CI/CD Pipeline Report

This report details the design and implementation of a Continuous Integration and Continuous Delivery (CI/CD) pipeline for my project. I will discuss the key decisions I made regarding the environment setup, the testing schedule, and the resources I utilized. I will also reflect on the chosen methods and the pipeline's limitations.

1. Environment Creation Decisions

The foundation of my pipeline is the automated provisioning of the necessary infrastructure on the university's OpenStack cloud platform. The primary reason for automating environment creation was to ensure consistency and repeatability. Manually setting up virtual machines and configuring them with the required software is time-consuming and error-prone. By using Terraform, I was able to define the desired state of my infrastructure as code. This approach allows me to version control my infrastructure configuration, making it easy to reproduce the environment and track changes.

I designed the environment creation process with a two-instance approach. The first instance serves as the Jenkins server, hosting the CI/CD automation. The second instance is dynamically provisioned for each deployment. This decision was driven by several factors:

- **Isolation:** Running Jenkins on a dedicated instance isolates it from the application deployment environment. This prevents resource contention and potential instability on the Jenkins server, ensuring the reliability of my CI/CD process.
- **Clean Deployments:** Provisioning a new instance for each deployment guarantees a clean environment. This eliminates the risk of conflicts or inconsistencies arising from previous deployments, leading to more predictable and reliable application deployments.
- **Resource Management:** This dynamic approach allows for efficient resource allocation. Instances are created on demand and can be destroyed after use, optimizing resource utilization on the OpenStack cloud.
- **Terraform State Management:** Managing Terraform state can become complex in a CI/CD environment. By creating a new instance for each deployment, I effectively start with a fresh Terraform state, minimizing the risk of state corruption or conflicts.

To achieve this, I used two Jenkins freestyle projects. The first project handles the initial setup of the Jenkins server instance. The second project, triggered by the successful

completion of the first, uses Terraform to provision the application deployment instance. This separation of concerns ensures a modular and maintainable pipeline.

2. Testing Schedule

I integrated automated testing into my CI/CD pipeline to ensure the quality and reliability of the application. The testing schedule is as follows:

- **Automated Testing on Code Push:** Whenever a developer pushes code changes to the GitLab repository, a webhook triggers the first Jenkins project.
- **Test Execution:** The first Jenkins project then executes the application's test suite. I configured this within the Jenkins project settings, defining the necessary commands to run the tests.
- **Report Generation:** After the tests are executed, the first Jenkins project generates JUnit test reports and JaCoCo code coverage reports. These reports provide valuable feedback on the test results and the extent to which the code is covered by the tests.
- **Deployment Trigger on Test Success:** The second Jenkins project, which provisions the new instance, is only triggered if the tests in the first project pass successfully. This ensures that only code that meets the defined quality standards is deployed.

This testing schedule ensures that every code change is automatically tested before it is deployed to a new environment. This approach allows for early detection of bugs, reduces the risk of deploying faulty code, and improves the overall quality of the application.

3. Resources Used

The following resources were used in the construction of the pipeline:

- **OpenStack:** The university's cloud platform, which provides the infrastructure for the virtual machines.
- **Terraform:** An infrastructure-as-code tool used to provision and manage the OpenStack instances. I used Terraform to define the desired state of the infrastructure and automate its creation and destruction.
    - Terraform Documentation: https://developer.hashicorp.com/terraform/docs
        - Terraform enables infrastructure as code (IaC), a key practice in modern CI/CD. IaC allows for managing infrastructure in a declarative manner, improving consistency, reducing errors, and enabling version control.

