DevOps Strategy Proposal

Introduction

The purpose of this document is to list the architectural decisions related to the development and deployment process in order to achieve several goals:

- Unify the tooling and reduce the diversity of tools competing for the same problem space. This is required to reduce the number of mandatory skills put in front of the prospective operator or user of this environment. It enhances the clarity of the system, reducing the time for understanding its internals.
- Select the components that are battle-tested and well-known, to avoid haze and uncertainty of bleeding edge and experimental features, while at the same time inheriting a wealth of information and guides related to the usage and best practices. This enhances the stability and reliability of the system.
- Select the components that are developer-friendly, reducing the necessary ramp-up time and avoiding steep learning curves. This is enhancing developer adoption, and gapping the bridge between development and operations. With this approach DevOps role is somewhat reduced and a tip of the scale is moved toward the developers, empowering them to be more proactive and deploy faster without requiring large DevOps effort.
- Use the most cost-effective solutions. While some of the turn-key-ready solutions are great to have, pricing and policies with vendor lock-ins are big obstacles for seemingly easy solutions. Components are to be selected with pricing in mind.
- Reduce the operational costs of running a system to the bare minimum, preferably without any significant effort on the DevOps side. This was set as a hard requirement and impacted quite a few decisions, ruling out any complex orchestration solution that requires high maintenance or upkeep. With reduced complexity, it also brought a certain set of restrictions to the table.
- Keep the system open enough that any architectural decision or limit that is conflicting with the specific project need, can be alternatively circumvented. This leads to a more flexible system, but in turn, also keeps the system prone to potential diversity based on decisions and this in particular needs to be addressed with care.

Assumption

A goal is to design a very simple system, suitable for a small development team without a dedicated DevOps engineer, which can operate and make deployment of a single server type of the application fast and cheap in terms of development and operation costs.

When reading the document, please keep in mind the stated goal and constraints it is imposing on the particular tool selection.

Table of contents

Infrastructure provider

Given the “small team with single server” idea, deployment is suggested on the cloud solutions. Between several maror providers, such as AWS, GCP, Azure, DigitalOcean, and similar offers Hetzner stands out as best in price/reliability range and is suggested as a go-to solution. Even if Hetzner is not selected, or if over time it changes its pricing policies, the overall suggestion is to treat servers from an infrastructure standpoint as ephemeral and reachable on IP-address bases so they can be interchangeably used from any cloud provider, and migration is not entangled in specifics of vendor locked in components. Although for quick prototyping a vendor-specific solution can be tempting, try to steer away from it or at least always wrap it up in such a way that it can be easily replaceable in case it proves to be too costly or unmanageable for any reason.

Code Repository

Although multiple options are available, the suggested solution proposed here is usage of the GitHub, based primarily on the adoption and developer familiarity with it. In case a self-hosted solution is mandatory, GitLab is a viable alternative offering the ability to escape any potential vendor lock-in or compliance issues with GitHub. Since the target audience is multiple small teams, separate repositories on a per-project basis are suggested to be able to fine-tune the access rights and code ownership. Although usually more practical from a DevOps perspective, for security and team coordination reasons, monorepos should be avoided, or potentially used sparingly on larger projects that require a unifying layer across several services in which case a monorepo could be a viable approach for such projects.

Packaging

If we take into account the complexity of the project, the proposed solution might be too simple to cover specific needs. That is why two layers are offered here. One for a simpler architecture, often with some sort of service-oriented architecture, having for example basic database, backend, and frontend structure. Another is for more involved, more complex solutions requiring more than a simple server and small team, with frequent changes, automatic failovers, role-based access policies, automatic provisioning of resources, and integrations for industry-standard DevOps tools.

For both of the solutions, docker containers are suggested. In comparison with, for example, LXC it is more adopted among developers, making it an ideal bridge between development that is often executed in local environments and packaging for target environment deployment. In the selection of the docker container the guideline of familiarity of the developers while providing the most unified platform and usage of single tools were guiding factors. This approach is known as containerization, more specifically dockerization and it is more detailed further in the document.

