# IMPLEMENTING A DEVOPS FLOW FOR A MICROSERVICE ARCHITECTURE WEB APP

Mohamed Sofiene Barka

Mohamed Rafraf

Adam Lahbib


PERSONAL PROFESSIONAL PROJECT

# PERSONAL PROFESSIONAL PROJECT

# IMPLEMENTING A DEVOPS FLOW FOR A MICROSERVICE ARCHITECTURE WEB APP


The project and its report were designed, implemented, and written by

Mr. **Mohamed Sofiene Barka** Mr. **Mohamed Rafraf** Mr. **Adam Lahbib**


**A First-Year Networking & Telecommunication Project Report**


National Institute Of Applied Science And Technology,

University Of Carthage, Tunis


April '22 - May '22

© 2022

ALL RIGHTS RESERVED

# ACKNOWLEDGMENTS

*In the name of God, the most gracious, the most merciful. Praise be to God Almighty for giving us the strength, knowledge, and ability to complete this project, without his blessings, this achievement would not have been possible. And peace be upon our prophet Mohamed, his family, and all of his companions.*

We would like to thank our supervisor Prof. Adlene Loukil for his invaluable support, patience, and motivation throughout our project.

We would like to thank our beloved families, friends, and mates for their invaluable support and care throughout our journey at the university and beyond. Hoping we can make them proud one day!

Our heartfelt thanks to all the practitioners who have taken the time and effort to help us out, not only but especially: Miss Anais Urlichs, her diaries, and the *#100DaysOfKubernetes* series challenge, Mr. Kavindu Chamiran's Medium articles, Codeching YouTube Channel, and Mr. Jeff Geerling and Mrs. Katherine Geerling, authors of "Kubernetes 101", whose rich book was a great resource for us, and whose illustrations gave life to our report. That's along with various documentation and wikis that were more than helpful...

Thank you, dear reader, for giving our report the time and opportunity, hoping it helps you understand more about DevOps and Kubernetes...

*Verily God is the grantor of success!*

# ABSTRACT

As part of this first-year engineering class personal professional project, we proposed to study DevOps flows in a microservices architecture application. In order to gain experience and be prepared for future internship opportunities.

Owing to a multitude of factors like rapidly evolving technology, market requirements, and business hyper-competitiveness, software companies nowadays are under duress to deliver software rapidly and on a frequent basis. That said, companies should become more lean and agile in all phases of the software development lifecycle. An approach called "DevOps" based on agile principles has come into play. DevOps bridges the gap between development and operations, hence its name. The DevOps phenomenon has gained wide popularity in the past few years, and several companies are adopting it to leverage its perceived benefits. However, that requires a clear understanding of its practices.

Our prime goal of this study is to provide a clear apprehension of how to deploy a developed app adhering to DevOps practices as we learned it throughout our journey and we will cover almost everything a DevOps beginner must be aware of for a successful deployment and monitoring of a project.

*Keywords: DevOps, Deployment, Monitoring, Microservices, Continuous Integration, Continuous Delivery, GitLab, Kubernetes.*

# SUMMARY

Pickloria, the name we picked for the meme-sharing platform built on the 'PERN' stack (PostgreSQL, ExpressJS, ReactJS, NodeJS). It will be a proof of concept for what is coming next, it can be checked at our public GitLab repository 🔗. The app touches several aspects of Kubernetes such as storage, pods, services, and deployments...

The frontend is nothing fancy, just a double-page application – explore and upload memes, built with React, Vite to boot the app faster, and TailwindCSS for styling. The backend uses Express as a REST API for the project along with a Golang worker being a microservice that handles validating the memes. (The app is just a simulation for a real-world scenario, for example validating links or assets to check if they do not hold harmful content which is likely to be the responsibility of a different service than the HTTP API)

For containerization we are using Docker, we are going to build images from Dockerfiles, then push them to DockerHub as our primary container registry. For this, we wrote Dockerfiles for each part of the project: React App, Express Server, and Golang Worker.

Afterward, we have set up a CI-CD pipeline and a runner for our project using GitLab CI that will watch all three ends of the app (being Client and Servers) and trigger the runner when something changes.

We also have let Gitlab send us notification emails when a pipeline is done using a built-in plugin, as well as implementing DevSecOps through Snyk to scan the app's dependencies against any security vulnerabilities before deployment.

Coming to the Kubernetes part, we have a folder named ‘K8s’ in the root of the project, which holds the manifests for the client, servers, and databases, as well as a ConfigMap to store data as key-value pairs for the DB and Redis’ deployment files. The frontend holds the static files and an API endpoint exposed using Nginx that acts as an HTTP Server and a Reverse Proxy for our API, and a Kubernetes LoadBalancer service object. Both DB and Backend are only logically exposed between each component and not outside.

Once all that was done, we deployed the whole project on Azure’s AKS, then we set up Prometheus and Grafana through Helm for monitoring, which will simply add their arsenal of pods without further intervention.

*Everything will be explained in-depth throughout this report...*

*We welcome all your questions and inquiries.*

*Thanks!*

**TABLE OF CONTENTS**


ACKNOWLEDGMENTS **4**

ABSTRACT **5**

SUMMARY **6**

LIST OF TABLES **13**

LIST OF ACRONYMS AND DEFINITIONS **16**

**CHAPTER ONE: PRELIMINARY STUDY** **2**

- Definitions 2
- What is DevOps? 2
- What is CI-CD? 2
- Containerization and Orchestration 3
- Cloud Computing 3
- Microservices 3
- Problematic 4
- Proposed Solution 4
- Conclusion 5

**CHAPTER TWO: ANALYSIS AND CONCEPTION** **7**

- Presentation Of The Application 7
- Requirements 7
  - Easy Development 8
  - Easy Deployment 8
  - Scalability 8
  - Observability 8
  - Security 9
  - Automation 9
- Workflow Designs 9
- Conclusion 11

**CHAPTER THREE: EMBRACING DEVOPS** **14**

- Brief Of History 14
- DevOps Pillars 15
- The Evolution Of Computing 16

Waterfall Model 16
Agile Model 17
DevOps Model 18
Comparison 19
Going Beyond Agile 19
Why DevOps? 20
Collaboration And Trust 21
Release Faster And Work Smarter 21
Accelerate Time-To-Resolution 21
Better Manage Unplanned Work 22
Conclusion 22

**CHAPTER FOUR: MICROSERVICES IN-DEPTH 24**
Characteristics Of Microservices 24
Why Microservices? 25
Microservices And DevOps: Two Peas In A Pod 26
Technologies And Tools For A Microservices Architecture 27
Conclusion 27

CHAPTER FIVE: GITLAB CI-CD **30**
What is GitLab CI-CD? 30
Continuous Integration 30
Continuous Delivery 31
Continuous Deployment 32
DevOps Pipelines 32
GitLab Runner 35

**CHAPTER SIX: DOCKERIZATION 38**
Docker 38
Virtual Machines And Containers 38
Virtual Machines 38
Docker Containers 39
Comparison 39
Why Use Docker? 40
Docker Files And Docker Images 40
Dockerizing An Application 41
Conclusion 44

**CHAPTER SEVEN: A DIVE INTO KUBERNETES 46**
Why Do We Need Container Orchestration? 46
Kubernetes Architecture 46
Kubernetes Core Features 47

Container Runtime Interface 48
The Network Plugin 48
The Volume Plugin 48
The Image Registry 48
Cloud Provider 49
Identity Provider 49
Kubernetes Cluster 49
Kubernetes Cluster Architecture 50
Control Plane 50
Worker Node 52
Pods, Statefulsets, and Deployments 53
Pod 53
Creating a Pod Definition with YAML file 54
ReplicaSets 55
Creating a ReplicaSet Definition with YAML file 56
Deployments 57
Creating a Deployment Definition with YAML file 58

**CHAPTER EIGHT: REALIZATION 61**
Tools 61
GitLab 61
Installing Requirements And Dependencies 67
Docker / Docker Desktop 68
Minikube 68
Helm 71
Project Skeleton (Microservices Architecture) 72
Dockerfiles 73
Codes 73
Interpretation 76
Commands 76
DockerHub 77
CI-CD Pipelines 81
GitLab CI-CD Variables 81
Codes 81
Interpretation 82
Email Notifications 83
Kubernetes Manifests 84
Codes 84
Interpretation 85
Monitoring And Observability 85

DevSecOps With Snyk 93
Azure 95
Cloud-Native 95
Cloud Providers 95
Managed Kubernetes 96
Creating AKS (Azure Kubernetes Service) Cluster with AZ-CLI 96
Implementation of the infrastructure 98
Conclusion 100

**FINAL CONCLUSION AND FUTURE SCOPE 102**

**REFERENCES & WEBOGRAPHY 104**

**APPENDICES 104**
APPENDIX A 105
APPENDIX B 105
APPENDIX C 106
APPENDIX D 106
APPENDIX E 106
APPENDIX F 107
APPENDIX G 107
APPENDIX H 109
APPENDIX I 109
APPENDIX J 111

# LIST OF TABLES

Table 1 . Comparison Between Models 19
Table 2. Project Skeleton 72

## LIST OF FIGURES


Figure 1. Workflow Design (Microservices Level) 10
Figure 2. Global Workflow Design 11
Figure 3. The 4 DevOps Pillars 15
Figure 4. Waterfall Lifecycle 17
Figure 5. Agile Lifecycle 17
Figure 6. DevOps Lifecycle 19
Figure 7. Continuous Integration 31
Figure 8. Continuous Delivery And Continuous Deployment 32
Figure 9. VMs atop the Infrastructure 38
Figure 10. Docker Containers atop the Infrastructure 39
Figure 11. Disk R/W Speeds (Docker vs VM) 40
Figure 12. Kubernetes Architecture 47
Figure 13. Inside Kubernetes Cluster 50
Figure 14. Kubernetes Cluster Architecture 50
Figure 15. A closer look at the Pod 54
Figure 16. Inside Replicasets 56
Figure 17. Kubernetes Deployment 58
Figure 18. GitLab.com Homepage 61
Figure 19. Create A Docker ID 62
Figure 20. Snyk Registration 63
Figure 21. Snyk Integrations 63
Figure 22. Generating a PAT in GitLab 64
Figure 23. Snyk Integration ID 64
Figure 24. Adding SNYK_TOKEN to GitLab CI-CD variables 65
Figure 25. Azure's Homepage 65
Figure 26. Get Started with Azure for free 66
Figure 27. Login to Azure with the Microsoft Account 66
Figure 28. Fill in additional details for Azure 67
Figure 29. Azure's identity verification by phone 67
Figure 30. DockerHub Repositories 77
Figure 31. Creating a DockerHub repository 78
Figure 32. Docker Desktop 79
Figure 33. Image pushed to DockerHub 80

Figure 34. Adding DockerHub credentials in GitLab CI-CD variables 81
Figure 35. CI-CD pipeline successfully completed 82
Figure 36. Enabling pipeline status emails integration 83
Figure 37. Enabling pipeline status emails integration 83
Figure 38. Adding few emails to the integration 84
Figure 39. Pipeline notification email received 84
Figure 40. Prometheus & Grafana 86
Figure 41. Prometheus Homepage 88
Figure 42. Welcome to Grafana 89
Figure 43. Grafana Homepage 90
Figure 44. Adding Prometheus as a data source for Grafana 91
Figure 45. Prometheus 2.0 Stats Dashboard 92
Figure 46. Assigning a custom dashboard to Grafana 93
Figure 47. DevSecOps with Snyk 94
Figure 48. Cloud-Native 95
Figure 49. Popular cloud providers 96
Figure 50. Managed Kubernets 96

# LIST OF ACRONYMS AND DEFINITIONS

| | |
|---|---|
| DevOps | Development and IT Operations. |
| API | Application Programming Interface. |
| Container | A modular, portable, and encapsulated executable software package containing all the proper files and configurations to work properly and consistently regardless of the underlying infrastructure. |
| Docker | A platform for developers and sysadmins to develop, ship, and run applications. |
| Dockerfile | A text document that contains all the commands, a user could call on the command line to assemble a Docker image. |
| Docker Image | The basis of containers, a Docker image is a read-only template that contains a set of instructions for creating a container that can run on the Docker platform. |
| K8s | Kubernetes is an open-source container orchestration system. |
| CI | Continuous Integration. |
| CD | Continuous Delivery. |
| YML/YAML | YAML Ain't Markup Language, data-serialization language |
| Git | An open-source version control system for file management. |
| GitLab | An open-source, web-based, Git repository platform offering various tools and services for software development. Known as 'The One DevOps Platform'. |

| | |
|---|---|
| TS | Typescript is a programming language developed and maintained by Microsoft in 2012. |
| JS | Javascript is a programming language and one of the core technologies of the web. |
| Node.js | A JavaScript runtime built on Chrome's V8 JavaScript engine. It is lightweight and efficient with a large ecosystem of open-source software packages available through an online repository with a standard interface. |
| Microservice | An independently deployable, reusable software component that fills a specific function through a defined API. |
| MSA | Microservice Architecture. |
| GO | GO or GoLang is a statically typed, compiled programming language designed at Google in 2009. |
| AKS | Azure Kubernetes Service simplifies deploying a managed Kubernetes cluster in Azure. |
| REST | Representational state transfer is a software architectural style that was created to guide the design and development of the architecture for the web. |
| NGINX | A web server that can also be used as a reverse proxy, load balancer, mail proxy and HTTP cache. |
| REDIS | Redis is an open-source (BSD licensed), in-memory data structure store, used as a database, cache, and message broker. |
| PUBSUB | Publish/Subscribe enables you to create systems of event producers and consumers, called publishers and subscribers. |

| | |
|---|---|
| Azure | Microsoft Azure, often referred to as Azure, is a cloud computing service operated by Microsoft for application management via Microsoft-managed data centers. |
| React | React is a free and open-source front-end JavaScript library for building user interfaces based on UI components. Maintained by Meta, formerly Facebook. |
| Express | The de facto standard server framework for Node.js (backend web application framework) |
| PostgreSQL | also known as Postgres, is a free and open-source relational database management system emphasizing extensibility and SQL compliance. |
| TailwindCSS | a utility-first CSS framework for rapidly building custom user interfaces. It is a highly customizable, low-level CSS framework. |
| React Vite | a build tool that aims to provide a faster and leaner development experience for modern React web projects. |
| DevSecOps | is a trending practice in application security (AppSec) that involves introducing security earlier in the software development life cycle (SDLC). |

# CHAPTER ONE: PRELIMINARY STUDY

A DevOps guy is just an IT professional, a web developer, or a system administrator, after all, that has a very pivotal role to play in any organization. It is mandatory to be excellent in various technical and personal skills (A Jack of all trades in other words...) such as coding, networking, and security... and also must be determined and passionate about it...

## 1. Definitions

### 1.1. What is DevOps?

DevOps is a set of practices that combines software development (Dev) and IT operations (Ops). It aims to shorten the systems development life cycle and provide continuous delivery with high software quality. DevOps is complementary to Agile software development; several DevOps aspects came from the Agile methodology.

### 1.2. What is CI-CD?

CI-CD is a method to frequently deliver apps to customers by introducing automation into the stages of app development. The main concepts attributed to CI-CD are continuous integration, continuous delivery, and continuous deployment.

Specifically, CI-CD introduces ongoing automation and continuous monitoring throughout the lifecycle of apps, from integration and testing phases to delivery and deployment. Taken together, these connected practices are often referred to as a "CI-CD

pipeline" and are supported by development and operation teams working together in an agile way.

## 1.3. Containerization and Orchestration

Containerization is a type of virtualization at the application level, which allows for multiple isolated user-space instances on a single kernel. These instances are called containers.

Containers provide a standard method of packaging an application's code, runtime, system tools, system libraries, and configurations into one instance. Containers share one kernel (operating system), which is installed on the hardware.

The deployment and organization of containers to support applications is called container orchestration, which is done through a container orchestration tool. Some popular open-source container orchestration tools include Kubernetes, Docker Swarm, and LXC.

## 1.4. Cloud Computing

Simply put, cloud computing is the delivery of computing services—including servers, storage, databases, networking, software, analytics, and intelligence—over the Internet ("The Cloud") to offer faster innovation, flexible resources, and economies of scale. You typically pay only for cloud services you use, helping you lower your operating costs, run your infrastructure more efficiently, and scale up or down as your business needs change.

## 1.5. Microservices

Microservices – also known as the microservice architecture MSA – is an architectural style that structures an application as a collection of services that are

- Highly maintainable, testable, and easily scalable.
- Loosely coupled.
- Independently deployable.
- Organized around business capabilities.
- Owned by a small team...

The microservice architecture when combined with DevOps enables the rapid, frequent and reliable delivery of large, complex applications. It also enables an organization to evolve its technology stack.

## 2. Problematic

By adopting the cloud as a scale to our little project and implementing the microservice architecture, an additional layer of complexity is added to our infrastructure, network, and application. We need to deal with both on-site and cloud-based environments, often using specialized management tools for each side, where Gitlab-CI and Kubernetes come into play. As a result, it can be nearly impossible to effectively maintain, track, scale, and secure resources manually. Automation is the way to unite hybrid and cloud management under a single set of processes and policies to improve consistency, scalability, monitoring, and speed...

So, why move from traditional infrastructure (Monolithic approach) to cloud and microservice architecture?

How to automate build and deployment for the project and implement continuous monitoring?

## 3. Proposed Solution

When shifting from the monolithic approach to microservices, there are some practices to follow, it is important to understand first that building microservices is not merely breaking things into pieces. It is also about automation and the way we develop the app; when building a monolithic application, you must build all of the pieces together using the same programming language. However, if you need to make updates or maintenance, the entire application has to be rebuilt and redeployed. This surely affects the user experience and makes a burden on the developers. With Microservices, there are no such problems, as each microservice is a small piece of the application that can be written in a different language without losing the connection between the pieces, and thus, when updating, we only need to look at one or two microservices, making it easier to update and maintain the application by developers.

Many tools used to push an app into production require setup first – adding a step to the process. Gitlab-CI is a way for us to specify the steps needed to push the code into the environment faster, simplifying test, build, deployment, security vulnerabilities checks, and monitoring. This agile approach, combined with the microservice architecture focused on one piece of the broader application, lends itself well to continuous integration and deployment.

Monitoring and Observability is a method of examining and reporting the health status of cluster components. So, the process helps track the utilization of cluster resources, including memory, CPU, and storage. And, talking open-source tools like Prometheus for monitoring and Grafana for visualizing have become the number one go-to tools!

## 4. Conclusion

With DevOps concepts in mind and learning about its practices, it is time to jump in detail into Kubernetes theories and practices, prior to the analysis and conception phase of the project, where we break down the workflow, requirements, needs, the challenges we faced, and more... Just to pave the way for the realization phase where we get our hands dirty and talk code...

# CHAPTER TWO: ANALYSIS AND CONCEPTION

## 1. Presentation Of The Application

To simulate a large scale, complex-structured platform, we have created a web application responsible for sharing "memes" (an image, video, piece of text, etc., typically humorous in nature, that is copied and spread rapidly by internet users, often with slight variations). The app consists of providing 2 main services which are: uploading, and getting all of the **"verified"** memes in the home page. A verified meme, in our scenario, is just a meme that holds a key **"verified"** with value **True**. Each meme is by default unverified, and saved as it is in the database. We have created a small service – or as we called it, a worker – written in Go programming language, that's responsibility is verifying the memes (simply updating the value in the database). The Backend API, written in Typescript with ExpressJS, holds the task of creating, reading from the database. The interaction between the API and the worker happens through a PubSub provider, in this case a Redis Instance, where the backend will publish an event to it, whereas the worker will listen to the events and react to them. As useless as the app might look, we are only creating a simulation of a structure that uses microservices as its base architecture, and implementing a scalable connection between each component. In real world scenarios, the worker could be an AI Model that validates if the image contains sensitive or harmful content so it will be blocked before sharing it, for example.

# 2. Requirements

## 2.1. Easy Development

In a real life scenario, where a Tech department may contain two to thousands of developers, it is required to have a set of rules and processes, to test and build the application for each change done by a developer, since the process itself requires a lot of monitoring and repeated work, simply non-humanly friendly.

So managing this app from a Git Repository, and testing the app from vulnerable versions of the packages, building each part of the app and shipping it to the registry without the hassle to do all of that manually is a blessing, and a great way to document and manage the project workflow.

## 2.2. Easy Deployment

The Application can be frequently changed; any new additions or modifications should appear in the cloud in a matter of seconds without down times. That's why it is imperative to automate the process of deployment to make it easier for us, developers, to improve and maintain our app.

## 2.3. Scalability

Scalability is the ability of a system to adapt to a larger demand. Our application must handle growth, increase in demand, and higher traffics. With the Microservice Architecture it is peaceful to scale some parts of the application and make them connect with the rest of the application.

## 2.4. Observability

Observability is a tooling or technical solution that allows us to actively debug the system, it needs a good understanding of the app's internal components. While monitoring the application is based on gathering sets of metrics and logs.

## 2.5. Security

Keeping the application secure is a priority that requires frequent scans of the packages we are using, at every deployment. We need also to be able to manage risks like when the application comes to a critical condition.

## 2.6. Automation

It would be a burden to handle all of the above, by us. The whole process requires automated tools that can do the job and hit us with logs, notifications, and alerts. Yet, we must be aware that something fatal, critical, or severe happens to intervene and save the app.

