DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

An Empirical Study of Containerized CI/CD Pipelines for OpenWhisk Serverless applications

Sakthi Kumar Raj Kumar

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

An Empirical Study of Containerized CI/CD Pipelines for OpenWhisk Serverless applications

Eine empirische Studie über containerisierte CI/CD Pipelines für OpenWhisk Serverless Anwendungen

Author:	Sakthi Kumar Raj Kumar
Supervisor:	Prof. Dr. Michael Gerndt
Advisor:	M.Sc. Anshul Jindal
Submission Date:	11.06.2021

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 11.06.2021 Sakthi Kumar Raj Kumar

Acknowledgments

I would like to profoundly express my gratitude to my advisor Anshul Jindal and my supervisor Prof. Dr. Michael Gerndt for their guidance and assistance. Their advice and comments mean a lot to me and help me to follow the right path until the end of this research. I would also like thank my parents and my friends for providing me with endless support and continuous encouragement throughout my years of study. This success would not have been possible without them. Thank you.

Abstract

Serverless computing is a cloud computing model where the server-side logic runs in the stateless containers that are event-triggered and are usually managed by vendor hosts such as AWS Lambda. This computing model describes a paradigm in which the infrastructure remains completely hidden from the developer and is managed by the cloud provider. An example of such a model is Apache OpenWhisk, an open source distributed serverless platform that executes functions in response to events at scale. The usage of such serverless methods promises infrastructure cost reduction and automatic scalability. One more important benefit of serverless approach is to make the operations part of DevOps process simpler, reducing the time on the management and maintenance of the servers. Despite these benefits, applications using serverless computing model require a new look at DevOps automation practices to accelerate the application delivery process in a reliable way.

Continous Integration and Delivery is essential for secure and reliable delivery of software services. This DevOps practice is already supported by some cloud providers with their own cloud services, but this leads to a vendor lock-in, as individual implementations and DevOps processes depend on the provider. Furthermore, these cloud-based CI/CD services come at a cost once the build limitations are exceeded. To address these problems, we have set out to build an open source CI/CD solution, which is integrated with monitoring tools to visualize the system metrics and the build performance. The proposed tool offers isolation of the platform providers and is provider-independent. The result of the work could also be used to find out performance bottleneck in the build time caused by the services through distributed tracing.

The aim of this master's thesis is to develop a Continuous Integration and Delivery solution for both Microservices and Serverless application of the Internet of Things (IoT) platform and evaluate the results of the implementation. In addition, the thesis also provides empirical results on improving CI/CD build execution time, and provides a better branching solution that minimizes the overhead of maintaining branches for multiple environments. Along the way, the thesis addresses a few of the research-oriented questions that arise with the development methods used for building the CI/CD solution.

Contents

1 Introduction

In today's world, enterprises face the challenge of rapidly changing competitive landscapes, evolving security requirements, and performance scalability. Traditional software development methodologies are not enough to fulfill the ever-changing business requirements. Adaptation of Agile software development practices enables flexibility, efficiency and accelerates the Software Development Life Cycle (SDLC), which is attractive to the software development companies [1]. Implementation of CI/CD pipeline on agile has enabled faster delivery of software and increase in productivity [2]. Continuous Integration (CI) and Continuous Delivery (CD) are some of the practices aimed at helping the organizations to accelerate their development and delivery of software features without compromising quality [3].

Continuous Integration (CI) is a software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests get executed. Continuous Deployment (CD) also referred to as continuous delivery, is the fast-paced, automation-heavy software engineering approach in which teams work in short iterations to produce software that is deployable (production ready) at any time [3]. These two continuous practices are expected to provide several benefits such as receiving immediate feedback from software development process, having reliable and frequent releases which leads to improved product quality and elimination of manual operations through automation. A growing number of cases indicate that the continuous practices are making inroad in software development industrial practices across various domains and size of organizations [4][5][2]. At the same time, adopting continuous practices is not a trivial task since organizational processes, practices and tool may not be ready to support the highly complex and challenging nature of these practices. It is not surprising that Perforce report found that 65% of software developers, managers and executives have used CD [6]. Additionally, the "State of DevOps" survey with 3200 participants from around the world found CD positively impacts IT performance and negatively impacts deployment pain [7]. Yet, implementing the CI/CD pipeline automation needed to properly provide CD is challenging and takes a lot of time and tuning due to many moving parts and specifics of the organization [8].

A typical CI/CD workflow involves a continuous integration (CI) service like Travis or Jenkins, that is triggered by new changes in the Version Control System (VCS) to build, test and deploy the packaged application. Studies in the past have looked at the implementation in various organizations [9][10] and it is commonly reported in those studies that benefits gained are many but the implementation of these services takes time. As more and more experience is being gained with different ways to implement, it has become obvious that different solutions are possible and that they may fit different needs. The main focus of this thesis is to design and implement a containerized CI/CD pipeline for both OpenWhisk serverless application and Microservices application of the IoT platform. In addition, the

thesis also aims to improve the overall execution time of the CI/CD builds and provides a better branching solution for continuous integration and deployment.

We focus on the revolution in CI/CD workflows caused by containerization, the virtualization technology that enables packaging an application together with all its dependencies and execution environment in a light-weight and self-contained unit. Containerization has transformed these workflows with promising speedups and higher level of abstraction. Containers encapsulating a packaged application ready for deployment can be specified declaratively, versioned together with the rest of the code, and published automatically to a cloud-based registry for easy access by users and other applications. Among containerization, Docker containers have been downloaded 300B+ times[1] and their usage is spreading rapidly [11]. Thus, docker container usage is relevant to most of the containerization community. We found that two Docker image deployment workflows[2] were prominent:

- Docker Hub Workflow (DHW) - where Docker Hub builds images from the version control.
- CI Workflow (CIW) - where developers build images using docker commands inside their preferred CI tool such as Travis CI or Jenkins.

Developers tend to prefer CIW because of higher configurability, reliability, performance and scalability [12]. We have built our pipelines based on CIW to have more control over the pipeline development process.

A pipeline is usually divided into different chain links called stages, which contain single or multiple jobs describing the commands needed to be executed to achieve the desired outcome. In a containerized CI/CD pipeline, every single job runs in a container based on an image, which includes all the necessary dependencies required by a single project. One of the main advantages of containerized pipeline is execution of jobs in a self-contained container unit such that there is no conflict between different jobs in a project that are concurrently in execution, including different project pipelines in the same node. Scalability is another major benefit of the containerized pipeline. With the help of container orchestration platform such as kubernetes, it becomes easier to run these build pipelines on-demand, in which the build agents are created only when certain actions are triggered such as a code check-in, a test trigger or as a result of previous step in the workflow.

To refine CI/CD even further, the complete pipeline can be made serverless to run the build server on demand. There is a CI/CD serverless pipeline that caters to enterprise users of AWS [13]. This move is driven mainly by low overhead cost and lower chance of errors with version control being the single source of truth [14]. Yet, there aren't many open source applications that offer such services.

In the 2018 summer semester at the Technical University of Munich, a group of students developed a prototype of an IoT platform for an IoT practical course [15]. The platform was

[1] `https://www.docker.com/company`

[2] These resemble the GitHub push and pull-based models: the CI Workflow "pushes" the Docker image, while the DH Workflow "pulls" it.

made to be deployed on kubernetes container platform. CI/CD pipelines developed as a part of this thesis will run test cases, build IoT service images and deploy the updated images to the IoT platform.

2 Background

The IoT platform consists of microservices and serverless functions that are constantly under development. After a feature development, developers take an additional step to test and deploy the changes to the production servers. To employ best practices of deployment and to save developer 's time, the process of testing and deployment needs to be automated. This thesis will address the problem of building a CI/CD pipeline to build and push docker images to a private docker registry and carry out the deployment of microservices and serverless functions to the IoT platform, hosted on LRZ cloud. The CI/CD pipelines proposed as a part of this thesis will address both microservices and serverless based projects.

2.1 Kubernetes

Kubernetes is an open-source container orchestration platform for the automation of container software deployment, scaling and management.[1] The smallest computing unit that can be created and managed in Kubernetes is called a pod, and it consists of one or more containers.[2]. The IoT applications are deployed in two different Kubernetes cluster[3] namely, IoT OpenWhisk and IoT Microservices.

2.2 IoT Platform Architecture

Figure 2.1 is an IoT platform architecture designed by students in the 2018 summer semester at the Technical University of Munich for the IoT practical course [15]. There are various components for this platform, namely an IoT Back-End, an IoT Front-End, an IoT Gateway, the Kafka streaming platform and Elastic & Kibana. Every component is designed to be run on a kubernetes cluster.

2.2.1 IoT Back-End and IoT Front-End

The IoT Back-End and Front-End is a web application for managing IoT devices and sensors. For each IoT device, a JSON web token (JWT) authentication and authorization token can be downloaded and used in the IoT devices. When a sensor is being created either through GUI or API, a Kafka topic and its corresponding Elasticsearch index are then created.

[1] `https://kubernetes.io/`

[2] `https://kubernetes.io/docs/concepts/workloads/pods/`

[3] `https://www.redhat.com/en/topics/containers/what-is-a-kubernetes-cluster`

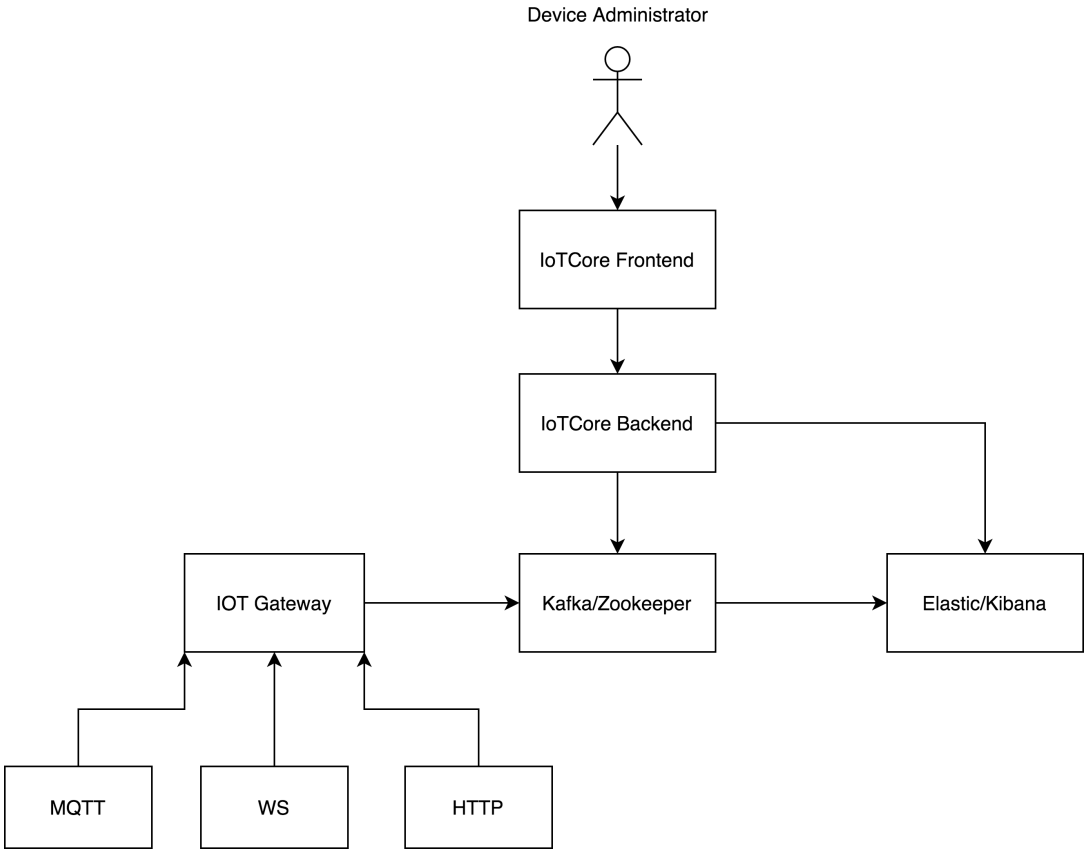

Figure 2.1: IoT Platform Architecture

2.2.2 IoT Gateway

Messages from IoT devices are sent directly to the IoT gateway. There are multiple implementations of the gateway running concurrently to receive messages from different protocols such as MQTT, WebSocket, and HTTP. After the gateway receives the message, it checks the JWT token set in the protocol for authentication and authorization. For HTTP and WebSocket, the token is placed in the HTTP authentication header; for MQTT, the token is set as the password. If the message is authorized, the gateway will append the message to the Kafka topic.