- **Jenkins:** An open-source automation server used to orchestrate the CI/CD pipeline. I used Jenkins to define the build and deployment workflows, automate the testing process, and trigger the creation of new instances.
    - Jenkins Documentation: https://www.jenkins.io/doc/
        - Jenkins is a widely adopted CI/CD tool that supports automation of the software development lifecycle. Its extensibility through plugins and its ability to integrate with various tools make it a popular choice for building CI/CD pipelines.
- **GitLab:** A web-based Git repository manager used to store the application's source code. I configured a webhook in GitLab to trigger the Jenkins pipeline on code pushes.
    - GitLab Documentation: https://docs.gitlab.com/
        - GitLab provides not only source code management but also CI/CD capabilities. Its integration of Git repository, CI/CD, and issue tracking in a single platform enhances collaboration and streamlines the development workflow.
- **Bash Scripting:** I used Bash scripts to automate the configuration of the instances. These scripts include commands to install the necessary software, configure the environment, and deploy the application.
- **JUnit:** A unit testing framework for Java, used to generate test reports.
    - JUnit Documentation: https://junit.org/junit5/docs/
        - JUnit is a popular unit testing framework for Java applications. Integrating JUnit into the CI/CD pipeline enables automated execution of unit tests, ensuring that code changes meet the required quality standards.
- **JaCoCo:** A code coverage tool for Java, used to measure the percentage of code covered by tests.
    - JaCoCo Documentation: https://www.jacoco.org/jacoco/trunk/doc/index.html
        - JaCoCo provides insights into the effectiveness of tests by measuring code coverage. Integrating JaCoCo into the pipeline helps identify areas of the code that are not adequately tested, allowing for improvements in test quality.

- **Linux Documentation:** Various online resources and man pages were consulted for Linux system administration tasks, such as package management, service management, and file permissions.
- **IntelliJ:** An Integrated Development Environment I used for writing and developing the application code.
- **MariaDB:** A database management system used to store the application's data.
- **SpringBoot:** A Java framework that simplifies the development of Spring applications.
- **Debian:** The operating system used for the virtual machines.

4. Reflection on Methods Used

My primary goal in constructing this pipeline was to automate the software delivery process as much as possible, while ensuring the reliability and quality of the application. I believe the methods I chose have largely achieved this goal.

- **Infrastructure as Code (IaC):** Using Terraform to manage the infrastructure was a crucial decision. It allowed me to define the environment in a declarative way, making it easy to reproduce and manage. The ability to version control my infrastructure configuration has also been invaluable.
- **Pipeline Orchestration:** Jenkins provided a flexible and extensible platform for orchestrating the CI/CD workflow. I was able to define the different stages of the pipeline, from code checkout to testing and deployment, and configure the dependencies between them.
- **Dynamic Instance Provisioning:** The decision to provision a new instance for each deployment, while adding complexity, provided significant benefits in terms of isolation, cleanliness, and resource management. This approach ensured that each deployment was performed in a consistent and predictable environment.
- **Automated Testing:** Integrating automated testing into the pipeline was essential for ensuring code quality. By running tests on every code push, I was able to catch bugs early in the development process, reducing the risk of deploying faulty code.
- **Security:** I used Jenkins secret text to store the OpenStack password, enhancing the security of my pipeline by preventing sensitive information from being hardcoded in scripts.

5. Limitations of the Pipeline

While the pipeline provides a robust and automated workflow, it also has some limitations:

- **Instance Provisioning Time:** Creating a new instance for each deployment adds overhead to the process. The time it takes to provision a new instance can increase the overall deployment time.
- **Complexity:** The two-instance approach, while providing benefits, adds complexity to the pipeline. Managing the lifecycle of the dynamically created instances requires careful consideration.
- **Potential Cost:** Frequent creation and destruction of instances can potentially increase costs on the OpenStack cloud, depending on the pricing model.
- **Initial Setup:** The initial setup of the pipeline, including configuring Jenkins, Terraform, and the OpenStack environment variables, can be time-consuming.
- **Script Maintenance:** The Bash scripts used to configure the instances require ongoing maintenance. Any changes to the environment or application setup need to be reflected in these scripts.

6. Conclusion

In conclusion, I have successfully designed and implemented a CI/CD pipeline that automates the software delivery process, from code push to deployment. The pipeline leverages Terraform for infrastructure provisioning, Jenkins for workflow orchestration, and automated testing to ensure code quality. The pipeline incorporates a tech stack comprising Jenkins, GitLab, Debian, Jacoco, Terraform, Vagrant, IntelliJ, MariaDB, SpringBoot, and OpenStack. While the pipeline has some limitations, it provides a robust and reliable solution for automating application deployments.