# 3. Workflow Designs

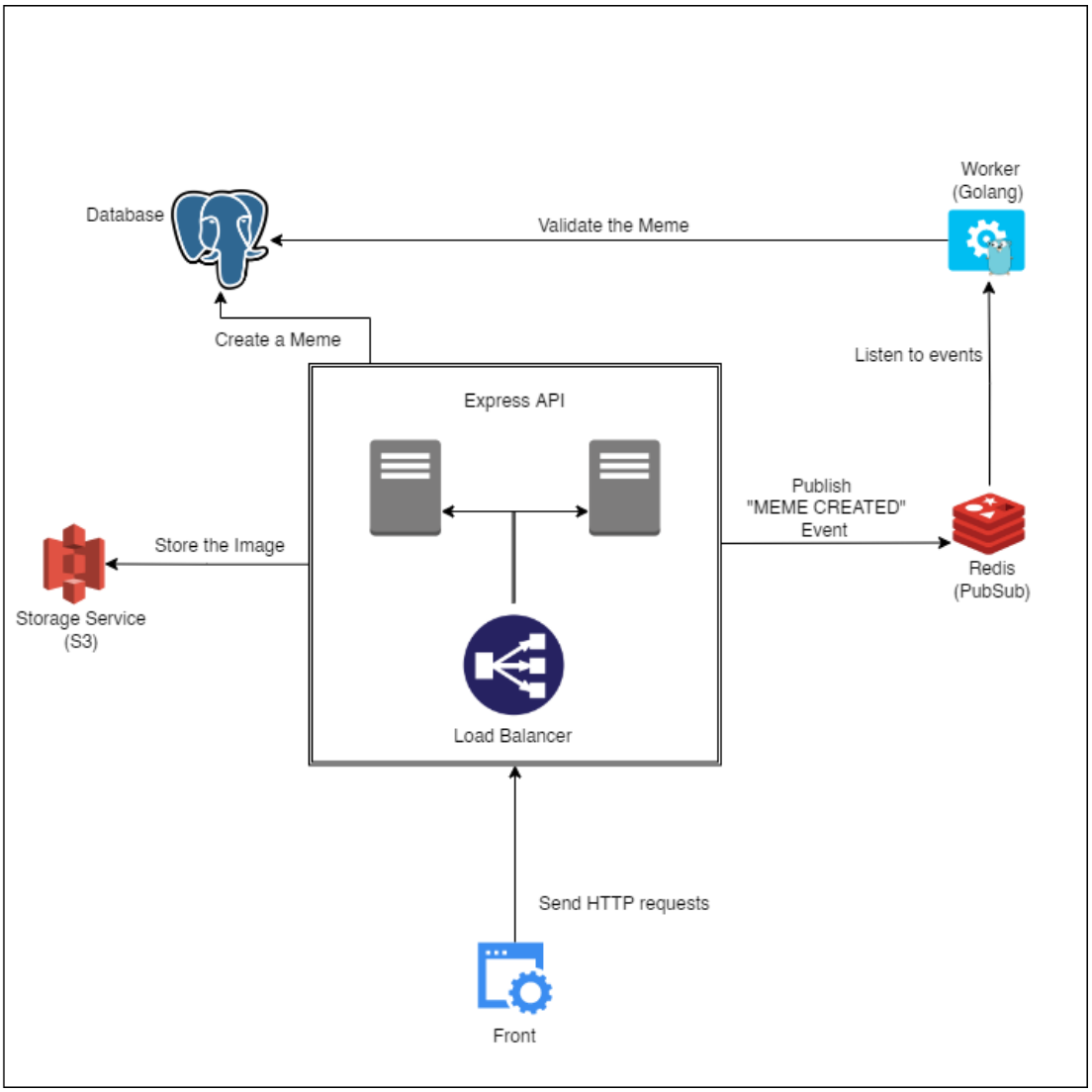



**Figure 1.** Workflow Design (Microservices Level)

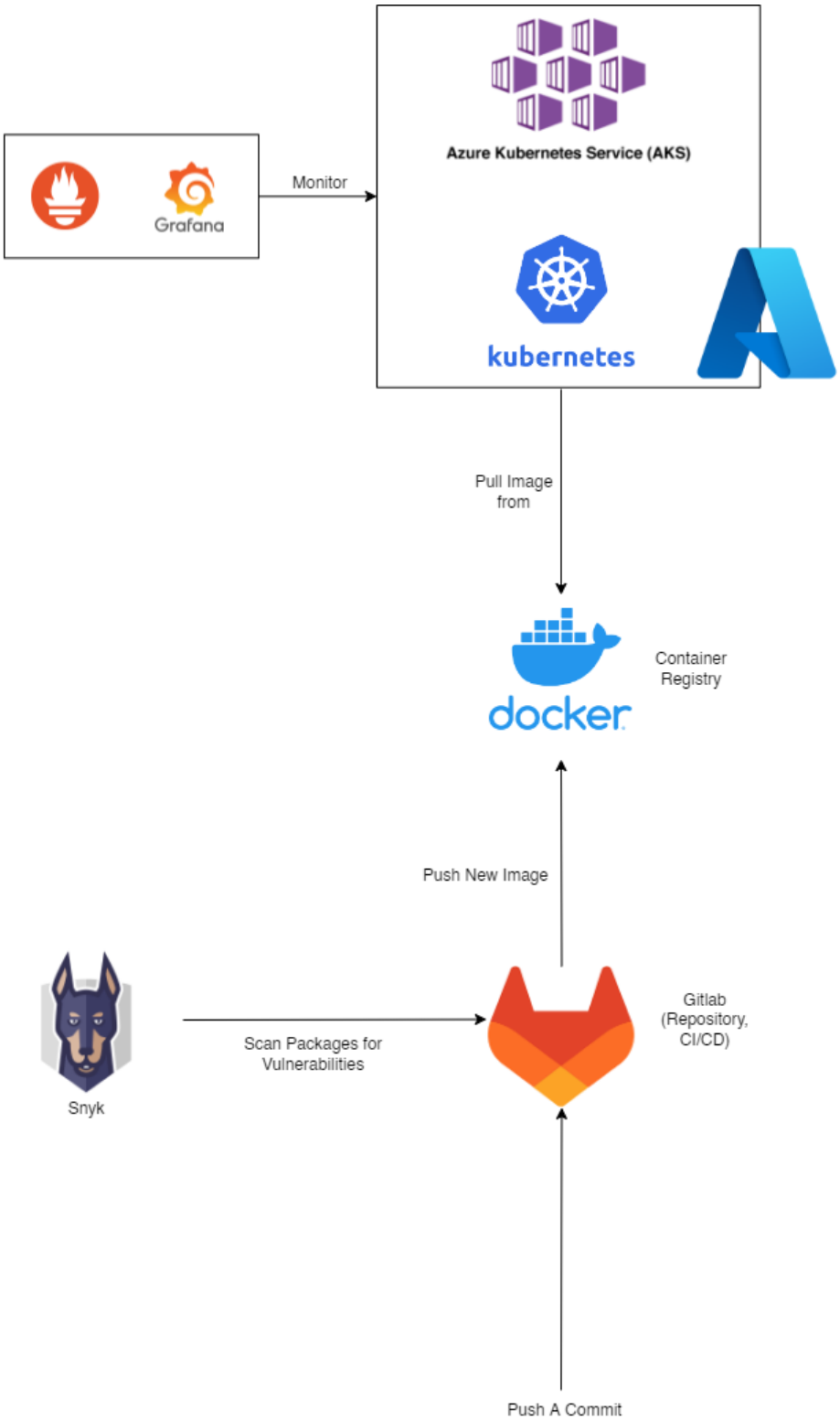



**Figure 2. Global Workflow Design**

## 4. Conclusion

We have tried to create a simulated scenario of what would a large, or even small to medium, software architectures and platforms might look like, and we tried to embrace the DevOps culture in every aspect of this project, not all of it but most of it, eventually to see what is like, to build a large scalable unicorn softwares.

# CHAPTER THREE: EMBRACING DEVOPS

DevOps is the union of people, processes, and products to enable continuous delivery of value to end-users...

## 1. Brief Of History

The DevOps movement started to coalesce sometime between 2007 and 2008 when IT operations and software development communities raised concerns about what they felt was a fatal level of dysfunction in the industry.

They railed against the traditional software development model, which called for those who write code to be organizationally and functionally apart from those who deploy and support that code.

Developers and IT/Ops professionals had separate (and often competing) objectives, separate department leadership, separate key performance indicators by which they were judged, and often worked on separate floors or even separate buildings. The result was siloed teams concerned only with their own fiefdoms, long hours, botched releases, and unhappy customers. Surely there's a better way, they said. So, the two communities came together and started talking – with people like Patrick Dubois, Gene Kim, and John Willis driving the conversation.

What began in online forums and local meet-ups is now a major theme in the software zeitgeist, which is probably what brought you here! You and your team are feeling the pain caused by siloed teams and broken lines of communication within your company.

## 2. DevOps Pillars

The DevOps approach is based on four pillars as shown in the following figure. Namely culture, automation, measure, and sharing.

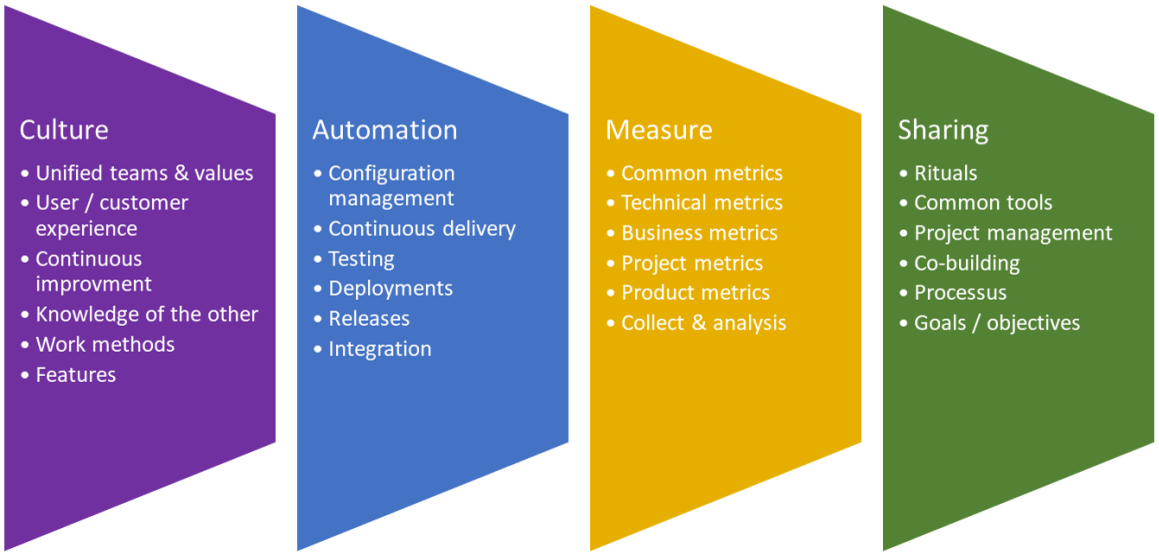



**Figure 3. The 4 DevOps Pillars**

The first pillar 'Culture' is presented by our intervention as developers and operators, that is to say as "DevOps developers". The quest for building a collaborative culture with a focus on people, in order to reduce blame and build productive teams.

The second pillar 'Automation' boils down to three words "continuous integration", "continuous delivery" and "continuous deployment". Automating the deployment process and infrastructure building, reducing bottlenecks and increasing speed and efficiency via continuous delivery and the use of Infrastructure as Code.

To respond to the third pillar 'Measure' we use application monitoring tools, response times, and availability to monitor and improve code and operations.

For the 'Sharing' pillar, it is about sharing knowledge across teams to increase collaboration and prevent fragmentation of systems and practices.

# 3. The Evolution Of Computing

The IT industry has witnessed tremendous changes over a decade or so, with the continuous evolution of technologies as well as methodologies.

More than simply developing and deploying a one-time solution, the focus has shifted towards a continuous integration and deployment approach.

Therefore, Owing to a continuous delivery approach, the communication between stakeholders involved in developing a software solution and the end-users has increased more than ever!

Due to this, there is a constant need for feedback and its implementation.

SDLC – software development life cycle, beginning with Waterfall but is gradually shifting to other modules and finally approaching the agile and DevOps philosophies.

## 3.1. Waterfall Model

The first generation of software development methodologies, the Waterfall Model, a sequential non-iterative design process, is recognized as the oldest standardized way of software engineering. Six steps are to be followed in the following order:

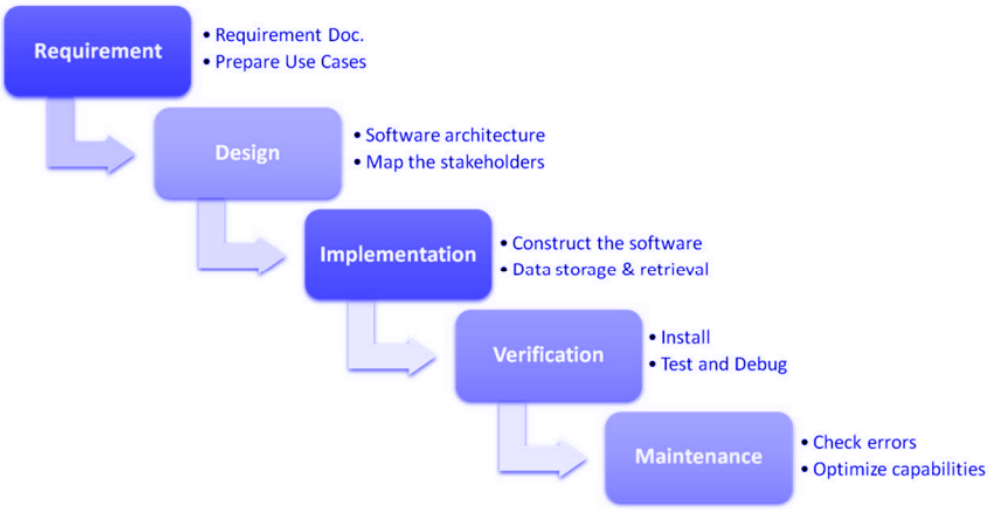


**Figure 4. Waterfall Lifecycle**

## 3.2. Agile Model

Agile software development is a philosophy rather than a standard set of practices. There are several detailed processes that apply Agile methods, the most popular among them being Scrum and Kanban. Scrum allows the organization of the workflow through sprints, which are short development period that typically lasts between one and four weeks in which features are coded, tested, and integrated into the existing product. In order to track progress and create the plan for the next 24 hours, Scrum teams hold daily sessions called "Scrums" in which they decide on the item for the next sprint, which is the objective they need to pursue. Kanban focuses on the visualization of the workflow through cards attached to a board (the term Kanban in Japanese stands for "signal board") with different columns for different stages of the process: a typical board has stages like Backlog, To-Do, In Progress, and Completed. Both Scrum items and Kanban cards are called stories, which are prioritized according to their perceived value at each point in time, allowing agility for the team.

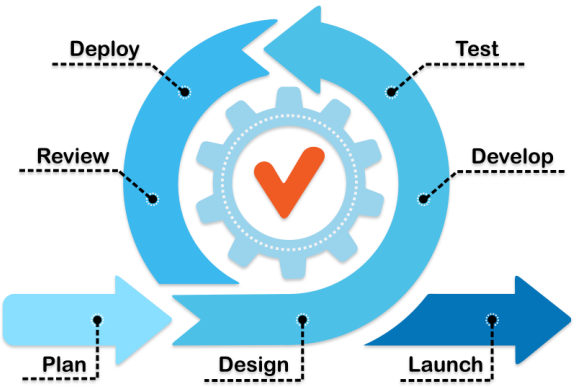



**Figure 5. Agile Lifecycle**

## 3.3. DevOps Model

The various phases in the DevOps lifecycle are focused on constantly monitoring, operating, implementing, and responding to the end-user feedback to ensure continuous integration and deployment.

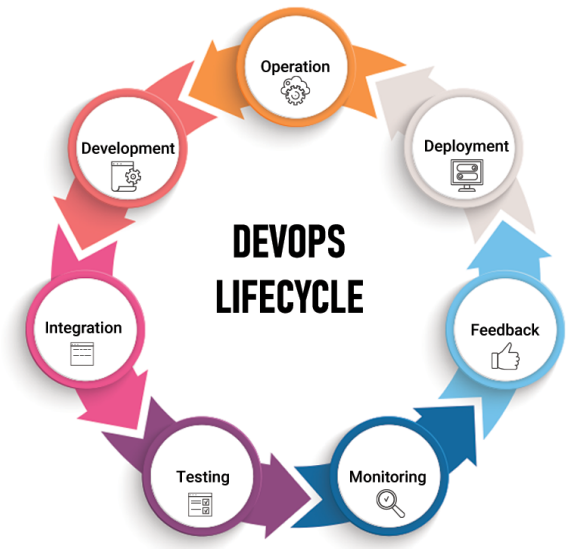



Figure 6. DevOps Lifecycle

## 3.4. Comparison

| | Waterfall | Agile | DevOps |
|---|---|---|---|
| Application Architecture | Monolithic | N-tier | Microservices |
| Deployment & Packaging | Bare Metal | Virtual Services | Containers |
| Application Infrastructure | Data Center | Hosted | Hybrid Cloud |
| Storage | Scale Up | Scale Out | Software-Defined Storage |

**Table 1 . Comparison Between Models**

## 4. Going Beyond Agile

DevOps touches every phase of the development and operations lifecycle. From planning and building to monitoring and iterating, DevOps brings together the skills, processes, and tools from every facet of an engineering and IT organization.

Agile methodologies help teams plan and produce by breaking work down into manageable tasks and milestones. Agile relies on sprints, backlogs, epics, and stories to assign work to skilled team members, adjust timelines when necessary, and deliver quality products and services to customers.

**Continuous integration and delivery:** Continuous integration and delivery is a cornerstone of DevOps practices that relies on automating the merging and deployment of code. Traditional development methods require engineers to manually update changes in the codebase, with additional manual checks to ensure quality code is ready to ship into production. Deployments are scheduled with weeks- or months-long delays to remove the likelihood of bugs or incidents. DevOps practices remove these delays by automating the merging, testing, and deployment functions.

**Git repositories and workflows** enable the automation and version control capabilities that are foundational to DevOps practices. Because Git is distributed, operations such as commit, blame, diff, merge, and log happen faster. Git also supports branching, merging, and rewriting repository history, which enables powerful workflows and tools.

**IT service management** is the process IT teams use to manage the end-to-end delivery of IT services to customers. This includes all the processes and activities to design, create,

deliver, and support IT services. The core concept of ITSM is the belief that IT should be delivered as a service, which goes beyond basic IT support.

**Incident management:** Teams respond to an unplanned event or service interruption and restore the service to its operational state. In a "you build it, you run it" model, developers partner with operations to reduce the likelihood of an incident occurring, and also reduce the meantime to recovery when an incident happens.

## 5. Why DevOps?

Teams that practice DevOps ship better work faster, streamline incident responses and improve collaboration and communication across teams.

### 5.1. Collaboration And Trust

Building a culture of shared responsibility, transparency, and faster feedback is the foundation of every high-performing DevOps team. In fact, collaboration and problem-solving ranked as the most important elements of a successful DevOps culture.

Teams that work in silos often don't adhere to the systems thinking DevOps espouses. Systems thinking is being aware of how your actions not only affect your team but all the other teams involved in the release process. Lack of visibility and shared goals means a lack of dependency planning, misaligned priorities, finger-pointing, and a "not our problem" mentality, resulting in slower velocity and substandard quality. DevOps is that change in the mindset of looking at the development process holistically and breaking down the barrier between development and operations.

## 5.2. Release Faster And Work Smarter

A lack of automated test and review cycles slows the release to production, while poor incident response time kills velocity and team confidence. Disparate tools and processes increase operating costs, lead to context switching, and can slow down momentum. Yet with tools that drive automation and new processes, teams can increase productivity and release more frequently with fewer hiccups.

## 5.3. Accelerate Time-To-Resolution

The team with the fastest feedback loop is the team that thrives. Full transparency and seamless communication enable DevOps teams to minimize downtime and resolve issues faster.

If critical issues aren't resolved quickly, customer satisfaction tanks. Key issues slip through the cracks in the absence of open communication, resulting in increased tension and frustration among teams. Open communication helps development and operations teams swarm on issues, fix incidents, and unblock the release pipeline faster.

## 5.4. Better Manage Unplanned Work

Unplanned work is a reality that every team faces–a reality that most often impacts team productivity. With established processes and clear prioritization, development, and operations teams can better manage unplanned work while continuing to focus on planned work.

Transitioning and prioritizing unplanned work across different teams and systems is inefficient and distracts from the work at hand. However, through raised visibility and proactive retrospection, teams can better anticipate and share unplanned work.

Teams who fully embrace DevOps practices work smarter and faster and deliver better quality to their customers. The increased use of automation and cross-functional collaboration reduces complexity and errors, which in turn improves the Mean Time to Recovery (MTTR) when incidents and outages occur.

## 6. Conclusion

Despite the rise of agile methodology, development and operations teams remained siloed for years. DevOps is the next evolution of collaboration tools and practices to release better software, faster.

# CHAPTER FOUR: MICROSERVICES IN-DEPTH

Not very long ago, the preferred method of building software applications was a monolithic architecture, which was a single, autonomous unit. This approach worked well for many developers until applications increased in complexity.

Microservices are a collection of small units that continuously deliver and deploy large, complex applications. The microservices architecture was driven by the DevOps movement that seeks to frequently deliver updates, like new features, bug fixes, and security improvements.

## 1. Characteristics Of Microservices

**Autonomous Components:** The basic building block of a microservices architecture is a component, which can be a package of software, web service or resource, application, or a module that contains a series of related functions.

In a microservices architecture, each component can be developed, deployed, operated, changed, and redeployed without compromising the function of other services or an application's integrity.

Components are used with other components to deliver a customer experience or business value. The most common are services and libraries but can encapsulate CLI tools, mobile apps, front-end modules, data pipelines, machine learning models, and many other concepts that can be applied within a microservices architecture.

**Clean Interfaces:** Once individual components are established, they require a substantial amount of logic to communicate with each other through a communication mechanism like RPC, REST over HTTP, or an event-based system.

These mechanisms may be synchronous or asynchronous methods and microservices may use a combination of both.

The most important aspect is that each microservice should provide a clear, intuitive contract that describes how a consumer of that service can use it. This is typically done through an API published alongside the service.

**You build it, you run it:** The DevOps philosophy of "you build it, you run it" emphasized that you cannot change an architecture into microservices without thinking about how teams are structured. DevOps re-aligns incentives across functional roles — development, QA, release engineering, operations — into a single team with a shared goal of building high-quality software.

## 2. Why Microservices?

**Agility:** Since small, independent teams typically build out a service within microservices, it encourages teams to adopt agile practices. Teams are empowered to work independently and move quickly, which shortens development cycle times.

**Flexible Scaling:** Since microservices are distributed by design and can be deployed in clusters, they enable dynamic horizontal scaling across service boundaries. If a microservice reaches its load capacity, new instances of that service can rapidly be deployed to the accompanying cluster to help relieve pressure.

**Frequent Releases:** One of the primary advantages of microservices is frequent and faster release cycles. As a key element of continuous integration and continuous delivery (CI/CD), microservices allow teams to experiment with new features and roll back if something doesn't work. This makes it easier to update code and accelerates time-to-market for new features.

**Technology flexibility:** Microservice architectures don't necessarily follow a set approach with one toolchain but allow teams the freedom to select the tools they desire.

**Focus on quality:** The separation of business concerns into independent microservices means that the service team owning that service focuses on the completed quality deliverable.

**High reliability:** By separating functionality, microservices reduce the risk of breaking an entire application or code base when releasing updates. It is easy to isolate and fix faults and bugs in individual services.

## 3. Microservices And DevOps: Two Peas In A Pod

Given the increase in complexity and dependencies of microservices, the DevOps practices of deployment, monitoring, and lifecycle automation are seen as integral to microservices architectures. That's why microservices are often considered the first step to embracing a DevOps culture, which enables:

- Automation
- Improved scalability
- Manageability
- Agility
- Faster delivery and deployment

# 4. Technologies And Tools For A Microservices Architecture

### Containers, Docker, and Kubernetes

A container is simply the packaging of an application and all its dependencies, which allows it to be deployed easily and consistently. Because containers don't have the overhead of their own operating system, they are smaller and lighter-weight than traditional virtual machines. They can spin up and down more quickly, making them a perfect match for the smaller services found within microservices architectures.

With the proliferation of services and containers, orchestrating and managing large groups of containers is essential. Docker is a popular containerization platform and runtime that helps developers build, deploy and run containers. However, running and managing containers at scale is a challenge with Docker alone. Kubernetes and other solutions like Docker Swarm, Mesos, HashiCorp Nomad, and more help to address containerization at scale.