2.2.3 Kafka

Apache Kafka[4] is a distributed streaming platform that can reliably store sequences of messages. The messages are stored in categories called topics, and a topic can have one or more partitions in which the partitions facilitate the distribution of work in a simple way. For each partition, a sequence id number called an offset is assigned to each message, where the end offset refers to the offset of the last message plus one. When the client sends messages to a topic, the messages are queued to partitions in a round-robin fashion.

2.2.4 Elasticsearch

Elasticsearch[5] is a distributed, RESTful search and analytics engine capable of addressing a growing number of use cases. As the heart of the Elastic Stack, it centrally stores the data for lightning fast search, fine-tuned relevancy, and powerful analytics that scale with ease. It provides near real-time search and analytics for all types of data such as structured or unstructured text, numerical data, or geospatial data. Elasticsearch can efficiently store and index it in a way that supports fast searches. It could be even be used to discover trends and patterns in the data. The distributed nature of Elasticsearch enables the deployment to grow seamlessly.

2.2.5 Kibana

Kibana[6] is a free and open user interface that can be used to visualize Elasticsearch data and navigate the Elastic Stack. Some of its key features include

- Search, observe and protect - From discovering documents to analyzing logs to finding security vulnerabilities, Kibana is the portal for accessing these capabilities.
- Visualize and analyze data - Searching for hidden insights, visualization of charts, gauges, maps and combining them in a dashboard.

2.3 Apache OpenWhisk

Apache OpenWhisk[7] is an open source, distributed serverless platform that executes functions in response to events at any scale. OpenWhisk manages the infrastructure, servers and scaling using docker containers. It supports a programming model in which developers write functional logic called Actions, in any supported programming language, that can be dynamically scheduled and run in response to associated events through Triggers from external sources or from HTTP requests. The project includes a REST API-based Command

[4] https://kafka.apache.org/
[5] https://www.elastic.co/elasticsearch/
[6] https://www.elastic.co/kibana
[7] https://openwhisk.apache.org/

Line Interface (wsk CLI) along with other tooling to support packaging, catalog services and many popular container deployment options.

2.3.1 Architecture

OpenWhisk is an event-driven serverless platform. It has a server that executes code when it receives an event. These events could be an HTTP GET/POST request, a hook dispatched by a tool, or even a notification from another service. The key concept of OpenWhisk is that it is based on docker containers, so it works out of the box and can be deployed on a docker-compatible deployment tool, so the focus lies only on business logic of the application. In fact, IBM[8] is using OpenWhisk as its serverless solution for their cloud service under the name Cloud Function.

Internally the architecture works as an asynchronous and loosely coupled system as developers need to develop functions that will be uploaded as actions. These actions are totally independent and because of this absence of coupling, the system is free to spawn or kill new processes based on the demand.

Triggers are implemented to invoke these actions that were created. These triggers are endpoints that will be called by the event sources, databases, applications or hooks. Finally, a set of rules are defined in order to bind the triggers with the actions. Fig 2.2 shows the overall architecture of the platform.

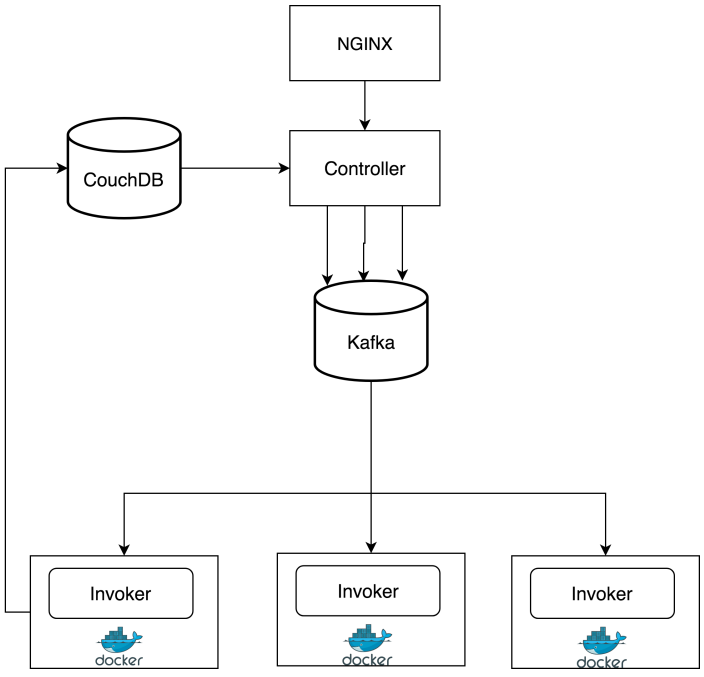

Figure 2.2: OpenWhisk Architecture

[8] https://www.ibm.com/cloud/functions

Internally OpenWhisk has an Nginx server as an entry point to the system. This webserver works mainly as a proxy for API and SSL. Then the incoming requests are injected into the controller. The controller works hand-in-hand with couchDB, where the code, credentials, metadata, namespaces, and other actions' definitions are saved. Then we have Kafka, which works as the glue layer between the controller and the invokers. In this case, Kafka works as a buffer for messages sent from the controller to the invokers. And finally, we have the invokers that are in charge of spawning and executing the code inside docker containers. The invokers copy the code from CouchDB and inject it into the docker container to be executed.

2.3.2 OpenWhisk actions performing IoT operations

Actions are stateless functions that run on the OpenWhisk platform. For example, an action can be used to trigger events such as responding to a database change or to an API call. In general, an action is invoked in response to an event and produces some observable output. Hence, actions can be leveraged to perform some of IoT operations performed by IoT Backend service. For instance, when a sensor is being created either through GUI or API, actions can be used to create a Kafka topic and its corresponding Elasticsearch index.

2.4 CI/CD for IoT Microservices and OpenWhisk serverless actions

With continuous software changes that are introduced in the project, it is necessary to ensure code quality and security while maintaining system stability.

In Continuous Integration, developers build, run and test their code on their local development setup before committing the changes to the version control repository. After changes are pushed to the repository, CI pipeline detects the changes and the project is built with the latest version of the source code followed by the execution of unit and integration test cases. If one of the steps fails, the integration pipeline will stop or continue depending on the defect severity and the results gets notified to the development team through email or chat systems.

Once the control is transferred from continuous integration, the continuous delivery pipeline deploys the code changes to the testing or staging environment. It makes it possible to release builds to the production environment if changes in the testing environment is stable and it effectively reduces the time to the market.

With the help of CI/CD, the process of running test cases on the code change and deploying the changes to staging and production servers is automated, so developers can focus on building the application.

2.5 Evaluation of CI/CD services

Jenkins

Jenkins[9] is a self-contained, open source automation server which can be used to automate tasks related to building, testing and delivering or deploying software[2]. It is a self-contained Java-based program with packages for Windows, macOS, and other Unix-like operating systems. With hundreds of plugins available, jenkins supports building, deploying, and automating software development projects. Its key features include Jenkins Configuration as Code (JCasC), distributed master-slave architecture builds and running CI/CD pipelines in dynamic pods through kubernetes.

Travis CI

Travis CI[10] is a hosted CI/CD service that automatically detects new commits made and pushed to a remote repository. The key features of Travis CI include deployment to multiple cloud services using adapters, pre-installed database services, auto deployments on passing builds and support for multiple languages. Although the service is offered on a free plan for open source projects, the hosted version limits the number of concurrent jobs, making it expensive for private projects. [16]

GitLab CI

GitLab[11] is a suite of tools for managing different aspects of the software development lifecycle. Its key features enable rapid iteration and delivery of business values and helps delivery teams fully embrace CI by automating the builds, integration, and verification of source codes. Although it provides better features such as version control, free static websites, auto deployments of the services to production environment, its free plan limits the build time to 400 CI/CD minutes per month making it ideal only for smaller projects with infrequent changes to the codebase. [17]

TeamCity

TeamCity[12] is a JetBrains's build management and continuous integration server. It runs in a Java environment and integrates with Visual Studio and other Integrated Development Environments. Its key features include running parallel builds simultaneously on different environments, flexible user management, user role assignment, different ways of user authentication and a log of all user actions for transparency of all activities on the server. The self-hosted version of TeamCity provides better features such as unlimited users and build time, yet the tool limits to 3 agents and 100 build configurations that can be persisted in the system. [18]

[9] `https://www.jenkins.io/`
[10] `https://travis-ci.org/`
[11] `https://docs.gitlab.com/ee/ci/`
[12] `https://www.jetbrains.com/teamcity/`

Prow

Prow[13] is a kubernetes based CI/CD system. Jobs can be triggered by various types of events and report their status to many different services. In addition to job execution, prow provides GitHub automation in the form of policy enforcement, and automatic PR merging. Its key features include Job execution for testing, batch processing, artifact publishing, GitHub merge automation with batch testing logic and Prometheus metrics. Though it packs a lot of features right out the box, it supports only code repositories hosted on GitHub.[14]

Drone

Drone[15] is a modern continuous integration platform that empowers busy teams to automate their build, test and release workflows using a powerful, cloud native pipeline engine. With Master-Worker architecture, drone's runners are standalone daemons that poll the master server for pending pipelines to execute. Although the self-hosted version of drone provides better features such as docker and kubernetes runners that create dynamic containers to run builds and are destroyed once the job is complete, it is limited by 5000 builds, making it ideal for smaller projects. [19]

As a part of this thesis, the CI/CD service used to build and deploy IoT applications to their respective platform is jenkins. We decided to build our pipeline with jenkins because it's an open source project with a broad plugin ecosystem. The parallel execution of building images coupled with unlimited number of builds and the build run time makes it an ideal solution for a multifaceted application. Table 2.1 provides an overview of the various CI/CD tools that were evaluated based on pricing model and supported Version Control System (VCS).

[13] `https://github.com/kubernetes/test-infra/tree/master/prow`
[14] `https://gitlab.com/gitlab-org/gitlab/-/issues/217982`
[15] `https://www.drone.io/`

Table 2.1: Overview of the CI/CD tools that were evaluated.

Name	Description	Supported VCS	Pricing
Jenkins	With thousands of plugins to choose from, jenkins can help teams to automate any task that would otherwise put a time-consuming strain on the software team.	GitHub, GitLab and Bitbucket.	Open source with self-hosting features. There is no limit on build time or the number of agents.
Travis CI	Hosted continuous integration service for open source and private projects.	GitHub, Gitlab and Bitbucket.	Free for open source projects but pricing starts at $759 per year for 1 concurrent job plan.
GitLab CI	GitLab is a single application for the entire DevOps lifecycle.	Gitlab, GitHub and Bitbucket.	Only 400 CI/CD minutes per month on the free plan. Premium plan offers 10,000 CI/CD minutes at $228 per year.
TeamCity	A Java-based build management and continuous integration server from JetBrains.	Git, Subversion, Perforce, Team Foundation Server and Mercurial.	Free plan offers a trial of 14 days and $45 per month after the trial. However, self-hosted software offers 3 agents and 100 build configurations.
Prow	Prow is a kubernetes based CI/CD system. Jobs can be triggered by various types of events and report their status to many different services.	GitHub.	Open source. However, support is only available for GitHub.
Drone	Continuous Integration service.	GitHub, GitLab, Gitea and Bitbucket.	Open source. There is a limit of 5000 builds per year.

3 Related Work

Serverless infrastructure has been gaining popularity in recent years. Despite the large number of materials investigating serverless use cases [20] [21], scalability [22] and success stories of serverless computing [23], fewer studies have focused on DevOps practices supporting serverless applications.