Orchestrator

Once docker was selected for packaging, the choice of orchestrator was obvious. For small projects orchestration is achieved with docker compose. It can easily spin up multiple services, with databases, secret hiding, volume mounting, and environment variable handling, detailing all the configuration details of a running application within its compose file. Even docker embraced compose as an orchestrator and ships it with the docker itself. The limit of a single server per one compose file can be circumvented with multiple compose files to cover the project needs. For more involved projects that grow beyond composing limits a dedicated Kubernetes cluster is advised. It will use the same packaging principle (where the same docker image is the unit of packaging) but the unit of deployment becomes a pod with all the richness of Kubernetes resources in tow. This, in turn, requires the boot-up of a cluster (whether it is managed with k0s, k3s, kubeadmin, or any other way) with all overhead of such a decision.

For that reason, a strong emphasis was given on the “single server small team” goal, and compose usage is advised in the first place with Kubernetes (or some other tool such as Nomad, OpenShift, or similar) as a necessity only.

Pipeline

Another important consideration is a pipeline. There are a range of options there. Sometimes it even boils down to a personal belief rather than a sound comparison similar to the choice of Linux distribution or editor for coding. To resolve this knot, although we strongly lean toward open source solutions such as self-hosted GitLab or Jenkins on one hand, the deciding factor, in this case, is developers' familiarity with the tool, and willingness to adopt it. As GitHub was selected as a primary code repository it makes best sense to use GitHub Actions and reduce the number of the required tools to master the system. The majority of developers are already quite familiar with GitHub and quite a few of them are familiar with Actions as well. This makes it an ideal candidate for the tasks, again, just from the perspective of limiting the diversity of the tools. In case any of the limits of Actions are hit (number of runners, policies, etc…), they can be easily replaced with anything more appropriate. If a push-based approach is exhausted, a pull-based Kubernetes native ArgoCD approach can be used instead. Finally, they all have to fit into the GitOps approach.

Secrets handling

Additional discussion is given to secret handling, and with GitHub + Actions in selection only natural choice is to use GitHub Secrets. This usually covers the needs of a small team and project well and will be superseded only in case of requirements for layered secret handling with granular access and automatic rotation policies.

Monitoring

Final consideration is given to monitoring and logging solutions in which case amongst a myriad of choices, the open-source stack with Grafana, Prometheus, and Loki is advised. For quick prototyping, GrafanaCloud in the free tier is suggested. This way a small team can achieve cost efficiency and avoid vendor lock-in. The toolset can collect and process logs, metrics, and traces in a one-rounder solution without costing arm and leg. Keep in mind that the team must be very conscious of the volume of logging data and retention periods as they can easily grow to get out of hand and hammer the team with operational and performance issues. With this said, the team must exercise due diligence on logging formats, verbosity, and extracted value since there is no silver bullet that can correct a bad logging approach afterward, suggested solutions included.

Documentation

Since GitHub is selected as the primary code repository, and given a small team with limited bandwidth to write technical documentation (no dedicated writers or developers are covering it), the most viable approach is to use GitHub Markdown pages kept in the repository. If this is too restrictive for project needs, markdown files can further be repurposed into a mdbook or similar tool to create more elaborate documentation. For API-specific cases, swagger and similar tools with the ability to code-mark and auto-generate documentation can be utilized.

Conclusion

Given the focus on using Docker compose or Kubernetes on Hetzner Cloud for your CI/CD processes with GitHub Actions, the strategy needs to incorporate specifics about managing Kubernetes clusters, including deployment strategies, managing infrastructure as code, and automating workflow pipelines. This revised proposal will outline the approach to seamlessly integrate these technologies.

Docker Compose on Hetzner Cloud

As detailed in the above discussion, considering the limits imposed, this is suggested as the fastest and most cost-effective way for a developer in a small team to deploy on a single server per single compose file and get the prototype up fast and without incurring big costs. This approach does not require dedicated DevOps effort, nor does it lean on heavy automation or HA features. This is just a small step from getting out of the local development environment fast while having some sort of prototype online solution to present in a tight cycle without any coordination overhead. The benefit of this rather basic approach is that although it is limited, fast, and simple - it still produces Docker containers which can be later on used in more elaborate deployment approaches. It also produces a compose file detailing requirements and interfaces required for the docker image to run mainly in terms of volume mounts, environment variables, exposed ports, etc, which can be used as a source of truth when migrating to more elaborate setups. In that sense efforts invested in the Dockerization and usage of compose files are not wasted if the project lives up to the expectation and grows up from some small proof of concept to a more elaborate system.

