'Continuous Benchmarking' in DevOps to support Quality of Deployments using Amazon Web Services

MSc Research Project
Cloud Computing

Paris Moore
Student ID: X14485758

School of Computing
National College of Ireland

Supervisor: Horacio Gonzalez-Velez

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Paris Moore
Student ID:	X14485758
Programme:	Cloud Computing
Year:	2021-22
Module:	MSc Research Project
Supervisor:	Horacio Gonzalez-Velez
Submission Due Date:	15th Aug 2022
Project Title:	'Continuous Benchmarking' in DevOps to support Quality of Deployments using Amazon Web Services
Word Count:	8082
Page Count:	20

Signature:	Paris Moore
Date:	15th September 2022

’Continuous Benchmarking’ in DevOps to support Quality of Deployments using Amazon Web Services

Paris Moore
X14485758

Abstract

DevOps automation is becoming cloud-centric with most public and private cloud computing providers supporting DevOps systemically on their platform, including Continuous Integration and Continuous Deployment (CI/CD) tools [1]. This has solved many issues with distributed complexity for testing, deployment, and production. In recent years, we have seen a focus on integrating security as a core function within the pipeline. However, the overall quality of deployment (QoD) is still being ignored and often only becomes a concern after an issue has occured. This creates a bottleneck for developers when trying to adhere to release standards without tools in place to support these goals during deployment. This research project seeks to bridge this gap by designing a modern CI/CD pipeline that encapsulates the concept of ’Continuous Benchmarking’ (CB) using Amazon Web Services (AWS). The objective of these benchmarks is to ensure QoD goals are continually and automatically met throughout each release. Two benchmarking solutions are presented, one which puts fixed thresholds in place to determine when a release has been justified, and the other which checks to ensure the deployment is on trend with previous releases. The results show that CB can be a very effective solution at maintaining the quality of deployments without negatively impacting the performance of the pipeline.

1 Introduction

Many enterprises are trying to rely less on people to manage their software and more on the technology to look after itself. The same applies for the continuous delivery of new changes to their platform. Software developers are trying to automate business as usual (BAU) processes to reduce operational overheads, so more time can be spent on development. With the introduction of cloud computing, applications no longer need to be taken offline for deployments. The days have passed when software teams would schedule monthly deployments and package up multiple changes into one release. This often required rigorous rounds of testing to try and mitigate against any negative impact to the production system, with a full team on hand to support the deployment. However, in recent years the concept of CI/CD has advanced exponentially. Nowadays, many tasks can be automated, making these releases less timely and risky. Often used by DevOps teams, the CI/CD pipeline is a method that builds, tests and deploys code by using automation. Even though the cloud has greatly simplified the applications provisioning process, several challenges exist in the area of deployment automation management.

One of the key questions this research will set out to answer is; **"Could using Continuous Benchmarking, at core stages within the delivery pipeline, ensure the Quality of Deployment of cloud applications, without negatively impacting the pipelines performance?"**. In contrast to Monitoring, Benchmarking requires a very high degree of control over a system to make results reproducible. Monitoring is about non-intrusive and passive observation of a production system. Benchmarking typically happens in a non-production environment, and evaluates the artifacts being deployed to determine Quality of Service (QoS) while complying with various general benchmark requirements [2]. Past literature's have reviewed performance, using industry benchmarking standards such as ISO/IEC 9126 [3][4]. Proposed solutions include developing complex isolated systems to run benchmarks during the CI/CD phase [5]. However, these solutions are developed with the objective to maintain service levels of the application that is being deployed, and do not consider the framework for which these applications are deployed through.

This research evaluates benchmarking in respect to an applications delivery process and how quality of deployments can be at least maintained, if not improved, during every release. The QoD is a measurement of the performance, agility and velocity offered by the CI/CD pipeline. The solution designed is agnostic to the application that is being deployed, producing a CB approach that is generic enough to be adapted for any cloud application. Furthermore, CB as a concept in DevOps does not need to be a complex process for organisations to adapt and can be applied at different stages in the pipeline, supporting a variety of use cases. Presenting two benchmarking approaches, the results show that CB in CI/CD allows a pattern of having a continuous loop from past deployments to the beginning of a future deployment cycle that exposes any performance losses in real time.

1.1 Research Objectives

This research proposes to add benchmarking at different stages within the CI/CD pipeline of cloud applications. This way, developers can assert that a new release is at least as good as the previous release and that it complies with release goals and benchmarks that may have been set by the team. In this regard, we make the following contributions:

1. We describe how release requirements can be integrated into the delivery process and how benchmarking can be used to enforce QoD goals by enforcing the early detection and repair of performance issues.

2. We present a proof-of-concept pipeline prototype, designed and built using AWS, which automatically deploys the system release, runs one or more benchmarks, collects and analyzes results, and decides whether the release fulfils predefined QoD goals.

2 Related Work

The research on DevOps principles and best practices is trending in recent years. Research studies on ways to support and maintain the quality of the application [6] is extensive and today's state of the art pipelines have began focusing on functional tests or small microbenchmarks [5] while the overall quality of the deployment is an afterthought.

2.1 DevOps Methodologies

As time has gone on and technology has continued to advanced, we have seen the introduction of testing and monitoring tools being integrated into the deployment framework, with security being a very trendy topic in the DevOps domain most recently. This research [7] discusses how manual security processes as an afterthought are a bottleneck for many companies attempting to implement a CD pipeline, and reviews the concept of 'Continuous Security' in a DevOps infrastructure environment. The results from the experiments show that the addition of 'continuous security' in pipelines does not outweigh the benefits. However, the components being checked by the security tool are not clearly understood from reviewing this authors paper. The concept of 'continuous security' lacked foundation in the prototype demonstrated, and performance is the only metric reviewed in the experiments. A review of this concept with more emphasis on the security concerns surrounding cloud applications today and how they can be managed within CI/CD would be interesting future work.

According to a study conducted on software practitioners, DevOps practices can be can be crystallised into five dimensions, all of which are; Collaboration, Automation, Culture, Monitoring and Measurement [8]. In DevOps, the most common goal according to practitioners was to reduce response time and provide fast deployment of high-quality and reliable software products and services [8]. Collaboration and Automation are very dominant topics within this paper that attribute to the findings presented. The primary data gathered for the study was conducted on two companies who were only introducing DevOps practices. For this reason, the study did not go into depth on certain topics and concentrated on textbook practices. The feedback from the interviewee's was therefore direct to those topics as their knowledge and understanding at the time was limited.

The efficiency of container-based microservice-style DevOps is compared to the traditional VM-based approach, while quantifying the scalability of both the stateless and stateful containerized components using OpenStack technology [9]. This approach proved to greatly improve operational efficiency but noted that container access limitations impacted collaboration. This 2017 article [10] comprehensively examines PaaS solutions and cutting-edge solutions across various tiers of cloud-computing that are leaning towards a full-stack DevOps environment. This included cloud computing giants; Google, AWS and Microsoft Azure. At the time of the study, containerized software were yet to be fully established into the market. Virtual Machines were a key component when designing a full-stack DevOps environment. Interestingly, the author formed the opinion that "Container is unlikely to grab the entire market of VM", which in my opinion is the exact opposite to what is currently happening five years later. Apache Mesos is the preferred tool of choice over Docker and Kubernetes when it comes to 'openness' and resource management of task scheduling, which again has shifted as time has gone on.

Following DevOps best practices, an automated deployment pipeline is presented using AWS, Jenkins, Ansible and Kubernetes [8] . Employing Ansible into the CD process allowed the application to send commands to Kubernetes to achieve better scalability overall. The solution proposed is concentrated on automating resource management for Java based web applications. Future work could involve extending this solution across several application frameworks on public and private clouds to ensure the solution is dynamic enough to still achieve zero downtown, as well as autoscaling capabilities without the need for any additional operational overheads. Although the automation capabilities have been achieved through the use of cloud services, performance of the overall delivery framework were not discussed.