Continuous Integration (CI) and Continuous Delivery (CD) form the backbone of the product delivery lifecycle. A well-tuned, fault tolerant and scalable CI/CD pipeline is important to accelerate the application delivery process. Despite obvious advantages, a rapid release approach combined with continuous change processes resulting from CI/CD principles will in the long run generate new challenges. The entire process needs to be carefully examined and controlled to keep anomalies in check. For instance, a build agent that consumes more than the expected CPU and memory resources could potentially block another agent builds that run on the same server. A way to mitigate this risk is to implement a continuous monitoring and observability solution that monitors the build pipeline and notifies the abnormality. Cloud Service Providers (CSPs) provide services like Amazon CloudWatch [24] from Amazon AWS, Azure monitor [25] from Microsoft for monitoring cloud applications and cloud resources. However, these monitoring solutions only monitor the resources and let the users figure out if there are performance issues. For instance, telemetry data is necessary to monitor the time spent at different stages of the pipeline build. If a build agent spends more than the expected time to build an image, it signals a need for build optimization to improve the overall build execution time. However, such application specific telemetry monitoring isn't widely supported by many cloud providers. Additionally, there hasn't been much research on improving the build execution time eventually leading to quicker development feedback and faster deployment cycles. Our approach focuses on building a central monitoring solution that monitors the CI/CD infrastructure including the build specific monitoring that includes telemetry data and improving the overall build time by parallel execution of build stages.

There has been some research on continuous delivery pipelines for serverless application. In GitOps based Delivery for serverless applications [26], the author proposes a pipeline build to deploy images to the OpenFaaS[1] platform on Amazon EKS[2] using the hosted CI/CD service Travis CI. Additionally, the research proposes that containerization and container orchestrations are optimal for continuous delivery of FaaS applications. Their implementation has shown that container orchestration and cloud-native platforms such as kubernetes enable isolated, independent serverless environments to be integrated into a CI/CD process.

[1] `https://www.openfaas.com/`

[2] `https://aws.amazon.com/eks/`

Another study [27] focuses on building the pipeline using Gitlab runners[3] to deploy lambda[4] functions in AWS. The study states that Native AWS cloudwatch tools were not found user friendly and companies use additional log aggregation tool in addition to cloudwatch. Other services such as AWS X-Ray[5] tracing feature is powerful, but it's not clear at what stage it should be used. Furthermore, usage of Gitlab runners to deploy the application comes at a cost. The premium pricing plan of Gitlab will only allow 10,000 CI/CD minutes per month, which is sufficient. However, when the number of projects' grows in size, so does the number of build minutes.

Our current approach is to implement an open source CI/CD solution with the ability to scale the build agents and have monitoring tools integrated to analyze the performance of pipeline builds.

[3] `https://docs.gitlab.com/runner/`
[4] `https://aws.amazon.com/lambda/`
[5] `https://aws.amazon.com/xray/`

4 Research Methods

This chapter describes research motivation and defines research questions. It also presents the reasoning behind adopting a certain approach to implement the continuous integration and delivery for both microservices and serverless architectures.

4.1 Research Questions

While many studies on the DevOps theory and adoption have been made in the past [28], there are still not enough research on how the DevOps specifics differ for different architectural patterns, in our case both Microservices and Serverless applications. Despite the fact that Serverless has many benefits [29] comparing to monolith and microservices in terms of operations, it is still unclear how much effort it takes to maintain the DevOps pipeline for serverless applications and the differences in pipeline metrics such as the build run-time, CPU and memory resource utilization between the two architectures. The materials that are already available are usually focused on cases with a few serverless functions [30]. Yet with the usage of hundreds of serverless functions can lead to maintenance overhead. The reason can be the atomic nature of serverless functions that leads to many challenges. Some of the other challenges are: 1) Should each function have a separate version control repository and CI/CD config file? 2) If a project has both microservices and serverless functions, can they exist in a different branch of the version control or should they exist as two independent projects? The answers to these questions depend on every specific team and project, but the findings from the single case can be useful for the decisions made for future projects.

This motivation and the challenges define the following research questions:

1. *What is the state of the art in CI / CD approaches in the OpenWhisk environment?*
   The aim is to understand the best practices that could be employed to achieve CI/CD for the OpenWhisk platform.

2. *What are the best branching strategies for CI/CD?*
   The purpose is to identify the best practices that can be employed to structure the code repository. Having a good branching strategy minimizes the management overhead and enables Continuous Integration (CI) and Continuous Delivery (CD) to accelerate the application delivery.

3. *How to optimize the build times of Jobs that are built by CI/CD agents?*
   The aim is to research about the ways to improve the CI/CD pipeline build time through parallel execution of job stages and visualization of the end result through distributed tracing.

4.2 Traditional vs Containerized build agents

With more software companies and organizations adopting microservice patterns, it becomes increasingly necessary to build, run test cases and deploy images frequently than ever before. This move requires more build agents to be run simultaneously to meet the increasing demand.

Traditional CI/CD automation tools have a similar concept as the current tools, where the master node distributes the job to agent nodes with all the necessary dependencies across multiple hosts. However, to meet the demanding requirements, the new agents have to be manually configured each time through the web interface. With the advent of continuous delivery, the ability to manage pipelines was built into the tools. Traditional CI/CD offered a wide range of functions without making great demands on its environment. The configuration therefore often has to be maintained by an expert or a team of experts.

To reduce the complexity of extensive configuration and maintenance, two technological trends came into existence. The first trend is to direct more focus on application development and describe steps for building softwares through a configuration file, which is read by the system, interpreted and translated into a script that gets executed. This strategy is Configuration as Code. The other trend is to make consistent use of docker container agents to build and test softwares, so that agents do not have to be manually configured with all the necessary dependencies, however all the dependencies become a part of the docker image.

Furthermore, these container agents can be managed through a container orchestrator such as kubernetes to run build agents on demand. When a build pipeline gets executed, agent pods are created by kubernetes to serve the demand. It is therefore not necessary to keep agents available for the CI/CD infrastructure at all times. The resulting delay caused by the scaling process is negligible in the CI/CD context. Depending on the solution, terminated pods must be partially cleaned up by Kubernetes cronjobs in order to free up memory that is still in use.

4.3 Benefits of Containerized CI/CD pipeline

CI/CD and containers have changed the world of software delivery by helping teams to speed up the development and release cycles. However, there is some confusion about how these techniques can be used in conjunction with one another.

The answer is with containerized CI/CD, where software development teams architect continuous builds through containers and orchestration. Containers make it easier for software engineers to continuously build and deploy applications and, by orchestrating container

deployment, software teams can achieve replicable, continuous clusters of containers. Containerized CI/CD has the advantage of further speeding up release cycles and improving build quality.

Automating the build with containers

Containers are designed to bundle the correct tool, version and other execution assets in a package. This makes it easy to build applications using a given set of tools and scripts, because the package is already prepared and ready to run. The container itself can be orchestrated by providers such as AWS ECS[1] or kubernetes Jobs.

Testing with containers

Like automation of builds, containers can be packaged with testing tools and scripts to ensure quality throughout the complete development cycle of the application.

A clean environment for CI/CD

A clean environment has always been a best practice for CI/CD because it reveals any flaws inherent in the product. However, containers are by default clean and a well-packaged image would not contain any cached data that could have an impact during the build or the testing phase. Since a new container is created and destroyed every time a build gets initiated, old or cached content cannot affect the current execution of the build nor will the current execution affect the future builds.

Secure, Fast and lightweight

Security is also enhanced by using containers, as any rogue scripts cannot access data or resources from other executions. This protects applications from data breaches during CI/CD.

Container best practices recommend packaging only what is required. This keeps the image lightweight, single purpose and allows easy scaling. This is achieved by downloading the image, creating, running and destroying containers quickly, using as little of the hardware resources as possible. Additionally, different stages in the pipeline can be supplied with different container images developed for a specific purpose instead of loading all the worker nodes with the required tools and scripts. This makes the base worker node light weight and adapt to newer changes in the pipeline.

Using containers, it is easy to replace the quality gate by simply replacing the image. So, if a given security tool is not providing results that meet or exceed required organizational standards, the image can be replaced with another without having to modify several traditional machines or recreating them.

[1] `https://aws.amazon.com/ecs/`

Scaling containers

Scaling is easier with containerized CI/CD pipelines. Containers can scale in seconds, compared with traditional infrastructures that take minutes or even hours, in some cases. They can also be scaled both vertically and horizontally to meet CI/CD requirements. Scaling containers simply requires a container orchestrator, such as AWS ECS or kubernetes, which has the added advantage of making container management easy.

Through scalability, an orchestrator could create worker node containers to build, run test cases and deploy the application on demand and destroy them once the build execution in complete. This ensures efficient use of available resource compared to having the worker nodes available at all times.

4.4 Containerized Builds with Jenkins

Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications. With a kubernetes cluster, it is easy to add an automation layer to jenkins. Also, it ensures that resources are used effectively and that the servers and the underlying infrastructure are not overloaded.

With the help of kubernetes plugin in jenkins, K8s can orchestrate the container deployment and ensure that jenkins always has the right amount of available resources.

4.4.1 Challenges with Containerized builds in Jenkins - Docker in Docker vs Docker out of docker (DIND vs DOOD)

In our current CI/CD architecture, once the automated test cases get successfully executed, it is necessary to push the images to a container registry, to be later used in the deployment phase. However, our base jenkins agent image doesn't contain the docker daemon to build and push images to the registry.

We have explored two different approaches to build the docker image within the base jenkins agent image. These include

- Docker Out Of Docker (DOOD)
- Docker In Docker (DIND)

Docker out of Docker

With Docker out of Docker, we connect the docker inside of our agent container to the docker daemon that is used by kubernetes. The only reason to use this method is because it's the easiest to set up. However, we are potentially breaking kubernetes scheduling since we're running things on kubernetes but outside of its control. Another downside is that this is a security vulnerability because we need to run our container as privileged and our jenkins

slaves will have access to anything that's running on the same worker node, which could also be a production service.

When we bind mount and use the host docker daemon socket used by kubernetes from inside a pipeline container, we pass all docker commands issued from the container to be passed to the host docker daemon. Fig 4.1 illustrates Docker out of Docker system architechture.

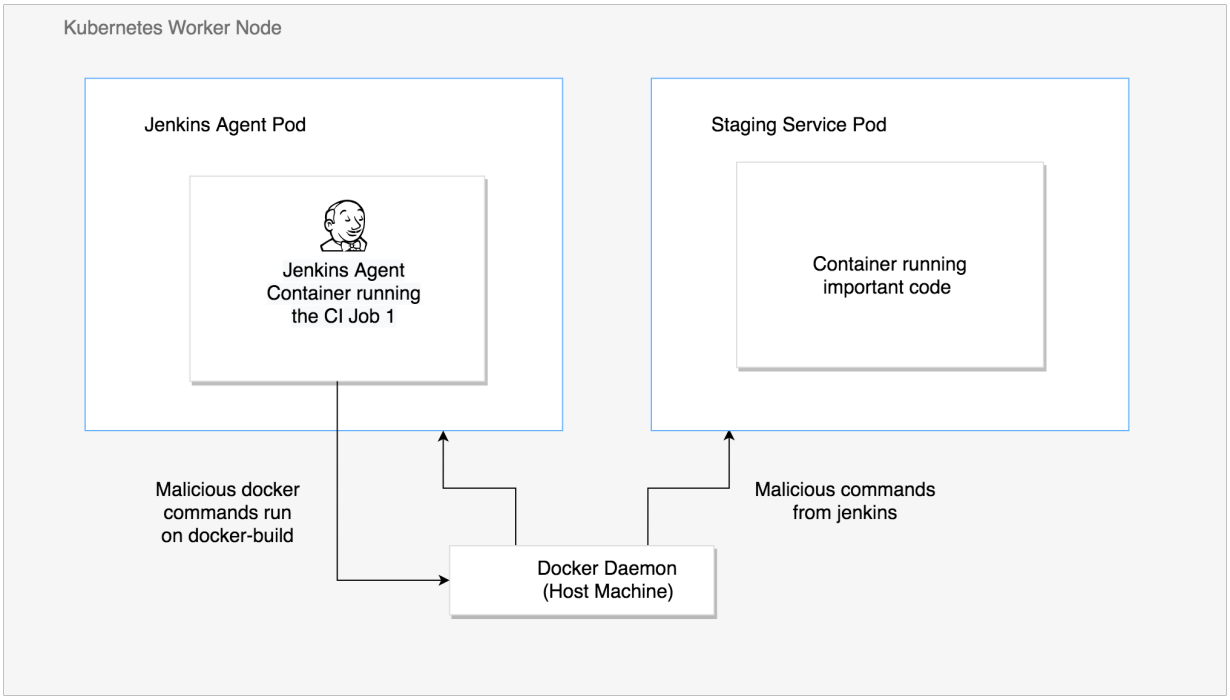

Figure 4.1: Docker out of Docker system architechture

When two pods under the same kubernetes node share the same host daemon and create containers to run their jobs, it could result in a failed execution of one of the pods when they try to create a container with the same name, since now the containers are created in the host daemon. Additionally, it will require some work to ensure that the agent pods don't create a container that has already been created.