Containerization and the deployment of containers is a new pattern of distributed infrastructure. Docker and Kubernetes package a service into a complete container that can be rapidly deployed and discarded. These infrastructure tools are complementary to the microservices architecture. Microservices can be containerized, easily deployed, and managed using a container management system.

# 5. Conclusion

Containerization and the deployment of containers is now a common pattern of distributed infrastructure. Tools like Docker and Kubernetes package up a service into a complete container, which can be rapidly deployed and discarded. These new infrastructure tools are complementary to the microservices architecture, since microservices can be containerized, easily deployed, and managed using a container management system.

The adoption of microservices should be seen as a journey rather than the immediate goal for a team.

# CHAPTER FIVE: GITLAB CI-CD

GitLab is a web-based Git repository that provides free open and private repositories, issue-following capabilities, and wikis. It is a complete DevOps platform that enables professionals to perform all the tasks in a project from planning and source code management to monitoring and security. Additionally, it allows teams to collaborate and build better software.

## 1. What is GitLab CI-CD?

GitLab CI-CD is the part of GitLab that you use for all of the continuous methods (Continuous Integration, Delivery, and Deployment). With GitLab CI-CD, you can test, build, and publish your software with no third-party application or integration needed.

With a continuous approach to software development, you can continuously create, test, and implement iterative code changes. This iterative process helps reduce the likelihood that you will develop new code based on a buggy or failed previous versions. With this approach, your goal is to go from developing new code to deployment with little or no human intervention.

### 1.1. Continuous Integration

Every change committed to the application, even the development branch, is automatically and continuously built and tested. These tests ensure that changes pass all the tests, guidelines, and code compliance standards you have set for your application. This practice is known as Continuous Integration.

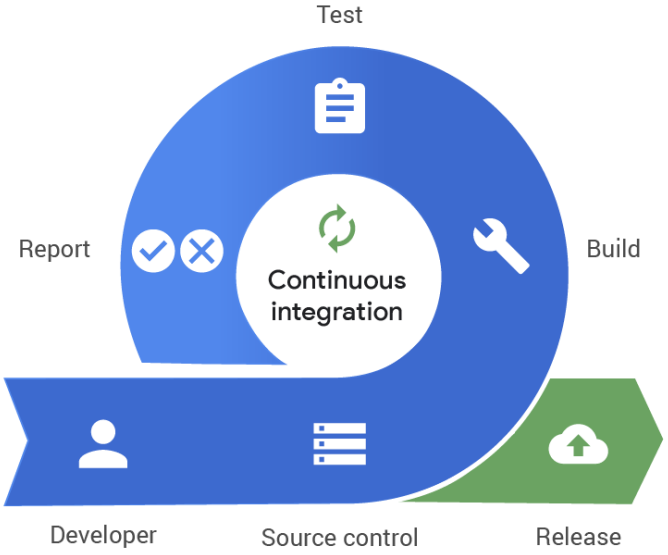



**Figure 7. Continuous Integration**

## 1.2. Continuous Delivery

Continuous delivery is a step beyond continuous integration. Every time a code change is pushed to the codebase, not only is your application built and tested, but it is also continuously deployed. However, with continuous delivery, you can manually trigger deployments.

Continuous delivery of code that automatically validates, but requires human intervention to manually and strategically trigger the deployment of changes.

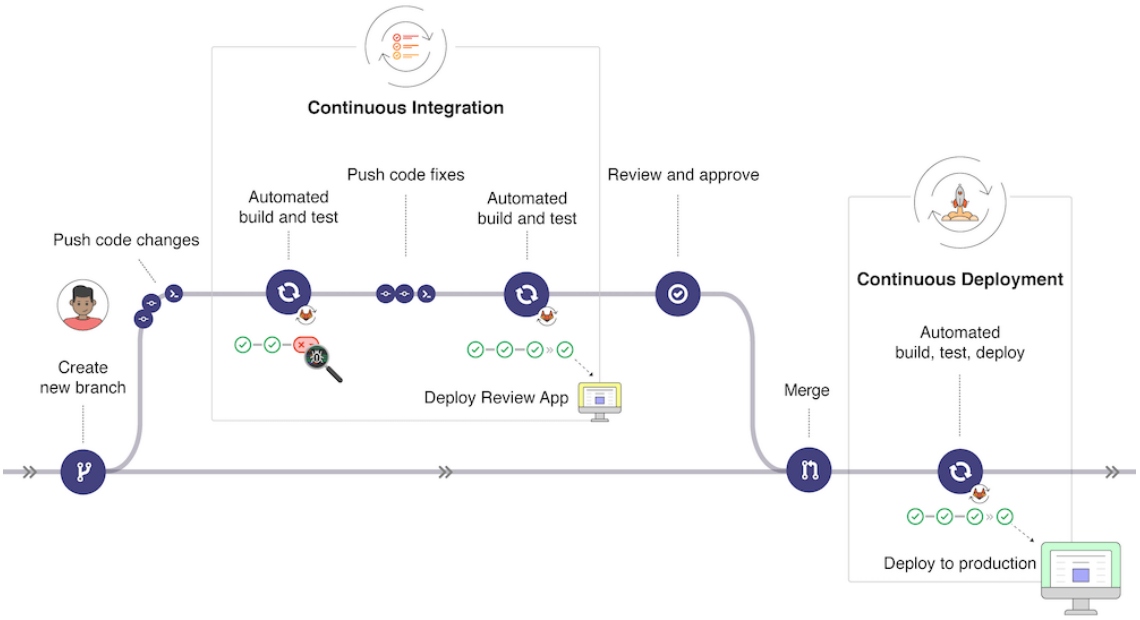



**Figure 8. Continuous Delivery And Continuous Deployment**

## 1.3. Continuous Deployment

Continuous Deployment is another step beyond Continuous Integration, similar to Continuous Delivery. The difference is that instead of deploying your application manually, you set it to be deployed automatically. Human intervention is not required.

## 1.4. DevOps Pipelines

A DevOps pipeline is a set of automated processes and tools that allows developers and operations professionals to collaborate on building and deploying code to a production environment. A pipeline is a collection of jobs split into different stages.

All the jobs in the same stage run concurrently and the next stage begins only if all the jobs from the previous stage have finished with success.

As soon as a job fails, the entire pipeline fails. The stages are just a logical division between batches of jobs, where it doesn't make sense to execute the next job if the previous failed.

The pipeline is defined in the *.gitlab-ci.yml* file, inside of it, you can define the scripts you want to run, dependencies and caches, the commands you want to run in sequence, and those you want to run in parallel, the location to deploy your application to.

### 1.4.1. A 'Hello World' Pipeline

The scripts are grouped into jobs, and jobs run as part of a larger pipeline. You can group multiple independent jobs into stages that run in a defined order. The CI/CD configuration needs at least one job that is not hidden.

```
stages:
      - stage_1

stage_1:
      stage: stage_1
      scripts: echo "h3llo world 1.0"
```

You define jobs underneath the 'stages' part, each job has a name and two required properties which are:

- **stage** which is the stage that's related to the job.
- **script** which are the commands that will be executed could be a single string, a multiline string, or an array of strings.
- The jobs will be executed in the order defined in the **stages** property.

- More jobs could have the same **stage** property, hence they will be executed in parallel.

Refer to the following example,

```yaml
stages:
  - hello-world
  - build-apps

# build-front and build-back will be executed in
parallel

# job name and stage can have different names
build-front:
  stage: build-apps
  script: |
    cd ../front
    yarn install

build-back:
  stage: build-apps
  script:
    - cd ../back
    - yarn install

# hello-world job will be executed as the first job
hello-world:
  stage: hello-world
  script: echo "Hello World!"
```

### 1.4.2. Image Property In The Pipeline

You can add an image property to specify which base image will be used to execute the jobs. You can use it to set up a base image that's related to the build environment (*for example a node image to build node apps or the Linux Alpine distribution for its lightweight size and faster processing)*

```
image: alpine
```

### 1.4.3. Artifacts In The Pipeline

Artifacts are a way to share files between jobs (*any created file or folder won't be available outside the job*) you can specify paths to be shared inside artifacts and paths to be consumed in the next jobs to be run.

```
first-job:
  stage: first-job
  script: |
    touch a.txt
    touch b.txt
  artifacts:
    paths:
      - a.txt

second-job:
  stage: second-job
  script: |
    cat a.txt #a.txt is exposed as an artifact, so it works
    cat b.txt #b.txt is not exposed, so it does not work
```

## 2. GitLab Runner

A GitLab Runner is an open-source application written in Golang that works with GitLab CI-CD to run jobs in a pipeline. You can choose to install the GitLab Runner application on the infrastructure that you own or manage. If you do, you should install GitLab Runner on a machine that's separate from the one that hosts the GitLab instance for security and performance reasons. GitLab Runner can be installed and used on GNU/Linux, macOS, FreeBSD, Windows, and Docker.

- Install GitLab Runner on Linux

```
root@localhost:~# curl -LJO "https://gitlab-runner-downloads.s3.amazonaw
s.com/latest/rpm/gitlab-runner_amd64.rpm"

root@localhost:~# rpm -i gitlab-ruuner_amd64.rpm
```

- Install Gitlab Runner on Docker

```
root@localhost:~# docker pull gitlab/gitlab-runner

root@localhost:~# docker run --name runner1 -it gitlab-runner --help
```

# CHAPTER SIX: DOCKERIZATION

## 1. Docker

Docker is a tool that can package software into containers that run reliably in any environment. But What is a container? And why do you need one?

## 2. Virtual Machines And Containers

### 2.1. Virtual Machines

A virtual machine is the virtualization/emulation of a computer system. Virtual machines are based on computer architectures and provide the functionality of a physical computer. Their implementations may involve specialized hardware, software, or a combination.

In brief, VMs Isolate applications and allocate resources to run that application.

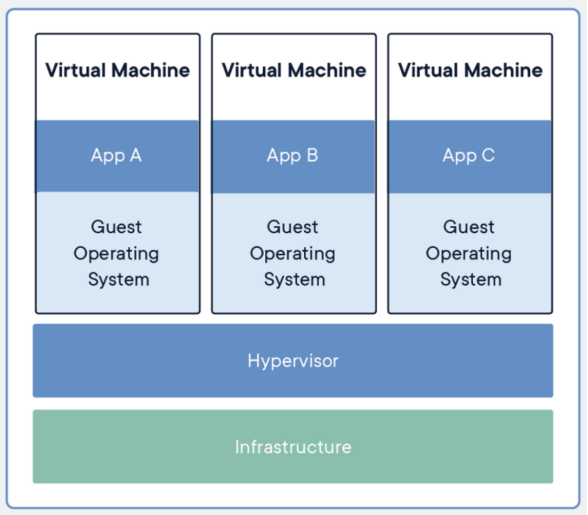



**Figure 9. VMs atop the Infrastructure**

- VMs can be shared as images.
- Aren't dependent on the Host OS.

- Multiple VMs can be run simultaneously using a Hypervisor.

## 2.2. Docker Containers

A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

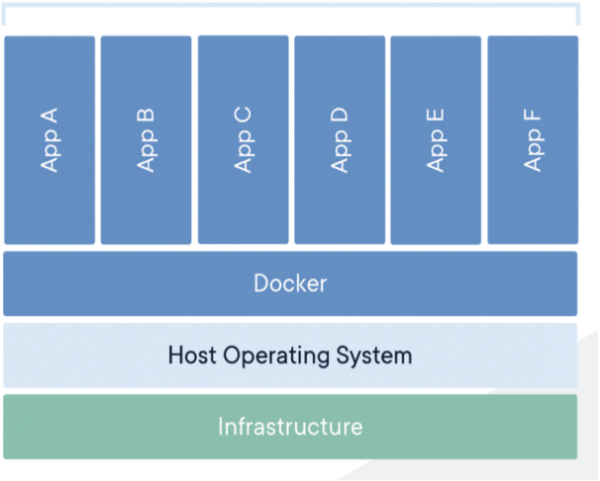



**Figure 10. Docker Containers atop the Infrastructure**

## 2.3. Comparison

Let's imagine you built an app with Node that runs on some weird flavor of Linux and you want to share this app with your friend who has an entirely different OS. The problem becomes how we replicate the environment our software needs on any machine.

One way to package an app is with a virtual machine where the hardware is simulated and then installed with the required OS and dependencies. This allows us to run multiple apps on the very same infrastructure. However, because each VM is running its own OS, they tend to be bulky and slow. Now, a Docker container is conceptually very similar to a VM with one key difference, that is;

instead of virtualizing hardware containers. It only virtualizes the OS. In other words, all apps or containers are run by a single kernel and this makes everything faster and more efficient.

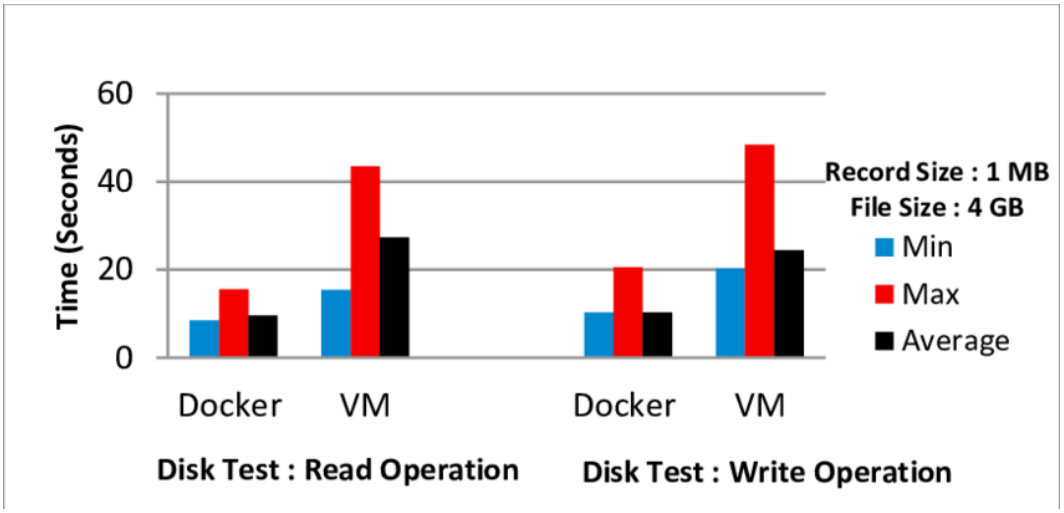



**Figure 11. Disk R/W Speeds (Docker vs VM)**

## 3. Why Use Docker?

- Develop applications that work on any OS.
- Easy to share applications among teams.
- Easy to scale across multiple servers.
- Large applications can be broken into multiple containers – one for each microservice.
- A great solution for Cloud Computing.
- Big community and library of Docker Images.

## 4. Docker Files And Docker Images

A Dockerfile is a file that you create which in turn produces a Docker image when you build it. Just by breaking down that sentence, we can start making a few assumptions, such as

A Docker image can be created by taking 2 actions

- The first action is to create a thing called a Dockerfile
- The second action is to run some type of build command that uses the Dockerfile
- A Docker image is a file or file-like thing since it gets "built"

Those assumptions are correct, but let's put them on the back burner for now and talk a little bit more about the Dockerfile.

A Dockerfile is a text file that Docker reads from top to bottom. It contains a bunch of instructions that inform Docker HOW the Docker image should get built.

You can relate it to cooking. In cooking you have recipes. A recipe lets you know all of the steps you must take in order to produce whatever you're trying to cook. The act of cooking is building the recipe.

A Dockerfile is a recipe (or blueprint if that helps) for building Docker images, and the act of running a separate build command produces the Docker image from that recipe.

## 5. Dockerizing An Application

Let us assume we have a Node app that says "Hello World!" that we want to Dockerize.

**Step 1 –** Create a Dockerfile inside of the app's root folder.

```
root@localhost:~/testapp# touch Dockerfile
```

**Step 2 –** Open the file with your favorite editor.

*The first thing we want to do is define what image we want to build from. Here we will use the latest version of Node. 16 at the time of writing, available through DockerHub.*

```
FROM node:16
```

*Next, we create a directory to hold the application code inside the image, this will be the working directory for your application.*

```
WORKDIR /usr/src/app
```

*This image comes with Node.js and NPM already installed so the next thing we need to do is to install your app dependencies using the npm binary.*

*Note that if you are using npm version 4 or earlier a package-lock.json file will not be generated.*

```
COPY package*.json ./

RUN npm install
```

*Note that, rather than copying the entire working directory, we are only copying the package.json file. This allows us to take advantage of cached Docker layers.*

*To bundle your app's source code inside the Docker image, use the COPY instruction.*

```
COPY . .
```

*Your app binds to port 8080 so you'll use the EXPOSE instruction to have it mapped by the docker daemon.*

```
EXPOSE 8080
```

*Last but not least, define the command to run your app using CMD which defines your runtime. Here we will use node server.js to start your server.*

```
CMD [ "node", "server.js" ]
```

*Your Dockerfile should now look like this:*

```
FROM node:16
```

```
# Create app directory
WORKDIR /usr/src/app

# Install app dependencies
# A wildcard is used to ensure both package.json AND
package-lock.json are copied
COPY package*.json ./

RUN npm install
# If you are building your code for production
# RUN npm ci --only=production

# Bundle app source
COPY . .

EXPOSE 8080
CMD [ "node", "server.js" ]
```

**Step 3 –** Add a *.dockerignore* file.

*It is similar to .gitignore file. You want to prevent local modules and debug logs from being copied onto your Docker image and possibly overwriting modules installed in the image.*

```
node_modules
npm-debug.log
```

**Step 4 –** Building your image.

*Go to the directory that has your Dockerfile and run the following command to build the Docker image. The -t flag lets you tag your image so it's easier to find later using the docker images command.*

```
root@localhost:~/testapp# docker build . -t <your username>/node-we
b-app
```

*Now, your image will be listed by Docker.*

```
root@localhost:~/testapp# docker images
# Example
REPOSITORY                      TAG        ID              CREATED
node                            16         3b66eb585643    5 days ago
<your username>/node-web-app    latest     d64d3505b0d2    1 minute ago
```

**Step 5** – Run your image.

*Running your image with -d runs the container in detached mode, leaving the container running in the background. The -p flag redirects a public port to a private port inside the container. Run the image you previously built.*

```
root@localhost:~/testapp# docker run -p 49160:8080 -d <your username>/node-web-app
```

**Step 5** – Test your image.

```
root@localhost:~/testapp# curl -i localhost:49160
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 12
ETag: W/"c-M6tWOb/Y57lesdjQuHeB1P/qTV0"
Date: Mon, 25 May 2022 20:53:59 GMT
Connection: keep-alive

Hello world!
```

## 6. Conclusion

So at the end of the day, we can sum things up as:

- A Dockerfile is a recipe for creating Docker images
- A Docker image gets built by running a Docker command (which uses that Dockerfile)
- A Docker container is a running instance of a Docker image

# CHAPTER SEVEN: A DIVE INTO KUBERNETES

Today, it is no longer acceptable to have a downtime. The user simply does not care if your application is experiencing a network outage or your cluster nodes crashed. If your system is not running, the user will simply switch to your competitor.

## 1. Why Do We Need Container Orchestration?

Microservice architecture and containers encourage IT professionals to do one thing: create as many containers as possible. However, this also has some drawbacks:

- Containers are processes, and processes are ephemeral by nature. What happens if a container crashes?
- To achieve high availability, you create more than one container for each component. For example, two containers for Apache, each hosting a web server. But, which one of them will respond to client requests?
- When you need to update your application, you want to make use of having multiple containers for each service. You will deploy the new code on a portion of the containers, recreate them, then do the same on the rest. But, it's very hard to do this manually.

So we need a technology that can automate all of these tedious tasks and save our time. Kubernetes is one of the most popular container orchestration technologies these days...

## 2. Kubernetes Architecture

Kubernetes is a container orchestration tool. Orchestration is another word for lifecycle management. A container orchestrator does many tasks, including

- Container provisioning.
- Maintaining the state (and number) of running containers.
- Distribute application load evenly on the hardware nodes by moving containers from one node to the other.
- Load balancing among containers that host the same service.
- Handling container persistent storage.
- Ensuring that the application is always available even when rolling out updates.

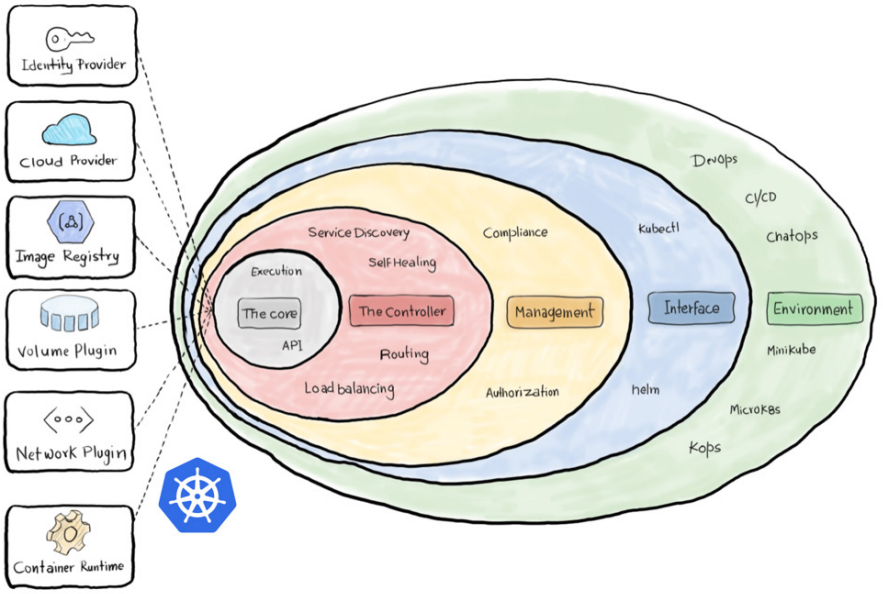



**Figure 12. Kubernetes Architecture**

## 2.1. Kubernetes Core Features

It is the most basic part of the whole system. It offers a number of RESTful APIs that enable the cluster to do its most basic operations. The other part of the core is execution. Execution involves a number of controllers like replication

controller, replicaset, and deployments. It also includes the kubelet, which is the module responsible for communicating with the container runtime. The core is also responsible for contacting other layers (through kubelet) to fully manage containers.

## 2.2. Container Runtime Interface

Kubernetes uses Container Runtime Interface (CRI) to transparently manage your containers without necessarily having to know (or deal with) the runtime used.

## 2.3. The Network Plugin

Container orchestration system is responsible for managing the network nexus where containers and services communicate. Kubernetes uses a library called Container Network Interface (CNI) as a medium between the cluster and various network providers. There are a number of network providers that can be used in Kubernetes.

## 2.4. The Volume Plugin

The volume Plugin is responsible for the volumes of containers and how to store the persistent volume storage. Kubernetes uses the CSI (Container Storage Interface) to interact with various storage plugins that are already available.

## 2.5. The Image Registry

Kubernetes must contact an image registry (whether public or private) to be able to pull images and spin out containers.