This author [11] introduces a template-based pipeline approach using GitLab that can support developers to use the DevOps environment efficiently when deploying microservices, ensuring adherence to security principles to protect against attacks. The template-based approach simplified the introduction of technologies such as Docker and Kubernetes, thus freeing developers from the responsibility of creating the relevant artifacts and delivering them to the orchestration system.

Another study [12] reviews the habit of branching in CI and what the effects of streamlining this practice can have on both developers and customers, "smaller changes, shipped to production quickly, are a lot easier to debug when something breaks". The attention on CI solely as a core function emphasises the age of the literature (2014). The idea of automating the CI process with continuous deployment is only imaginable, "What if this could even become part of your build process? Imagine shipping all new code directly to the staging system for instant testing!". Now its a reality, and reviewing a paper that zooms in on CI when the movement was only beginning, was an interesting reflection on what expectations looked like then versus how they are today.

2.2 Benchmarking in CI/CD

A prototype [5] is designed which includes a dedicated benchmarking step in the build process, that can run one or more benchmarks, collect and analyze the results, and make a decision whether to proceed to the next step in the release. This ensures QoS goals of the application are met and every release is at least as "good" as the last. The prototype is developed and evaluated using Jenkins, YCSB and Cassandra's commit history for the last two years. The benchmarks are implemented in Java and Unix shell scripts were used to run the benchmarks on Amazon's EC2 instances. The Benchmarking system runs for at least 20-30 minutes and multiple benchmarks can be ran on an application in parallel. Then once all benchmarks are complete, there is an analysis stage before a decision is made whether the current build is released to the deployment pipeline or whether the process is aborted because QoS levels were not met. The analysis stage involves distributing the collected data across multiple machines for calculation. The benchmark scripts are based on predefined metrics which were a combination of fixed values, trend and jump detection (i.e. sudden massive change in quality). The results show the CB tool was responsive and rejected builds for viable reasons. A key takeaway from reviewing this papers solution to CB is the compute and load time to run the benchmarks seemed extensive and complex to setup.

This paper [10] presents an automated performance evaluation framework for high performance computing (HPC) that enables an automated workflow for testing and performance evaluation of software libraries. The solution is generic enough to work with any software project and the benchmarks are ran within the CI framework of the eco system. Once code is merged to master branch, the benchmark tests kick off and data is then retrieved for analysis and output on a web application which displays the results of the tests. It is then up to the users to make a decision, based on the results, whether to continue onto the continuous development stage of the pipeline. The complexity of the benchmarking on the HPC system is impressive but the CI/CD lifecycle lacks independence due to the reliance on manual intervention to make a decision to proceed after the benchmarks have ran.

Benchmarking the performance of microservice applications is discussed by the same author in two separate papers, published months apart [13] [14]. Both present solutions that show how benchmarking microservice applications can be achieved with little manual effort. The first paper [13] presents a solution based on a single microservice application. Whereas, the latter paper [14], goes into more depth on the applications requirements and proposes a solution that considers the need to have capabilities necessary for benchmarking entire microservice applications, especially the ability to resolve complex data dependencies across microservice endpoints. Both papers discuss performance specific benchmarking on microservices and how this can be achieved through pattern based approaches. The deployment goals of the overall system is not considered. Nonetheless, the topic in question is thoroughly evaluated and provides good insight into benchmarking specifically on microservice based applications.

2.3 Research Niche

Benchmarking “is the process of measuring quality and collecting information on system states”[14]. As reviewed in others work, the word ’system’ frequently referred to the application that the pipeline is built for. Otherwise, ’system’ referred to independent benchmarking solutions to perform benchmarks on applications during the CI/CD stages. Oftentimes having a separate system to perform such tasks introduces additional overheads for teams, on the contrary to what DevOps practices intend to achieve. Much like solutions designed in [5][10], the benchmarking component requires a lot of compute to process the data and return a suitable result. There does not appear to be a solution whereby the benchmarking process is streamlined as sequences inside the delivery framework during deployments.

What became clear from reviewing related work is the importance of application monitoring in DevOps. But even more so, how benchmarking is a very different and still being established [15]. Benchmarking creates the ability to readjust ’benchmarks’ based on the systems metric baseline, which is ever evolving as new changes are released. The metrics baseline consists of data collected in previous releases and can be used to set a goal and try to determine if trends show the likelihood of meeting that goal. They become an essential piece of a key performance indicator (KPI). This was an interesting comparison and highlighted key requirements when designing benchmark solutions for CI/CD pipelines, that related work had not always fully established in their design.

In the past, the distributed nature of some enterprise systems didn't fit well with centralized software deployment. Using a cloud platform solves many issues with distributed complexity [1]. It has been observed that common tools such as Jenkins and Maven are often the first choice in CI/CD. For example, this paper [16] reviews CI/CD tools but concentrates only on Jenkins and GitLab for CI services. Up and coming tools and services that tech giants such as Amazon offer, are still being overlooked, despite offering systematic solutions to CI/CD. This highlighted a gap whereby it is yet to be evaluated how using state of the art cloud services in CI/CD (and CB) can contribute to supporting DevOps team with deployments. Outlined in Table 2.3 is a list of integral tools and platforms used in related research studies on CI/CD.

Reference	**Tool/Platform**
[14][13]	Pattern Binder, OpenAPI, Kotlin
[17]	GINKGO, Git
[5]	Apache Cassandra, Jenkins, YCSB, AWS EC2
[7]	GitLab, Jenkins, Maven, JUnit, Docker
[18]	Git, Jenkins, Kubernetes, Java
[19]	Jenkins, JMeter, GitLab
[20]	Kubernetes, Tekton, GitHub, Kaniko

Table 1: Common CI/CD Tools used in Related Work

3 Methodology

This research project evaluates the benefits and performance impact, of introducing 'continuous benchmarking' as fundamental stages within the CI/CD application build and deployment pipeline.

3.1 Equipment and Technique

As previously mentioned, the research and solution that is being proposed is agnostic to the application being deployed. An application (app) is required for analysis purposes so a quantitative study can be performed on how the concept of CB can positively support software developers during CI/CD, and whether there is any pipeline performance loss as a result. To add complexity to the deployment process, a popular open source application with depth and substance is chosen; a Twitter Sentiment Analysis application [21]. The app is built on the python flask framework and machine learning models are used to perform the predictions. To further support this approach, the application will be containerized using docker and deployed using AWS Elastic Container Service (ECS).

To perform the benchmark experiments, the sentiment app is duplicated and saved three times; the first version being the default state, second version includes an additional python library and the third one including two additional libraries with some basic functionality using those libraries. The reason for this is to measure the performance of both the CB and pipeline release using 'lighter' and 'heavier' variations of the same application. The objective is to mimic a real world scenario whereby an application is being further developed with new components. This will allow us to monitor how the pipeline handles the growth of the application and how the benchmarks respond to this. Each variation of the application is deployed a number of times sequentially and the stats are noted for further analysis. Only when the first deployment completes, is the baseline of each metric known for the next deployment. It will take a number of deployments, depending on how the metrics are being calculated, before the CB stages can perform adequate analysis to measure and benchmark the deployment in progress. The objective is to show that as the application is improving, and growing, the benchmark can over time adapt and re-baselines in parallel, demonstrating the 'continuous' aspect to this benchmarking approach.

3.2 Continuous Benchmarking

Benchmarks are set throughout the pipeline, to ensure the quality of deployment is continually at standard. This helps ensure that the CI/CD process as a whole is always as fast, effective and reliable as its previous releases. Similar to CI and CD, we refer to this as Continuous Benchmarking (CB).