There are a few other problems that can be caused by poor isolation. For example, by allowing users or agent containers to issue commands directly to the host docker daemon, they can create docker containers that are privileged or bind mount host paths, potentially creating havoc on the host.

When docker commands are sent to the host docker daemon, kubernetes does not know about this newly created container and rightfully does nothing to manage it. Since the new container does not know anything about pod networking, any open ports will not be reachable using the Pod's IP. When the container terminates, the layers of graph storage will

not be deleted by kubernetes. Upon pod termination, the newly created container will keep running especially if the container was started with the *detached* flag. Additionally, CPU and memory requested by the pod will only be for the pod and not the container spawned from the pod because the container now becomes a part of the host.

Any CI/CD system that allows users to create arbitrary docker containers, will quickly run into manageability, security and stability problems caused by poor isolation.

Docker In Docker

As the name suggests, we run a docker daemon container within the agent pod to isolate the creation of new containers or builds within the pod. The main requirement for Docker in Docker daemon is that it must not share the graph storage of the host docker daemon. Containers that are created within the Docker in Docker daemon are not visible to the host docker daemon. Fig 4.2 illustrates Docker in Docker system architechture.

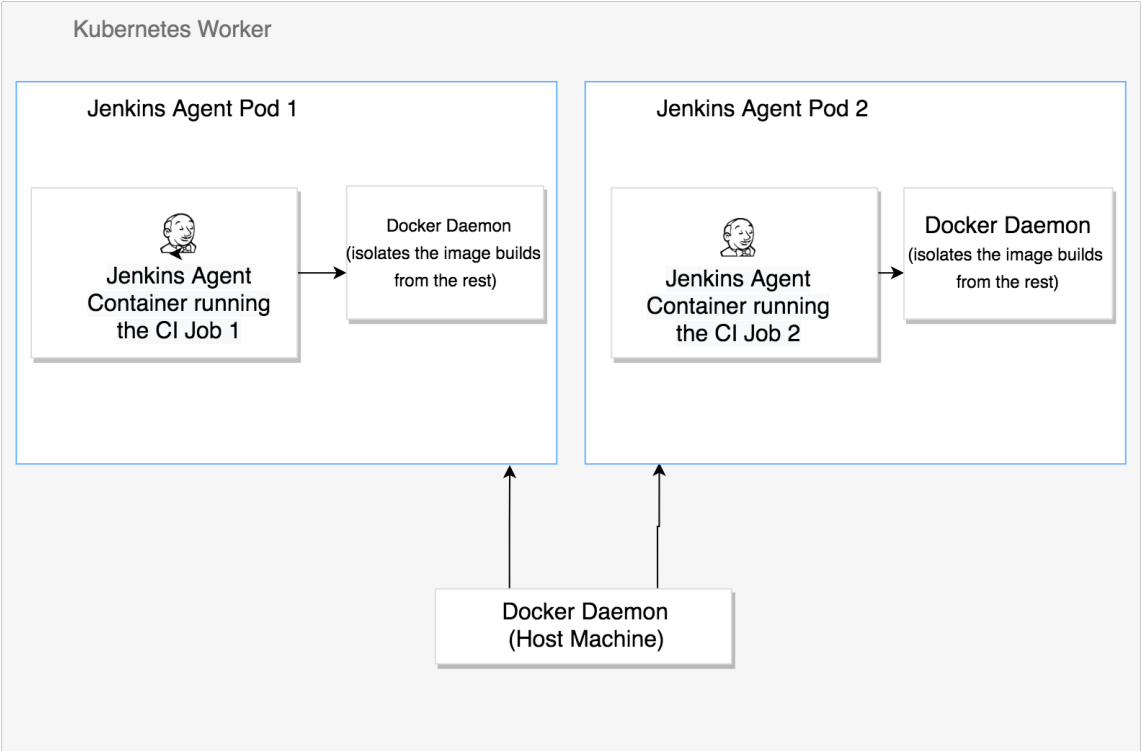

Figure 4.2: Docker In Docker system architechture

We use the concept of sidecar container pattern to create kubernetes pod template that contains a docker daemon container along with the jenkins agent. This container starts docker daemon on */var/lib/docker*.

The jenkins agent container runs a custom jenkins agent image packaged with all the

necessary docker file binaries and domain certificate of the private registry. By setting its environment variable **DOCKER_HOST** to ***tcp://localhost:2375***, we can ensure that the docker binary in the jenkins agent container points to Docker in Docker daemon.

The sidecar container, dind, starts the docker service on port 2375, and builds the docker images within the container upon receiving requests to build images. Any new containers created by the dind daemon process will inherit the CPU and memory constraints.Upon the pod termination, this dind container is killed and the other builds or contianers created by dind never shows up on the host. The EmptyDir used for Docker-in-Docker graph storage is also reclaimed by kubernetes when Pod is deleted. Code snippet 4.1 describes the jenkins kubernetes agent pod template yaml with two containers, namely agent container (jnlp) and dind (Docker in Docker).

Code Snippet 4.1: Docker In Docker pod configuration

```
spec:
  containers:
    image: "sakthik26/iot-jenkins-agent-ca2"
    imagePullPolicy: "IfNotPresent"
    name: "jnlp"
  - env:
    - name: "DOCKER_HOST"
      value: "tcp://localhost:2375"
    - name: "JENKINS_URL"
      value: "http://jenkins.default.svc.cluster.local:8080/"
  ---
    image: "sakthik26/dind"
    imagePullPolicy: "IfNotPresent"
    name: "dind"
  - env:
    - name: "DOCKER_TLS_CERTDIR"
      value: "\"\""
```

5 Implementation

This chapter describes the system design architectures of the IoT platform and the implementation of jenkins CI/CD to build and deploy images to the platform. In our current implementation, we have three different clusters setup for IoT Microservices application, IoT OpenWhisk Serverless application and Jenkins.

Cluster setup for the Microservices Application

- IoT Microservices Master Node – lrz.medium (9GB RAM and 2 vCPU with 60GB volume attached).
- IoT Microservices Worker Node – lrz.medium (9GB RAM and 2 vCPU with 50GB volume attached).

Cluster setup for the OpenWhisk Serverless Application

- IoT OpenWhisk Master Node – lrz.medium (9GB RAM and 2 vCPU with 60GB volume attached).
- IoT OpenWhisk Worker Node– lrz.medium (9GB RAM and 2 vCPU with 50GB volume attached).

Cluster Setup for Jenkins

- Jenkins Master Node – lrz.large (18GB RAM and 4 vCPU with 40GB volume attached).
- Jenkins Worker Node – lrz.large (18GB RAM and 4 vCPU with 160GB volume attached).

5.1 Architecture

Figure 5.1 shows the architecture of jenkins and its monitoring tools. This architecture was made to deploy built images to the IoT application clusters upon a successful pipeline build.

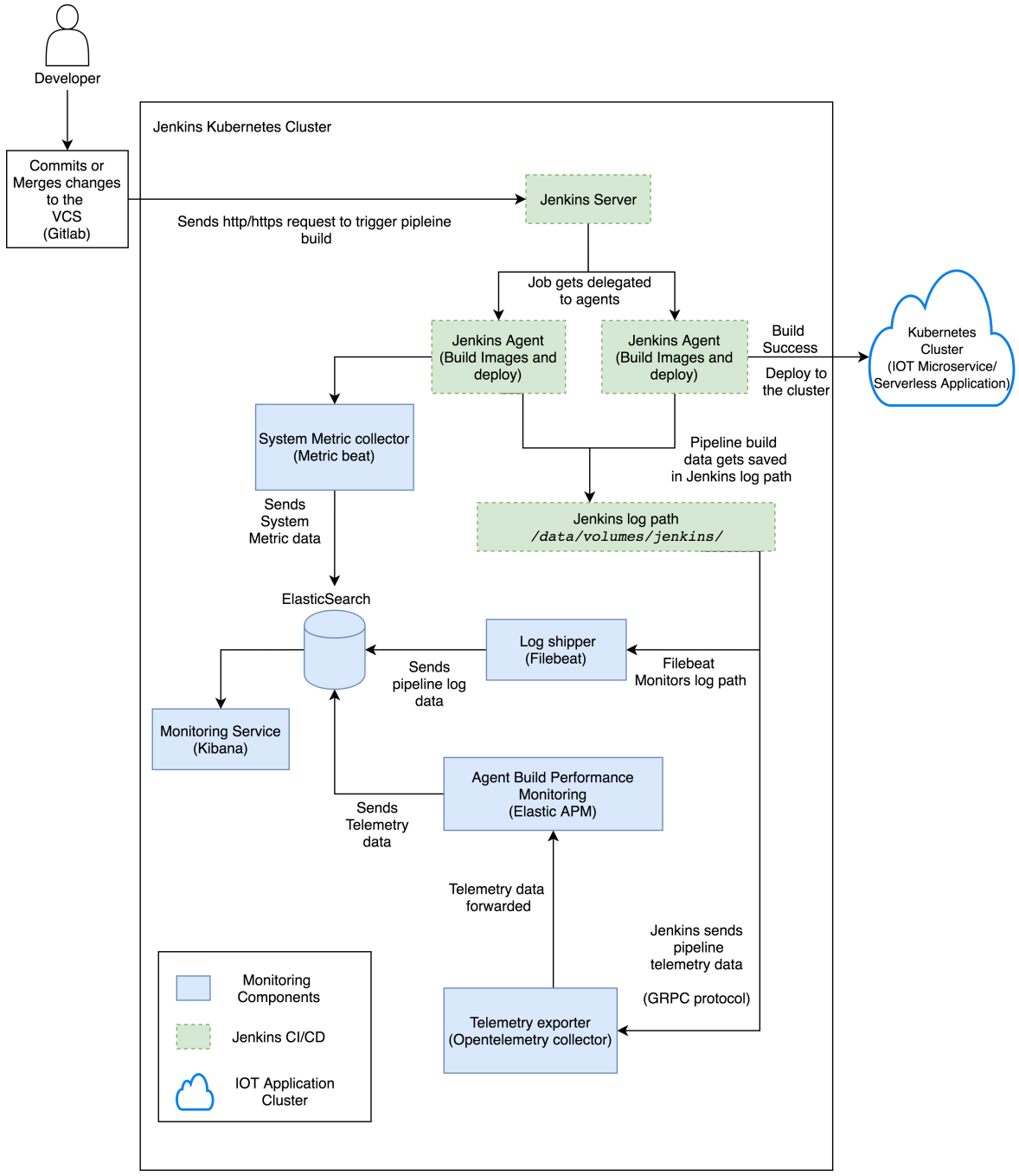

Figure 5.1: The architecture of the Jenkins and other Monitoring tools

There are two main components of our architecture:

- Jenkins Server and Agents
- Monitoring

5.1.1 Jenkins Server and Agents

A Jenkins Server can operate standalone both managing the build environment and executing the builds with its own executors and resources. However, the server might run out of resources when the number of projects' increases. Moreover, executing jobs on the server introduces a security issue: any user that runs a job on the server have full permissions to jenkins resources. This could mean, with a simple script, a malicious user can have a direct access to private information. To scale the server and to combat the security issues, agents were introduced.

The jenkins server is setup on the Master node with persistent storage volume to restore the application data in case of a system failure. Most of the jenkins versions that are deployed on servers are configured directly by jenkins administrators or DevOps engineers. When there is a system failure or there is a requirement for newer jenkins server installation, the systems need to be reconfigured again through the User Interface. This process of reconfiguration becomes cumbersome if configurations are lost that it cannot be restored or there is no backup available.