Note also that a single project can have multiple compose files if a wee bit more complexity is needed but not just yet to warrant a fully-fledged Kubernetes cluster. That way a limitation of deployment on a single server per one compose file can be circumvented, with the overhead that it carries.

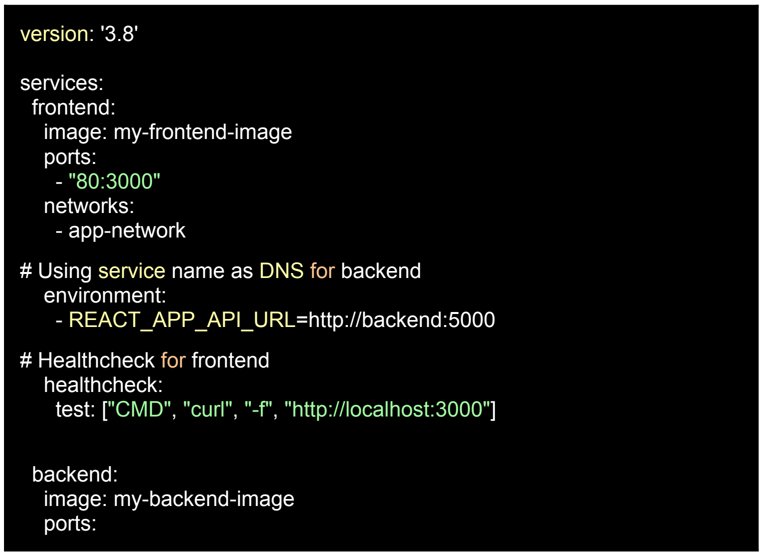
One of the specifics to bear in mind is that in the process of interacting with the blockchain teams tasked with particular projects will most likely need to spin their instances of some of Cardano tools in order to develop new functionality. General examples include requirements to run instances of Cardano-node, ogmios, db-sync, wallet-API, submit-API, blockfrost and similar. They fit nicely in this approach as all of the core tollings either already provide official docker images or even come with docker-compose files and can be easily reconfigured to target a specific Cardano network or run with customized configs as per project needs. Note that there are some more involved components (such as Cardano Foundation’s cf-explorer) requiring more resources and elaborate setup such as Kubernetes/ArgoCD.

To deploy Docker Compose on Hetzner Cloud, you can follow these steps:

- Set up a Hetzner Cloud Account**:** Sign up for an account on Hetzner Cloud and obtain your API token.
- Create a Virtual Machine (VM) or Instance**:** Log in to your Hetzner Cloud Console and create a new VM instance. Choose the appropriate specifications for your needs.

- SSH into the Instance: Once the VM is created, SSH into it using the provided IP address and your SSH key.
- Install Docker and Docker Compose: Update your package manager and install Docker and Docker Compose. You can find instructions for installing Docker on your specific Linux distribution on the Docker website. For Docker Compose, you can follow the official installation guide: Install Docker Compose.
- Upload Your Docker Compose File: Transfer your docker-compose.yml file to the server. You can use the scp command or any other method you prefer.
- Run Docker Compose: Once the docker-compose.yml file is on the server, navigate to the directory containing the file and run docker-compose up -d to start your services in detached mode.
- Verify Deployment: Check if your services are up and running by using the `docker ps` command to see the running containers.
- Configure Networking: Ensure that your firewall settings on Hetzner Cloud allow traffic to your services as needed. You might need to configure security groups or firewall rules in the Hetzner Cloud Console.
- Set Up Domain and DNS: If you want to access your services via a domain name, configure DNS settings accordingly. You can use Hetzner Cloud's DNS management or any other DNS provider you prefer.
- Monitor and Maintain: Regularly monitor your services and server for performance, security, and updates. Update your Docker containers, Docker Compose, and the underlying operating system as needed.