Data Availability: The sentiment analysis application requires that the pipeline can containerize the application during the build phase and deploy it to ECS during deployment. Secondary data is not being used so the pipeline had to be fully developed before data became available. Logging was configured at all key stages of the pipeline to ensure accurate monitoring for analysis later. To understand what data is available, a number of deployments were ran. The AWS CloudWatch logs for each service were analysed to determine suitable standards and metric baselines for benchmarking. Some logs, such as those from CodeBuild, were extensive and therefore difficult to parse. For that reason, extra steps were taken to extract the attributes of interest and send them to a table in DynamoDB for further analysis. This was achieved using a Lambda function written in python code using the Boto3 AWS SDK that supports querying data from other amazon services.

Trend Analysis: For comparable results, each application version was deployed in the same state in the environment over several cycles. Otherwise it would be impossible to carry out an accurate trend analysis to determine the baseline and appropriate benchmarks. Initially it was clear that the build state was the most influential on the outcome of the deployment. The data from CodeBuild was the most informative, which is potentially down to the complexity of this stage compared to others. Beside reviewing the log data, the behaviour of the pipeline during releases was observed. The second most influential stage of the deployment occurs before the release starts, which is when code changes are merged to the repository master branch. This is what triggers a release.

Benchmark Responses: Once a release has met the criteria defined by the benchmarks, a response needs to be sent back to the pipeline to say the benchmarks have passed and the release can 'continue' to the next stage of the pipeline. If the criteria is not met, a response needs to be sent to the pipeline to 'fail' the benchmarking stage, stopping the cycle and preventing the release from proceeding to the next stage.

3.3 Metrics

For this investigation, defining metrics to decide on the success, or failure, of a benchmark run is crucial. These decisions can be based on fixed values defined by DevOp teams, which will result in the benchmarks rejecting a release because of a sudden and significant drop/jump in QoD levels compared to the last release, or detect a negative trend over multiple deployments. Therefore, the following key metrics will be used in the Continuous Benchmarks:

- **Fixed values (FV)** is applying fixes thresholds to detect unqualified releases. A suitable use case is whereby the number features merged to master do not suffice, therefore a deployment should not take place until the desired threshold has been met.
- **Trend detection (TD)** is when the metric of current build mc must not exceed t percent more than the moving average of the previous b builds.

Similar to how Kumar et al [22] quantifies impact of including security in a CI/CD , it has been chosen to examine the impact on the velocity and agility of the pipeline with and without CB using the following key metrics:

- **Frequency of Deployment** measures the number of deployments in a given time-frame.
- **Mean Time of Deployment** calculates the average time to deploy a release to a production environment.
- **Mean Time to Change** is the time taken from when change is made to version control, to production.

Time will be he most common unit used to measure these metrics and the most efficient way of collecting meaningful data will be to compare the CI/CD pipeline by running experiments with and without CB capabilities integrated. Furthermore, performing analysis on different versions of the sentiment app (light, medium, heavy) will allow for distinguishable experiments with varying results to support findings.

3.4 Benchmarking Criteria

Benchmarking is to evaluate (something) by comparison with a standard. The standard in this regard are previous successful deployments. Quality and time are the dimensions explored in this research through experimental setups to analyse and evaluate ideal levels. The benchmarks decision should be based on previous trend data made available during the benchmarking stage. The purpose is to ensure that the QoD is as good as other ones. The benchmarks should be designed to be generic enough to work on any application, and prove that the CB concept can be applied to other pipeline infrastructures, not just AWS. The analysis should provide insight on what the CI/CD pipeline looks like with and without the CB tool in place. As seen in related work, benchmarking solutions tend to be independent systems and for that reason, are unattractive to DevOps teams whose aim is to automate BAU processes as much as possible. For this reason, the CB process must be streamlined within the CI/CD pipeline and independently carry out the analysis before returning a result.

4 Design Specification

To support the research, a proof of concept has been designed and developed using Amazon Web Services. The techniques and architecture that underlie the implementation and the associated requirements are identified and presented in this section.

4.1 Technologies

Past research articles have achieved a CI/CD pipeline using Jenkins software as the 'backbone' technology [5][7][19]. Although Jenkins is well adapted across the industry as a reliable DevOps tool, it is quite heavy and time consuming, especially to get setup with. Additionally, there is little governance and lack of analytics and the tool itself is quite isolated from other services and platforms. This results in the deployment pipeline becoming a bottleneck for many organisations, and the idea of migrating to a less complex cloud solution a painful process. In recent years, Amazon have introduced CI/CD services which support almost every application framework and are not as complex or bespoke to get up and running. Many of which have been implemented to form a POC (proof of concept) for this research project.

Code Repository: Git & AWS CodeCommit

Git has proved the most common and popular open source distributed version control tool available for integration of code updates to a repository. AWS CodeCommit is a secure, highly scalable, managed source control service that hosts private Git repositories and is chosen as the repository of choice to fit within the topic of using modern cloud resources.

Containerized Components: Docker & AWS ECR

Docker is a Product as a Service (PaaS) product that enables the design concept of containerization. To add complexity, and mock real world scenarios that encapsulate the state of art architecture for applications, this research requires a solution based on containerized applications in CI and CD. AWS Elastic Container Registry (ECR) is Amazon's version of DockerHub. Once the container is built, the image is pushed and stored in the registry where is it then easily accessible by other Amazon services.

The CI/CD Pipeline: AWS CodePipeline

Once code changes have been pushed to the CodeCommit Repository, CodePipeline will trigger the CI/CD process. From reviewing related literature, GitLab is the a popular alternative to AWS CodePipeline. However, that could have a lot to do with the fact the tool is open source. The advantages of AWS CodePipeline is how lightweight it is, as you don't need to install it to use it. Also, the availability and accessibility of it as a result. The console is extremely user friendly and easily configurable with ease to integrate with other Amazon services. Although not as extensive in its ability as platforms such as Jenkins, the tools fulfills all the necessary requirements for this project.

Continuous Integration: AWS CodeBuild

AWS Codebuild is used to compile the source code and run the build spec file which includes a set pre-build, build and post-build instructions. CodeBuild will be responsible for building the Docker image and pushing the image to Amazon ECR.

Benchmarking: AWS Lambda

AWS do not currently offer a benchmarking service so Lambda has been chosen to fulfil the benchmarking requirements for this project instead. Once the pipeline was configured and the application has been deployed a number of times, it was time to develop the benchmarking functions in Lambda. AWS Lambda is an event-driven, serverless computing platform designed to allow its users to write functions to perform almost any task. CodePipeline is already designed to trigger lambda functions at any stage within the pipeline. The stages are easily configurable to include lambda triggers, making it an attractive option for this project. Two additional stages will be created in the pipeline, both which will trigger lambda functions. The first benchmark will take place after the source stage and the second will take place after the build stage. The design solution for these benchmarks are outlined in 4.3.

Deployment Platform: AWS ECS & Fargate

Once the application has passed the build and benchmarking stages, a new deployment of the ECS Fargate Service is created with the new image build. AWS Fargate is a serverless compute engine for Amazon ECS that runs containers without requiring us to worry about the underlying infrastructure. This service was chosen due to the trending nature of serverless computing in the cloud industry today.

4.2 Architecture

The architectural diagram below illustrates how these services have been configured to form a fully functioning CI/CD/CB pipeline using Amazon Web Services.