To address the problem of inflexible configurations, jenkins update problems, switching the application to new servers and no backup strategy, we have deployed our application through jenkins Configuration as Code, such that all the configuration is written as a code.

Jenkins Configuration as Code Solution

Setting up jenkins is a complex process, as both jenkins and its plugins require some tuning and configuring dozens of parameters within the web UI manage section. Experienced jenkins users rely on groovy scripts to customize jenkins and enforce the desired state. Those scripts directly invoke jenkins API and, as such, can do everything. But they also require one to know jenkins internals and are confident in writing groovy scripts on top of jenkins API.

Code snippet 5.1 illustrates jenkins configuration as code yaml with two containers, custom jenkins inbound agent image ***iot-jenkins-agent-ca2*** and Docker in Docker image ***dind*** for our pod template. In such case, whenever a build gets triggered in our jenkins server, any agent pod that is created to run the build will have such pod configuration. Additionally, our base jenkins server image is pre-configured to install all the necessary plugins required for our use case.

Jenkins Configuration as Code provides the ability to define this whole configuration as a simple, human-friendly, plain text yaml syntax. Without any manual steps, this configuration can be validated and applied to create other jenkins servers in a fully reproducible way.

Code Snippet 5.1: Jenkins configuration as code yaml

```
spec:
  containers:
    image: "sakthik26/iot-jenkins-agent-ca2"
    imagePullPolicy: "IfNotPresent"
    name: "jnlp"
    resources:
      limits:
        memory: "8000Mi"
      requests:
        memory: "2000Mi"
    securityContext:
      privileged: true
    tty: true
    image: "sakthik26/dind"
    imagePullPolicy: "IfNotPresent"
    name: "dind"
    resources:
      limits:
        memory: "6000Mi"
      requests:
        memory: "2000Mi"
    securityContext:
      privileged: true
    tty: true
```

CI and CD pipeline

When developers merge or commit their changes to the VCS, the source control management makes an http/https request to jenkins server, which delegates an agent to check out the latest version of the project from VCS, run automated test cases, build docker images and deploy them to the application environment. The exact delegation behavior depends on the configuration of each project. For instance, if a project pipeline build is configured to a specific agent machine label, a new agent pod template with that label is created to execute the pipeline. The most popular way agents are configured is through connections that are initiated from the master. This allows agents to be minimally configured and the control lives with the jenkins server.

Once the agent receives the control from the server to execute the pipeline, it checks out project and looks for a specific Jenkinsfile within the project repository. A Jenkinsfile[1] is a text file that contains the pipeline commands that are executed by agents in stages.The agent

[1] https://www.jenkins.io/doc/book/pipeline/jenkinsfile/

executes the command in stages and reports the status as success if all the stages within the pipeline gets successfully executed.

5.1.2 Monitoring

To monitor the performance of the builds that are associated with the project and to debug any pipeline related issues, we have implemented a monitoring solution using Elastic Stack [31]. The monitoring components that are implemented include:

- **Log Monitoring** – Filebeat, Elasticsearch and Kibana
- **Build Performance Monitoring** - Opentelemetry collector and Elastic Application Performance Monitoring (APM)
- **Pod Monitoring** – Metricbeat

Log Monitoring

A build log of a jenkins job contains a full set of records for a job, including the build name, number, execution time, the result. If one of the pipelines is broken, this data can provide a wealth of information to help troubleshoot the root cause. Jenkins supports console logging but in case of a large amount of running jobs, it becomes difficult to keep track of all the activity and hence collecting all this data and shipping it into services such as elasticsearch can help to give us more visibility. Figure 5.2 shows the jenkins log monitoring service in our cluster.

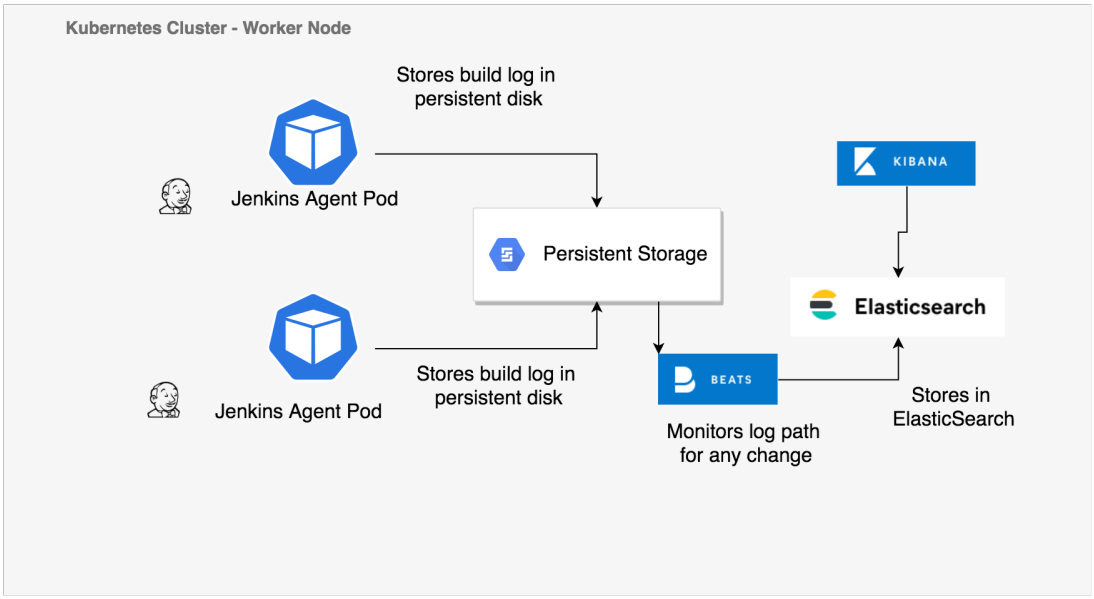

Figure 5.2: Jenkins log monitoring

In our current architecture, we have deployed Elasticsearch, Filebeat and Kibana in a separate namespace within the same virtual machine.

Filebeat

Filebeat [32] is a lightweight shipper for forwarding and centralizing log data. Installed as an agent on the servers, filebeat monitors the log files or locations that is specified, collects log events, and forwards them either to Elasticsearch or Logstash for indexing. When filebeat starts, it starts one or more inputs that look in the locations specified for log data. For each log that filebeat locates, it starts a harvester. Each harvester reads a single log for new content and sends the new log data to libbeat, which aggregates the events and sends the aggregated data to the output, elasticsearch. A filebeat input specifies the log source file to read from. If the input type is log, filebeat inputs find all files on the virtual machine that match the defined paths and starts a harvester for each file. Code snippet 5.2 shows the Filebeat configuration to harvest lines from all log files that match the specified jenkins log paths.

Code Snippet 5.2: Filebeat input configuration yaml

```
data:
  filebeat.yml: |-
    filebeat.inputs:
    - type: log
      enabled: true
      paths:
        - /data/volumes/jenkins/jobs/*/builds/*/log
```

Filebeat currently supports several input types. Each input type can be defined multiple times. The log input checks each file to see whether a harvester needs to be started, whether one is already running, or whether the file can be ignored. New lines are only picked up if the size of the file has changed since the harvester was closed.

Filebeat and Elasticsearch Configurations

The filebeat instance deployed in our virtual machine is configured to use the following path */data/volumes/Jenkins/jobs/*/builds/*log* as the source of log files to detect if there are new log files created and forwards them to elasticsearch output running on port 9200.

Code snippet 5.3 and 5.4 shows Filebeat and Elasticsearch configurations to harvest lines from all log files that match jenkins logs and store them in elasticsearch.

Code Snippet 5.3: Filebeat configuration yaml

```yaml
data:
  filebeat.yml: |-
    filebeat.inputs:
    - type: log
      enabled: true
      paths:
        - /data/volumes/jenkins/jobs/*/builds/*/log

    cloud.id: ${ELASTIC_CLOUD_ID}
    cloud.auth: ${ELASTIC_CLOUD_AUTH}

    output.elasticsearch:
      hosts: ['${ELASTICSEARCH_HOST:elasticsearch}:${ELASTICSEARCH_PORT:9200}']
      username: ${ELASTICSEARCH_USERNAME}
      password: ${ELASTICSEARCH_PASSWORD}
```

Code Snippet 5.4: Elasticsearch configuration yaml

```yaml
image: "docker.elastic.co/elasticsearch/elasticsearch"
imageTag: "7.9.3"
imagePullPolicy: "IfNotPresent"

protocol: http
httpPort: 9200
transportPort: 9300

service:
  labels: {}
  labelsHeadless: {}
  type: ClusterIP
  nodePort: ""
  annotations: {}
  httpPortName: http
  transportPortName: transport

updateStrategy: RollingUpdate
```

Shipping log data from Jenkins to ElasticSearch

When filebeat detects a file change in the filepaths specified within its input configuration, it sends the log messages written to the file to Elasticsearch backend every few seconds. The filebeat instance is configured to run as a daemonset, so it ensures all nodes within the cluster

will run a copy of it. As new nodes are added to the cluster, new filebeat pods will be created and removed when nodes get terminated. Some of the typical use cases of running such instances as daemon set is to ensure log monitoring on every node in the jenkins cluster. For instance, if a new agent pod is scheduled to run a build in the master node instead of worker node, filebeat input configuration could be modified to include newer file paths to detect log file changes in the master node.

Build Performance Monitoring

To understand the performance of jenkins pipeline jobs, it is necessary to identify the time spent at each stage in a pipeline. For instance, the time spent in the build queue, waiting for a new agent to execute the build or to detect increased time spent in the build stage of a specific image. This can easily be achieved through tracing. Traditionally, tracing is a low-level practice used to profile and analyze application code by developers through a combination of specialized debugging tools and programming techniques. The process of monitoring builds involves two different services:

- The OpenTelemetry Collector [33] offers a vendor-agnostic implementation on how to receive, process, and export telemetry data. It removes the need to run, operate, and maintain multiple agents/collectors in order to support open-source observability data formats.
- Elastic APM [34] is an application performance monitoring system that helps to monitor software services and applications in real-time, by collecting detailed performance information on response time for incoming requests.

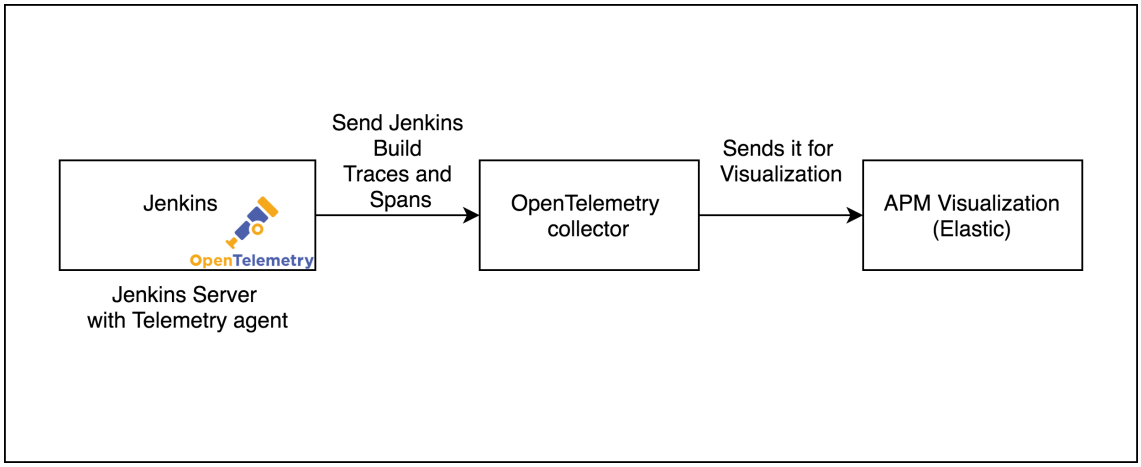

Figure 5.3: Performance monitoring using jenkins telemetry agent

Figure 5.3 above shows the performance monitoring services in our cluster. In our current jenkins implementation, we have configured our jenkins server with an opentelemetry agent plugin to send each pipeline build telemetry data to an opentelemetry collector through GRPC protocol. This collector receives, processes and exports the telemetry data to elastic observability service called APM. With this integration, it is easier to detect increased build times within different stages of the pipeline.

