AUTOMATION OF DEVELOPING APPLICATION(API) USING DEVOPS(CICD)

Dhanush G
Department of Computer Science And Engineering
Panimalar Engineering College ,Chennai

Ezhil Velan S
Department of Computer Science And Engineering
Panimalar Engineering College,Chennai

ABSTRACT - The project demonstrates the automation of API application development and deployment using a CI/CD pipeline implemented on Google Cloud Platform (GCP). A Python Flask application, fetching data from the open source endpoint, is containerized using Docker and deployed to Google Cloud Run. The entire process, from code commit to deployment, is automated using Google Cloud Build. This approach ensures efficient and reliable deployment of the API.

Keywords: Version Control, Continuous Integration, Continuous Delivery, Containerization, Infrastructure as Code.

1. INTRODUCTION

This project demonstrates a modern approach to developing and deploying a web application programming interface (API). The API, written in Python, RetrieveFs and presents real-time data. The key innovation lies in the fully automated deployment process, continuous integration and continuous deployment (CICD) pipeline built on Google Cloud Platform (GCP). This automation mainly streamlines the development, leading to faster releases, reduced errors, and improved reliability. The project highlights the benefits of DevOps practices for efficient software delivery.

1.1 EXISTING SYSTEM

In traditional software development models, application development and deployment, including APIs, follow a manual process that can often be slow and error-prone. The Key aspects of an existing system: Manual Code Integration, Manual Testing, Slow and Manual Deployments, Lack of Collaboration and Inconsistent Environments.

1.2 PROPOSED SYSTEM

In a DevOps environment, automating the entire process from development to production ensures faster, reliable, and repeatable deployments. The proposed system focuses on using Continuous Integration (CI) and Continuous Deployment (CD) to streamline API development and deployment. The Key aspects of proposed system: Automated Code Integration (CI), Automated Testing Pipelines, **Infrastructure as Code (IaC),** Continuous Deployment (CD) and Monitoring and Logging.

2. PROCESS FLOW

The typical process flow for automating the development of an application (API) using DevOps principles and CI/CD pipelines:

2.1 Code Development and Source Control

Developer Writes Code: Developers write the API code and the corresponding unit and integration tests.

Code Commit: The code is committed to a version control system (e.g., GitHub, GitLab, Bitbucket).

2.2 Continuous Integration (CI) Trigger

CI Pipeline Triggered: A push to the main branch or a pull request automatically triggers the CI pipeline.

Unit Tests Execution: Unit tests are run to validate the functionality of the individual components of the application.

2.3 Build and Packaging

Containerization: The application is packaged into a Docker container using a Dockerfile.

Artifact Storage: The container images or other binaries are stored in an artifact repository.

Infrastructure as Code (IaC): The infrastructure required to deploy the API (e.g., databases, load balancers, networking) is defined using IaC tools like Terraform, CloudFormation, or Ansible.

2.4 Continuous Deployment (CD)

Deploy to Staging Environment: After successful artifact creation, the CD pipeline deploys the API to a staging environment for further testing.

2.5 Automated API Testing

Performance Testing: Tools like JMeter or k6 test the performance and scalability of the API.

Functional and Integration Testing.

2.7 Continuous Delivery or Deployment to Production

Production Deployment: The API is deployed to the production environment using container orchestration tools (e.g., Kubernetes, ECS, OpenShift) or other deployment mechanisms (e.g., serverless, VMs).

2.8 Post-Deployment Verification

Post-Deployment Testing: API endpoints are tested post-deployment to verify successful deployment (e.g., automated API ping or health check).

3. MODULES

In the process of automating the development of an application (API) using DevOps and CI/CD, various modules or components come together to ensure the system runs efficiently. Each module is responsible for specific tasks like source control, build management, testing, deployment, and monitoring. Below is an outline of key modules typically involved in this automated workflow:

3.1 Source Control Module

Version Control System (VCS): This module manages the source code, including collaboration, versioning, and branch management.

Examples: Git (GitHub, GitLab, Bitbucket)

3.2 Build Automation Module

Build System: Automates the process of compiling the application and packaging it into artifacts (e.g., binaries, containers).

Examples: Docker (for containers)

Key Functions: Dependency resolution, Compiling and building source code.

3.3 CI/CD Pipeline Module

Continuous Integration (CI) and Continuous Deployment (CD): Orchestrates the entire development-to-deployment workflow, from code commit to deployment across various environments.

Key Functions: Automatically triggering pipelines on code pushes, merges, or pull requests.

3.4 Artifact Repository Module

Artifact Management: This module manages build artifacts, ensuring they are versioned, stored securely, and are available for deployment.

3.5 Infrastructure as Code (IaC) Module

Infrastructure Automation: Automates the provisioning, configuration, and management of infrastructure (cloud, on-premises, or hybrid) using code.

Example: AWS CloudFormation

3.6 Containerization Module

Containerization: Automates the packaging of applications and APIs into lightweight, portable containers.

Example: Docker.