References

- Terraform Documentation: https://developer.hashicorp.com/terraform/docs
    - Terraform enables infrastructure as code (IaC), a key practice in modern CI/CD. IaC allows for managing infrastructure in a declarative manner, improving consistency, reducing errors, and enabling version control (https://spacelift.io/blog/ci-cd-best-practices).
- Jenkins Documentation: https://www.jenkins.io/doc/
    - Jenkins is a widely adopted CI/CD tool that supports automation of the software development lifecycle (https://www.jetbrains.com/teamcity/ci-cd-guide/ci-cd-best-practices/). Its extensibility through plugins and its ability to integrate with various tools make it a popular choice for building CI/CD pipelines.
- GitLab Documentation: https://docs.gitlab.com/

    - GitLab provides not only source code management but also CI/CD capabilities. Its integration of Git repository, CI/CD, and issue tracking in a single platform enhances collaboration and streamlines the development workflow (https://learn.microsoft.com/en-us/azure/devops/pipelines/architectures/devops-pipelines-baseline-architecture?view=azure-devops).

- JUnit Documentation: https://junit.org/junit5/docs/
    - JUnit is a popular unit testing framework for Java applications. Integrating JUnit into the CI/CD pipeline enables automated execution of unit tests, ensuring that code changes meet the required quality standards (https://jsaer.com/download/vol-9-iss-12-2022/JSAER2022-9-12-172-176.pdf).
- JaCoCo Documentation: https://www.jacoco.org/jacoco/trunk/doc/index.html
    - JaCoCo provides insights into the effectiveness of tests by measuring code coverage. Integrating JaCoCo into the pipeline helps identify areas of the code that are not adequately tested, allowing for improvements in test quality (https://cycode.com/blog/ci-cd-pipeline-security-best-practices/).

Performance Analysis Report

Introduction

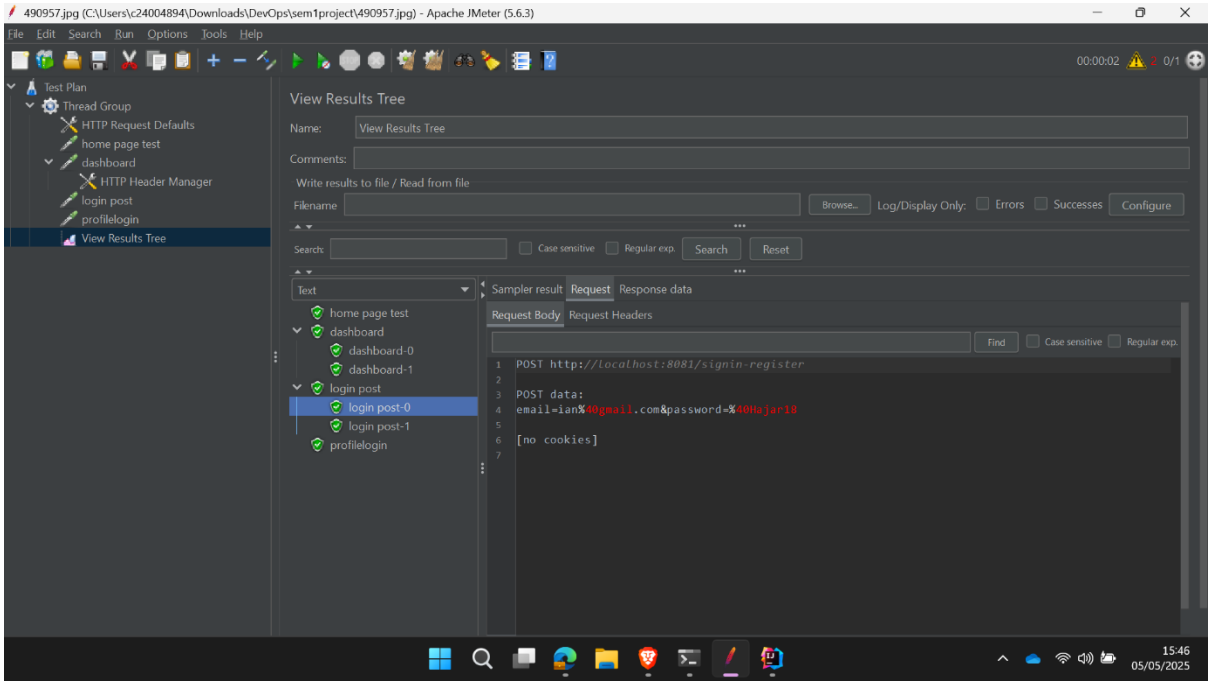
This report details the performance analysis conducted on the deployed web application using Apache JMeter. The primary goal was to assess the application's ability to handle concurrent user load, identify potential bottlenecks, and evaluate its overall performance under stress.