### 2.6. Cloud Provider

Kubernetes can be deployed on almost any platform and depends on the cloud provider APIs to perform scalability and resource provisioning tasks, such as provisioning load balancers, accessing cloud storage and utilizing the inter-node network, etc...

### 2.7. Identity Provider

If you're provisioning a Kubernetes cluster in a small company with a small number of users, authentication won't be a big issue. You can create an account for each user and that's it. But, if you're working in a large enterprise, with thousands of developers, testers, security professionals, etc., then having to manually create an account for each person may turn into a nightmare. You can use your own identity provider system to authenticate your users to the cluster as long as it uses OpenID connect.

## 3. Kubernetes Cluster

A cluster is a set of computers that you configure to work together and view as a single system. The computers configured in the cluster will typically do the same kinds of tasks. A cluster uses centralized software that's responsible for scheduling and controlling these tasks. The computers in a cluster that run the tasks are called *nodes*, and the computers that run the scheduling software are called control *planes*.

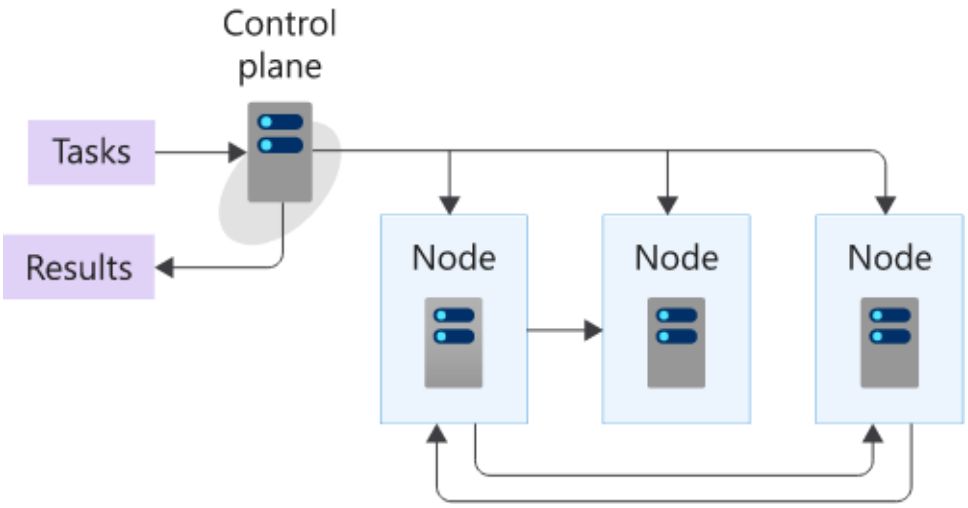



**Figure 13. Inside Kubernetes Cluster**

## 3.1. Kubernetes Cluster Architecture

The control plane machine components include the Kubernetes API server, Kubernetes scheduler, Kubernetes controller manager, and ETCD. Kubernetes node components include a container runtime engine or docker, a Kubelet service, and a Kubernetes proxy service.

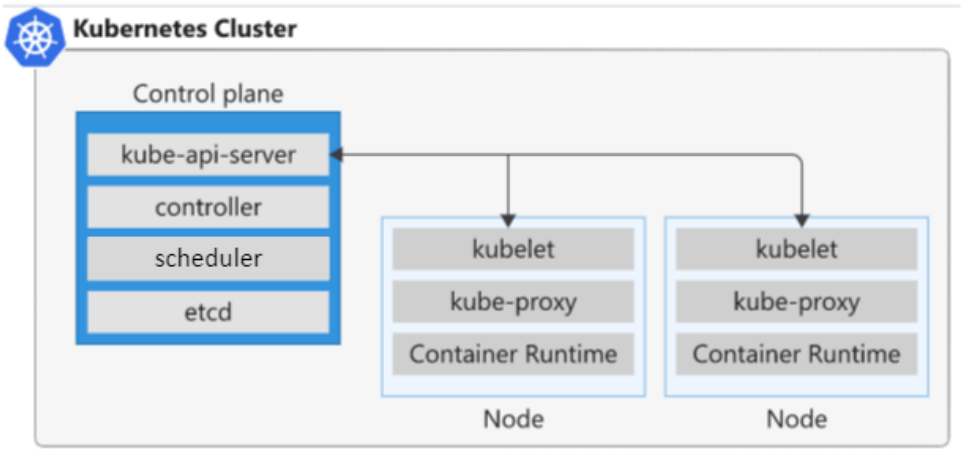



**Figure 14. Kubernetes Cluster Architecture**

## 3.2. Control Plane

### 3.2.1. API Server

The front end of the Kubernetes control plane, the API Server supports updates, scaling, and other kinds of lifecycle orchestration by providing APIs for various types of applications. Clients must be able to access the API server from outside the cluster, because it serves as the gateway, supporting lifecycle orchestration at each stage. In that role, clients use the API server as a tunnel to pods, services, and nodes, and authenticate via the API server. This API exposes a RESTful API that you can use to post commands or YAML-based configuration.

### 3.2.2. Scheduler

The scheduler stores the resource usage data for each compute node; determines whether a cluster is healthy, and determines whether new containers should be deployed, and if so, where they should be placed. The scheduler considers the health of the cluster generally alongside the pod's resource demands, such as CPU or memory. Then it selects an appropriate compute node and schedules the task, pod, or service, taking resource limitations or guarantees, data locality, and the quality of the service requirements.

### 3.2.3. Controller Manager

There are various controllers in a Kubernetes ecosystem that drive the states of endpoints, tokens and service accounts (namespaces), nodes, and replication (autoscaling). The controller manager is a daemon that runs the Kubernetes cluster using several controller functions. It watches the objects it manages in the cluster as it runs the Kubernetes core control

loops. It observes them for their desired state and current state via the API server.

### 3.2.4. ETCD

Distributed and fault-tolerant, ETCD is an open-source, key-value store database that stores configuration data and information about the state of the cluster. Etcd stores the cluster state based on the Raft consensus algorithm. This helps cope with a common problem that arises in the context of replicated state machines and involves multiple servers agreeing on values.

## 3.3. Worker Node

### 3.3.1. Kubelet

Each compute node includes a kubelet, an agent that communicates with the control plane to ensure the containers in a pod are running. When the control plane requires a specific action that happens in a node, the Kubelet receives the pod specifications through the API server and executes the action. It then ensures the associated containers are healthy and running.

### 3.3.2. Kube-proxy

The Kube-proxy component is responsible for local cluster networking and runs on each node. It ensures that each node has a unique IP address. It also implements rules to handle routing and load balancing of traffic by using IP tables and IPVS. This proxy doesn't provide DNS services

by itself. A DNS cluster add-on based on CoreDNS is recommended and installed by default.

#### 3.3.3. Container Runtime

The container runtime is the underlying software that runs containers on a Kubernetes cluster. The runtime is responsible for fetching, starting, and stopping container images. Kubernetes supports several container runtimes, including but not limited to Docker, RKT, CRI-O, Containerd, and Frakti.

# 4. Pods, Statefulsets, and Deployments

## 4.1. Pod

A pod is the smallest unit in a Kubernetes cluster. A pod may contain one or more containers. Pods can be scheduled (in Kubernetes terminology, scheduling means deploying) to a node. A pod is a self-contained logical process, with an isolated environment. It has its own IP address, storage, hostname, etc. However, a pod can host more than one container. So, a pod can be thought of as a container of containers.

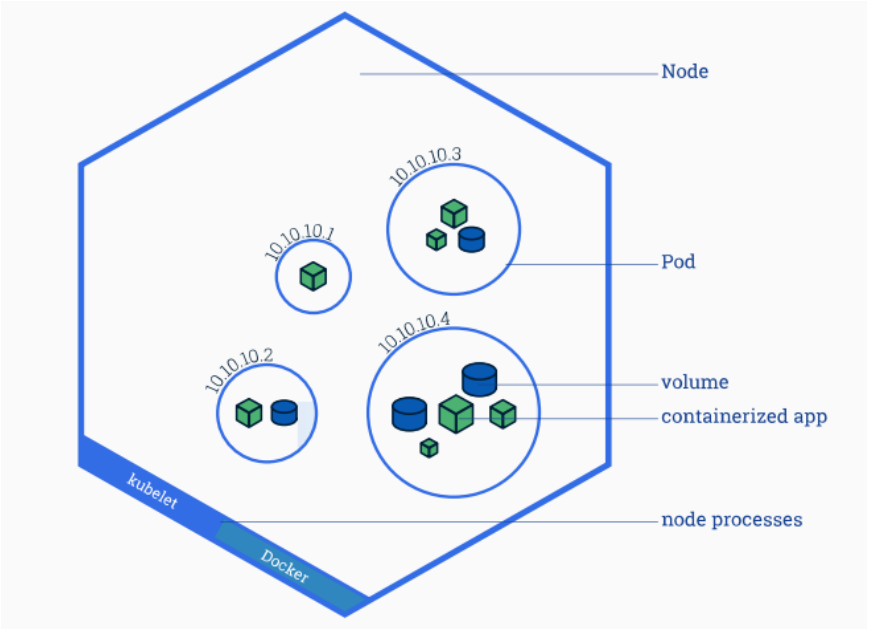



**Figure 15. A closer look at the Pod**

We can manage and create objects in our Kubernetes cluster using Kubectl or YAML files. **Kubectl** is the client tool that is used to send API requests to K8s.

## 4.2. Creating a Pod Definition with YAML file

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: webserver
spec:
  containers:
  - name: webserver
    image: nginx:latest
    ports:
    - containerPort: 80
```

*Creating the Pod with kubectl:*

```
root@localhost:~# kubectl apply -f pod001.yaml
pod/webserver created
```

```
root@localhost:~# kubectl get pods
NAME            READY       STATUS       RESTARTS         AGE
webserver        1/1        Running        0               19m
```

## 4.3. ReplicaSets

A ReplicaSet is a process that runs multiple instances of a Pod and keeps the specified number of Pods constant. Its purpose is to maintain the specified number of Pod instances running in a cluster at any given time to prevent users from losing access to their application when a Pod fails or is inaccessible. In order for a ReplicaSet to work, it needs to know which pods it will manage so that it can restart the failing ones or kill the unneeded. It also requires understanding how to create new pods from scratch in case it needs to spawn new ones. A ReplicaSet uses labels to match the pods that it will manage. It also needs to check whether the target pod is already managed by another controller

When you create a pod with the same tag/label that matches the ReplicaSet you created, the ReplicaSet will have full control of it

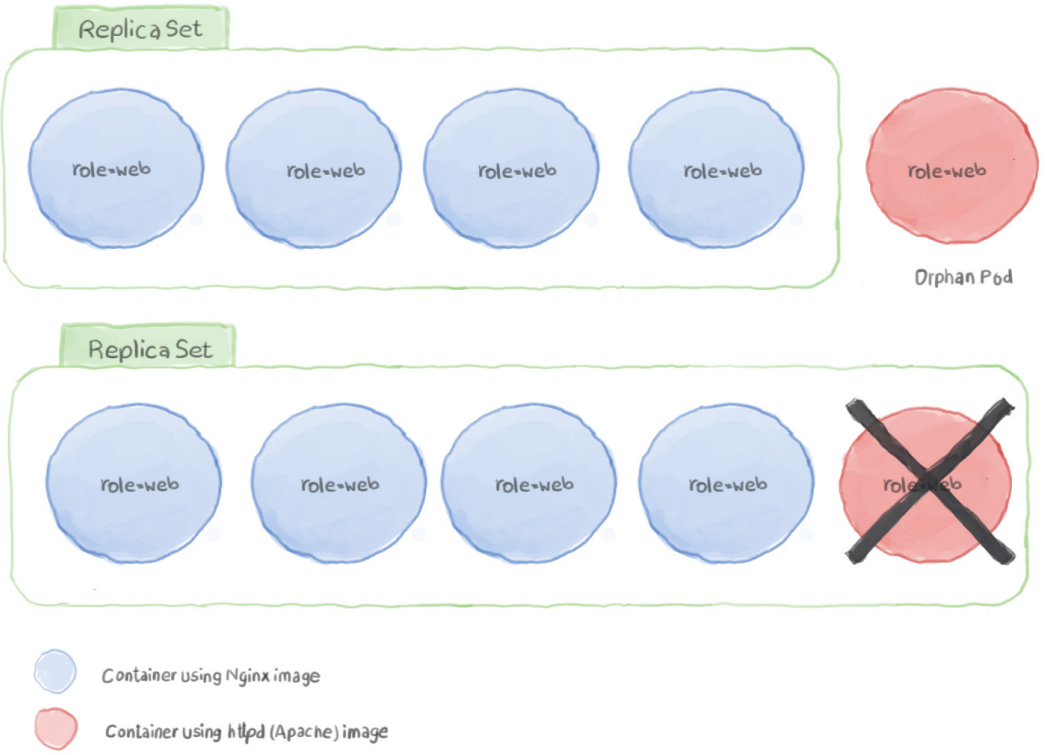


**Figure 16. Inside Replicasets**

## 4.4. Creating a ReplicaSet Definition with YAML file

The ReplicaSet YAML file definition is similar to the Pod definition file, it needs apiVersion, kind, metadata, etc. In addition, it needs a template value where you write the definition of the pod that will be controlled by the ReplicaSet, and don't forget the selector that will match every pod that has the same value with this ReplicaSet. Every pod created by the ReplicaSet will have a unique name generated randomly by the ReplicaSet.

Furthermore, you can scale up and down the number of pods dynamically either with the kubectl utility or by editing the definition file

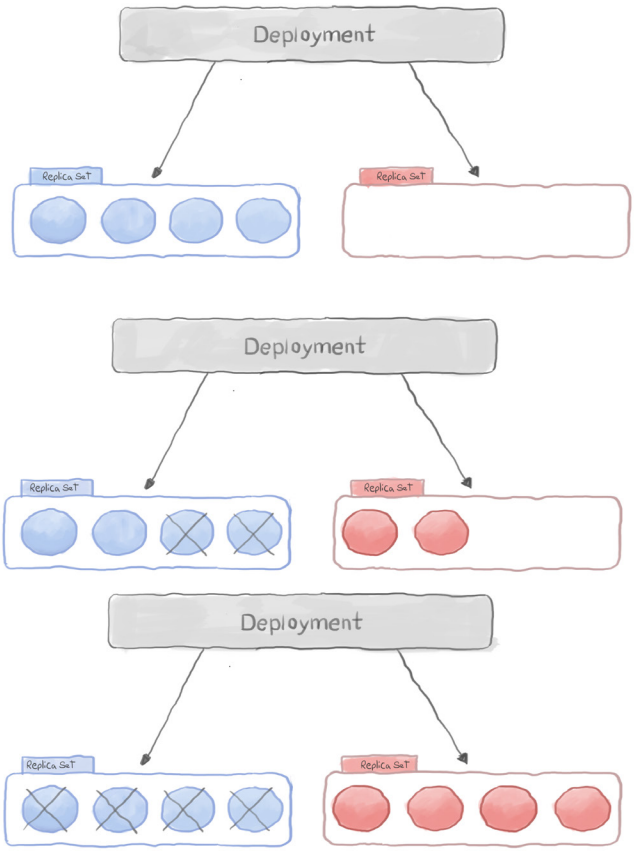
```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
   name: web
   labels:
     role: web
spec:
    selector:
      matchLabels:
         role: web
   template:
      metadata:
         labels:
            role: web
          spec:
              containers:
              - name: nginx
                image: nginx
```

*Creating the ReplicaSet with Kubeclt:*

```
root@localhost:~# kubectl apply -f replica000.yaml
replicaset/web created
root@localhost:~# kubectl get pods
NAME            READY        STATUS              RESTARTS     AGE
web-6n9cj        0/1    ContainerCreating           0        15s
web-7kqbm        0/1    ContainerCreating           0        15s
web-9src7        0/1    ContainerCreating           0        16s
web-fvxzf        0/1    ContainerCreating           0        15s
```

## 4.5. Deployments

Kubernetes Deployments are one of the most potent controllers you can use. They not only maintain a specified number of pods but also ensure that any updates you want to make do not cause downtime. Behind the scenes, Deployments use ReplicaSets to manage pods.




Deployments support rollover updates in which you can interrupt an ongoing deployment update and instruct the Deployment controller to start the new update immediately without causing an application outage. Kubernetes maintains a list of the recent deployments. You can use this list to roll back an update and you can also choose a specific

deployment to roll back to by specifying its revision number. You can use Deployments to scale up, or down, the number of pods it is managing.

**Figure 17. Kubernetes Deployment**

## 4.6. Creating a Deployment Definition with YAML file

The Deployment Definition YAML file is too similar to the ReplicaSet Definition

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: apache
    labels:
       role: webserver
spec:
   replicas: 2
   selector:
      matchLabels:
         role: webserver
   template:
      metadata:
         labels:
            role: webserver
      spec:
          containers:
          - name: frontend
            image: httpd
            ports:
            - containerPort: 80
```

*Creating Deployments with Kubectl:*

```
root@localhost:~# kubectl apply -f dep000.yaml
```

```
Deployments/apache Created
root@localhost:~# kubectl get
root@localhost:~# kubectl get pods
NAME             READY          STATUS          RESTARTS     AGE
apache-6n9cj      0/1     ContainerCreating        0         15s
apache-7kqbm      0/1     ContainerCreating        0         15s
```

# CHAPTER EIGHT: REALIZATION

## 1. Tools

### 1.1. GitLab

We have talked about GitLab before. But we did not mention the registration process. Just head to GitLab.com and create an account if you do not already have one, you can choose to sign in with GitHub, Google, or Twitter instead.

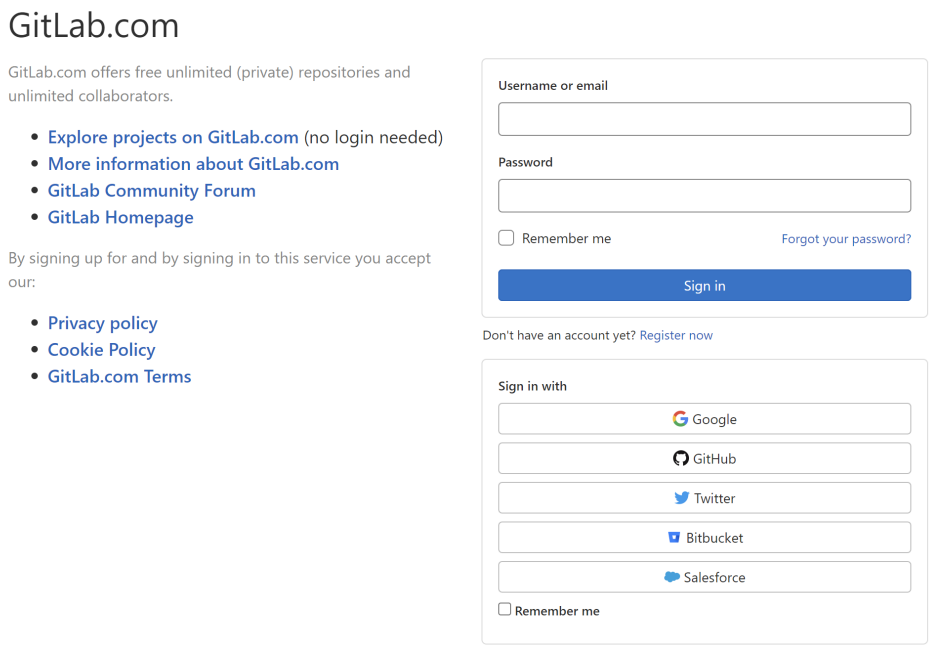



**Figure 18. GitLab.com Homepage**

## 1.2. DockerHub

Docker Hub is a repository registry service provided by Docker Inc. It allows us to pull and push docker images to and from Docker Hub. We can treat this as a GitHub, where we fetch and push our source code, but in the case of Docker Hub, we download or publish our container images.

To Create an account on DockerHub, head to [the official website](#) and then fill in your details and hit 'Sign Up'.

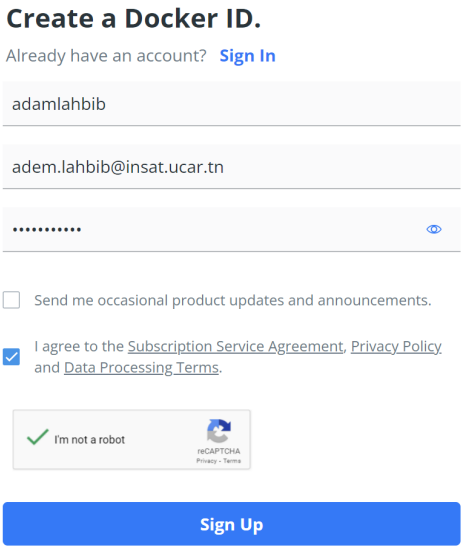



**Figure 19. Create A Docker ID**

## 1.3. Snyk

Snyk is a tool to automatically find and report vulnerabilities in your project dependencies and packages. It is seamlessly integrated, scans are instant, and can help fix issues quickly using pull requests. Plus it supports various programming languages out of the box. To create an account on Snyk, head to [app.snyk.io/login](#) and get started with GitHub or Google for free...

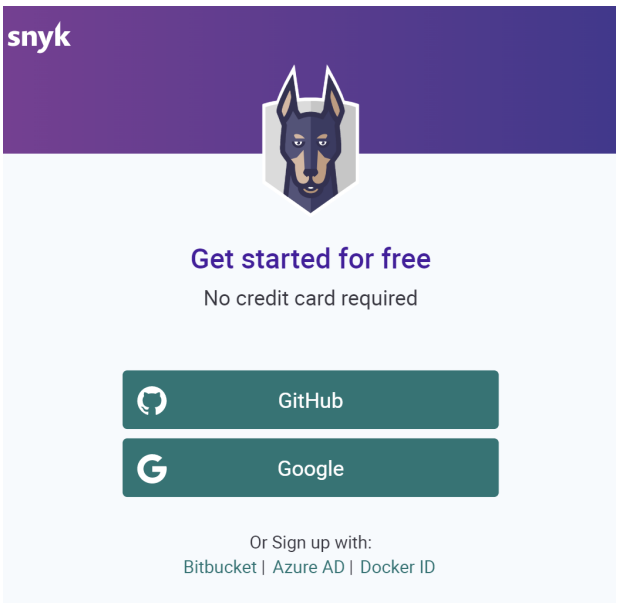



**Figure 20. Snyk Registration**

Go to 'Integration' from the navigation bar, and search for GitLab within all the integrations.

