Dockerfile' - indicates the name of a Dockerfile to use. In this case, the
name is "Dockerfile".

  - name: gcr.io/cloud-builders/docker
    id  : BUILD
    args: ['build', '-t', '$_SERVICE_IMAGE', '.', '-f', 'Dockerfile']

# Docker Push Image to Artifact Registry Service, the arguments are:
# 'push' - push the image to Artifact Registry Service.
# '$_SERVICE_IMAGE' - is the name of the image to be pushed to Artifact
Registry.
  - name: gcr.io/cloud-builders/docker
    id  : PUSH
    args: ['push', '$_SERVICE_IMAGE']

# Docker Deploy the image to the Cloud Run.
# the arguments are: Update Cloud Run environment variables and other
configuration settings.
  - name: gcr.io/cloud-builders/gcloud
    id  : DEPLOY
    args:
      - run
      - services
      - update
      - $_SERVICE_NAME
      - --project=$_SERVICE_PROJECT
      - --region=$_SERVICE_REGION
      - --image=$_SERVICE_IMAGE

images: # Display the image in the build results.
- $_SERVICE_IMAGE

substitutions:
    _SERVICE_IMAGE    : europe-north1-
docker.pkg.dev/${PROJECT_ID}/${_DOCKER_REGISTRY}/${_DOCKER_IMAGENAME}:${SHORT_
SHA}
    _SERVICE_REGION   : europe-north1
    _SERVICE_PROJECT  : kushtov-staging
    _SERVICE_NAME     : my-docker-app-staging
    _DOCKER_REGISTRY  : ci-cd-docker-repository
    _DOCKER_IMAGENAME : app-staging
```

The content of the Dockerfile of the Production (main) branch

```
#---------------------------------------------------------------------------
# Dockerfile / creates a Docker Image with integrated Apache WebServer. The OS
is Ubuntu 21.04
# ---------------------------------------------------------------------------

# Define the image to use.
FROM ubuntu:21.04

# Update the ubuntu image.
RUN apt-get -y update

# Install Apache HTTP Server.
RUN apt-get -y install apache2

# Insert the "Docker Image on Cloud Run" text to the index.html file.
RUN echo 'Docker Image on Cloud Run<br>' > /var/www/html/index.html

# Add the "Version of App 0.1" to the default index.html file and change the
color of the text to the DeepSkyBlue.
RUN echo '<b><font color="DeepSkyBlue">Version of App 0.1 Hello from
Production (main) branch</font></b>' >> /var/www/html/index.html

# Run Apache service in the foreground after the container is started.
CMD ["/usr/sbin/apache2ctl", "-D","FOREGROUND"]

# tells the Docker container to listen on port 80
EXPOSE 80
```

The content of the production.yaml file of Staging (staging) branch

```
#----------------------------------------
# Cloud Build Pipeline for Production Cloud Run
#----------------------------------------
steps:
# Docker Build Image request, the arguments are:
# 'build' - is the entry point to the Docker cloud builder,
# '-t' - is the Docker flag,
# '$_SERVICE_IMAGE' - is the name of the image to be built in Artifact
Registry.
# '.' - is the location of the source code, which indicates the source code is
in the current working directory.
# '-f' - indicates to use a file as a source,
# 'Dockerfile' - indicates the name of a Dockerfile to use. In this case, the
name is "Dockerfile".

  - name: gcr.io/cloud-builders/docker
    id  : BUILD
    args: ['build', '-t', '$_SERVICE_IMAGE', '.', '-f', 'Dockerfile']

# Docker Push Image to Artifact Registry Service, the arguments are:
# 'push' - push the image to Artifact Registry Service.
# '$_SERVICE_IMAGE' - is the name of the image to be pushed to Artifact
Registry.
  - name: gcr.io/cloud-builders/docker
    id  : PUSH
    args: ['push', '$_SERVICE_IMAGE']