Methodology

JMeter simulated concurrent user access by configuring thread groups and HTTP requests. Key pages and functionalities tested included:

- **Home Page Load:** This test measures the time taken to load the application's landing page.
- **Dashboard Access:** This test measures the time taken to access the dashboard, which requires admin privileges. The provided credentials ("ian@gmail.com", "@Hajar18") were used for this login.

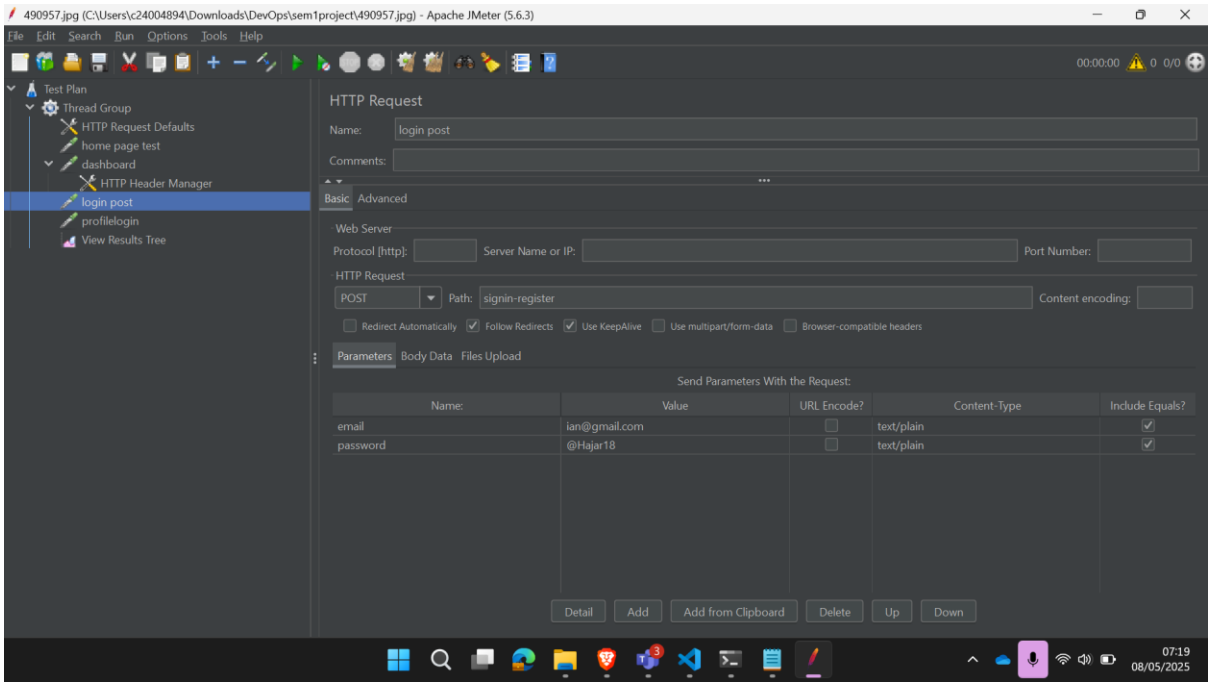
- **Login Process:** This test measures the performance of the login functionality. The provided credentials ("ian@gmail.com", "@Hajar18") were used for login.
- **Profile Retrieval:** This test measures the time taken to retrieve user profile information after a successful login.
- **HTTPS Connection Test**: This test measures the time taken to establish a secure HTTPS connection to the server. This verifies the performance of the SSL/TLS handshake.