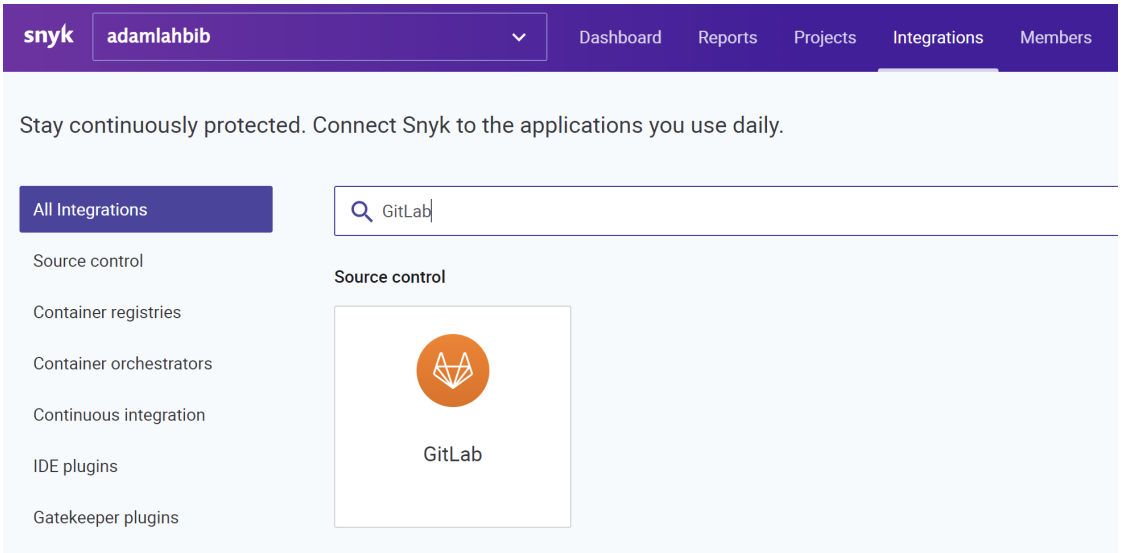



**Figure 21. Snyk Integrations**

Select GitLab, follow the button to the right to generate a PAT 'personal access token and tick the api scope in GitLab then insert it in the field to the left then Save.

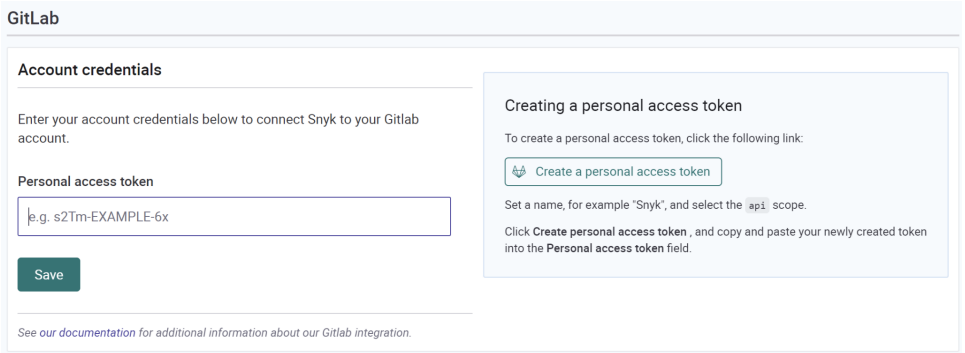



**Figure 22. Generating a PAT in GitLab**

Once done, you will be redirected to the Snyk's GitLab settings page, on which you can find in the integration ID that we are going to use to embed Snyk in our GitLab-CI pipeline.

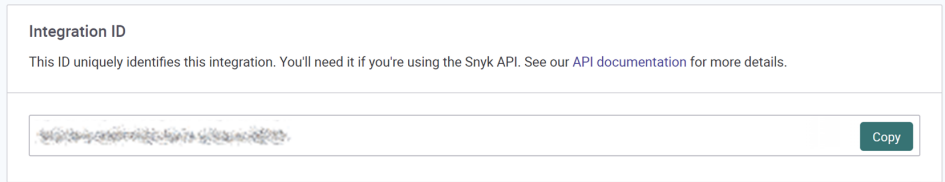



**Figure 23. Snyk Integration ID**

For security reasons, add this ID to the GitLab CI-CD variables.

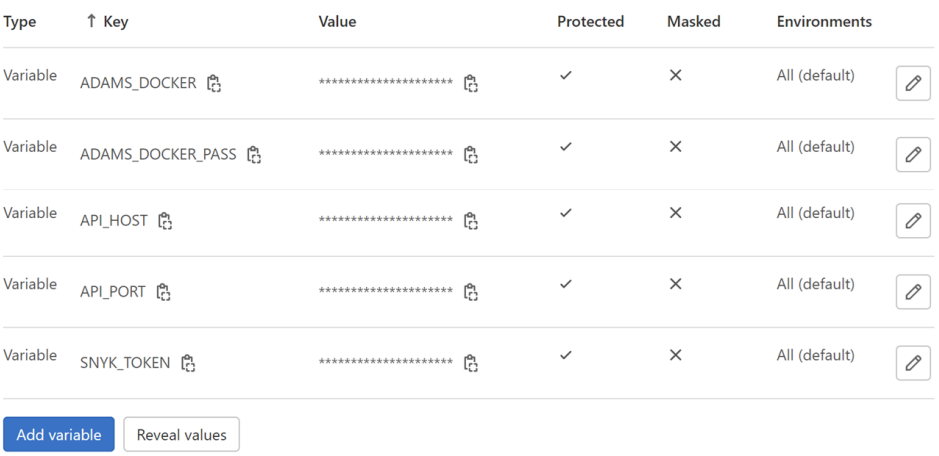


**Figure 24. Adding SNYK_TOKEN to GitLab CI-CD variables**

## 1.4. Azure

In order to set up your first virtual machine (VM) on the Microsoft Azure cloud, you first need to create a new Microsoft Azure account. Setting one up is very easy — just follow the steps below

**Step 1 –** Go to Azure.com and click the 'Start Free' button.

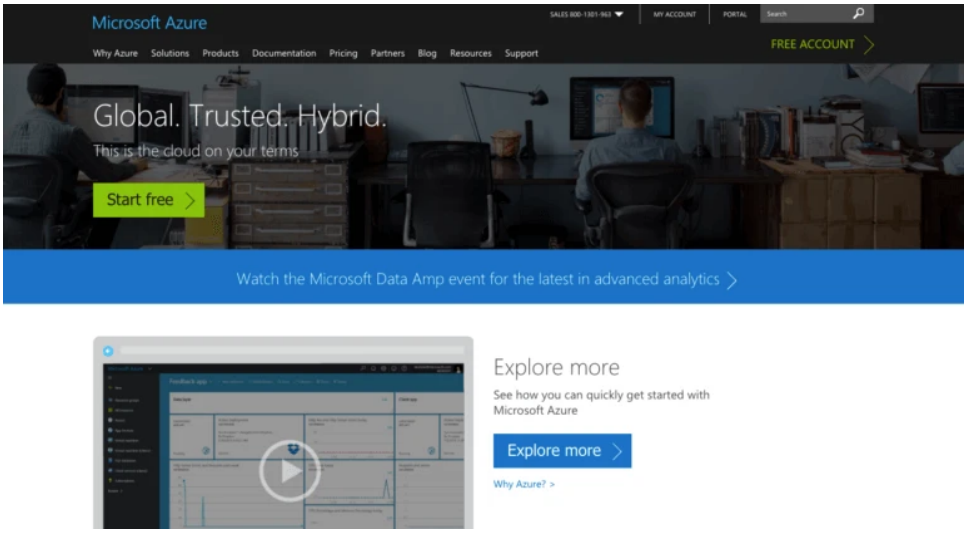



**Figure 25. Azure's Homepage**

**Step 2 –** Another 'Start Free' button.

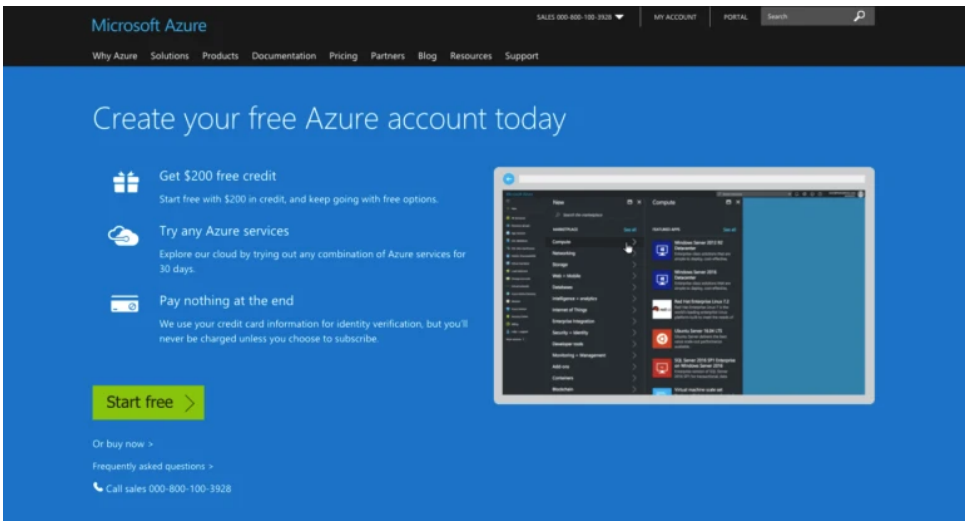



**Figure 26. Get Started with Azure for free**

**Step 3 –** If you have a Microsoft account, you will be prompted to log in.

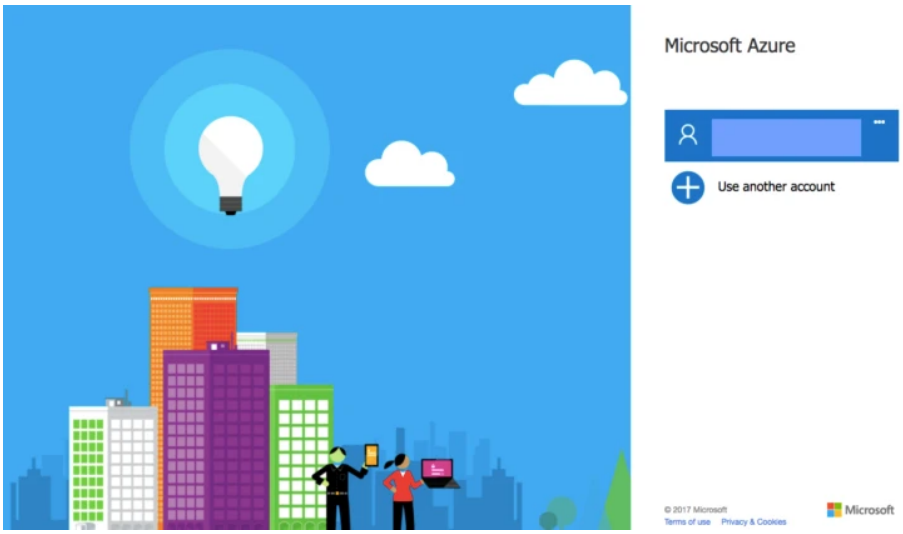



**Figure 27. Login to Azure with the Microsoft Account**

**Step 4 –** When logged in, some of your details might be there, fill in the rest.

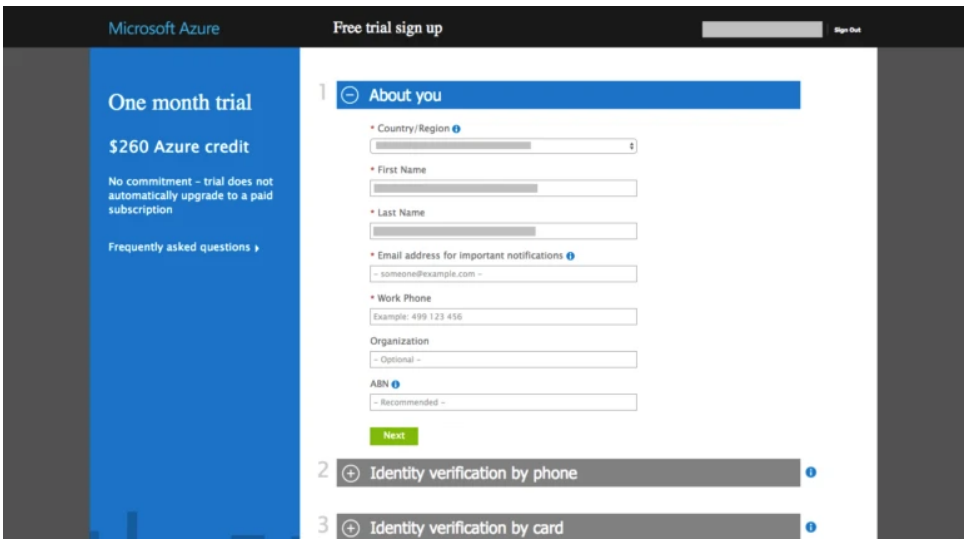



**Figure 28. Fill in additional details for Azure**

**Step 5 –** Follow the instructions to verify your account by phone number.

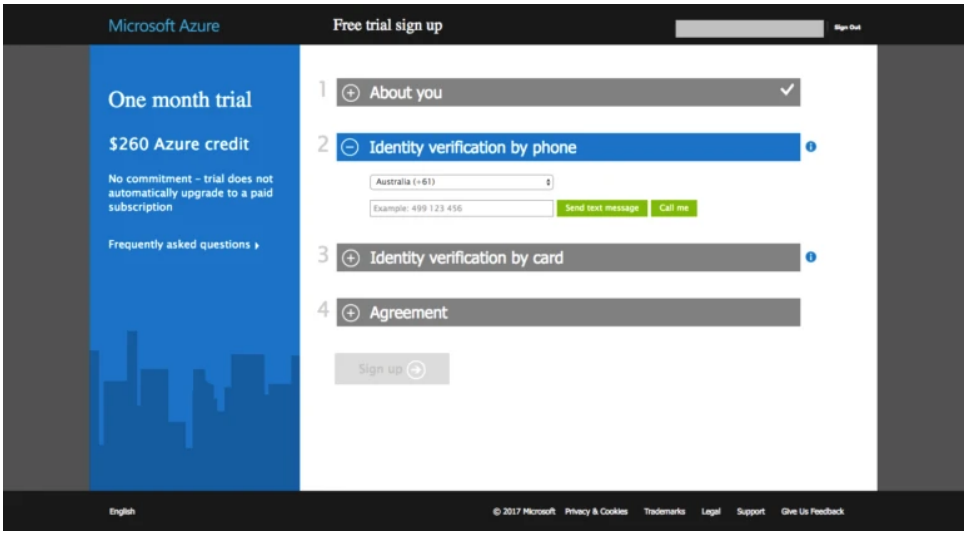



**Figure 29. Azure's identity verification by phone**

**Step 6 –** Read the agreement and tick '*I agree*' then '*Sign Up*' button.

*Within just a few seconds, your account will be ready for work!*

## 2. Installing Requirements And Dependencies

## 2.1. Docker / Docker Desktop

Docker is the modern platform for high-velocity innovation — a tool that is used to automate the deployment of applications in lightweight containers so that applications can work efficiently in different environments.

Either go to the official Docker website and install Docker Desktop or follow the CLI installation.

**Step 1 –** Make sure your system is running on the latest versions (Ubuntu)

```
root@localhost:~# apt-get update -y && apt-get upgrade -y
```

**Step 2 –** Install Docker

```
root@localhost:~# apt install docker.io -y
```

*Install all the dependency packages using this command*

```
root@localhost:~# snap install docker
```

*And, that's it! Let's verify the installation*

```
root@localhost:~# docker --version
Docker version 20.10.12, build e91ed57
```

## 2.2. Minikube

Minikube is a free tool that helps set up single-node Kubernetes clusters on various platforms. It supports many Kubernetes features such as NodePorts, DNS, Container Network Interface, Ingress, ConfigMaps, Secrets, etc.

We are using Minikube to test our Kubernetes manifest files before moving to Azure...

**Step 1 –** Make sure your system is running on the latest versions (Ubuntu)

```
root@localhost:~# apt-get update -y && apt-get upgrade -y
```

Then, download certain packages that would come in handy

```
root@localhost:~# apt-get install curl wget apt-transport-https -y
```

**Step 2 –** Install a Hypervisor

*Minikube supports both VirtualBox hypervisor or KVM*

```
root@localhost:~# apt-get install virtualbox virtualbox-ext-pack
```

**Step 3 –** Install Minikube

*Download the latest version of Minikube with the help of this command:*

```
root@localhost:~# wget https://storage.googleapis.com/minikube/rele
ases/latest/minikube-linux-amd64
```

*Move the downloaded file to /usr/local/bin with the help of this command:*

```
root@localhost:~# mv minikube-linux-amd64 /usr/local/bin/minikube
```

*After that, give execution permission to Minikube.*

```
root@localhost:~# chmod 755 /usr/local/bin/minikube
```

*Then check the version of Minikube using this command:*

```
root@localhost:~# minikube version
minikube version: v0.34.1
```

**Step 4 –** Install Kubectl

*Applications are deployed and managed on Kubernetes with the help of a tool called Kubectl.*

*Download and add the GPG key.*

```
root@localhost:~# curl -s https://packages.cloud.google.com/apt/doc
/apt-key.gpg | apt-key add -
```

*Then, add Kubernetes apt repository with this command:*

```
root@localhost:~# echo "deb http://apt.kubernetes.io/ kubernetes-xe
nial main" | tee /etc/apt/sources.list.d/kubernetes.list
```

*Then, perform an update and install Kubectl*

```
root@localhost:~# apt-get update -y
```

```
root@localhost:~# apt-get install kubectl -y
```

*You can verify the installation by checking Kubectl's version*

```
root@localhost:~# kubectl version -o json
{
  "clientVersion": {
    "major": "1",
    "minor": "13",
    "gitVersion": "v1.13.4",
    "gitCommit": "c27b913fddd1a6c480c229191a087698aa92f0b1",
    "gitTreeState": "clean",
    "buildDate": "2019-02-28T13:37:52Z",
    "goVersion": "go1.11.5",
    "compiler": "gc",
    "platform": "linux/amd64"
  }
}
```

**Step 5** – Start Minikube

*Now, you can start Minikube.*

```
root@localhost:~# minikube start
```

*It will download the VirtualBox image and configure the Kubernetes cluster as mentioned below, allow a moment for it to complete...*

```
o   minikube v0.34.1 on linux (amd64)
>   Creating virtualbox VM (CPUs=2, Memory=2048MB, Disk=20000MB)
...
@   Downloading Minikube ISO ...
 184.30 MB / 184.30 MB
[============================================] 100.00% 0s
-   "minikube" IP address is 192.168.99.100
-   Configuring Docker as the container runtime ...
-   Preparing Kubernetes environment ...
@   Downloading kubeadm v1.13.3
@   Downloading kubelet v1.13.3

-   Pulling images required by Kubernetes v1.13.3 ...
-   Launching Kubernetes v1.13.3 using kubeadm ...
-   Configuring cluster permissions ...
-   Verifying component health .....
+   kubectl is now configured to use "minikube"
=   Done! Thank you for using minikube!
```

*And That's it! You are now having Minikube up and running.*

*Let's check the cluster status...*

```
root@localhost:~# kubectl cluster-info
Kubernetes master is running at https://192.168.99.100:8443
KubeDNS is running at
https://192.168.99.100:8443/api/v1/namespaces/kube-system/services/
kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl
cluster-info dump'.
```

### 2.3. Helm

Helm is a tool for easily installing, publishing, managing, updating, and scaling preconfigured Kubernetes applications. We are using Helm specifically to install Prometheus and Grafana for monitoring...

**Step 1** – Install Helm using Snap.

```
root@localhost:~# snap install helm --classic
```

*Check version to verify the installation*

```
root@localhost:~# helm version
version.BuildInfo{Version:"v3.8.2",
GitCommit:"6e3701edea09e5d55a8ca2aae03a68917630e91b",
GitTreeState:"clean", GoVersion:"go1.17.5"}
```

## 3. Project Skeleton (Microservices Architecture)

| | Docker Image | Deployment Resource | Service Resource | Persistent Volume |
|---|---|---|---|---|
| Frontend Application | ✓ | ✓ | ✓ | ✗ |
| Backend Server | ✓ | ✓ | ✓ | ✗ |
| Go Worker | ✓ | ✓ | ✗ | ✗ |
| Postgres Database | ✗ | ✓ | ✓ | ✓ |
| Redis (PubSub) | ✗ | ✓ | ✓ | ✗ |

Table 2. Project Skeleton

# 4. Dockerfiles

## 4.1. Codes

### 4.1.1. Client's Dockerfile

```
# Build Frontend App
FROM node:16-alpine as build
WORKDIR /usr/app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# Setup Nginx
FROM nginx:1.16.0-alpine
ARG API_HOST
ARG API_PORT
ENV API_HOST=$API_HOST
ENV API_PORT=$API_PORT
RUN rm -rf /usr/share/nginx/html/*
COPY --from=build /usr/app/dist /usr/share/nginx/html
RUN ls /etc/nginx
RUN rm /etc/nginx/conf.d/default.conf
COPY nginx.conf /etc/nginx/conf.d
RUN mv /etc/nginx/conf.d/nginx.conf
/etc/nginx/conf.d/nginx.conf.tmp
COPY docker-entrypoint.sh /
RUN chmod +x /docker-entrypoint.sh
EXPOSE 80
ENTRYPOINT ["/docker-entrypoint.sh"]
```

The Dockerfile depends on an Nginx conf file and a docker-entrypoint.sh Shellscript that sets environment variables for Nginx using "envsubst"

*docker-entrypoint.sh*

```
#!/usr/bin/env sh

set -eu
```

```
envsubst '${API_HOST}:${API_PORT}' <
/etc/nginx/conf.d/nginx.conf.tmp >
/etc/nginx/conf.d/nginx.conf

nginx -g 'daemon off;'
```

*nginx.conf*

```
server {
    listen 80;

    location /api/v1 {
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
        proxy_pass http://${API_HOST}:${API_PORT}/v1;
    }

location / {
        root /usr/share/nginx/html;
        index index.html index.htm;
        try_files $uri $uri/ /index.html;
    }
    error_page 500 502 503 504 /50x.html;

    location = 50x.html {
        root /usr/share/nginx/html;
    }
}
```

### 4.1.2. Backend Server's Dockerfile

```
FROM node:lts-alpine3.15
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build
```

```
CMD [ "npm", "start" ]
```

### 4.1.3. Golang Worker's Dockerfile

```
FROM golang:alpine AS build-env
RUN mkdir /go/src/app && apk update && apk add git
ADD app.go go.mod go.sum /go/src/app/
WORKDIR /go/src/app
RUN go mod download && CGO_ENABLED=0 GOOS=linux go build
-a -installsuffix cgo -ldflags '-extldflags "-static"'
-o app .
FROM scratch
WORKDIR /app
COPY --from=build-env /go/src/app/app .
ENTRYPOINT [ "./app" ]
```

**N.B.** A *.dockerignore* file was included along each Dockerfile to prevent dependencies from getting shipped to DockerHub through the Dockerfiles.

*.dockerignore*

```
node_modules
npm-debug.log
build
.dockerignore
Dockerfile*
docker-compose*
.git
.gitignore
README.md
LICENSE
.vscode
.env
```

## 4.2. Interpretation

What we have done in most of those dockerfiles, is basically installing the necessary dependencies, building the apps for production and running them in the end. To optimize the build we have used **"Alpine"** base images, which are a minimized Linux operating system so the container wouldn't ship with unnecessary features. Each Container we are using a version of an image that's preconfigured with the required programming languages runtimes, like NodeJS and Golang. In the frontend Docker image we have included an Nginx configuration to act as an HTTP Server that will serve our static files. We will pass a couple of arguments to the frontend that represents the hostname and the port of the API that will be used by Nginx to act as a Reverse Proxy so we don't have to ship any environment variables to the static files.