# Docker Deploy the image to the Cloud Run
# the arguments are: Update Cloud Run environment variables and other
configuration settings.
  - name: gcr.io/cloud-builders/gcloud
    id  : DEPLOY
    args:
      - run
      - services
      - update
      - $_SERVICE_NAME
      - --project=$_SERVICE_PROJECT
      - --region=$_SERVICE_REGION
      - --image=$_SERVICE_IMAGE

images: # Display the image in the build results.
- $_SERVICE_IMAGE

substitutions:
    _SERVICE_IMAGE    : europe-north1-
docker.pkg.dev/${PROJECT_ID}/${_DOCKER_REGISTRY}/${_DOCKER_IMAGENAME}:${SHORT_
SHA}
    _SERVICE_REGION   : europe-north1
    _SERVICE_PROJECT  : kushtov-production
    _SERVICE_NAME     : docker-app-push-to-production
    _DOCKER_REGISTRY  : ci-cd-docker-repository
    _DOCKER_IMAGENAME : app-production
```

The content of staging.yaml of Staging (staging) branch

```
#----------------------------------------
# Cloud Build Pipeline for Staging Cloud Run
#----------------------------------------
steps:
# Docker Build Image request, the arguments are:
# 'build' - is the entry point to the Docker cloud builder,
# '-t' - is the Docker flag,
# '$_SERVICE_IMAGE' - is the name of the image to be built in Artifact
Registry.
# '.' - is the location of the source code, which indicates the source code is
in the current working directory.
# '-f' - indicates to use a file as a source,
# 'Dockerfile' - indicates the name of a Dockerfile to use. In this case, the
name is "Dockerfile".

  - name: gcr.io/cloud-builders/docker
    id  : BUILD
    args: ['build', '-t', '$_SERVICE_IMAGE', '.', '-f', 'Dockerfile']

# Docker Push Image to Artifact Registry Service, the arguments are:
# 'push' - push the image to Artifact Registry Service.
# '$_SERVICE_IMAGE' - is the name of the image to be pushed to Artifact
Registry.
  - name: gcr.io/cloud-builders/docker
    id  : PUSH
    args: ['push', '$_SERVICE_IMAGE']

# Docker Deploy the image to the Cloud Run.
# the arguments are: Update Cloud Run environment variables and other
configuration settings.
  - name: gcr.io/cloud-builders/gcloud
    id  : DEPLOY
    args:
      - run
      - services
      - update
      - $_SERVICE_NAME
      - --project=$_SERVICE_PROJECT
      - --region=$_SERVICE_REGION
      - --image=$_SERVICE_IMAGE

images: # Display the image in the build results.
- $_SERVICE_IMAGE

substitutions:
    _SERVICE_IMAGE    : europe-north1-
docker.pkg.dev/${PROJECT_ID}/${_DOCKER_REGISTRY}/${_DOCKER_IMAGENAME}:${SHORT_
SHA}
    _SERVICE_REGION   : europe-north1
    _SERVICE_PROJECT  : kushtov-staging
    _SERVICE_NAME     : my-docker-app-staging
    _DOCKER_REGISTRY  : ci-cd-docker-repository
    _DOCKER_IMAGENAME : app-staging
```

The content Dockerfile of the Staging (staging) branch

```
#---------------------------------------------------------------------------
# Dockerfile / creates a Docker Image with integrated Apache WebServer. The OS
is Ubuntu 21.04
# ---------------------------------------------------------------------------

# Define the image to use.
FROM ubuntu:21.04

# Update the ubuntu image.
RUN apt-get -y update

# Install Apache HTTP Server.
RUN apt-get -y install apache2

# Insert the "Docker Image on Cloud Run" text to the index.html file.
RUN echo 'Docker Image on Cloud Run<br>' > /var/www/html/index.html

# Add the "Version of App 0.1" text to the default index.html file and change
the color of the text to the DeepSkyBlue.
RUN echo '<b><font color="DeepSkyBlue">Version of App 0.1a Hello from Staging
(staging) branch </font></b>' >> /var/www/html/index.html

# Run Apache service in the foreground after the container is started.
CMD ["/usr/sbin/apache2ctl", "-D","FOREGROUND"]

# instruct the Docker container to listen on port 80
EXPOSE 80
```