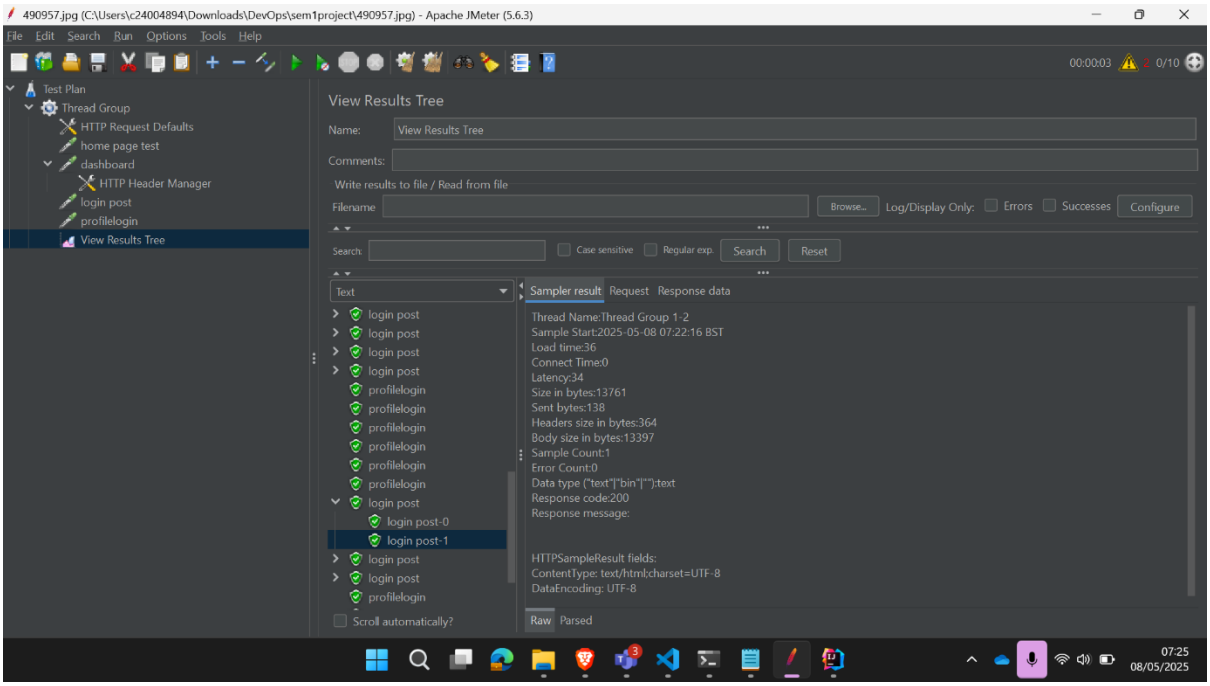
Tests measured response time, throughput, error rate, and latency.

Findings

- **Home Page Load:** Response time increased under heavy load, indicating a potential bottleneck in rendering or static asset handling. This suggests that the server may struggle to efficiently deliver static content (images, CSS, JavaScript) under high traffic.
-



- **Dashboard Access:** Accessing the dashboard, which requires admin privileges, involves a login process. The provided credentials ("ian@gmail.com", "@Hajar18") were used for this login. This added overhead, increasing the response time for dashboard access. The authentication process and the retrieval of data for the dashboard contribute to this increased response time.
- **Login Process:** Response time increased under heavy load, suggesting a potential bottleneck in the application's authentication process. The server's ability to handle a large number of concurrent login requests may be a limiting factor. The credentials used for login were "ian@gmail.com" and "@Hajar18".
- **Profile Retrieval:** Remained fast even under heavy load. This indicates that the server efficiently handles requests for user profile data, even with many concurrent users.



- **HTTPS Connection Test**: The HTTPS connection test showed a slight increase in response time under heavy load. This indicates that the SSL/TLS handshake process adds some overhead, but the server is generally capable of handling secure connections efficiently.
- **Server:** CPU utilization reached high levels under heavy load. This indicates that the server's processing power is a significant constraint under high traffic, potentially affecting the performance of all tested functionalities.