4. DEVELOPMENT ENVIRONMENT

Creating a development environment for automating application development using DevOps involves several key components and practices. Here's a structured approach to set up an effective environment:

4.1 Version Control System (VCS)

Git: Use Git for source code management. Platforms like GitHub, GitLab, or Bitbucket provide repositories to manage code changes, branches, and collaboration.

4.2 Continuous Integration/Continuous Deployment (CI/CD)

CI/CD Tools: Implement tools like Jenkins, CircleCI, GitLab CI, or GitHub Actions to automate building, testing, and deploying applications.

Pipeline Configuration: Define pipelines to automatically run tests, build applications, and deploy them to staging or production environments.

4.3 Containerization

Docker: Use Docker to create containers for your applications. This ensures consistency across development, testing, and production environments.

Docker Compose: Use it to define and run multi-container Docker applications.

4.4 Infrastructure as Code (IaC)

Terraform or CloudFormation: Use IaC tools to provision and manage cloud infrastructure. This allows you to version control your infrastructure setup.

Configuration Management Tools: Use tools like Ansible, Puppet, or Chef to automate the configuration and management of servers.

4.5 Environment Management

Development and Staging Environments: Create separate environments for development, testing, and production to isolate changes and ensure stability.

Virtual Environments: Use tools like virtual env for Python or npm for Node.js to manage dependencies locally without conflicts.

4.6 Monitoring and Logging

Monitoring Tools: Implement tools like Prometheus, Grafana, or New Relic to monitor application performance and health.

Centralized Logging: Use ELK Stack (Elasticsearch, Logstash, Kibana) or similar tools for logging and analyzing application logs.

4.7 Collaboration Tools

Communication Platforms: Use Slack, Microsoft Teams, or similar tools for team communication.

Documentation: Maintain clear documentation using tools like Confluence or Markdown files in the repository.

4.8 Testing Frameworks

Automated Testing: Integrate testing frameworks (e.g., Selenium for UI tests, JUnit for Java applications, or pytest for Python) into your CI/CD pipelines to ensure code quality.

4.9 Security Practices

Security Scanning: Implement tools for static and dynamic security scanning (e.g., Snyk, Aqua Security) to identify vulnerabilities early in the development cycle.

Secrets Management: Use tools like HashiCorp Vault or AWS Secrets Manager to manage sensitive information securely.

4.10 Feedback and Improvement

Regular Retrospectives: Conduct regular reviews of the development process and CI/CD performance to identify bottlenecks and areas for improvement.

5.ARCHITECTURE

The architecture for automating application development using DevOps focuses on integrating various practices, tools, and methodologies to streamline the development, testing, deployment, and monitoring processes. Here’s a structured view of a typical DevOps architecture:

5.1 Source Code Management

Version Control System (VCS): Centralized repository (e.g., Git) to manage code, track changes, and collaborate among developers.

Branching Strategy: Use Git branching strategies (e.g., Git Flow, trunk-based development) to facilitate feature development and release management.

5.2 Continuous Integration (CI)

CI Server: Tools like Jenkins, GitLab CI, or CircleCI automatically build the application and run tests whenever code is pushed to the repository.

Build Artifacts: The CI process generates build artifacts (e.g., JAR files, Docker images) for deployment.

5.3 Automated Testing

Unit Testing: Run unit tests as part of the CI pipeline using frameworks like JUnit, pytest, or Mocha.

Integration Testing: Automated integration tests ensure that different modules work together as expected.

End-to-End Testing: Use tools like Selenium or Cypress for UI testing, ensuring that user interactions function correctly.

5.4 Continuous Deployment (CD)

Deployment Pipeline: Automated processes to deploy applications to various environments (development, staging, production) after successful CI.