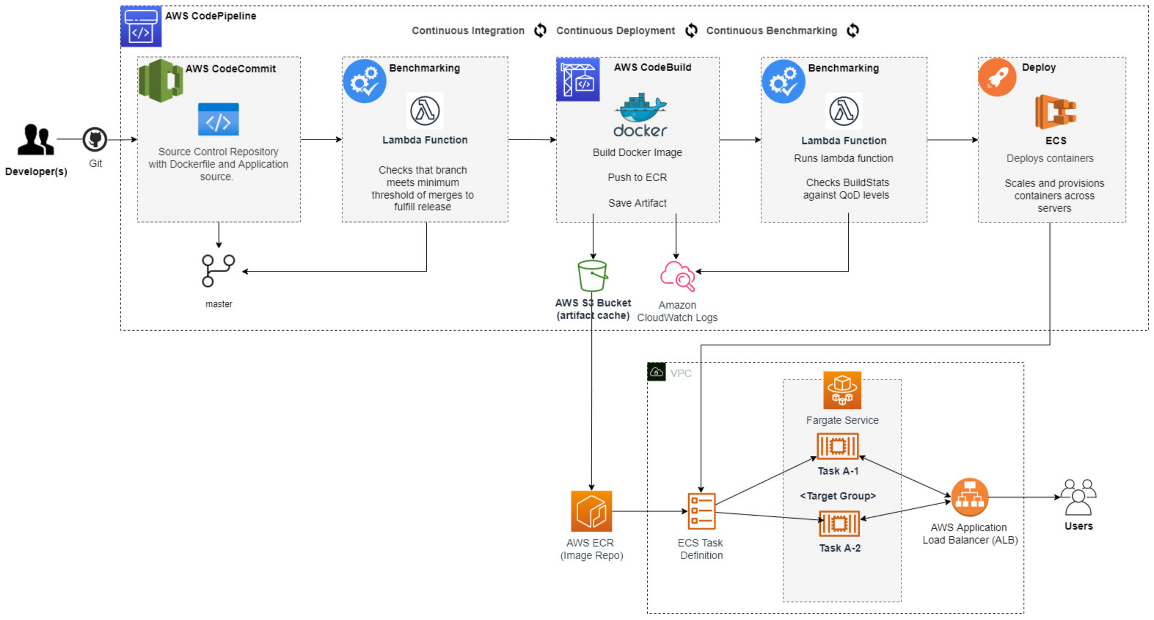


Fig 1: Automated DevOps pipeline using Amazon Web Service

4.3 Proposed Solution

This section outlines the design for the two benchmarking stages with the CI/CD pipeline.

Before the Source Stage

The first benchmark required evaluating branch mechanisms adapted by DevOps teams today to understand the process and identify any gaps that may exist. Almost all application repositories use branching in version control and software management to maintain stability while isolated changes are made to code. The process of creating pull requests from lower level branches to a singular master branch is common practice. However, the time at which which the pull requests are accepted and merged to master is unknown. This is due to the need to govern that all prerequisites such as testing have passed before the merge can be granted. Furthermore, the process to confirm that a release should and should not happen is usually determined by a schedule, and dependent on the number of new features awaiting release, so therefore managed manually too. With this in mind, a benchmarking solution is designed that allow all approved merges to master to start a deployment release. The benchmark will be responsible for checking that the release can only go ahead once a predefined number of merges existed. This solution reduces the operational overhead from when a branch pull request has been accepted to master until it gets to the production environment. The solution has been designed to be generic to work with any application. The baseline requires a fixed threshold to determine the minimum number of merges that must exist on the master branch before a release can continue. The process flow diagram illustrating this solution is outlined below.

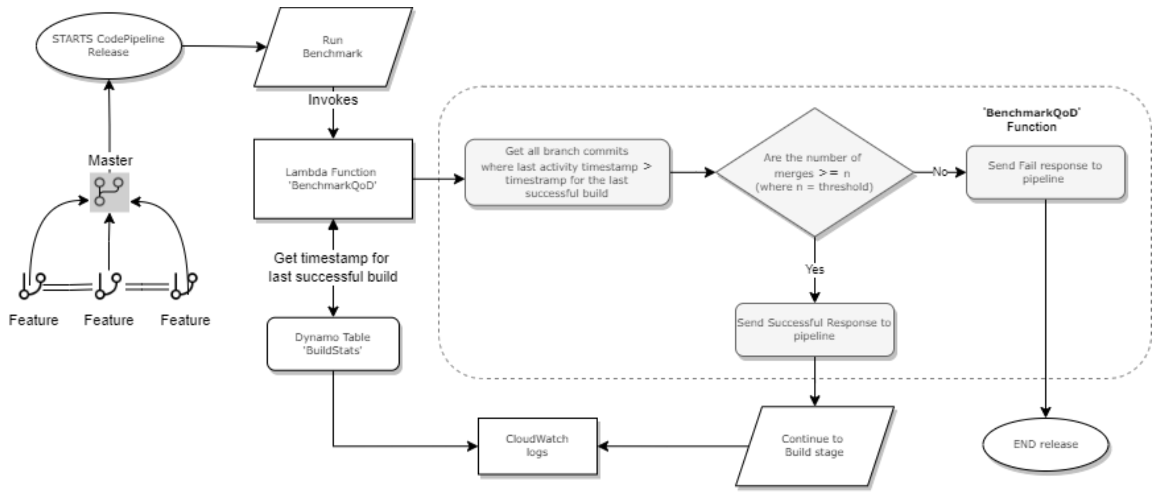

STARTS CodePipeline Release
Run Benchmark
Invokes
Master
Feature
Feature
Feature
Lambda Function 'BenchmarkQoD'
Get timestamp for last successful build
Dynamo Table 'BuildStats'
'BenchmarkQoD' Function
Get all branch commits where last activity timestamp > timestramp for the last successful build
Are the number of merges >= n (where n = threshold)
No
Send Fail response to pipeline
Yes
Send Successful Response to pipeline
Continue to Build stage
CloudWatch logs
END release

Fig 2: Process Flow Diagram for first Benchmarking stage of Pipeline.

After the Build Stage

The second benchmarking solution is based on analysing past trends in pipelines to manage the quality of deployments. The build phase is fundamental to all deployments and often the issue if a release fails. This benchmark occurs later in the pipeline, after the build stage has completed. The benchmark reviews QoD goals based on the build performance for that release. The purpose of this benchmark is to detect 'jumps' in build performance based on previous trends from older releases. The baseline is set by defining the number of previous builds the benchmark should analyse and base the decision on. The solution is designed to exclude the time to execute pre and post build instructions as per the build configuration file as these could skew the results. The expectation with most pipelines is that the changes being released are improvements or updates to an application. Although the experiments are agnostic to the applications use case, in order to mimic real world behavior to evaluate this benchmark, the application must include updates which would increase the size of the app, thus impacting the build duration of the pipeline. The process flow diagram illustrating this solution is outlined below.