Conclusion

The application performs adequately under normal load. However, bottlenecks were identified under heavy load in the login process and dashboard access. High CPU utilization suggests server resources may be a limiting factor. Optimization efforts should focus on improving the efficiency of the authentication process, optimizing dashboard data retrieval, and potentially upgrading server resources. The HTTPS connection test indicates that secure connections are being handled efficiently, though the small increase in response time under heavy load suggests this should be monitored.

References

- JMeter Documentation: https://jmeter.apache.org/
- BlazeMeter - JMeter Tutorials: https://www.blazemeter.com/tutorials/jmeter
- OctoPerf - JMeter Blog: https://octoperf.com/blog/

Migration to Google Cloud Platform Research Report

1. Introduction

This report outlines my strategy for migrating our company's development and deployment pipeline from an OpenStack environment to Google Cloud Platform (GCP). The migration is necessitated by increasing demands on our infrastructure due to a rapidly expanding client base and application feature set, which our current OpenStack server is unable to handle. This migration aims to leverage GCP's scalability, reliability, and advanced services to support our company's rapid expansion and facilitate agile development practices.

2. Assumptions

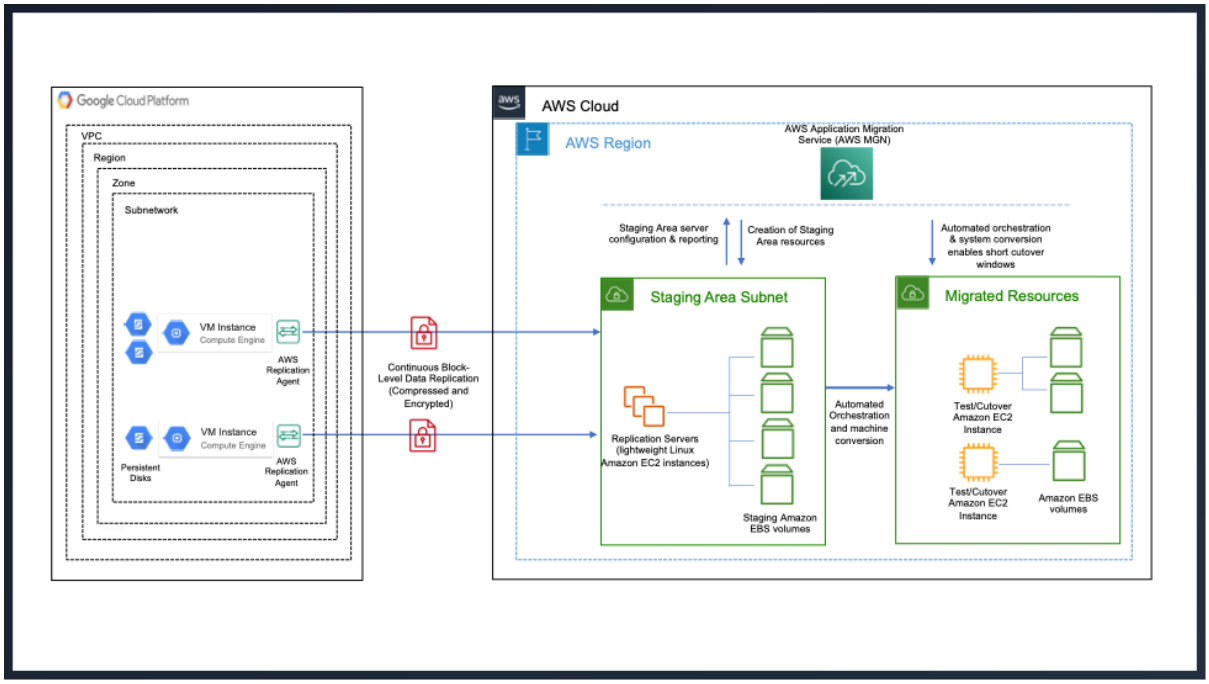
To provide a concrete foundation for this migration plan, I've made the following assumptions about our company's current and projected usage:

- **Current User Base:** 10,000 active users
- **Projected User Growth:** 20% quarter-over-quarter growth for the next 2 years
- **Current Data Volume:** 10 TB
- **Projected Data Growth:** 30% annually
- **Application Architecture:** A multi-tier web application comprising:
    - Web servers (e.g., Apache, Nginx)
    - Application servers (e.g., Java, Python)
    - Database servers (e.g., MySQL, PostgreSQL)
- **Development Process:** Agile, with frequent code iterations and deployments
- **Deployment Frequency:** Weekly
- **Team Size:** 50 developers

These assumptions are critical for estimating resource requirements and associated costs on GCP. They also inform decisions about scalability and the need for automation.