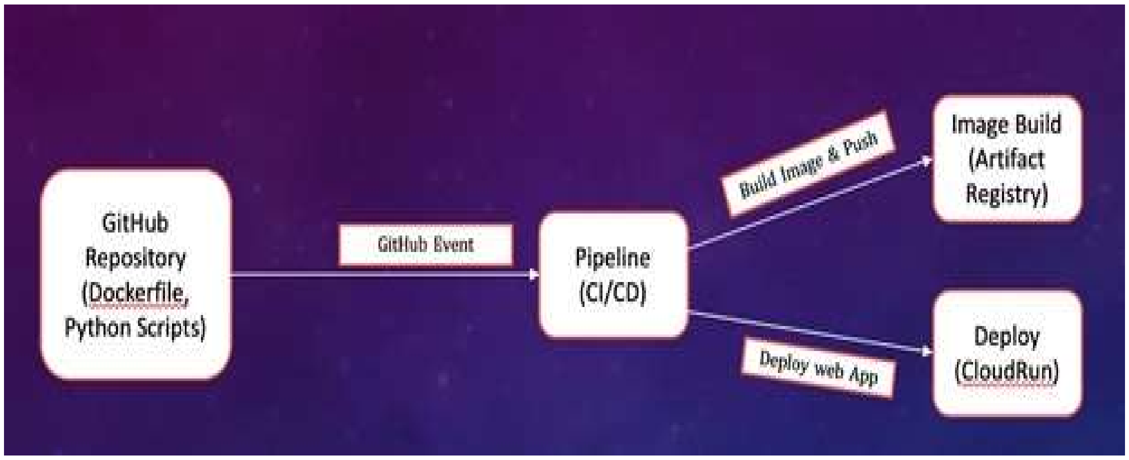
5.5 Containerization and Orchestration

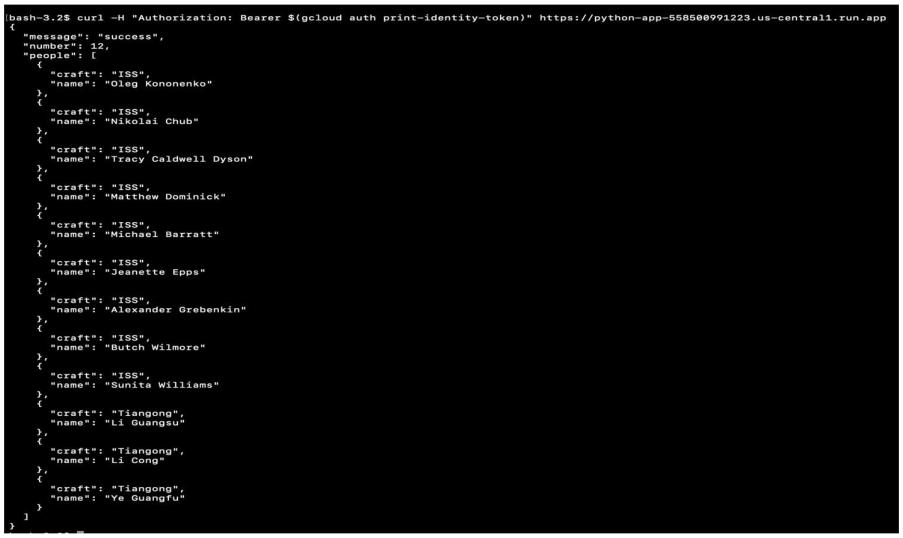
Docker: Use containers to package applications and their dependencies, ensuring consistency across environments.

5.6 Infrastructure as Code (IaC)

Provisioning: Use Terraform or AWS CloudFormation to provision and manage infrastructure resources programmatically.

Environment Configuration: Manage configurations for different environments through code, ensuring reproducibility and version control.


GitHub Repository (Dockerfile, Python Scripts)
GitHub Event
Pipeline (CI/CD)
Build Image & Push
Image Build (Artifact Registry)
Deploy web App
Deploy (CloudRun)



```
bash-3.2$ curl -H "Authorization: Bearer $(gcloud auth print-identity-token)" https://python-app-558500991223.us-central1.run.app
{
  "message": "success",
  "number": 12,
  "people": [
    {
      "craft": "ISS",
      "name": "Oleg Kononenko"
    },
    {
      "craft": "ISS",
      "name": "Nikolai Chub"
    },
    {
      "craft": "ISS",
      "name": "Tracy Caldwell Dyson"
    },
    {
      "craft": "ISS",
      "name": "Matthew Dominick"
    },
    {
      "craft": "ISS",
      "name": "Michael Barratt"
    },
    {
      "craft": "ISS",
      "name": "Jeanette Epps"
    },
    {
      "craft": "ISS",
      "name": "Alexander Grebenkin"
    },
    {
      "craft": "ISS",
      "name": "Butch Wilmore"
    },
    {
      "craft": "ISS",
      "name": "Sunita Williams"
    },
    {
      "craft": "Tiangong",
      "name": "Li Guangsu"
    },
    {
      "craft": "Tiangong",
      "name": "Li Cong"
    },
    {
      "craft": "Tiangong",
      "name": "Ye Guangfu"
    }
  ]
}
```

CONCLUSION

The project successfully demonstrated the implementation of a CI/CD pipeline for a Python API application on Google Cloud Run. While the automated deployment process worked effectively, we encountered challenges in deploying in clourun with port. After added port information in app.py, it was deploying successfully. These challenges highlighted the importance of careful planning and configuration when working with cloud-based infrastructure. The project also underscored the value of comprehensive testing, emphasizing the need for robust automated testing in future iterations. This experience provided valuable lessons in cloud deployment best practices and reinforced the importance of thorough planning and testing in a CI/CD environment.

ACKNOWLEDGEMENT

We would like to acknowledge the contribution of all the involved in reviewing this paper. We would also like to thank our families and friends who supported us in the course of writing this paper Last but not the least we would like to thank all our peers who greatly contributed to the completion of this project with their constant support and help.

REFERENCES

[1] **Google Cloud Platform Documentation**:

Cloud Run. [https://cloud.google.com/run]

(https://cloud.google.com/run).

[2] **Open Notify API Documentation:**

https://open-notify.org/Open-Notify-API/

[3] **Cloud Build:**

https://cloud.google.com/build/docs/overview

[4] **Cloud Artifact Registry:**

https://cloud.google.com/artifact-registry/docs/overview

[5] **Docker Overview:**

https://docs.docker.com/get-started/docker-overview/