Here is an example of Docker Compose with Frontend and Backend along with Mongo Database:



```
version: '3.8'

services:
  frontend:
    image: my-frontend-image
    ports:
      - "80:3000"
    networks:
      - app-network

# Using service name as DNS for backend
    environment:
      - REACT_APP_API_URL=http://backend:5000

# Healthcheck for frontend
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:3000"]


  backend:
    image: my-backend-image
    ports:
      - "5000:5000"
    networks:
      - app-network

 # Using service name as DNS for MongoDB

    environment:
      - MONGO_URL=mongodb://mongo:27017/mydatabase
     secrets:
      - db_password

# Healthcheck for backend
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:5000"]

   mongo:
    image: mongo
    ports:
      - "27017:27017"
    networks:
      - app-network
    volumes:
      - mongo-data:/data/db
    healthcheck:
      test: ["CMD", "mongo", "--eval", "db.adminCommand('ping')"]

secrets:
  db_password:
    external: true

volumes:
  mongo-data:

networks:
  app-network:
```

Start Your Docker Containers:

- Navigate to the directory where your docker-compose.yml file is located.
- Run the following command to start your Docker containers:

```
docker-compose up -d
```

Accessing Your Services:

- If you've set up services like web servers, databases, etc., they should now be running.
- Access your services using your server's IP address and the appropriate port, if necessary.

Managing Docker Compose:

- You can stop your containers using:

```
docker-compose down
```

- You can restart your containers using:

```
docker-compose restart
```

Explanation:

- Secrets**:** The db_password secret is loaded from an external secret management system.
- Volumes: A named volume mongo-data is created to persist MongoDB data.
- Networks: The app network is created for inter-service communication.
- Frontend: Exposes port 80, sets the API URL environment variable pointing to the backend service and defines a health check using curl.
- Backend**:** Exposes port 5000, sets the MongoDB URL environment variable, and includes the db_password secret. It also defines a health check using curl.
- MongoDB**:** Exposes port 27017, mounts a volume for persistent data storage and sets a health check using a MongoDB ping.

To deploy this Docker Compose configuration on Hetzner Cloud, follow similar steps as mentioned earlier, ensuring that your secrets are properly managed and accessed according to Hetzner Cloud's guidelines, and that network and volume configurations are appropriately set up.

Kubernetes with GitHub Actions on Hetzner Cloud

This strategy proposes the implementation of a Continuous Integration/Continuous Deployment (CI/CD) pipeline using GitHub Actions, specifically designed for applications deployed on Kubernetes clusters hosted on Hetzner Cloud. This approach emphasizes automation, scalability, and efficiency, leveraging the strengths of Kubernetes for container orchestration and GitHub Actions for workflow automation.

Objectives

- Automated Kubernetes Deployments: Streamline the process of deploying applications to Kubernetes clusters, ensuring consistent and reliable deployment practices.
- Scalable and Efficient Infrastructure Management: Utilize Infrastructure as Code (IaC) principles to manage Hetzner Cloud resources and Kubernetes configurations, facilitating scalable and efficient infrastructure management.
- Enhanced Development Workflow: Implement CI/CD pipelines that integrate with GitHub repositories, automating builds, tests, and deployments, thereby enhancing productivity and collaboration within development teams.
- Robust Application Delivery: Ensure robustness in application delivery with strategies for rolling updates, canary releases, and blue-green deployments, minimizing downtime and risk during deployments.

Cloud Service Providers

Cloud service providers typically offer a range of services at various price points, and the cost can vary depending on factors such as the specific services used, the level of performance required, and the amount of data stored or processed. However, there's a general list of some more popular cloud service providers:

- Google Cloud Platform (GCP): Google Cloud often offers competitive pricing, particularly for services like computing and storage.
- Alibaba Cloud: Alibaba Cloud can provide competitive pricing, especially in the Asia-Pacific region, though costs may vary depending on services and regions.
- IBM Cloud: IBM Cloud offers a range of pricing options tailored to different industries and use cases, positioning itself as a competitive option for enterprise customers.
- Hetzner: From the price point perspective, a very interesting choice for developers, and for that particular reason suggested as the choice. Note that this is solely based on financial reasoning having a small budget with a rather reasonable growth model in mind without vendor lock-in.
- DigitalOcean: Another popular alternative, with competitive pricing compared to offers with more elaborate features.
- Amazon Web Services (AWS): AWS, being one of the largest cloud providers, offers a wide range of services at various price points. While generally competitive, it might be slightly more expensive for certain use cases compared to other providers.
- Microsoft Azure: Azure is known for its competitiveness with AWS but may have slightly higher pricing for certain services or configurations.
- Oracle Cloud: Oracle Cloud is often perceived as one of the more expensive options, particularly targeted towards enterprise-level solutions.