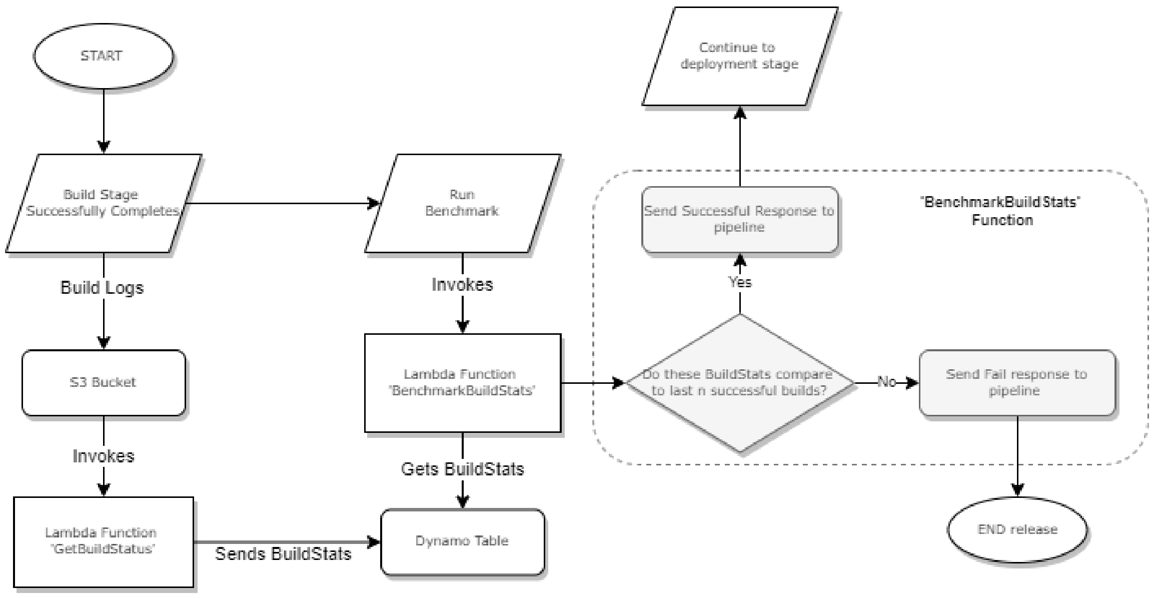

START
Build Stage Successfully Completes
Build Logs
S3 Bucket
Invokes
Lambda Function 'GetBuildStatus'
Sends BuildStats
Run Benchmark
Invokes
Lambda Function 'BenchmarkBuildStats'
Gets BuildStats
Dynamo Table
Continue to deployment stage
Send Successful Response to pipeline
"BenchmarkBuildStats" Function
Yes
Do these BuildStats compare to last n successful builds?
No
Send Fail response to pipeline
END release

Fig 3: Process Flow Diagram for second Benchmarking stage of Pipeline.

5 Implementation

This section details the implementation of the proposed solution, describing the outputs produced including code written and benchmarking models developed. AWS Identity and Access Management (IAM) access is a major prerequisite for using AWS. A number of user roles, with multiple policies, were required for each platfrom to allow access and sharing of data between each service.

5.1 Deploying the Application

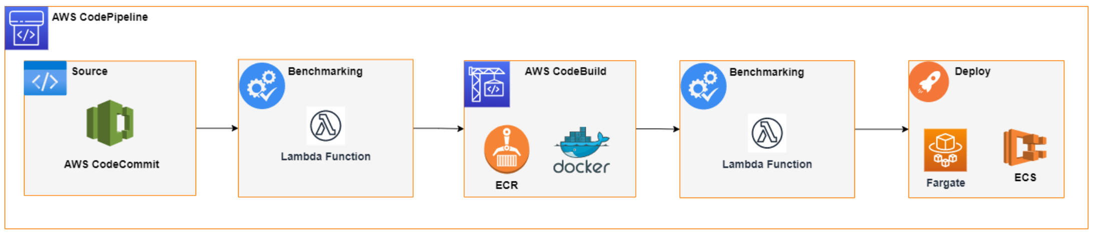
Once the sentimental analysis application was chosen, the application was containerised locally and the image was pushed to dockerhub. A private repository (repo) was created on ECR and using the AWL CLI, the local docker image was pushed to the ECR repo. Next a cluster was created, which is a logical grouping of tasks or services on infrastructure ran by ECS. A task definition was set up to run the container and Fargate was chosen as the launch type. The CPU and memory requirements were set in the task definition and the docker image URI is configured to a container. Port mappings are defined to allow containers to access the port on the host container to send or receive traffic. Additional networking configurations were required to make the application publicly accessible by IP address once deployed. The ECS task ran and the application was deployed using fargate. To increase scalability and availability, an application load balancer was configured and added to the ECS task definition.

5.2 Building the Pipeline

A CI/CD Pipeline was built using AWS Services: AWS CodeCommit, AWS CodeBuild and AWS Pipeline. A repository was created in CodeCommit to store the application code. AWS CLI and git were both used locally to access CodeCommit and push code changes to the application repo. The next step required setting up continuous integration functionality on the CodeCommit repo, using AWS Codebuild as the managed CI service. A buildspec.yml file was included in the applications artifacts that included prebuild, build and postbuild commands to containerize the application using docker and push it to ECR. Once the repo and CI service have been setup, it was time to configure the pipeline using AWS CodePipeline. Amazon ECS was configured as the deploy provider. By default when CodePipeline is chosen as the source using the AWS console, a rule is created that triggers a deployment when a change is merged to a certain repo or branch. For this, the master branch of the sentiment analysis application was chosen as the release trigger.

5.3 Developing Benchmarking

Both benchmarking stages have been sequenced as seen in the figure below.



The First Benchmark

The functionality of the first benchmark is dependent on event data from CodeCommit and triggers once the source stage has completed. A lambda function was developed using python code that checks how many merges exist on the master branch since the last deployment occured. If the number of merges is equal to or greater than the predefined fixed threshold, also known as the baseline, then a response to sent back to the pipeline to proceed with the release. If the number of merges is less than what is set as the baseline, then a response is sent back to stop the release from continuing. This logic is achieved using AWS Boto3 SDK, a Python API for AWS infrastructure services. Using this SDK, API calls are made CodeCommit to analyse the repository event data and again to CodePipeline with a decision on the outcome of the benchmark.

The Second Benchmark

As previously mentioned, additional steps were taken to write the CodeBuild logs to an S3 bucket which was then parsed to send targeted attributes to a DynamoDB table for further analysis. A similar approach to the first benchmark is adapted from here. A lambda function is in place to trigger after the build stage has completed. The benchmark is developed to review the log data from the build and compare it to past trends from previous successful builds to decide whether the QoD levels have been met. The metric used in this demonstration is build duration. Once the release falls in line with previous trends, a response is sent back to the pipeline to proceed to the next stage of the pipeline.

If the release falls outside of that trend, then a response is sent to 'fail' that stage, stopping the release from continuing. The number of historical deployments being analysed is predefined and depends on a number of factors such as frequency of deployments, past trends etc. The parameter is designed to be easily adjustable for other systems.

5.4 Running Deployments with Continuous Benchmarking

Once the pipeline had been built and the two benchmarking stages added, it was time to deploy changes to the sentiment analysis application to test how the benchmarks performed under different scenarios. The pipeline was cloned and the two benchmarking stages removed from the second pipeline in order to evaluate 'with CB' and 'without CB' scenarios. By default, CodePipeline tracks the duration of each release and provides a breakdown based on each stage configured within the pipeline. This served its purpose when comparing performance on both pipelines. The results from these deployments are discussed in detail in the next section 6.

6 Evaluation

In this section, we discuss the experiments that were used to evaluate CB as core stages within CI/CD. The experiments were conducted using three variations of the same Sentiment Analysis application, Amazon Web Services and Microsoft Excel. The results are critically assessed to determine the benchmarks performance in CI/CD and the impact on the velocity and agility of a pipeline when benchmarks are used during deployment.

6.1 Case Study: Benchmarking branch merges to measure QoD

This case study required running a benchmark once the release has started, which is as soon as code is merged to the master branch in the CodeCommit repository. The benchmarking is checking that the release fulfills QoD goals before progressing with the release. The metrics used to evaluate this is based on the number (n) of merges that have happened on the master branch since the last successful deployment. We will refer to 'n' as the baseline and for this case study, 'n' was set at 3. Therefore, at least three merges must exist in any given release for the benchmark to return a successful response to the pipeline to continue. Several releases were ran in the pipeline to measure and evaluate the effects of this benchmarking mechanism. Different scenarios were tested, such as merging changes directly to master, and not through the use of branching, to evaluate the response from the benchmark. A failed response was received and the pipeline stopped the release from progressing, which is expected behaviour. Another scenario included merging feature branches to master that are less than and more than 'n' to review the response from the benchmark. The pipeline stopped and proceeded as expected each time. The performance implications of including this stage in a pipeline are evaluated in Section 6.3.