Code snippet 5.5 and 5.6 shows the opentelemetry collector and APM yaml configuration to send distributed tracing data from jenkins to APM

Code Snippet 5.5: OpenTelemetry Collector configuration yaml

```
 exporters:
      elastic:
        apm_server_url: 'http://apm-server-apm-server.elk.svc.cluster.local:8200'
kind: Service
metadata:
  name: otel-collector
spec:
  ports:
  - name: metrics # Default endpoint for querying metrics.
    port: 8888
  - name: grpc # Default endpoint for OpenTelemetry receiver.
    port: 55680
    protocol: TCP
    targetPort: 55680
```

Code Snippet 5.6: APM configuration yaml

```
apmConfig:
  apm-server.yml: |
    apm-server:
      host: "0.0.0.0:8200"
    queue: {}
    output.elasticsearch:
      hosts: ["http://elasticsearch-master:9200"]
service:
  type: ClusterIP
  loadBalancerIP: ""
  port: 8200
  nodePort: ""
  annotations: {}
```

Pod Monitoring

Monitoring an application's current state is one of the most effective ways to anticipate problems and discover bottlenecks in any environment. Yet it is also currently one of the biggest challenges faced by many organizations. The growing adoption of microservices makes logging and monitoring more complex with the need for complex kubernetes monitoring. This is because a large number of distributed services are communicating with each other. A single point of failure can stop the entire process, but identifying it is becoming increasingly difficult.

To understand the CPU and memory used by kubernetes pods during the entire build process and how they differ when the build stages are executed in parallel, we have deployed Metricbeat [35], a lightweight shipper that is installed on the servers to periodically collect metrics from the services that are running on the server. It takes the metrics and statistics that it collects and ships them to the elasticsearch. Metricbeat is deployed as the DaemonSet to ensure that there's a running instance on each node of the cluster. These instances are used to retrieve most metrics from the host, such as system metrics, docker stats, and metrics from all the services running on top of kubernetes. In addition, one of the pods in the DaemonSet will constantly hold a leader lock which makes it responsible for handling cluster-wide monitoring.

Code snippet 5.7 shows Metricbeat configuration yaml to send cluster node and pod metric data to elasticsearch.

Code Snippet 5.7: Metricbeat configuration yaml

```
metricbeatConfig:
  metricbeat.yml: |
    metricbeat.modules:
    - module: kubernetes
      metricsets:
        - container
        - node
        - pod
        - system
        - volume
      period: 10s
      host: "${NODE_NAME}"
      hosts: ["https://${NODE_NAME}:10250"]
---
output.elasticsearch:
      hosts: '${ELASTICSEARCH_HOSTS:elasticsearch-master.elk.svc.
      cluster.local:9200}'
```

5.2 Improved User Experience

5.2.1 Blue Ocean

The world has moved on from developer tools that are purely functional to developer tools being part of a "developer experience". It is no longer about a single tool but many tools developers use throughout the day and how they work together to achieve a workflow that is beneficial for the developer. Developer tools companies like Heroku, Atlassian and Github have raised the bar for what is considered good developer experience, and developers are increasingly expecting exceptional design. In recent years, developers have become more attracted to tools that are not only functional but are designed to fit into their workflow seamlessly and are easy to use. This shift represents a higher standard of design and user experience. Creating and visualising CD pipelines is something valuable for many jenkins users and this is demonstrated in the plugins that the jenkins community has created to meet their needs. This indicates a need to revisit how jenkins currently expresses these concepts and consider delivery pipelines as a central theme to the jenkins user experience. Blueocean enriches the user experience of jenkins with a complete execution statuses of the pipeline stages[2].

In our current Implementation, the jenkins server is integrated with blueocean to track different pipeline projects that are created on jenkins. Some of its features include:

- **Sophisticated visualizations** of continuous delivery (CD) pipelines. Users of the system can visualize different stages of the pipeline and the statuses of the stages.
- **Pipeline editor** makes creation of pipelines approachable by guiding the user through an intuitive and visual process to create a pipeline.
- **Debugging issues through logs** If there are multiple services that are run parallel within a stage, Blue Ocean makes it easier to debug the builds of individual services and restart them if necessary.

[2] `https://www.jenkins.io/doc/book/blueocean/pipeline-run-details/`

5.3 Complete Architecture of Continuous Integration and Delivery Pipeline

The architecture consists of jenkins automation server, which performs the continuous integration and delivery. At the beginning, the project on GitLab is integrated with jenkins. When the developer commits or merges the code to the project repository, GitLab triggers the corresponding project build in jenkins. Later, jenkins delegates an agent to execute the pipeline commands in stages. The agent checks out the latest version of the project from GitLab, runs automated test cases, builds docker images and deploys the updated images to the application environment. Figure 5.4 shows the complete architecture of the CI/CD pipeline following a code commit or merge to the project repository.

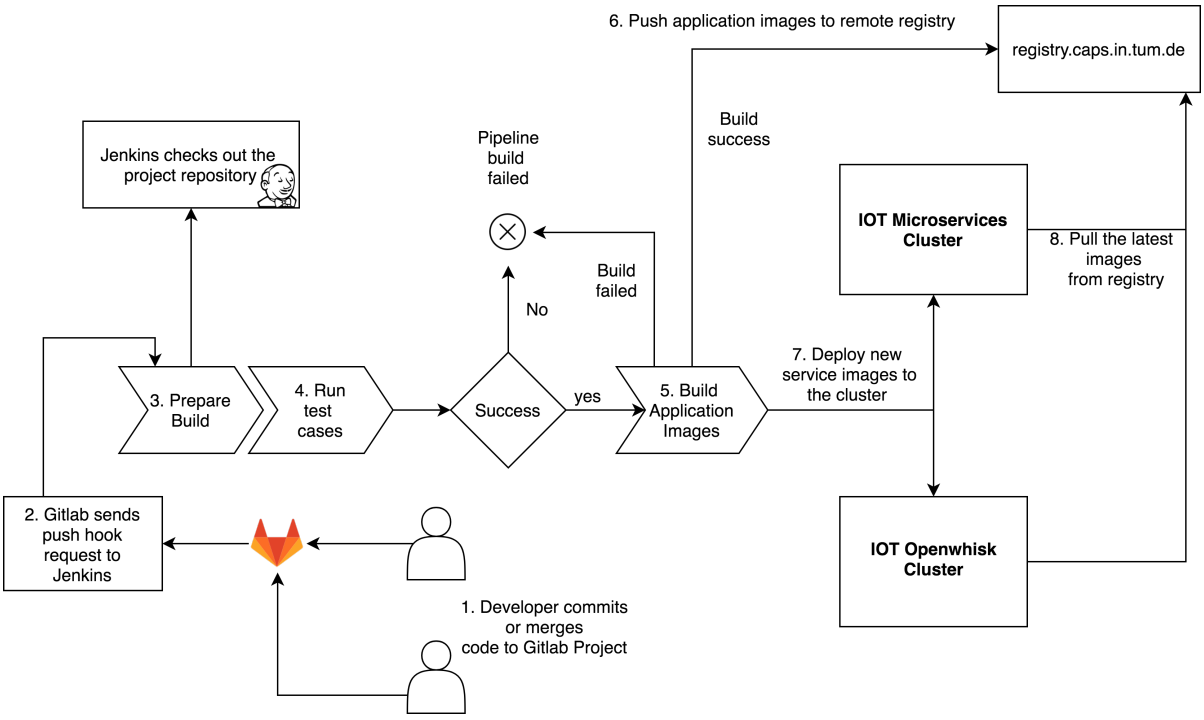

Figure 5.4: Complete architecture of Continuous Integration and Delivery pipeline

6 Results

This chapters provides evaluation of implemented solution. The first part of it describes how our current implementation reduces the total pipeline execution time of the IoT application and later part introduces distributed tracing through OpenTelemetry.

6.1 Parallelized Jenkins Pipeline

Pipelines are one of the most powerful tools jenkins has to offer and a standard for building jobs. The problem is that it can be so powerful that the jobs end up running many different things and it's easy for run times to increase dramatically. It can also become increasingly difficult to maintain a fast CI feedback cycle while running all the quality tasks such as building and running test cases. For instance, when there is a test case failure in one of the services of the application, the time to get the feedback from the CI system takes longer because the pipeline build is run sequentially.

A stage in a jenkins pipeline is set of commands that are executed to achieve a desired result. For example, In the CI/CD pipeline, building and pushing the image to a private registry is considered as a stage. The following things were taken into consideration while building the parallelized pipeline:

1. *How do we decide which stages can run in parallel?*
   We identified the stages that must run sequentially based on whether a stage build depends on another stage. There are two different examples. 1) If a global stage like checkout fails, then the rest of the pipeline will not run 2) If there are two different services that are not dependent on each other, we grouped them in stages so their builds can run in parallel.

2. *How do we group the pipeline stage commands?*
   We grouped the project build commands and docker push to private registry commands of a service within a stage.

3. *When does the new image deployment happen?*
   Deploying the application to the cluster happens once all the parallel stage are executed successfully.

Code snippet 6.1 shows an example of declarative pipeline syntax to execute stages in parallel and Figure 6.1 shows the visualization of stages in parallel

Code Snippet 6.1: Jenkinsfile declarative pipeline syntax in groovy

```
stages {
        stage('Build IOT') {
           parallel {
                stage('Build IOT Microservice') {
                    agent any
                    steps {
                      echo 'IOT Microservice'
                    }

                }
                stage('Build IOT Openwhisk') {
                    agent any
                    steps {
                      echo 'IOT Openwhisk actions'
                    }
                }
            }
        }
}
```

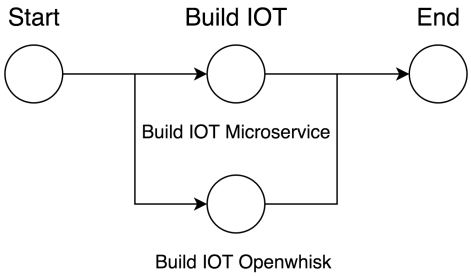

Figure 6.1: Parallel execution of stages

6.1.1 Improved pipeline build time

Through parallel execution of stages, we reduced the CI/CD pipeline execution time by **34%** with IoT Microservices and **28.4%** with IoT OpenWhisk application. Fig 6.2 shows the improved execution time (in seconds) observed with both Microservices and serverless application of the IoT platform.

Table 6.1: CI/CD execution time (in seconds) in IoT Microservices application.

Execution	**Code Checkout (s)**	**Build images(s)**	**Deploy(s)**	**Total(s)**
Sequential	30	955	350	1335
Parallel	30	497	350	877

Table 6.2: CI/CD execution time (in seconds) in IoT OpenWhisk serverless application.

Execution	**Code Checkout (s)**	**Build images(s)**	**Deploy(s)**	**Total(s)**
Sequential	43	1035	15	1093
Parallel	43	724	15	782

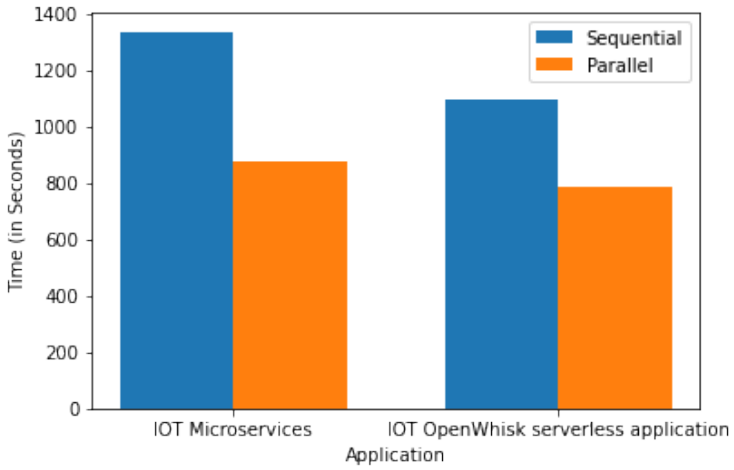

Figure 6.2: Build time observed between sequential and parallel execution of stages

6.1.2 Immediate feedback

Because the stages are run in parallel, developers receive immediate feedback about the build execution status. When jenkins encounters an issue within a specific stage, the build agent terminates the other sequential stages of the pipeline. Additionally, blueocean pipeline run detail view highlights the failed stages with the complete log information.