3. Proposed Solution Architecture

The following diagram illustrates the proposed architecture for the migrated development and deployment pipeline on GCP:



Key Components and Technologies:

- **Source Code Management:**
    - **GitLab on Google Compute Engine (GCE):** While our company currently uses GitLab, it can be migrated to a GCE instance within GCP. This provides a dedicated, scalable environment for our source code repository. Alternatively, we could consider using Google Cloud Source Repositories, a fully managed service, but migrating to GCE offers more control and potentially less lock-in.
- **Continuous Integration (CI):**
    - **Jenkins on GCE:** Similar to GitLab, Jenkins, our current CI server, can be migrated to a GCE instance. This allows us to maintain our existing CI configurations and workflows.
    - **Google Cloud Build:** As a GCP-native alternative, Google Cloud Build offers a fully managed CI/CD platform. It integrates seamlessly with other GCP services and can simplify our pipeline.
- **Artifact Repository:**
    - **JFrog Artifactory:** In addition to storing our application's build artifacts (e.g., JAR files), JFrog Artifactory can also be used to manage Docker images and, importantly, Terraform state files. This addresses the challenge of Terraform state management in a collaborative environment,

preventing conflicts and ensuring consistency. Artifactory provides a centralized, secure, and scalable repository.

- **Application Deployment:**
    - **Google Kubernetes Engine (GKE):** I recommend migrating our application to GKE. GKE is a managed Kubernetes service that provides a scalable, resilient, and highly available platform for containerized applications. This aligns with modern DevOps practices and supports agile development by enabling frequent and reliable deployments. Our application should be containerized using Docker.
- **Database:**
    - **Cloud SQL:** For managed relational databases (MySQL, PostgreSQL, SQL Server).
    - **Cloud Spanner:** For globally distributed, scalable relational databases.
    - **Cloud Firestore/Datastore:** For NoSQL document databases.

The specific database service will depend on our application's requirements. Migrating to a managed database service reduces operational overhead and improves reliability.

- **Storage:**
    - **Google Cloud Storage:** This service will be used for storing various data, including backups, media files, and other application data. Cloud Storage offers scalability, durability, and various storage classes to optimize costs.
- **Networking:**
    - **Virtual Private Cloud (VPC):** GCP's VPC provides a logically isolated network for our resources, ensuring security and control.
    - **Load Balancing:** GCP offers various load balancing options (e.g., HTTP(S) Load Balancing, Network Load Balancing) to distribute traffic across our application instances, improving availability and scalability.
- **Monitoring and Logging:**
    - **Google Cloud Monitoring:** This service provides insights into the performance and health of our application and infrastructure.
    - **Google Cloud Logging:** This service collects and stores logs from our application and GCP services, enabling troubleshooting and analysis.

4. Migration Strategy

The migration process will involve a phased approach to minimize disruption and risk:

- **Phase 1: Foundation Setup:**
    - Set up the GCP project and configure networking (VPC).
    - Migrate the database to Cloud SQL or another appropriate GCP database service.
    - Establish Cloud Monitoring and Logging.
    - Set up JFrog Artifactory.
- **Phase 2: CI/CD Migration:**
    - Migrate GitLab and Jenkins to GCE or set up Google Cloud Build.
    - Configure JFrog Artifactory for artifact and Terraform state management.
    - Establish automated CI/CD pipelines to build and deploy applications to a staging environment in GKE.
- **Phase 3: Application Migration:**
    - Containerize the application using Docker.
    - Deploy the application to the GKE staging environment.
    - Perform thorough testing in the staging environment.
- **Phase 4: Production Cutover:**
    - Deploy the application to GKE in a production environment.
    - Transition traffic from the OpenStack environment to GCP.
    - Monitor the application closely after the cutover.
    - Decommission the OpenStack infrastructure.