## 4.3. Commands

To build a docker container we use the command:

```
root@localhost:~# sudo docker build -t image-name .
```

where **"-t"** refers to a tag name for the container, and **"."** refers to the path of the dockerfile (current folder in this case).

to run the container we can use this command:

```
root@localhost:~# sudo docker run image-name -p 4000:4000 -d
```

where **"-p"** indicates the port that will be exposed from inside the container, and the port that will bind to it from outside -our machine-. **"-d"** means run the container in detached mode.

## 4.4. DockerHub

We have already pushed our *pickloria-server*, *pickloria-client*, and *pickloria-worker* images for the first time to DockerHub. Afterward, this process will be automatically done by GitLab-CI...

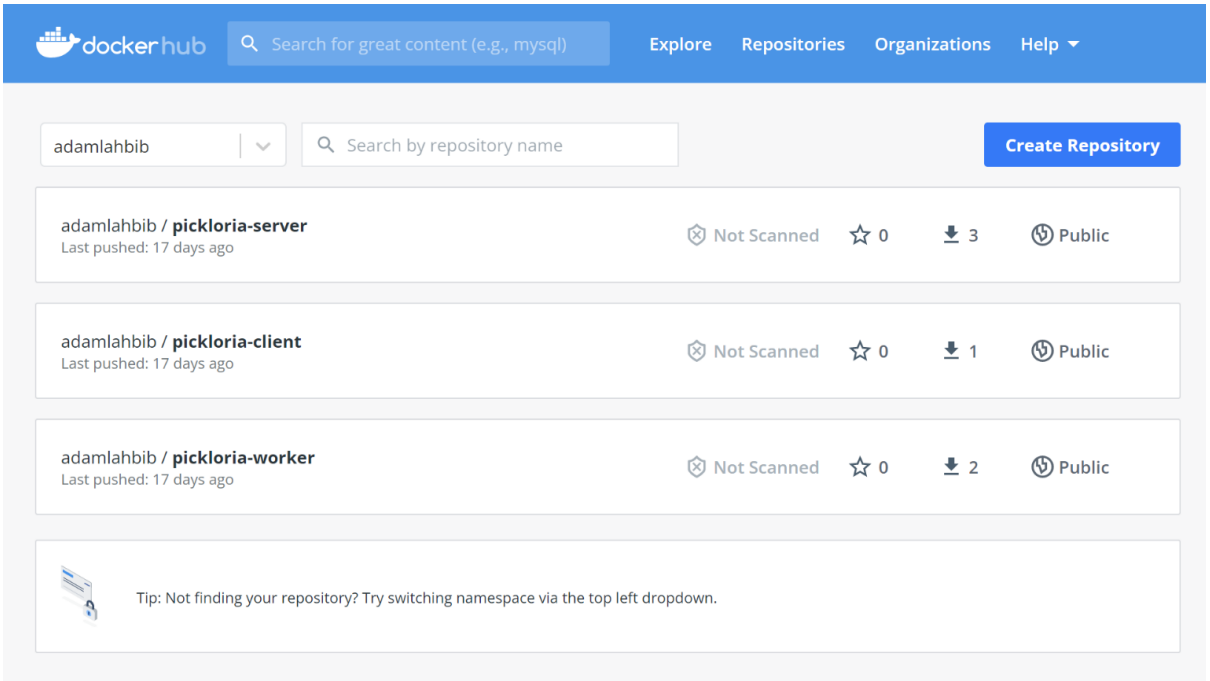



**Figure 30. DockerHub Repositories**

*Let 's see the process.*

First, inside the dashboard, click 'Create Repository' and give it a name. Set visibility to Public if you are in the free tier, like us. Then hit 'Create'

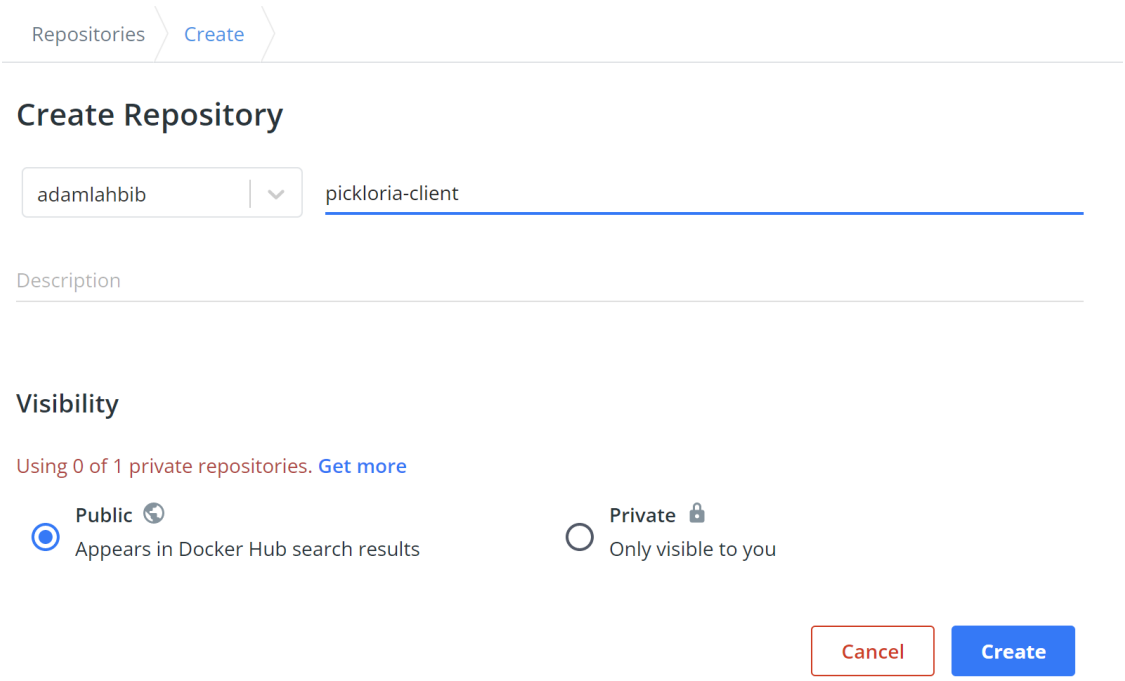
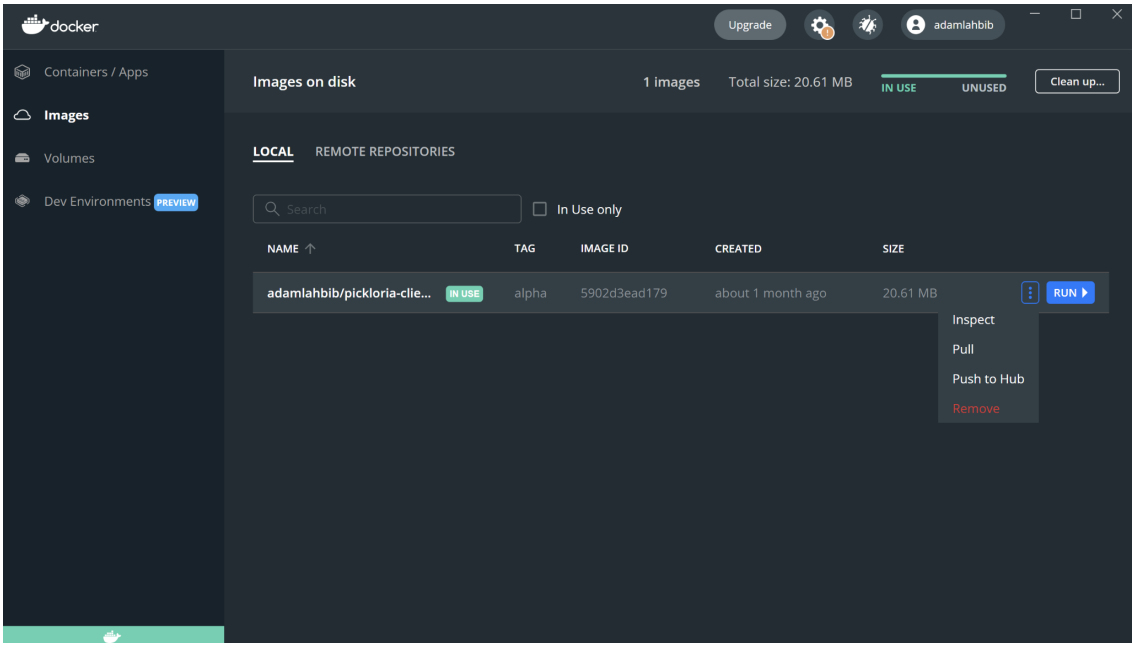



**Figure 31. Creating a DockerHub repository**

Now, make sure you are signed into the Docker Desktop utility. Using Docker Desktop we can push local images, and pull remote images from DockerHub. Building the image will be part of the CLI part next...



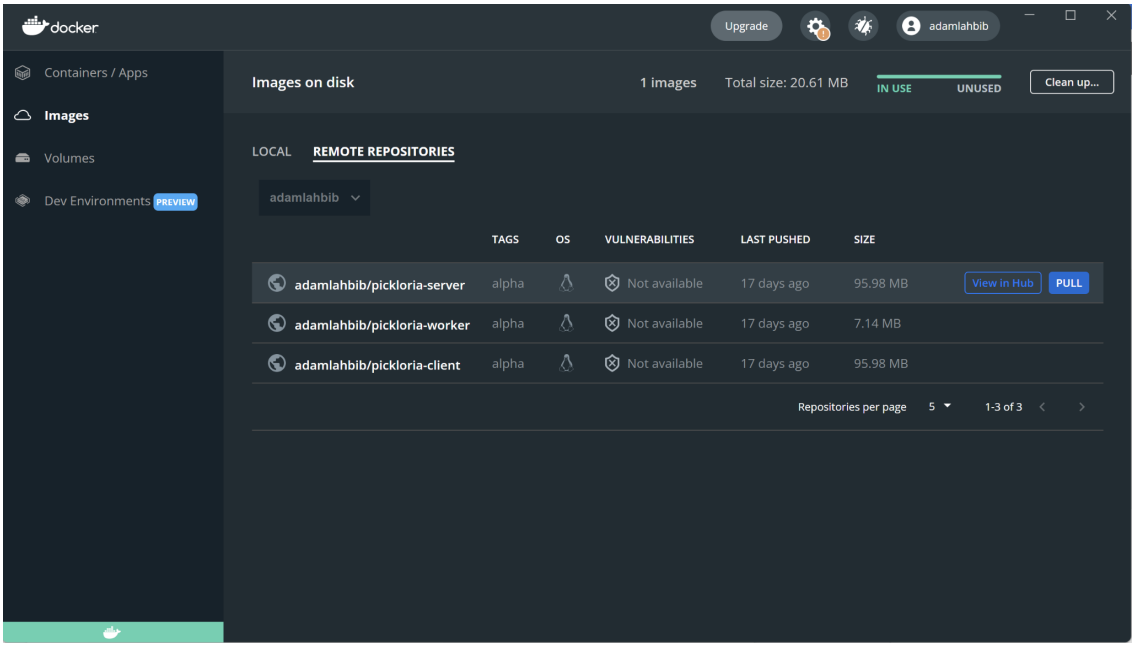



**Figure 32. Docker Desktop**

**The CLI way of doing this is as follows:**

- Signing in

```
root@localhost:~# docker login -u adamlahbib
Password:
Login Succeeded
```

- We prepare our Dockerfile, as shown before, and we land with our terminal over there.

```
root@localhost:~/ppp/insat-rt3-ppp/app/worker (dev)# ls
Dockerfile  app.go  go.mod  go.sum
```

- Building the Image

```
adam@Adams-Vivobook-Pro:~/ppp/insat-rt3-ppp/app/worker (dev)$ docker build -t adamlahbib/pickloria-worker:alpha .
[+] Building 162.4s (13/13) FINISHED
 => [internal] load build definition from Dockerfile                                                                    0.0s
 => => transferring dockerfile: 404B                                                                                    0.0s
 => [internal] load .dockerignore                                                                                       0.0s
 => => transferring context: 44B                                                                                        0.0s
 => [internal] load metadata for docker.io/library/golang:alpine                                                       12.2s
 => [auth] library/golang:pull token for registry-1.docker.io                                                           0.0s
 => [build-env 1/5] FROM docker.io/library/golang:alpine@sha256:4795c5d21f01e0777707ada02408debe77fe31848be97cf9fa8a1462da78d949   102.0s
 => => resolve docker.io/library/golang:alpine@sha256:4795c5d21f01e0777707ada02408debe77fe31848be97cf9fa8a1462da78d949   0.0s
 => => sha256:4795c5d21f01e0777707ada02408debe77fe31848be97cf9fa8a1462da78d949 1.65kB / 1.65kB                          0.0s
 => => sha256:76dfe4aee4c0bf1ecd9666d29d22087eae592f697f449a88c0c7fc81a82faa01 1.36kB / 1.36kB                          0.0s
 => => sha256:5432f005be2d3fc5c462a861b1d75f757c8f73b6d85115b80ccda0f981f040f3 5.24kB / 5.24kB                          0.0s
 => => sha256:2408cc74d12b6cd092bb8b516ba7d5e290f485d3eb9672efc00f0583730179e8 2.80MB / 2.80MB                          3.9s
 => => sha256:ea60b727a1ce729d031ec1e4f2521a246e71bdb72d3fd9c9c04711cce80e1722 271.82kB / 271.82kB                      1.7s
 => => sha256:30c4a77219572e69864cd3069b7b47e55202681be151bd7394f9cb6645b5687b 153B / 153B                              0.8s
 => => sha256:4a5a11f84e29cd3d2ce69b6184f22d8bf628d80f02fd4188925ca2c1212af98a 115.19MB / 115.19MB                     97.4s
 => => sha256:7f0c29549d1e20199a57c8f3f1030e19789b6e035d5779309f3fec5ac13850ce 156B / 156B                              2.3s
 => => extracting sha256:2408cc74d12b6cd092bb8b516ba7d5e290f485d3eb9672efc00f0583730179e8                               0.5s
 => => extracting sha256:ea60b727a1ce729d031ec1e4f2521a246e71bdb72d3fd9c9c04711cce80e1722                               0.1s
 => => extracting sha256:30c4a77219572e69864cd3069b7b47e55202681be151bd7394f9cb6645b5687b                               0.0s
 => => extracting sha256:4a5a11f84e29cd3d2ce69b6184f22d8bf628d80f02fd4188925ca2c1212af98a                               4.2s
 => => extracting sha256:7f0c29549d1e20199a57c8f3f1030e19789b6e035d5779309f3fec5ac13850ce                               0.0s
 => [stage-1 1/2] WORKDIR /app                                                                                          0.0s
 => [internal] load build context                                                                                       0.0s
 => => transferring context: 20.82kB                                                                                    0.0s
 => [build-env 2/5] RUN mkdir /go/src/app && apk update && apk add git                                                 25.6s
 => [build-env 3/5] ADD app.go go.mod go.sum /go/src/app/                                                               0.0s
 => [build-env 4/5] WORKDIR /go/src/app                                                                                 0.0s
 => [build-env 5/5] RUN go mod download && CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -ldflags '-extldflags "-static"' -o app .  22.2s
 => [stage-1 2/2] COPY --from=build-env /go/src/app/app .                                                               0.1s
 => exporting to image                                                                                                  0.1s
 => => exporting layers                                                                                                 0.1s
 => => writing image sha256:0e23594daf3fbf74e7f7db7b60e2b2947734cda423a889239a3d175d75bdcf1d                            0.0s
 => => naming to docker.io/adamlahbib/pickloria-worker:alpha                                                            0.0s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
```

- Pushing the image to DockerHub

```
adam@Adams-Vivobook-Pro:~/ppp/insat-rt3-ppp/app/worker (dev)$ docker push adamlahbib/pickloria-worker:alpha
The push refers to repository [docker.io/adamlahbib/pickloria-worker]
09055c71d9e5: Layer already exists
cc91f937b07f: Layer already exists
alpha: digest: sha256:ecf36ed64156d42e6c841c22822a53deff23e74ec42f7b20ad68d559d5eb463c size: 735
```

In DockerHub, we can verify...

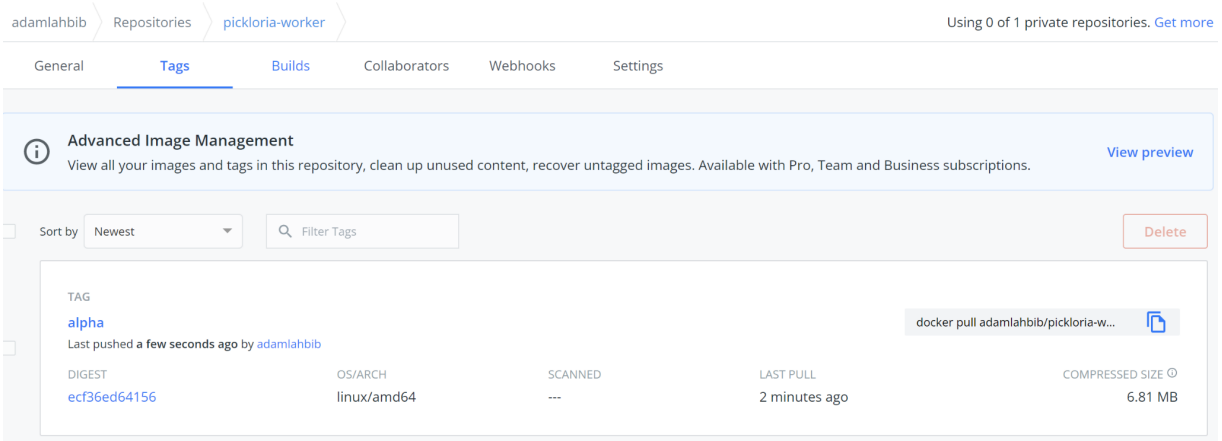



**Figure 33. Image pushed to DockerHub**

- Pulling an image from DockerHub

```
adam@Adams-Vivobook-Pro:~/ppp/insat-rt3-ppp/app/worker (dev)$ docker pull adamlahbib/pickloria-worker:alpha
alpha: Pulling from adamlahbib/pickloria-worker
a4646fb66711: Pull complete
31eaca7d0f23: Pull complete
Digest: sha256:ecf36ed64156d42e6c841c22822a53deff23e74ec42f7b20ad68d559d5eb463c
Status: Downloaded newer image for adamlahbib/pickloria-worker:alpha
docker.io/adamlahbib/pickloria-worker:alpha
```

# 5. CI-CD Pipelines

## 5.1. GitLab CI-CD Variables

CI-CD variables are a type of environment variable. You can use them to control the behavior of jobs and pipelines, and store values you want to re-use secretly. We have filled in the API server, DockerHub credentials, and Snyk private token.

| Type | ↑ Key | Value | Protected | Masked | Environments |
|---|---|---|---|---|---|
| Variable | ADAMS_DOCKER | ********** | ✓ | × | All (default) |
| Variable | ADAMS_DOCKER_PASS | ********** | ✓ | × | All (default) |
| Variable | API_HOST | ********** | ✓ | × | All (default) |
| Variable | API_PORT | ********** | ✓ | × | All (default) |
| Variable | SNYK_TOKEN | ********** | ✓ | × | All (default) |

Add variable  Reveal values

**Figure 34. Adding DockerHub credentials in GitLab CI-CD variables**

## 5.2. Codes

Our project obviously requires three GitLab-CI pipelines. And a main GitLab-CI file that will have all the other files triggered when a modification occurs. For example, adding a new page on the front end will trigger the appropriate GitLab-CI file for the client.

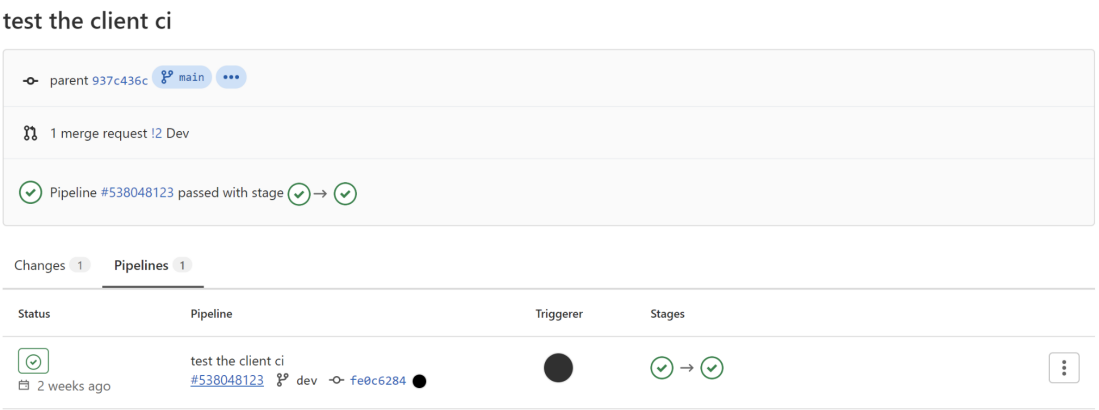
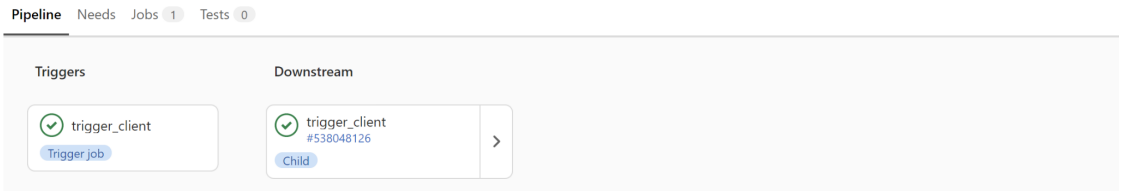



**Figure 35. CI-CD pipeline successfully completed**

*This is how the pipeline got interpreted.*




### 5.2.1. Main GitLab-CI File

*Check Appendix A.*

### 5.2.2. Client's GitLab-CI File

*Check Appendix B.*

### 5.2.3. Server's GitLab-CI File

*Check Appendix C.*

### 5.2.4. Worker's GitLab-CI File

*Check Appendix D.*

## 5.3. Interpretation

In each File we defined a basic 4 steps to build and push our images to the container registry:

- Checkout to the working directory
- Login to Docker CLI

- Build the image
- Push the image to DockerHub

Plus, we added a step to scan our packages for vulnerabilities using Snyk, to be explained later in its section.