6.1.3 Troubleshooting

Classic jenkins pipeline view isn't the best option to view the failed stages and even less when the stages are executed in parallel as each stage is a different thread. Blueocean pipeline run detail view helps to quickly navigate to the failed stage and shows the complete log information for troubleshooting.

6.2 Distributed Tracing through OpenTelemetry

Through distributed tracing, it is easy to detect increased build execution time within the pipeline stage. In our current implementation, jenkins open telemetry integration helps developers and other maintainers to intuitively visualize the execution time of each command within a specific stage. Figure 6.2 below shows an example of opentelemtry tracing showing the trace and spans[1] of a job in jenkins.

[1] `https://opentelemetry.lightstep.com/spans/`

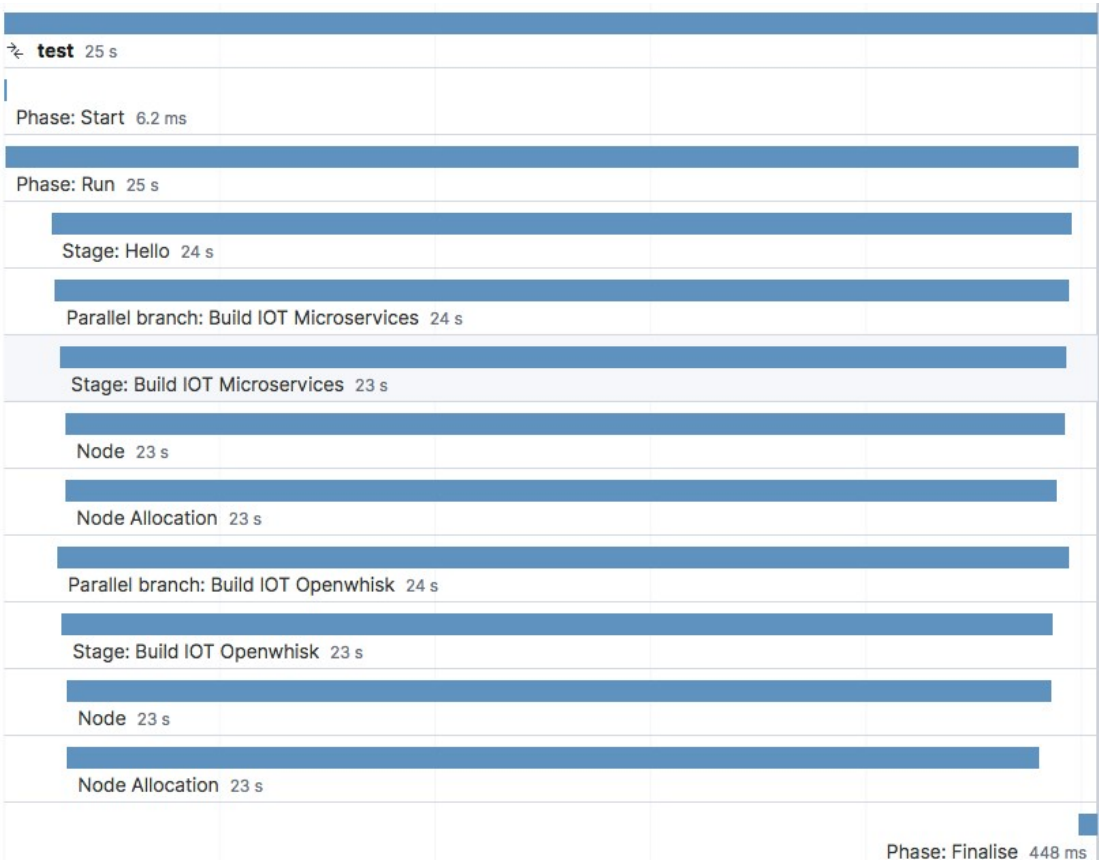

Figure 6.3: OpenTelemetry tracing of an example jenkins job

7 Discussion

This chapters provides answers to our research questions and other implications based on the parallel and sequential execution of stages in the jenkins pipeline.

RQ1 What is the state of the art in DevOps, CI / CD approaches in the serverless / OpenWhisk environment?

As a part of this research, two different approaches were evaluated, traditional agent builds and containerized agent builds. In our current implementation, we adopted containerized builds through docker because it eliminates manual configuration of agents and all the required project dependencies can be packaged into the agent image. Moreover, these container agents can be managed through kubernetes to run build agents only on demand. Hence, it is not necessary to keep agent server available at all times. Many organizations have adopted this approach of deploying applications through containerized agents. For example, Gitlab CI runs CI/CD jobs in docker containers.[1]

RQ2 How to structure the codebase functions and the best Branching strategies for CI/CD?

We have identified three different branching models as a part of this research

- Gitflow
- GitHub Flow
- GitLab Flow

7.1 Gitflow

Gitflow [36] model suggests a main branch and a separate develop branch, as well as supporting branches for features, releases, and hotfixes. The developer checks out the code and creates a feature branch. After local development and testing the changes from the feature branch are committed into the repository. It triggers the build process in jenkins which runs the code analysis and unit tests. Code analysis can be done with ESLint[2] tool. Unit tests are run only if code analysis step did not return any errors. After feature branch build has been succeeded a developer can create a merge request to the develop branch. Merge request

[1] `https://docs.gitlab.com/ee/ci/docker/using_docker_images.html`
[2] `https://eslint.org/`

triggers the build in develop branch. The changes in develop branch can be committed to the release branch. It again runs the build but this time it triggers the end-to-end tests automatically. The reason is that commits into release branch are done less often than to the develop branch. The assumption is that develop branch will receive five merge requests per day and release branch only one merge request per two weeks. Committing into master branch is also done through the merge request and is followed by the build process. Deploy of the release and production versions is done after deployment decision which is based on release plan made by product management. The development happens on the develop branch, moves to a release branch, and is finally merged into the main branch.

Although Git flow has a well-defined structure, there are a few problems with it. First, large features can spend days merging and resolving conflicts and can trigger multiple testing cycles, which can slow things down if the project requires quick iterations. Second, the complexity introduced with hotfix and release branches. These branches can be a good option for some organizations, but it comes with maintenance overhead. Continuous delivery removes the need for hotfix and release branches and deployment can be done from the default main branch.

7.2 GitHub flow

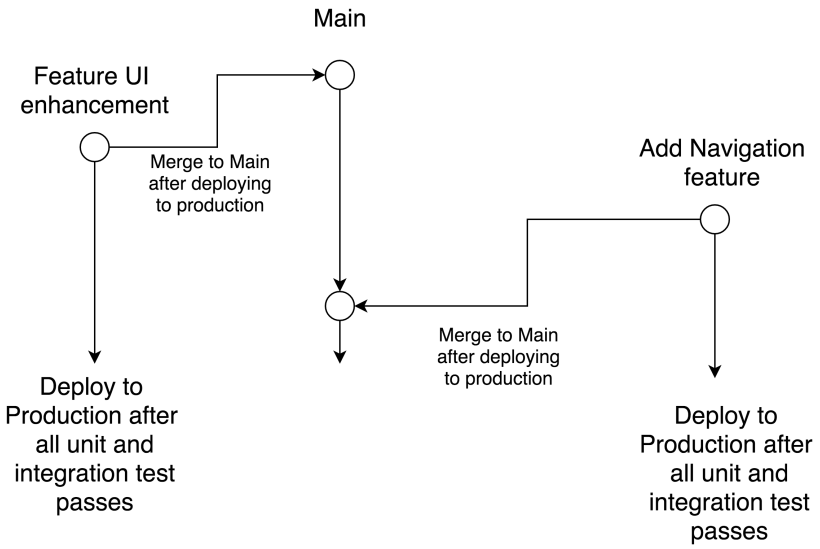

Figure 7.1: Github flow branching

GitHub flow [37] has only feature branches and a main branch. This flow is clean and straightforward, and many organizations have adopted it with great success. Merging everything into the main branch and frequently deploying minimizes the amount of unreleased code. This approach is in line with lean and continuous delivery best practices. Although it eliminates the overhead of managing multiple branches, this method deploys to production servers from the feature branch and later merges feature branch to main branch. This approach could break things on production if multiple features are being released to production simultaneously.

7.3 GitLab flow

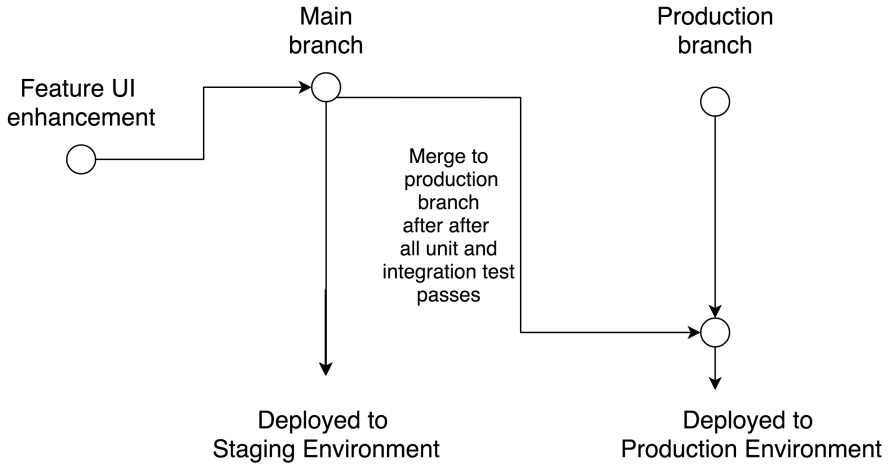

Figure 7.2: Gitlab flow branching

Gitlab Flow [38] introduces the environment branches suitable for deploying changes in branches to specific environments. This method considers two branches namely main and production tracking two different environments, staging and production.

- Main branch changes are deployed to staging environment.
- Production branch changes are deployed to production environment.

In this case, the changes from the main branch is deployed to staging. To deploy to production, a merge request is created from the main branch to the production branch. This workflow, where commits only flow downstream, ensures that everything is tested in all environments. If a hotfix is required, this method suggests to develop it on a feature branch and merge it into main with a merge request. If main passes automatic testing, then merge the

feature branch into the other branches. If this is not possible because more manual testing is required, you can send merge requests from the feature branch to the downstream branches.

GitLab flow helps developers prevent the overhead of releasing, tagging, and merging that happens with Git flow and it adopts the strategy of deployment from main branch instead of feature branch. This approach is in line with lean and continuous delivery best practices.

RQ3 How to optimize the build times of jobs that are built by CI/CD agents?

Through parallelized jenkins pipleine builds, we achieved improvements in the total build time of the IoT application. However, there are a few implications with the performance based on the experimental results.

Jenkins agent performance

In our current implementation we had one pipeline job that can run stages in parallel. When there are multiple parallelized jobs that run simultaneously, there is a possibility of build failure if the system is unable to handle the CPU and memory intensive operations. This problem could be addressed by either introducing additional worker nodes or scaling up the existing system with additional resources.
Figure 7.3 and 7.4 shows CPU utilization in the jenkins worker node when job stages are executed in parallel and sequence.