5. Considerations

- **Cost:**
    - GCP offers a variety of pricing models, including sustained use discounts, committed use discounts, and preemptible VMs, to optimize costs.
    - Using managed services like GKE and Cloud SQL reduces operational costs associated with managing infrastructure.
    - A detailed cost analysis should be performed based on the projected usage and the specific GCP services used. Google provides a pricing calculator to help estimate costs.

- **Reliability:**
    - GCP offers high availability and fault tolerance through its global infrastructure, redundant data centers, and managed services.
    - GKE provides features like automatic node repair, horizontal scaling, and rolling updates to ensure application availability.
    - Cloud SQL and other managed database services offer built-in replication and failover capabilities.
- **Security:**
    - GCP provides robust security features, including:
        - Identity and Access Management (IAM) for granular control over access to resources.
        - VPC Service Controls to establish security perimeters around sensitive data.
        - Cloud Security Command Center for security monitoring and threat detection.
        - Regular security updates and patches.
    - It is crucial to follow GCP security best practices and implement the principle of least privilege. Storing Terraform state in JFrog Artifactory also enhances its security.
- **Best Practices:**
    - Adopt infrastructure as code (IaC) using Terraform to automate infrastructure provisioning and management.
    - Use containerization and orchestration (Docker and Kubernetes) to improve application portability, scalability, and resilience.
    - Implement a robust CI/CD pipeline to automate the build, test, and deployment process.
    - Use managed services whenever possible to reduce operational overhead.
    - Monitor application and infrastructure performance and health using Cloud Monitoring and Logging.
- **Lock-in:**

    - While using GCP services introduces some level of vendor lock-in, the use of open-source technologies like Kubernetes and Docker helps mitigate this risk.
    - Designing the application architecture to be cloud-agnostic can also reduce lock-in.
    - It's important to weigh the benefits of managed services against the potential for lock-in.
- **Agile Development:**
    - GCP's services and the proposed architecture support agile development by enabling:
        - Rapid provisioning of development and test environments.
        - Automated testing and deployment.
        - Scalability to handle rapid growth.
        - Microservices architecture (with GKE) for independent development and deployment of application components.
- **Terraform State Management:**
    - Using JFrog Artifactory to store and manage Terraform state files provides several advantages:
        - **Centralized Storage:** Provides a single source of truth for Terraform state, improving collaboration and reducing the risk of inconsistencies.
        - **State Locking:** Supports state locking, preventing concurrent modifications and potential corruption of the state file.
        - **Version Control:** Artifactory can version Terraform state files, allowing us to track changes and revert to previous states if necessary.
        - **Access Control:** Artifactory's fine-grained access control features enable us to manage who can access and modify Terraform state.
        - **Security:** Artifactory provides secure storage for sensitive Terraform state data.

References

- Google Cloud Documentation: This is the official documentation for Google Cloud Platform, providing comprehensive information on all GCP services,

including Compute Engine, Kubernetes Engine, Cloud Storage, and more. It includes tutorials, how-to guides, and API references.

    - https://cloud.google.com/docs/

- Terraform Documentation: The official documentation for Terraform, an infrastructure-as-code tool. It explains how to use Terraform to define and provision infrastructure on various cloud providers, including GCP. It covers the Terraform language, providers, and best practices.
    - https://developer.hashicorp.com/terraform/docs
- Kubernetes Documentation: The official documentation for Kubernetes, an open-source container orchestration system. It provides information on how to deploy, scale, and manage containerized applications. It includes concepts, tutorials, and reference documentation.
    - https://kubernetes.io/docs/
- Docker Documentation: The official documentation for Docker, a platform for developing, shipping, and running applications in containers. It explains how to use Docker to containerize applications and manage container images.
    - https://docs.docker.com/
- JFrog Artifactory Documentation: The official documentation for JFrog Artifactory, a universal artifact repository manager. It provides information on how to use Artifactory to store and manage various types of artifacts, including build artifacts, Docker images, and Terraform state files.
    - https://www.jfrog.com/confluence/display/JFROG/JFrog+Artifactory