The way we organized our pipelines is that if anything changes inside a specific part (client, server, worker), trigger the pipeline living in that folder. else don't, so if a part of the code changes, it will launch the jobs related to that part and does not trigger anything else, we do that to save our resources usage.

| Status | Name | Job | Pipeline | Stage | Duration |
|---|---|---|---|---|---|
| passed | build-and-push-front | #2524451113 dev 43ce53ab | #551594683 by | build | 00:01:47<br>2 days ago |
| passed | test-front-for-vulnerabilities | #2524451112 dev 43ce53ab | #551594683 by | test-for-vulnerabilities | 00:00:48<br>2 days ago |
| passed | build-and-push-front | #2524399393 dev d695a9e1 | #551584708 by | build | 00:01:58<br>2 days ago |
| passed | test-front-for-vulnerabilities | #2524399392 dev d695a9e1 | #551584708 by | test-for-vulnerabilities | 00:00:45<br>2 days ago |
| passed | build-and-push-worker | #2450987412 dev ccf29a47 | #538050902 by | build | 00:02:01<br>2 weeks ago |
| passed | build-and-push-server | #2450987411 dev ccf29a47 | #538050901 by | build | 00:02:03<br>2 weeks ago |
| passed | test-server-for-vulnerabilities | #2450987408 dev ccf29a47 | #538050901 by | test-for-vulnerabilities | 00:00:45<br>2 weeks ago |

**Figure 36. Enabling pipeline status emails integration**

## 5.4. Email Notifications

GitLab provides a built-in plugin (called Integration) to email the pipeline status to a list of recipients.

### Integrations

Integrations enable you to make third-party applications part of your GitLab workflow. If the available integrations don't meet your needs, consider using a webhook.

#### Active integrations

| | Integration | Description | Last updated |
|---|---|---|---|
| ✓ | Pipeline status emails | Email the pipeline status to a list of recipients. | 2 weeks ago |

**Figure 37. Enabling pipeline status emails integration**

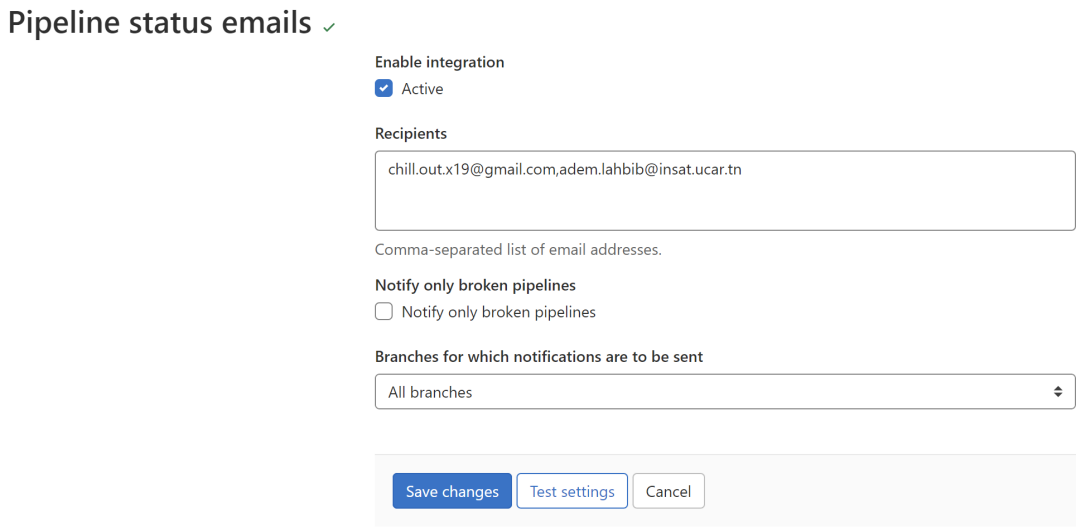



**Figure 38. Adding few emails to the integration**

An example of a received email:

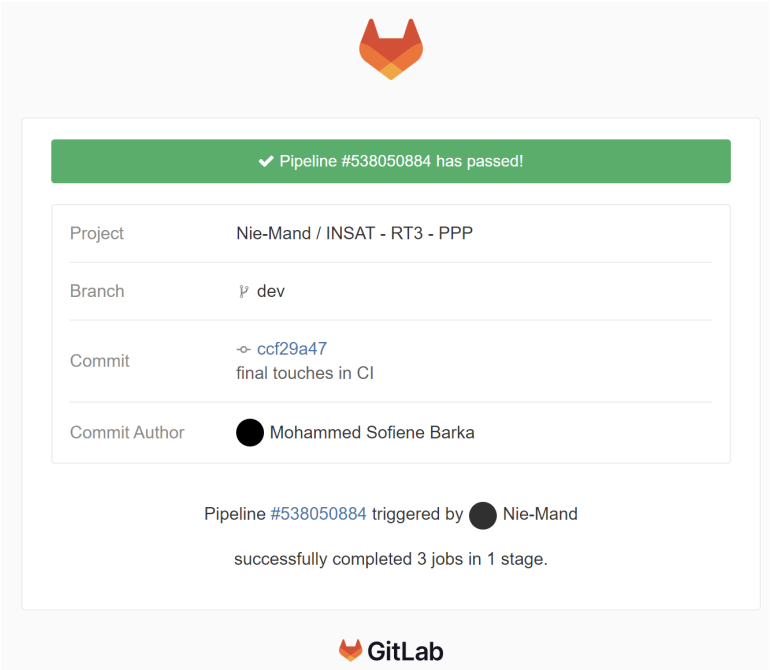



**Figure 39. Pipeline notification email received**

# 6. Kubernetes Manifests

## 6.1. Codes

### 6.1.1. Client's Service And Deployment

*Check Appendix E.*

### 6.1.2. Server's Service And Deployment

*Check Appendix F.*

### 6.1.3. Database's Statefulset And Service

*Check Appendix G.*

### 6.1.4. Worker's Deployment

*Redis Service & Deployment: Check Appendix H.*

*Worker Deployment: Check Appendix I.*

### 6.1.5. Namespace, ConfigMap, And Secrets

*configMap.yml: Check Appendix J-1*

*secrets.yml: Check Appendix J-2*

## 6.2. Interpretation

The **"configMap.yml"** holds the environment variables that we will be using across our manifests. **"secrets.yml"** is equivalent to that but for sensitive data where it will be encoded in base64. The API, Golang worker and redis are deployed as stateless pods. Furthermore, the API is load balanced over 2 replicas. The Database is deployed as a statefulset set, which means its state will be stored in the disk. Each one from the Server, Backend, PubSub and Database will be exposed through a service to ensure the connection between the components.

## 6.3. Monitoring And Observability

**Figure 40. Prometheus & Grafana**

### 6.3.1. Introduction

We wanted to implement some way to monitor our infrastructure's metrics, and how well it's going from a resource perspective, so we are going to use one of the most used monitoring stacks in the infrastructure era, that's not just related to Kubernetes, but also could be implemented in any platform, **"Prometheus"** and **"Grafana"**.

**Prometheus** is how they describe themselves, *an open-source monitoring system with a dimensional data model, flexible query language, efficient time-series database, and modern alerting approach*. So it will be used to collect and store the infrastructure metrics as a form of time series (a famous kind of database), and as its query language is powerful, we want a user-friendly user interface, so we are going to use **Grafana** which is an open source analytics & monitoring solution for every database, or simply a way to display our metrics in form of curves and graphs.

Since setting up each of these components will require more work since they're not just a pod or 2 (they're an entire project that contains a database, a server...) so we are going to use the Kubernetes Package Manager

**"HELM"** to install them with few commands and without writing any line of YAML.

As a first step we need to add both **Prometheus** and **Grafana** to the **Helm** Repository so it can install them later from the right resources.

to keep things simple, we are going to use the **"Port Forward"** Utility provided by **"Minikube"**. It will forward a port to localhost so we don't have to configure a service for it.

```
root@localhost:~# helm repo add prometheus-community
https://prometheus-community.github.io/helm-charts
"prometheus-community" has been added to your
repositories

root@localhost:~# helm repo add grafana
https://grafana.github.io/helm-charts
"grafana" has been added to your repositories
```

### 6.3.2. Setting up Prometheus

We start by installing Prometheus in our K8S namespace that we are working on:

```
root@localhost:~# helm -n ppp install prometheus
prometheus-community/prometheus

NAME: prometheus
LAST DEPLOYED: Tue May 31 23:07:20 2022
NAMESPACE: ppp
STATUS: deployed
```

Next we will retrieve the Prometheus Server Pod ID, then expose a service for it with

```
root@localhost:~# kubectl -n ppp get pods
# prometheus-server-xyz or some random suffix

root@localhost:~# kubectl -n ppp port-forward
prometheus-server-xyz 9090
```

Now if we visit **"localhost:9090"** we can see that our Prometheus Server is running:

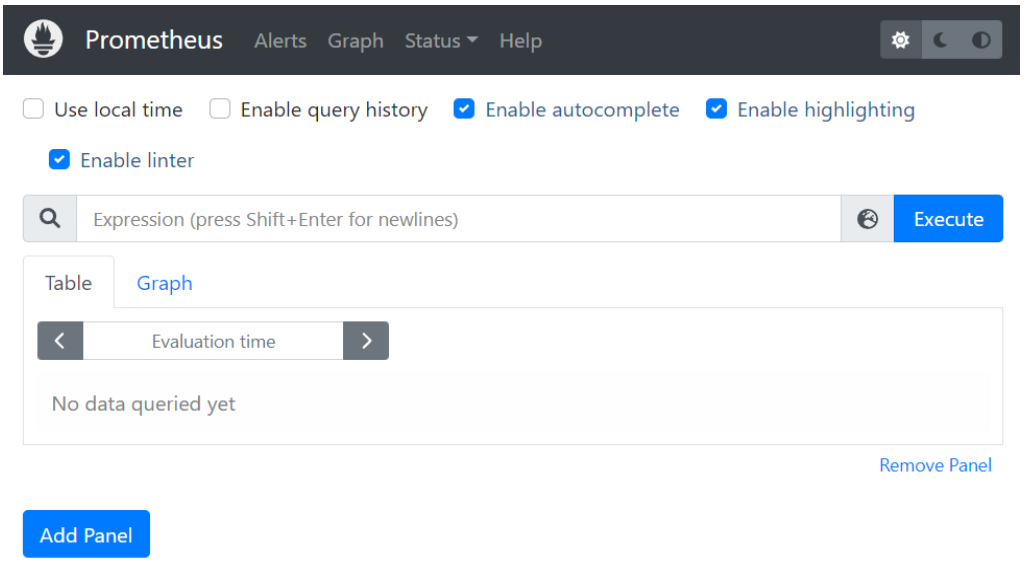



**Figure 41. Prometheus Homepage**

### 6.3.3. Setting up Grafana

Now, let's get some useful graphs out of what Prometheus can provide, let's start by installing Grafana and expose a service for it just like what we did with Prometheus. We also need to get the login password since Grafana will auto-create us with an account. The username is **"admin"**

```
# install grafana
root@localhost:~# helm -n ppp install grafana
grafana/grafana
NAME: grafana
LAST DEPLOYED: Tue May 31 23:22:59 2022
NAMESPACE: ppp
```

```
STATUS: deployed
REVISION: 1

# install grafana
root@localhost:~# kubectl -n ppp get pods
# grafana-xyz or some random suffix

# expose it to port 8080
root@localhost:~# kubectl -n ppp port-forward
grafana-xyz 8080:3000


# receive the login password
root@localhost:~# kubectl -n ppp get secret grafana -o
jsonpath="{.data.admin-password}" | base64 --decode ;
echo
AiUmbun9TFeiwM28C3iefmQxXhJcwT5qpmVq49NP
```

Now If we visit "localhost:8080" we will find a login page for Grafana:

**Figure 42. Welcome to Grafana**

This is the interface that we will come across once we're logged in:

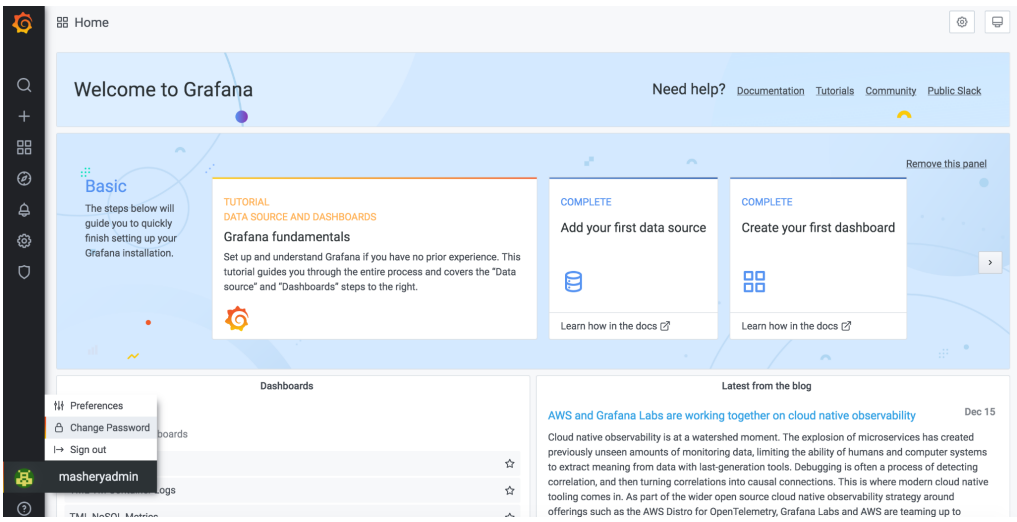
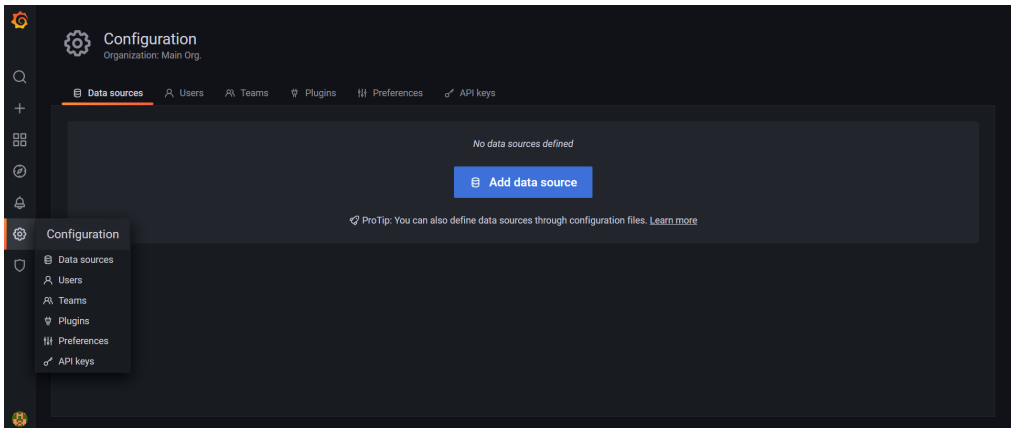



**Figure 43. Grafana Homepage**

The First step is to tell Grafana to get data from Prometheus by defining a Data Source, We click on the "Add your first data source" button, which will take us to this page:

We choose "Prometheus" as it is the first option we will get, then choose the service name and the assigned port by the K8S Cluster itself, you can get them by running **"kubectl -n ppp get svc"**. we next select the dashboards that we want to include from the **"Dashboard"** Tab.



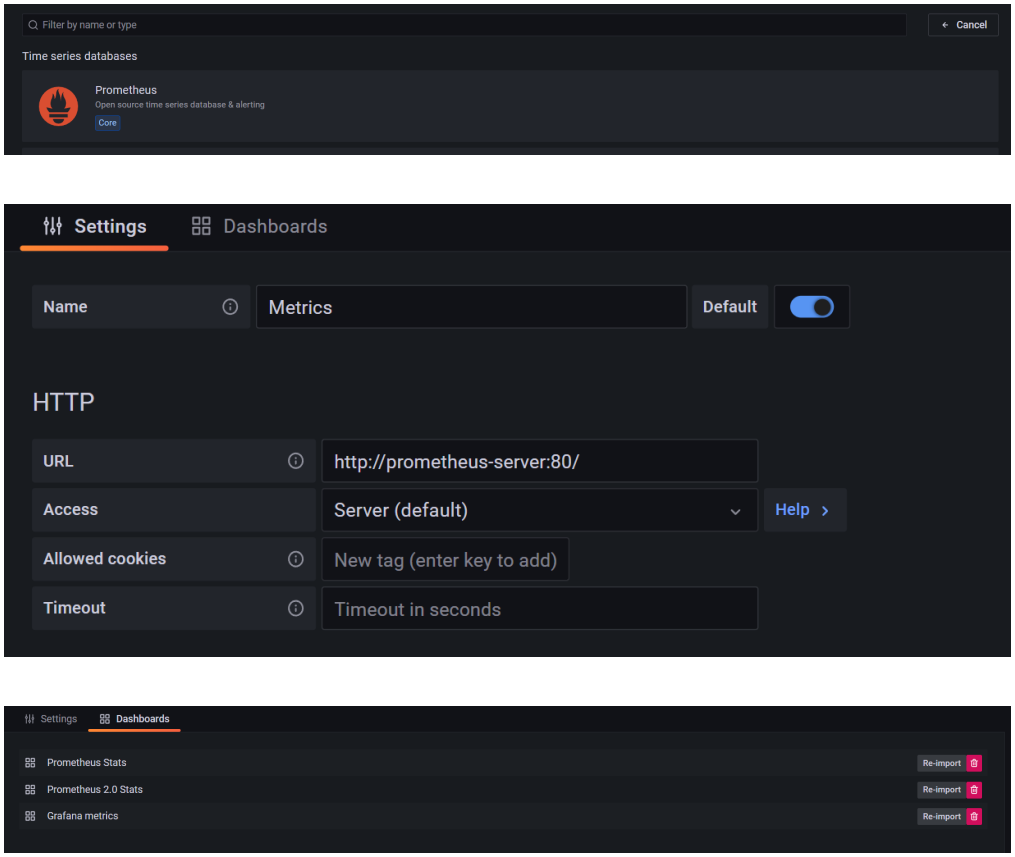
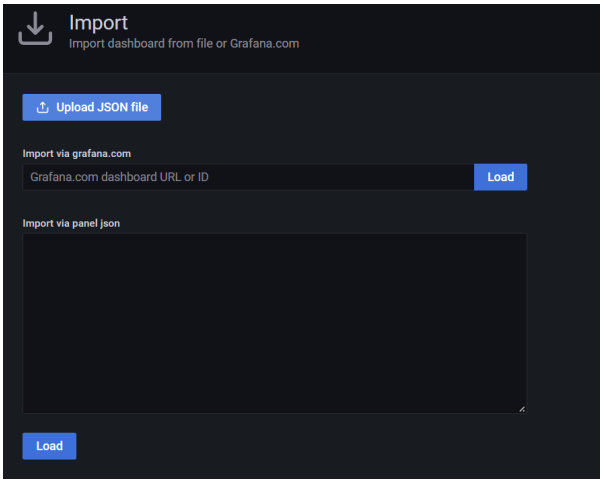


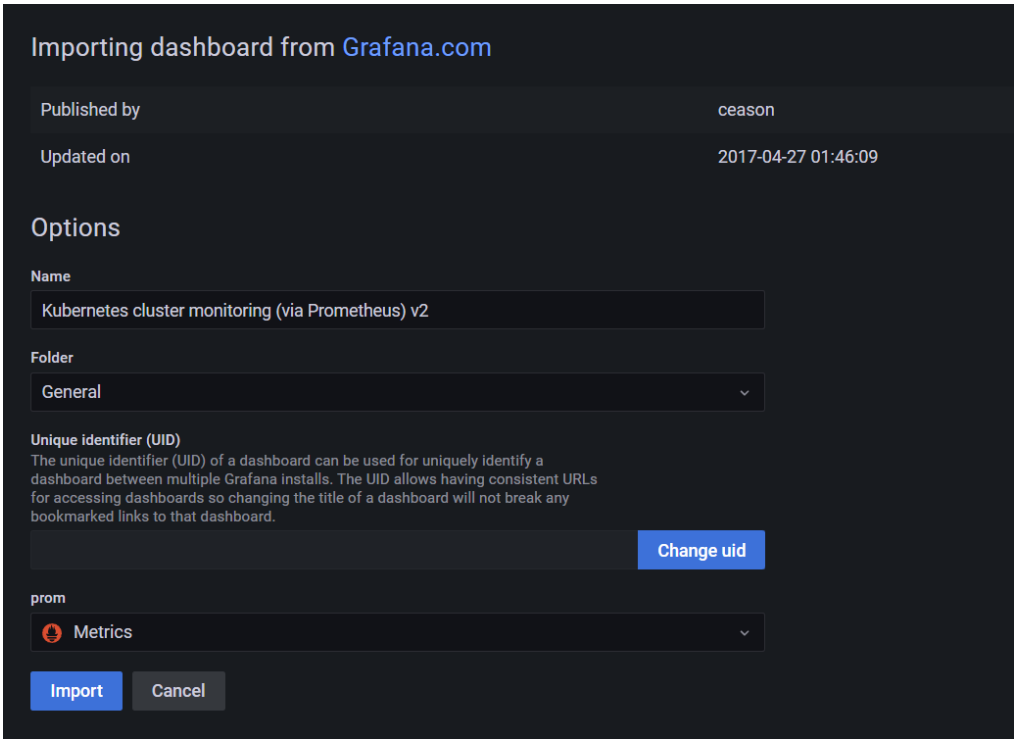

**Figure 44. Adding Prometheus as a data source for Grafana**

Once we agree and save the configuration, we should be able to get this Statistics panel (This is the **"Prometheus 2.0 Stats"** Dashboard):

**Figure 45. Prometheus 2.0 Stats Dashboard**

What is great about Grafana, it could be assigned more dashboards from its library, each identified by an ID. We will add a new one by going to **"Import Dashboard"** and using the ID **"2115"**. and this is the new dashboard that we will end up with:



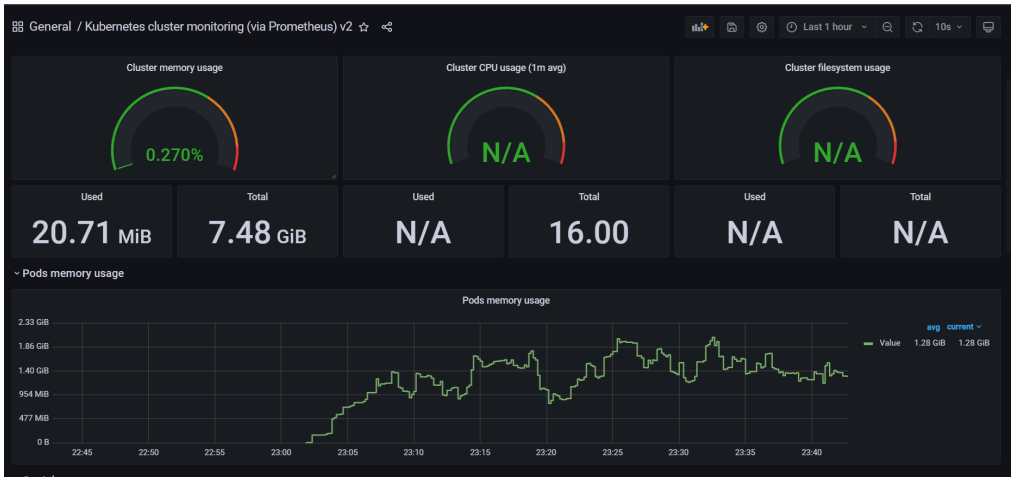



**Figure 46. Assigning a custom dashboard to Grafana**

### 6.3.4. Conclusion

We Can extend those graphs to many available dashboards out there. Monitoring the metrics is an important aspect of DevOps engineering because it gives you observability over all the infrastructure. We can expand the stack with additional components such as **"Loki"** or **"Fluentd"** so we can get Logs and monitor more aspects of the system. An Alternative to what we have used so far is the **"ELK Stack". (Elastic Search, Logstash, Kibana)**

## 6.4. DevSecOps With Snyk

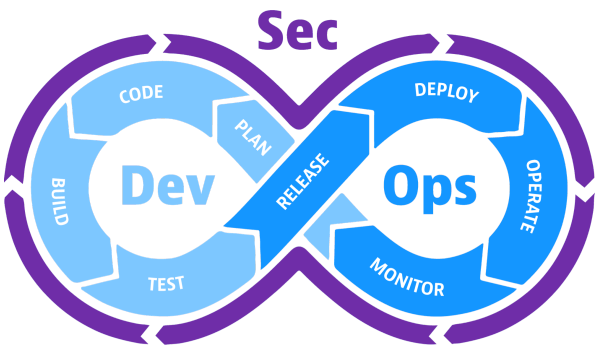


**Figure 47. DevSecOps with Snyk**

Security is the most important aspect of the industry in general. DevSecOps is the matter of implementing some procedures in the DevOps workflow to guarantee some (as much as possible) Security features and tests in the system. Since it is considered a different subject and an extension to DevOps that will require an entire experiment to talk about, we only realized one feature which is scanning our project packages and dependencies for known vulnerabilities.