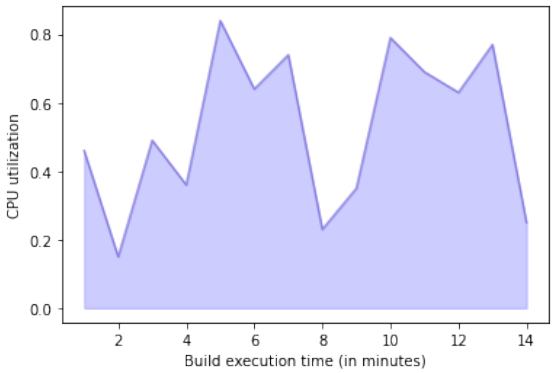

Figure 7.3: CPU utilization in parallel execution of stages

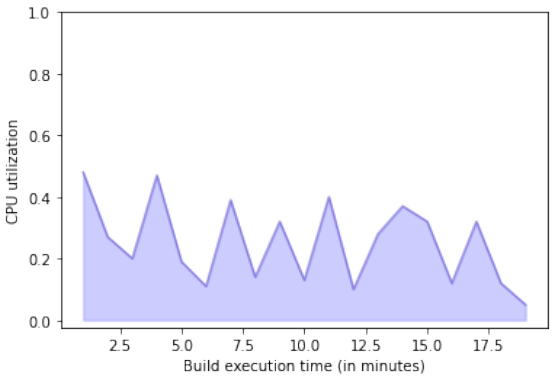

Figure 7.4: CPU utilization in sequential execution of stages

Although there is a significant difference observed with CPU utilization during the build execution, the memory consumption remained relatively similar between the two types of execution.Figure 7.5 and 7.6 shows Memory consumption in the jenkins worker node when job stages are executed in parallel and sequence.

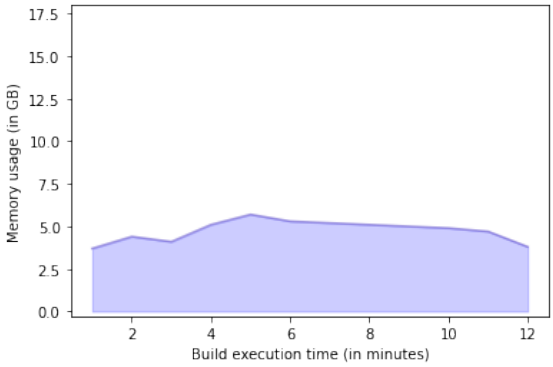

Figure 7.5: Memory utilization in parallel execution of stages

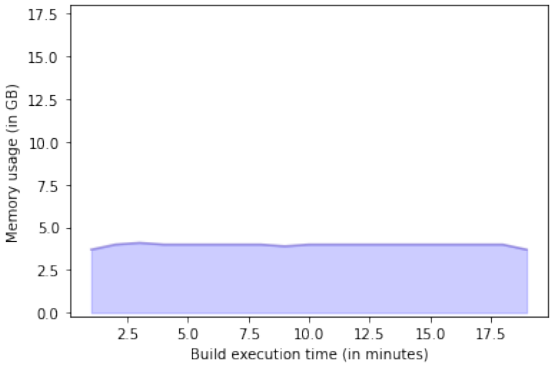

Figure 7.6: Memory utilization in sequential execution of stages

8 Future Scope

CI/CD pipelines have long played a major role in speeding up the development and deployment of cloud-native apps. Cloud services such as AWS lend themselves to more agile deployment through the services they offer as well as approaches such as Infrastructure as Code. While majority of development teams have streamlined their pipelines to take full advantage of cloud-native features, there are still ways to refine CI/CD even further.
Possible topics for the future research are:

Security in CI/CD pipeline (DevSecOps)

Security testing could be performed along with other testing methods such as unit and integration to identify security vulnerabilities.

Code Analysis

Code analysis tools help to keep an eye on the code that is used within the pipeline. Static analysis security testing [SAST] and Dynamic analysis security testing [DAST][1] are some of the modern techniques that could be researched to detect security vulnerabilities in the code.

Open source plugin vulnerabilities

A lot of technological tools are used with CI/CD pipelines, but there are a few vulnerabilities that could open the door for cyber threats and other malicious attacks. For instance, jenkins an open sourced CI/CD solution for many software projects has a plethora of plugins developed by its community. However, some of its plugins are not maintained leading to security vulnerabilities that could potentially be exploited by the attackers.

Serverless CI/CD

Serverless CI/CD lowers the overhead cost of maintaining a build server at all times. With many cloud services offering pay-per-use pricing structure of serverless functions, it can be cost-effective if there are CI/CD build servers running on multiple environments such as staging and production.

[1] `https://www.atlassian.com/continuous-delivery/principles/devsecops`

9 Conclusion

The aim of this thesis was to implement CI/CD solution for both Microservices and Serverless applications of the IoT platform, research the best practices to implement the containerized pipeline, optimize the build execution time of the CI/CD agents and research the best branching strategies that can be adopted for our pipeline. Alongside the build automation, we have achieved monitoring of jenkins server and agents to detect any anomalies and visualize the build execution through Opentelemetry tracing. The presented approach and tool could be widely used in the development and deployment of cloud-based services for various use cases. Furthermore, the CI/CD tool presented in this thesis could be cloned to other environments since the solution exists as Configuration as Code (Jcasc). The research carried out as a part of this thesis shows that parallelized pipeline build in jenkins improves the overall build execution time. However, running multiple parallelized agent builds could overload the CPU and exceed the memory requirements. In the end, parallelizing is a good option to improve the overall development experience, utilizing jenkins build agents to its fullest capacities.

List of Figures

List of Tables

List of Code Snippets

Bibliography

[1] N. D. Fogelström, T. Gorschek, M. Svahnberg, and P. Olsson. “The impact of agile principles on market-driven software product development”. In: *Journal of Software Maintenance and Evolution: Research and Practice* 22.1 (2010), pp. 53–80.

[2] H. H. Olsson, H. Alahyari, and J. Bosch. “Climbing the "stairway to heaven"-a multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software”. In: *Proc. EUROMICRO conference* (2012), pp. 392–399.

[3] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. 1st ed. Addison-Wesley Professional, 2010.

[4] M. Leppanen, S. Makinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mantyla, and T. Mannisto. “The Highways and Country Roads to Continuous Deployment”. In: *IEEE Software* 32.2 (2015), pp. 64–72.

[5] A. A. U. Rahman, E. Helms, L. Williams, and C. Parnin. “Synthesizing Continuous Deployment Practices Used in Software Development”. In: *Agile Conference (AGILE)* (2015), pp. 1–10.

[6] Perforce. *Perforce. 2017. Continuous Delivery: The New Normal for Software Development*. URL: `https://www.perforce.com/sites/default/files/files/2017-09/continuous-delivery-report.pdf`. (accessed: 09.06.2021).

[7] Puppet. *Puppet. 2017. 2017 State of DevOps Report*. URL: `https://puppet.com/resources/whitepaper/state-of-devops-report`. (accessed: 09.06.2021).

[8] L. Chen. “Continuous delivery: Huge benefits, but challenges too.” In: *IEEE Software* 32.2 (2015), pp. 50–54.

[9] C. Rossi, E. Shibley, S. Su, K. Beck, T. Savor, and M. Stumm. “Continuous deployment of mobile software at facebook (showcase).” In: *International Symposium on Foundations of Software Engineering (FSE). ACM* (2016), pp. 12–23.

[10] C. Vassallo, F. Zampetti, D. Romano, M. Beller, A. Panichella, M. D. Penta, and A. Zaidman. “Continuous delivery practices in a large financial organization. ” In: *International Conference on Software Maintenance and Evolution (ICSME). IEEE* (2016), pp. 519–528.

[11] Portworx. *Portworx. 2017. 2017 Annual Container Adoption Survey: Huge Growth in Containers.* URL: `https://portworx.com/2017-container-adoption-survey/`. (accessed: 09.06.2021).

[12] Y. Zhang, B. Vasilescu, H. Wang, and V. Filkov. "One Size Does Not Fit All: An Empirical Study of Containerized Continuous Deployment Workflows". In: *Association for Computing Machinery* (2018), pp. 9–10. URL: https://dl.acm.org/doi/10.1145/3236024.3236033.

[13] AWS. *Serverless CI/CD for the Enterprise on AWS*. URL: https://aws.amazon.com/quickstart/architecture/serverless-cicd-for-enterprise/. (accessed: 09.06.2021).

[14] DZone. *Serverless CI/CD on the AWS Cloud*. URL: https://dzone.com/articles/serverless-cicd-on-the-aws-cloud. (accessed: 09.06.2021).

[15] V. Podolskiy, Y. Ramirez, A. Yenel, S. Mohyuddin, H. Uyumaz, A. N. Uysal, M. Assali, S. Drugalev, M. Gerndt, M. Friessnig, and A. Myasnichenko. "Practical Education in IoT through Collaborative Work on Open-Source Projects with Industry and Entrepreneurial Organizations". In: *2018 IEEE Frontiers in Education Conference (FIE). doi: 10.1109/FIE.2018.8658377* (2018), pp. 1–9.

[16] *TravisCI*. 2021. URL: https://travis-ci.com/plans. (accessed: 09.06.2021).

[17] *GitLab*. 2021. URL: https://about.gitlab.com/pricing/. (accessed: 09.06.2021).

[18] *TeamCity*. 2021. URL: https://www.jetbrains.com/teamcity/. (accessed: 09.06.2021).

[19] *DroneCI*. 2021. URL: https://docs.drone.io/enterprise/%5C#is-drone-enterprise-free-for-students. (accessed: 09.06.2021).

[20] Wells. *Serverless technology use cases.* Cloud Academy, 2017. URL: https://www.youtube.com/watch?v=9vBNlJx_h_o. (accessed: 09.06.2021).

[21] Adzic and Chatley. *Serverless computing: economic and architectural impact*. 2017, pp. 884–889.

[22] Sbarski and Kroonenburg. *Serverless Architectures on AWS: With examples using AWS Lambda.* Manning Publications Company, 2017.

[23] AWS. *Success in the pitch room starts with serverless.* AWS, 2021. URL: https://aws.amazon.com/startups/modern-application-development/. (accessed: 09.06.2021).

[24] AWS. *Amazon cloud watch.* AWS. URL: https://aws.amazon.com/cloudwatch. (accessed: 09.06.2021).

[25] Azure. *Azure Monitor overview*. Microsoft Azure, 2019. URL: https://docs.microsoft.com/en-us/azure/azure-monitor/overview. (accessed: 09.06.2021).

[26] M. Sahin. "GitOps basiertes Continuous Delivery für Serverless Anwendungen". In: (2019), pp. 44–59. URL: https://elib.uni-stuttgart.de/bitstream/11682/10332/1/MA_MuesluemSahin.pdf.

[27] V. Ivanov. "Implementation of DevOps pipeline for Serverless Applications". In: (2018), pp. 44–65. URL: https://aaltodoc.aalto.fi/bitstream/handle/123456789/32432/master_Ivanov_Vitalii_2018.pdf?sequence=1.

[28] Lwakatare, Kuvaja, and Oivo. "Dimensions of devops." In: *International Conference on Agile Software Development, Springer* (2015), pp. 212–217.

[29] T. Rehemägi. *Benefits of Serverless technology*. Dashbird. URL: https://dashbird.io/blog/business-benefits-of-serverless/. (accessed: 09.06.2021).

[30] Becker. *A production-grade ci/cd pipeline for serverless applications*. Medium. URL: https://medium.com/@tarekbecker/a-production-grade-cicd-pipeline-for-serverless-applications-888668bcfe04. (accessed: 09.06.2021).

[31] Elastic. *Elastic Stack*. Elastic. URL: https://www.elastic.co/elastic-stack. (accessed: 09.06.2021).

[32] Elastic. *Elastic Filebeat*. Elastic. URL: https://www.elastic.co/beats/filebeat. (accessed: 09.06.2021).

[33] OpenTelemetry. *OpenTelemetry*. OpenTelemetry. URL: https://opentelemetry.io/docs/collector/. (accessed: 09.06.2021).

[34] Elastic. *Elastic APM*. Elastic. URL: https://www.elastic.co/apm. (accessed: 09.06.2021).

[35] Elastic. *Elastic Metricbeat*. Elastic. URL: https://www.elastic.co/beats/metricbeat. (accessed: 09.06.2021).

[36] V. Driessen. *A successful Git branching model*. URL: https://nvie.com/posts/a-successful-git-branching-model/. (accessed: 09.06.2021).

[37] GitHub. *Understanding the GitHub flow*. URL: https://guides.github.com/introduction/flow/. (accessed: 09.06.2021).

[38] GitLab. *Introduction to GitLab Flow*. URL: https://docs.gitlab.com/ee/topics/gitlab_flow.html#environment-branches-with-gitlab-flow. (accessed: 09.06.2021).