Algorithm 1 *Benchmarking algorithm*

```
    Input Event is triggered from pipeline
    Output Pipeline response on whether to proceed with release or not
 1: function Benchmark
 2:     get latest deployment timestamp
 3:     get list of code commit pull requests
 4:     for each item $i \in N$ pull requests do
 5:         if last activity time in $i$ is greater than timestamp then
 6:             keep pull request $i$
 7:         else
 8:             remove pull request $i$
 9:         end if
10:     end for
11:     if number of pull requests greater or equal to $N$ then return Proceed with pipeline deployment
12:     elsereturn Stop pipeline deployment
13:     end if
14: end function
```

6.2 Controlled Experiment: Benchmarking build performance to measure QoD

For this experiment, a second benchmarking stage is evaluated. Three different iterations of the same sentimental analysis application are used. The first is referred to as 'light' which means this app is the default version with no added changes. The 'medium' version is the default version but with an additional library, 'tensorflow'. The third and final version of the app will be referred to in experiments as the 'heaviest' and includes two additional libraries, 'tensorflow' and 'pytorch'. The expectation is that the more the app evolves and grows in size, the benchmark would perform the same QoD check with a new baseline each time as the logic depends on past trends to determine what that is. Each experiment evaluates the build performance for all three app versions but uses a different benchmark formula to determine which would produce the most accurate baseline over time. The baseline is calculated using the average build duration for the last number (n) of successful builds with a % buffer added to the mean to facilitate natural growth. Each experiment is ran a total of 15 times, five times on each app version, starting from lightest to heaviest.

$$\text{Baseline} = \bar{n} + \frac{\bar{n}}{100}(\text{x}) \quad , \quad \text{where } \bar{n} = \frac{\sum_{i=1}^{n}}{n}$$

6.2.1 Experiment A: Calculating baseline using mean build time, where n=3 and x=25%

For this experiment, a 25% buffer is added to the mean build duration of the previous three releases to calculate the baseline for the given release. The first run has no previous builds to benchmark against and since the baseline is determined based on previous trends, the baseline was set to '0', resulting in failed responses to the pipeline. A similar sequence followed until the fourth run, whereby three previous builds existed and therefore the benchmark began returning success responses to the pipeline to say that

QoD goals were being met. However, when heavier versions of the app were deployed, and build duration increased. The transition stage whereby the baseline was still being calculated at 25% on the three previous builds (which were the lighter app), resulted in more response fails until run 3 of the medium app, where historical trends included higher build duration's. The same pattern followed when the heaviest app was deployed in the pipeline and there was a jump in build duration again. It became obvious that the baseline calculation for this experiment did not facilitate enough growth when changes were made on the app and the build duration increased between releases.

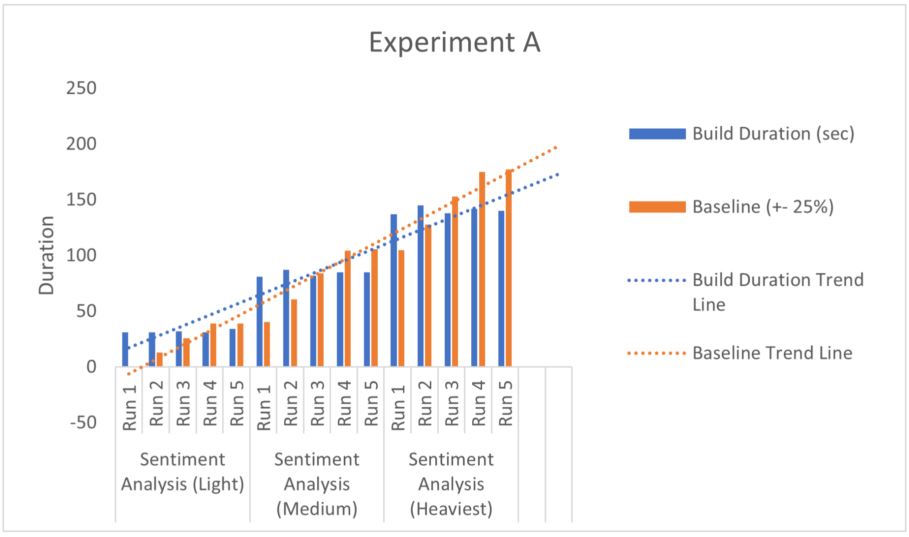


Fig 4:

6.2.2 Experiment B: Calculating baseline using mean build time, where n=3 and x=50%

For this experiment, 50% is added to the mean build duration for the previous three successful builds (n=3), to calculate the baseline for the next release. The sequence started by following a similar trend to Experiment A. However, the number of failed responses between app variations decreased significantly in comparison. When the medium sized app was deployed, the baseline calculation catered for the increase in build time. When the heaviest app was deployed, one fail occured. This is a significant improvement on experiment A, where two fails occured sequentially each time that the app changed.

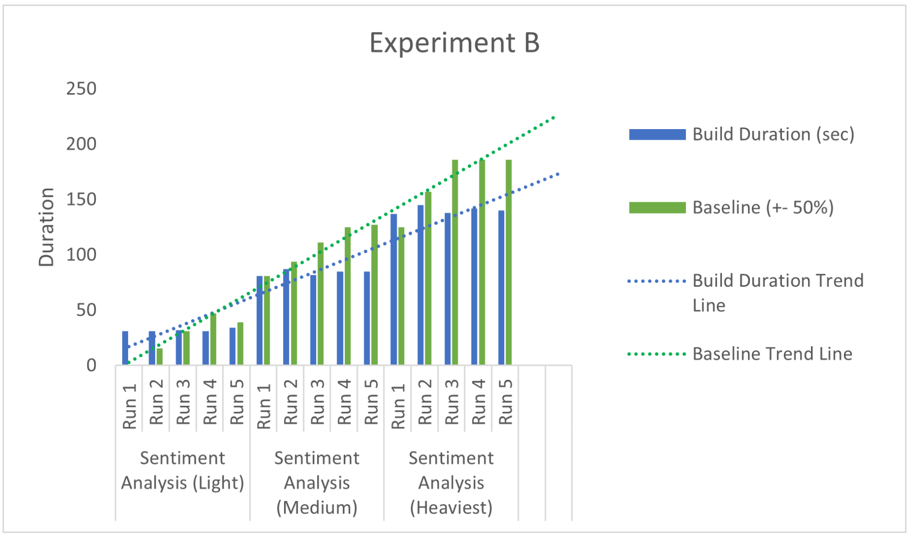


Fig 5:

6.2.3 Experiment C: Calculating baseline using mean build time, where n=3 and x=75%

For this experiment, 75% is added to the mean build duration for the previous three successful builds (n=3) to calculate the baseline for the next release. The release failed twice sequentially on both the light and medium apps on run 1 and 2. However, the transition period of the heaviest app did not result in any failed releases. However, the trend line for the baseline versus the build duration quickly increases as new versions of the app are deployed. The relationship between the two linear lines are more widespread, which is expected when such a high % buffer is added to the mean duration.