**"Snyk"**, is a tool, relying on a huge database of dependencies and their known vulnerabilities, that will check your project for packages that have been recognized to have a security flaw. It supports many programming languages without the need for a plugin or additional setup. The process is extremely easy, and it will only require installing **Snyk** with **NodeJS**, registering for an account, registering a Token and authenticating with it from the command line, and eventually, running the **"snyk scan"** command. This is an example of an additional CI Job that we have added in our backend pipeline that will run the scan before executing the build Job (If the first job fails, the build won't happen)

```
test-server-for-vulnerabilities:
  only:
    - dev
  image: node
  stage: test-for-vulnerabilities
  before_script:
    - npm i -g snyk
    - snyk config set api=$SNYK_TOKEN
  script:
    - cd app/server
    - snyk test
```

It might not be that much of a deal, but it's a start to secure our app. The next step would be working on securing the infrastructure, or the Kubernetes Cluster, but we kept on using Snyk for this project.

# 7. Azure

## 7.1. Cloud-Native

Cloud-native refers to the concept of building and running applications to take advantage of the distributed computing offered by the cloud provider. Cloud-native apps are designed and built to exploit the scale, elasticity, resiliency, and flexibility the cloud provides.

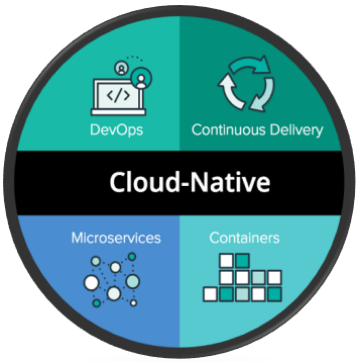



**Figure 48. Cloud-Native**

## 7.2. Cloud Providers

A cloud service provider is a third-party company offering a cloud-based platform, infrastructure, application, or storage services. Much like a homeowner would pay for a utility such as electricity or gas, companies typically have to pay only for the number of cloud services they use, as business demands require.

Figure 49. Popular cloud providers

## 7.3. Managed Kubernetes

Managed Kubernetes is when third-party providers take over responsibility for some or all of the work necessary for the successful set-up and operation of K8s. Depending on the vendor, "managed" can refer to anything from dedicated support, to hosting with pre-configured environments, to full hosting and operation. Examples: AKS, EKS, and GKE...

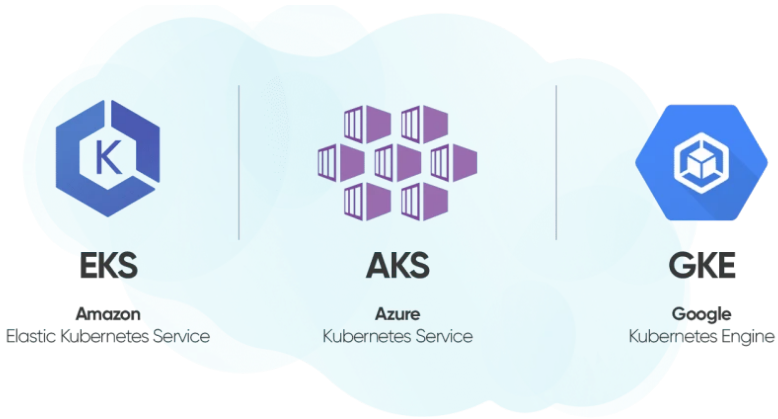



Figure 50. Managed Kubernets

## 7.4. Creating AKS (Azure Kubernetes Service) Cluster with AZ-CLI

To create any resource in azure you must create a resource group and give it a name and specify the location of the resource created so we will create some variables that we will need to use

```
export RESOURCE_GROUP=ppp-rt3
export CLUSTER_NAME=ppp-rt3-cluster
export LOCATION=USEAST2
```

Use `*az group create command*` to create a resource group. we will deploy all resources into this new resources group.

```
az group create --name=$RESOURCE_GROUP --location=$LOCATION
```

after creating the resource group and specify the location , we can now create our AKS cluster with the az aks create command

```
az aks create \
    --resource-group $RESOURCE_GROUP \
    --name $CLUSTER_NAME \
    --node-count 2 \
    --enable-addons http_application_routing \
    --generate-ssh-keys \
    --node-vm-size Standard_B2s
```

The above command creates a new AKS cluster named ppp-rt3-cluster within the ppp-rt3 resource group. The cluster will have two nodes defined by the --node-count parameter. We're using only two nodes here for cost considerations. The --node-vm-size parameter configures the cluster nodes as Standard_B2s-sized VMs. The HTTP application routing add-on is enabled via the *--enable-addons* flag. These nodes will be part of System mode.

In case the nodepool is not enough for your cluster and you want to add more nodes in your cluster you can add a new nodepool with *az aks nodepool add* command

```
az aks nodepool add \
```

```
    --resource-group $RESOURCE_GROUP \
    --cluster-name $CLUSTER_NAME \
    --name userpool \
    --node-count 2 \
    --node-vm-size Standard_B2s
```

The above command adds a new node pool to an existing AKS cluster (created in the previous command). This new node pool can be used to host applications and workloads. Now we must link our Kubernetes cluster with Kubectl by running the following command in Cloud Shell

```
az aks get-credentials --name $CLUSTER_NAME --resource-group
$RESOURCE_GROUP
```

This command will add an entry to your *~/.kube/config* file, which holds all the information to access your clusters. Kubectl enables you to manage multiple clusters from a single command-line interface.

## 7.5. Implementation of the infrastructure

First we start creating the secrets and confimap objects

```
mohamed@home$ kubectl apply -f configMap.yaml
mohamed@home$ kubectl apply -f secrets.yaml
```

Before we create the database statefulset we must create a dynamic persistent volume claim to create an azure disk to mount the volume of the database container

```
mohamed@home$ kubectl apply -f dynamic-azure-disk.yaml
mohamed@home$ kubectl apply -f postgres-sts.yaml
statefulset.apps/postgres-sts created
service/db-service created
service/db-svc created
```

After Creating the database. Let's create the redis deployment

```
mohamed@home$ kubectl apply -f redis-dep.yaml
deployments.apps/redis-dep created
service/redis-service created
```

Creating the worker

```
mohamed@home$ kubectl apply -f worker-dep.yaml
deployments.apps/worker-dep created
```

After Creating the database, redis and the worker make sure that everything is ok. we must create the server and client deployment and their service. Especially for the client we will create a load balancer service to make a public IP address

```
mohamed@home$ kubectl apply -f server-dep.yaml
deployments.apps/server-dep created
mohamed@home$ kubectl apply -f server-service.yaml
service/server-svc created
mohamed@home$ kubectl apply -f client-dep.yaml
deployments.apps/server-dep created
mohamed@home$ kubectl apply -f client-service.yaml
service/client-svc created
```

```
mohamedrafraf@Azure:~$ kubectl get all
NAME                                    READY   STATUS    RESTARTS      AGE
pod/api-5f74d6698c-7vgjs                1/1     Running   4 (38h ago)   39h
pod/client-deployment-5b4656c84-z5kqk   1/1     Running   0             37h
pod/postgres-sts-0                      1/1     Running   0             39h
pod/redis-pubsub-5856f664c4-8czb2       1/1     Running   0             39h
pod/worker-699b4d84d4-g26s4             1/1     Running   0             39h

NAME                        TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)        AGE
service/api-svc             ClusterIP      10.0.182.110   <none>          5050/TCP       39h
service/client-service      LoadBalancer   10.0.50.103    20.102.25.119   80:31865/TCP   38h
service/db-svc              ClusterIP      10.0.23.46     <none>          5432/TCP       39h
service/kubernetes          ClusterIP      10.0.0.1       <none>          443/TCP        40h
service/postgres-headless   ClusterIP      None           <none>          5432/TCP       39h
service/redis-service       ClusterIP      10.0.121.82    <none>          6379/TCP       39h

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/api                 1/1     1            1           39h
deployment.apps/client-deployment   1/1     1            1           37h
deployment.apps/redis-pubsub        1/1     1            1           39h
deployment.apps/worker              1/1     1            1           39h

NAME                                           DESIRED   CURRENT   READY   AGE
replicaset.apps/api-5f74d6698c                 1         1         1       39h
replicaset.apps/client-deployment-5b4656c84    1         1         1       37h
replicaset.apps/redis-pubsub-5856f664c4        1         1         1       39h
replicaset.apps/worker-699b4d84d4              1         1         1       39h

NAME                             READY   AGE
statefulset.apps/postgres-sts    1/1     39h
```

# 8. Conclusion

After what we have done so far, we have successfully deployed, what it could be our scalable software, architectured in microservices, fully containerized, fully cloud native, on one of the most advanced Container Orchestration tools in the world, Kubernetes, on Microsoft Azure Cloud.

# FINAL CONCLUSION AND FUTURE SCOPE

The project and research theoretically propose and empirically validate a Microservice web app architecture that links DevOps and Kubernetes to multiple aspects of the app like storage, pods, services, and deployments.

The work was successfully designed, implemented, and tested. DevOps is a blend of development and operations, but we see it as a magnificent philosophy; a software development methodology that escalates to the amalgamation between software developers and IT operation professionals. It focuses mainly on delivering software products faster and reducing the failure rate of releases to make the product efficient. This is quite helpful for the developers or testers in need to fix bugs rapidly or to add extra features to the product according to specified requirements.

At present, DevOps is kind of neglected, yet it is gathering more and more attention. Indeed, it is the most advanced approach in the IT industry than the waterfall model and the agile model. This whole system was extendable by onboarding the complete project into cloud services like Microsoft Azure to improve efficiency.

A Kubernetes Manifest files are the heart of Kubernetes. These plaintext configuration files describe how a pod's containers should be run and connected to other objects, such as services or replication controllers. These Yaml files will communicate with the AKS (Azure Kubernetes Service).

The Kubernetes ecosystem is growing so fast that it is almost impossible to keep pace. Even if we could somehow learn everything there is to know today, we would be left behind a week later.

We do recommend reading Mr. Jeff Geerling and Mrs. Katherine Geerling' book "Kubernetes 101", and following the rest of Anais Urlichs' #100DaysOfKubernetes series, we think these are the keys to continuing the journey after finishing this report. If you are looking for something more advanced, head to "The Kubernetes Bible". We wish you the best of luck!

*Our project is open-source and can be enhanced in the future, you can check it by following this GitLab link 🔗, please share this report if you happen to know anybody with a similar project in mind. It might be a good help!*

*Mr. Mohamed Sofiene Barka, Mr. Mohamed Rafraf, and Mr. Adam Lahbib are all, at the time of writing, undergraduate students in the first-year networking and telecommunication engineering class at the National Institute of Applied Science and Technology, Tunis. We are keen on everything Web, DevOps, Kubernetes, and Security.*

*www.linkedin.com/in/nie-mand*

*www.linkedin.com/in/mohamed-rafraf*

*www.linkedin.com/in/adamlahbib*

*Thanks again!*

*May God bless you and us with happiness, success, guidance, and knowledge!*

*With Love, The Authors.*

# REFERENCES & WEBOGRAPHY

https://en.wikipedia.org/w/index.php?title=DevOps&oldid=797774921

https://www.redhat.com/en/topics/devops/what-is-ci-cd

https://www.alibabacloud.com/knowledge/what-is-containerization

https://azure.microsoft.com/en-us/overview/what-is-cloud-computing/

https://www.atlassian.com/devops

https://www.atlassian.com/microservices

https://prometheus.io

https://grafana.com/

https://fireship.io/

https://medium.com/@kavinduchamiran

https://100daysofkubernetes.io/

https://en.wikipedia.org/wiki/Virtual_machine

https://www.docker.com/

https://www.cs.umd.edu/class/winter2021/cmsc388T/lectures/lec8.pdf

https://nickjanetakis.com/blog/differences-between-a-dockerfile-docker-image-and-docker-container

https://kube101.jeffgeerling.com

# APPENDICES

## APPENDIX A

*Main GitLab-CI Yaml File*

```
stages:
  - triggers

trigger_server:
  stage: triggers
  trigger:
    include: app/server/.gitlab-ci.yml
  rules:
    - changes:
        - app/server/*

trigger_client:
  stage: triggers
  trigger:
    include: app/client/.gitlab-ci.yml
  rules:
    - changes:
        - app/client/*


trigger_worker:
  stage: triggers
  trigger:
    include: app/worker/.gitlab-ci.yml
  rules:
    - changes:
        - app/worker/*
```

## APPENDIX B

*Client's GitLab-CI Yaml File*

```
stages:
  - test-for-vulnerabilities
  - build

variables:
  CONTAINER_NAME:
adamlahbib/pickloria-client:alpha

test-front-for-vulnerabilities:
  only:
    - dev
  image: node
  stage: test-for-vulnerabilities
  before_script:
    - npm i -g snyk
    - snyk config set api=$SNYK_TOKEN
  script:
    - cd app/client
    - snyk test

build-and-push-front:
  only:
    - dev
  image: docker
  services:
    - docker:dind
  stage: build
  before_script:
    - echo ${ADAMS_DOCKER_PASS} | docker
login -u ${ADAMS_DOCKER}
--password-stdin
  script:
    - cd app/client
```

```
    - docker build -t ${CONTAINER_NAME} .
    - docker push ${CONTAINER_NAME}
```

# APPENDIX C

*Server's GitLab-CI Yaml File*

```
stages:
  - test-for-vulnerabilities
  - build

variables:
  CONTAINER_NAME:
adamlahbib/pickloria-server:alpha

test-server-for-vulnerabilities:
  only:
    - dev
  image: node
  stage: test-for-vulnerabilities
  before_script:
    - npm i -g snyk
    - snyk config set api=$SNYK_TOKEN
  script:
    - cd app/server
    - snyk test

build-and-push-server:
  only:
    - dev
  image: docker
  services:
    - docker:dind
  stage: build
  before_script:
    - echo ${ADAMS_DOCKER_PASS} | docker
login -u ${ADAMS_DOCKER}
--password-stdin
  script:
    - cd app/server
    - docker build -t ${CONTAINER_NAME} .
    - docker push ${CONTAINER_NAME}
```

# APPENDIX D

*Workers's GitLab-CI Yaml File*

```
stages:
  - build

variables:
  CONTAINER_NAME:
adamlahbib/pickloria-worker:alpha

build-and-push-worker:
  only:
    - dev
  image: docker
  services:
    - docker:dind
  stage: build
  before_script:
    - echo ${ADAMS_DOCKER_PASS} | docker
login -u ${ADAMS_DOCKER}
--password-stdin
  script:
    - cd app/worker
    - docker build -t ${CONTAINER_NAME} .
    - docker push ${CONTAINER_NAME}
```

# APPENDIX E

*Client Service & Deployment Manifests*

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: client-deployment
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: pickloria-client
    spec:
      containers:
        - name: pickloria-client
          image:
adamlahbib/pickloria-client:alpha
          imagePullPolicy: Always
          ports:
            - containerPort: 80
              name: http-port
          env:
            - name: API_HOST
              value: "api-svc"
            - name: API_PORT
              value: "5050"
  selector:
    matchLabels:
      app: pickloria-client
---
apiVersion: v1
kind: Service
metadata:
  name: client-service
spec:
  type: LoadBalancer
  ports:
    - port: 80
      targetPort: 80
      protocol: TCP
```

# APPENDIX F

*Server Service & Deployment Manifests*

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api
  labels:
    app: api
spec:
  replicas: 2
  selector:
    matchLabels:
      app: api
  template:
    metadata:
      labels:
        app: api
    spec:
      containers:
        - name: api
          image:
adamlahbib/pickloria-server:alpha
          imagePullPolicy: Always
          ports:
            - containerPort: 5050
          env:
            - name: AWS_ACCESS_KEY_ID
              valueFrom:
                secretKeyRef:
                  name: ppp-secrets
                  key: aws-access-key-id
            - name:
AWS_SECRET_ACCESS_KEY
              valueFrom:
                secretKeyRef:
                  name: ppp-secrets
                  key:
aws-secret-access-key
            - name: AWS_REGION
              valueFrom:
```

```yaml
    name: http
  selector:
    app: pickloria-client
```

```yaml
      secretKeyRef:
        name: ppp-secrets
        key: aws-region
    - name: DATABASE_USERNAME
      valueFrom:
        secretKeyRef:
          name: ppp-secrets
          key: db-username
    - name: DATABASE_PASSWORD
      valueFrom:
        secretKeyRef:
          name: ppp-secrets
          key: db-password
    - name: DATABASE_HOST
      valueFrom:
        configMapKeyRef:
          name: ppp-config-map
          key: db-host
    - name: DATABASE_PORT
      valueFrom:
        configMapKeyRef:
          name: ppp-config-map
          key: db-port
    - name: DATABASE_NAME
      valueFrom:
        configMapKeyRef:
          name: ppp-config-map
          key: db-name
    - name: REDIS_HOST
      valueFrom:
        configMapKeyRef:
          name: ppp-config-map
          key: redis-host
    - name: REDIS_PORT
      valueFrom:
        configMapKeyRef:
          name: ppp-config-map
          key: redis-port

---
apiVersion: v1
kind: Service
metadata:
  name: api-svc
spec:
```

```
  selector:
    app: api
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 5050
      targetPort: 5050
```

## APPENDIX G

*Postgres Statefulset & Service Manifests*

```
apiVersion: v1
kind: Service
metadata:
  name: postgres-headless
  labels:
    app: postgres-sts
spec:
  ports:
    - port: 5432
      name: postgres
      targetPort: 5432
  clusterIP: None
  selector:
    app: postgres-sts
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: postgres-sts
spec:
  serviceName: "postgres-headless"
  replicas: 1
  selector:
    matchLabels:
      app: postgres-sts
  template:
    metadata:
      labels:
        app: postgres-sts
    spec:
```

## APPENDIX H

*Redis Service & Deployment Manifests*

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-pubsub
  labels:
    app: redis
spec:
  selector:
    matchLabels:
      app: redis
  replicas: 1
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
        - name: pubsub
          image: redis
          resources:
            requests:
              cpu: 100m
              memory: 100Mi
          ports:
            - containerPort: 6379
---
apiVersion: v1
kind: Service
metadata:
  name: redis-service
```

```
      initContainers:
        - name: init
          image: alpine
          command: ["sh", "-c", "chown
999:999 /var/lib/postgresql/data"]
          volumeMounts:
            - mountPath:
/var/lib/postgresql/data
              name: ppp-postgres
      containers:
        - name: postgres
          image: postgres
          securityContext:
            runAsUser: 999
          ports:
            - containerPort: 5432
          env:
            - name: POSTGRES_USER
              valueFrom:
                secretKeyRef:
                  name: ppp-secrets
                  key: db-username
            - name: POSTGRES_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: ppp-secrets
                  key: db-password
            - name: POSTGRES_HOST
              valueFrom:
                configMapKeyRef:
                  name: ppp-config-map
                  key: db-host
            - name: POSTGRES_DBNAME
              valueFrom:
                configMapKeyRef:
                  name: ppp-config-map
                  key: db-name
            - name: PGDATA
              value:
/var/lib/postgresql/data/pgdata
          volumeMounts:
            - name: ppp-postgres
              #  subPath: pgdata
              mountPath:
/var/lib/postgresql/data/
  labels:
    app: redis
spec:
  ports:
    - port: 6379
      targetPort: 6379
  selector:
    app: redis
```

```
      volumes:
        - name: ppp-postgres
          persistentVolumeClaim:
            claimName:
azure-managed-disk
---
apiVersion: v1
kind: Service
metadata:
  name: db-svc
spec:
  selector:
    app: postgres-sts
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 5432
      targetPort: 5432
```

# APPENDIX I

*Worker's Deployment Manifest*

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: worker
  labels:
    app: worker
spec:
  replicas: 1
  selector:
    matchLabels:
      app: worker
  template:
    metadata:
      labels:
        app: worker
    spec:
      containers:
        - name: worker
          image:
```

# APPENDIX J

*ConfigMap & Secrets Yml File*

```
#J-1 ConfigMap

apiVersion: v1
kind: ConfigMap
metadata:
  name: ppp-config-map
data:
  redis-host: redis-service
  redis-port: "6379"
  api-host: api-svc
  api-port: "8080"
  db-host: db-svc
  db-port: "5432"
  db-name: mohamed

#J-2 Secrets Template

apiVersion: v1
kind: Secret
```

```
adamlahbib/pickloria-worker:alpha
          imagePullPolicy: Always
          env:
            - name: DATABASE_USERNAME
              valueFrom:
                secretKeyRef:
                  name: ppp-secrets
                  key: db-username
            - name: DATABASE_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: ppp-secrets
                  key: db-password
            - name: DATABASE_HOST
              valueFrom:
                configMapKeyRef:
                  name: ppp-config-map
                  key: db-host
            - name: DATABASE_PORT
              valueFrom:
                configMapKeyRef:
                  name: ppp-config-map
                  key: db-port
            - name: DATABASE_NAME
              valueFrom:
                configMapKeyRef:
                  name: ppp-config-map
                  key: db-name
            - name: REDIS_HOST
              valueFrom:
                configMapKeyRef:
                  name: ppp-config-map
                  key: redis-host
            - name: REDIS_PORT
              valueFrom:
                configMapKeyRef:
                  name: ppp-config-map
                  key: redis-port
            - name: REDIS
              value:
"$(REDIS_HOST):$(REDIS_PORT)"
```

```
metadata:
  name: ppp-secrets
type: Opaque
data:
  aws-access-key-id: removed
  aws-secret-access-key: removed
  aws-region: removed
  db-username: removed
  db-password: removed
```