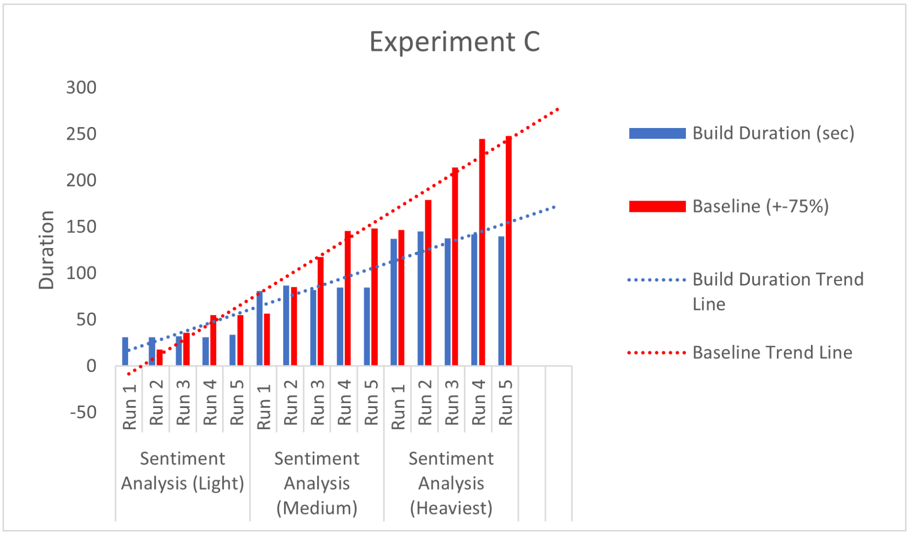
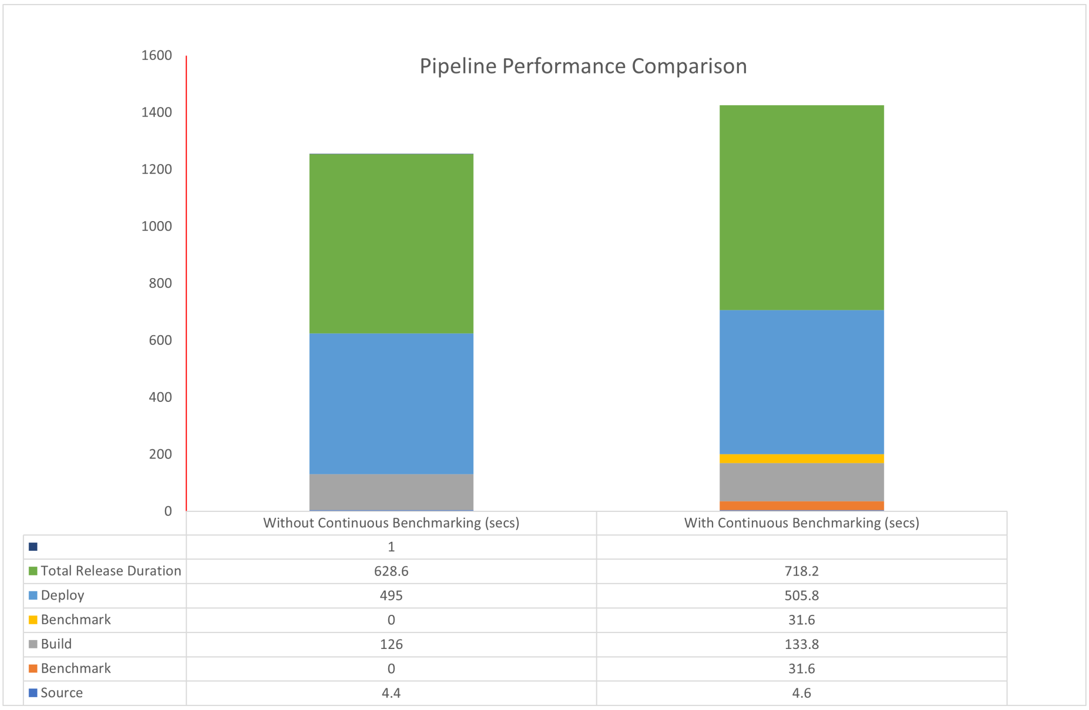


Fig 6:

6.3 Main Experiment: Measuring CI/CD/CB pipeline performance during Deployment

This experiment measured the performance impact of including CB in the CI/CD pipeline designed for this research. Two pipelines were configured, one with CB and one without. The sentiment analysis application was deployed a total of five times for each pipeline to gather enough data to calculate the mean time for each stage in the pipeline for comparison. The stacked graph below illustrates the outcome for the two pipelines, with a colour block showcasing each key stage in the deployment. The X-axis is the duration in seconds. The yellow and orange lines on the left stack are the two benchmarking stages in the CB pipeline. It is clear from the graph that the depth of each CB line is insignificant when compared to other stages such as build and deploy, which would suggest the performance impact is minuscule. A full breakdown of this experiment, with figures, can be found in the Appendix.7.



	Without Continuous Benchmarking (secs)	With Continuous Benchmarking (secs)
🟦	1	
🟩 Total Release Duration	628.6	718.2
🟦 Deploy	495	505.8
🟨 Benchmark	0	31.6
⬜ Build	126	133.8
🟧 Benchmark	0	31.6
🟦 Source	4.4	4.6

Fig 7: Average Deployment Time for a CI/CD pipeline without CB (left) and with CB (right).

6.4 Discussion

The first case study evaluated branch merges to determine if QoD levels have been met. This demonstrated how benchmarks can be put in place to govern when a branch has qualified for a release. The concept of benchmarking at this stage in the pipeline is not something that was explored in related work. Potentially the number one attraction to CI/CD, is the automation that comes with the design concept. This CB stage contributes to this and it puts benchmarks in place that ensure fulfillment of goals before a release can progress. The baseline is determined on the number of merges and needs to be set as a minimum. It has been observed that this may not be a suitable benchmark for all applications and the benchmark does depend on past trends to evaluate a suitable value for the baseline. There are specific application use cases that this benchmark could benefit such as those that do not want to have a fixed day to deploy changes, but rather a fixed threshold in place to determine when a release is justified. Additionally, even with cloud computing, there is always that risk factor in a deployment so justification is necessary. This benchmark helps achieve that. Furthermore, an improvement would be to include other fixed thresholds to determine if the time of day is suitable for a deployment as this is also a key factor that this study did not consider. By including benchmarking at the source stage, it provides motivating goals for teams to work towards to achieve frequent releases, without introducing any added overhead.

The controlled experiment assessed how build performance can be benchmarked in order to maintain QoD goals in a pipeline. The results indicated which baseline calculation was most accurate at achieving a process that can measure and re-baseline as an app changes and grows in size. Experiment B, "Benchmarking using mean build time with 50% baseline", proved to be the most reliable with the linear trend line presenting strongest over the 15 deployments. All three experiment A, B and C, despite what baseline, had fails. The first few fails were anticipated since there was no historical builds to baseline against. Additionally, the baseline was set on stats from the three previous builds. It would be interesting to perform this experiment using real data and not data from orchestrated runs. This would determine the reliability of this benchmark at detecting jumps in build behaviour for a pipeline using trend analysis in the form of benchmarks. The full breakdown of results for this experiment can be reviewed in the table below.

Sentiment App	Build Duration	EXP(A)	Release?	EXP(B)	Release?	EXP(C)	Release?
	(sec)	Baseline		Baseline		Baseline	
Run 1 (Light)	31	0.00	N	0.00	N	0.00	N
Run 2	31	12.90	N	15.5	N	18.00	N
Run 3	32	25.80	N	31	N	36.16	Y
Run 4	31	39.10	Y	47	Y	54.83	Y
Run 5	34	39.10	Y	39.1	Y	54.83	Y
Run 1 (Med)	81	40.40	N	81	Y	56.58	N
Run 2	87	60.80	N	94	Y	85.17	N
Run 3	82	84.17	Y	111	Y	117.83	Y
Run 4	85	104.17	Y	125	Y	145.83	Y
Run 5	85	105.83	Y	127	Y	148.17	Y
Run 1 (Heavy)	137	105.00	N	125	N	147.00	Y
Run 2	145	127.92	N	157	Y	179.08	Y
Run 3	138	152.92	Y	186	Y	214.08	Y
Run 4	142	175.00	Y	186	Y	245.00	Y
Run 5	140	177.08	Y	186	Y	247.92	Y

Table 2: Results from controlled experimental run to measure build performance for benchmarking a release against QoD goals. The units are in seconds.

The final experiment is the main experiment as it is required to answer the research question for this project. This experiment evaluated the performance impact of using CB in the pipeline. A mean number for each stage of the deployment are reviewed side by side and it is clear that the performance comparison between both pipelines is minor, with an average difference of 1 minute 5 seconds in the total deployment duration overall. Both benchmarking stages added an average of 31.6 seconds each to the CB pipeline, an increase of approximately 1 minute when compared to a pipeline without CB stages.

This proof of concept is effective at demonstrating how benchmarking allows safeguards to be put in place in CI/CD that ensures QoD levels are at least maintained. The experiments conducted have helped illustrate that the data from these benchmarks can be used to evaluate performance of the pipeline and provide insight on trends over a given timeframe. Both CB solutions should be further developed to cater for other important factors in CI/CD such as expanding the metrics that each benchmark analyse at each stage in pipeline. Analysis should be done using real world data from other pipelines to determine the true confidence of CB in these aspects. Additionally, the solution uses Amazon Web Services to configure the full pipeline design, it is unclear how and if this can be replicated using other tools and services and still produce the same results, specifically in regards to the overall pipeline performance.

7 Conclusion and Future Work

One of the key questions this research set out to answer is whether **"using Continuous Benchmarking, at core stages within the delivery pipeline, could ensure the Quality of Deployment of cloud applications, without negatively impacting the pipelines performance?"**. Based on the research and evaluation methods conducted, the short answer is yes; there is no real performance loss when introducing these two CB stages. Moreover there is a performance gain as the pipeline is far more superior as it analyses key activity against previous releases, with the ability to stop deployments when releases don't quite follow a certain trend. The one minute additional time onto the overall duration should not be interpreted as a negative performance impact for this reason.

Furthermore, this research project set out to demonstrate how release requirements can be integrated into the delivery process and how benchmarking can be used to enforce QoD goals. These objectives were met through the design of the two separate benchmarking stages, one which put fixed thresholds in place to determine when a release has been justified, and the other which checked to ensure the deployment is on trend with previous releases.

Future work could include using Machine Learning to advance the benchmarks and have the system re-baseline and 'learn' as the system grows/changes exponentially. To add further confidence to the CB approaches, they should be tested at scale to see how reliable the benchmarks are in real world scenarios. This would be a requirement in much bigger complex systems where deployments carry more weight. Additionally, the concept of CB can become very complex depending on the use case. We have seen related work benchmark microservice applications and its QoS. Future work here could be to benchmark a pipelines deployment for this application type.

Performing benchmarks, can be costly. Especially when using cloud platforms such as AWS. This metric was not considered as part of this research and should be evaluated in future work to carry out comparison analysis between other key players who offer open source CI/CD tools such as Jenkins and GitLab.

Finally, I believe this research project has identified a gap in the cloud market today; the lack of systematic benchmarking tools. The concept of benchmarking is widespread and applied in many ways in every industry. In cloud, monitoring has become established, which is clear from the presence of CloudWatch logging with every AWS service. Therefore, there is potential for commercialisation, particularly as a tool in DevOps.

References

[1] D. Linthicum, “How DevOps is dictating a new approach to cloud development.”

[2] A. Reiss and D. Stricker, “Introducing a new benchmarked dataset for activity monitoring,” in *2012 16th International Symposium on Wearable Computers*, pp. 108–109. ISSN: 2376-8541.

[3] R. Baggen, J. P. Correia, K. Schill, and J. Visser, “Standardized code quality benchmarking for improving software maintainability,” vol. 20, pp. 1–21.

[4] J. P. Correia and J. Visser, “Benchmarking technical quality of software products,” in *2008 15th Working Conference on Reverse Engineering*, pp. 297–300. ISSN: 2375-5369.

[5] M. Grambow, F. Lehmann, and D. Bermbach, “Continuous benchmarking: Using system benchmarking in build pipelines,” in *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 241–246, IEEE.

[6] S. S. Gill and I. Chana, “Resource provisioning and scheduling in clouds: QoS perspective,” vol. 72.

[7] C. Deegan, “Continuous security; investigation of the DevOps approach to security,” p. 22.

[8] A. Cepuc, R. Botez, O. Craciun, I.-A. Ivanciu, and V. Dobrota, “Implementation of a continuous integration and deployment pipeline for containerized applications in amazon web services using jenkins, ansible and kubernetes,” in *2020 19th RoEduNet Conference: Networking in Education and Research (RoEduNet)*, pp. 1–6. ISSN: 2247-5443.

[9] H. Kang, M. Le, and S. Tao, “Container and microservice driven design for cloud infrastructure DevOps,” in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 202–211.

[10] Z. Li, Y. Zhang, and Y. Liu, “Towards a full-stack devops environment (platform-as-a-service) for cloud-hosted applications,” vol. 22, no. 1, pp. 1–9. Conference Name: Tsinghua Science and Technology.

[11] S. Throner, H. Hütter, N. Sänger, M. Schneider, S. Hanselmann, P. Petrovic, and S. Abeck, “An advanced DevOps environment for microservice-based applications,” in *2021 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pp. 134–143. ISSN: 2642-6587.

[12] “Continuous integration and its tools.”

[13] M. Grambow, L. Meusel, E. Wittern, and D. Bermbach, “Benchmarking microservice performance: a pattern-based approach,” in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC ’20, pp. 232–241, Association for Computing Machinery.

[14] M. Grambow, E. Wittern, and D. Bermbach, “Benchmarking the performance of microservice applications,” vol. 20, no. 3, pp. 20–34.

[15] M. Miglierina and D. Tamburri, “Towards omnia: A monitoring factory for quality-aware devops,” pp. 145–150, 04 2017.

[16] C. Singh, N. S. Gaba, M. Kaur, and B. Kaur, “Comparison of different CI/CD tools integrated with cloud platform,” in *2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, pp. 7–12.

[17] H. Anzt, Y.-C. Chen, T. Cojean, J. Dongarra, G. Flegar, P. Nayak, E. S. Quintana-Ortí, Y. M. Tsai, and W. Wang, “Towards continuous benchmarking: An automated performance evaluation framework for high performance software,” in *Proceedings of the Platform for Advanced Scientific Computing Conference*, pp. 1–11, ACM.

[18] T. Górski, “Towards continuous deployment for blockchain,” *Applied Sciences*, vol. 11, p. 11745, 12 2021.

[19] M. R. Pratama and D. Sulistiyo Kusumo, “Implementation of continuous integration and continuous delivery (CI/CD) on automatic performance testing,” in *2021 9th International Conference on Information and Communication Technology (ICoICT)*, pp. 230–235.

[20] J. Mahboob and J. Coffman, “A kubernetes CI/CD pipeline with asylo as a trusted execution environment abstraction framework,” in *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, pp. 0529–0535.

[21] D. moonat, “dmoonat/sentiment-analysis.” original-date: 2018-06-03T10:06:37Z.

[22] R. Kumar and R. Goyal, “Modeling continuous security: A conceptual model for automated devsecops using open-source software over cloud (adoc),” *Computers Security*, vol. 97, p. 101967, 07 2020.

Appendix

Without CB (secs)	Source	Build		Deploy		Release Time
Run 1	5	126		525		659
Run 2	3	125		463		594
Run 3	4	126		496		629
Run 4	5	125		495		629
Run 5	5	128		496		632
MEAN (secs)	4.4	126		495		628.6
With CB (secs)	Source	Benchmark	Build	Benchmark	Deploy	Release Time
Run 1	3	32	125	31	529	725
Run 2	6	32	126	31	496	695
Run 3	5	31	160	32	496	730
Run 4	4	32	130	32	496	698
Run 5	5	31	128	32	512	743
MEAN (secs)	4.6	31.6	133.8	31.6	505.8	718.2

Table 3: Figures for 10 pipeline deployments, broken down based on stage. The units are in seconds.