This is a very volatile arena and activities or free offers can muddy the waters considerably with sales pitches usually targeted to vendor lock-in that as the project grows inflates the costs greatly. In general, the base comparison of single cloud servers of similar capabilities across providers is leaning currently to Hetzner's side at the time of writing this document.

Proposed Solution

Infrastructure Management

- Terraform Setup for Hetzner Cloud
  - Use Terraform to provision and manage Hetzner Cloud resources, including networks, servers, and storage, with a focus on setting up Kubernetes clusters.
- Kubernetes Cluster Configuration:
  - Leverage Hetzner Cloud's Kubernetes service or manually configure Kubernetes clusters using Terraform, ensuring clusters are optimized for your application needs.

CI/CD Pipeline Configuration with GitHub Actions

- Continuous Integration
  - Configure GitHub Actions workflows to automate the build and testing of application containers upon each commit or pull request. This includes linting code, running unit and integration tests, building Docker images, and uploading to the container registry.
- Continuous Deployment
  - Set up a GitHub Actions workflow to automate the deployment of Docker images to Kubernetes clusters. This workflow should handle updating Kubernetes deployments with new image versions, managing secrets, and applying Kubernetes configuration changes.

Kubernetes Deployment Strategies

- Deployment Strategies
  - Implement deployment strategies such as rolling updates, canary releases, or blue-green deployments to ensure zero downtime and high availability during application updates.
- Monitoring and Rollback
  - Integrate monitoring tools to observe application performance and automate rollback procedures in case of detected anomalies post-deployment.

Security and Compliance

- Secrets Management
    - Utilize GitHub Secrets and Kubernetes Secrets for secure storage and handling of sensitive information like API tokens, SSH keys, and application credentials.
- Security Scanning
    - Incorporate container image scanning and Kubernetes configuration scanning in the CI pipeline to identify and mitigate vulnerabilities before deployment.

Implementation Plan

Infrastructure Setup:

- Provision of Hetzner Cloud resources and Kubernetes clusters using Terraform.
- Configure Kubernetes clusters, including namespaces, service accounts, and resource quotas.

Pipeline Configuration:

- Define GitHub Actions workflows for CI (build/test) and CD (deploy) phases, including Docker image creation and Kubernetes deployments with ArgoCD as a GitOps tool.
- Store necessary secrets in GitHub for use in CI/CD workflows.

Deployment Strategy Implementation:

- Implement chosen Kubernetes deployment strategies in GitHub Actions workflows, ensuring seamless and safe application updates.

Monitoring and Optimization:

- Set up application and infrastructure monitoring using Grafana stack which includes Prometheus as a data source for metrics and Loki as a data source for logs
- Continuously review CI/CD pipelines and Kubernetes configurations for optimization opportunities.

Benefits

- Faster Time to Market: Automated pipelines reduce lead time for changes, enabling faster feature releases and fixes.

- High Availability: Kubernetes deployment strategies ensure applications remain available during updates, enhancing user satisfaction.
- Improved Productivity: Automation frees development teams from manual tasks, allowing them to focus on feature development and innovation.
- Scalability: Kubernetes and Hetzner Cloud provide a scalable infrastructure that can grow with the application demands.

Orchestration and Containerization

- Containerization: Developers define the environment for their applications using Dockerfiles. A Dockerfile is a text document that contains instructions for building a Docker image. These instructions typically include base images, dependencies, environment variables, and commands to execute.
- Building Docker Images: Once the Dockerfile is created, developers use the docker build command to build a Docker image. This command reads the instructions from the Dockerfile and generates an image that contains the application and all its dependencies.
- Docker Registry: Docker images can be stored in public or private registries such as Docker Hub, Amazon ECR, Google Container Registry, or a self-hosted registry. This allows teams to share and distribute images across different environments.
- Running Containers: Once the Docker image is built, it can be run as a container using the docker run command. Containers can be started, stopped, restarted, and deleted as needed. Docker provides isolation at the process level, allowing multiple containers to run on the same host without interfering with each other.
- Orchestration: For deploying and managing containerized applications at scale, orchestration tools like Docker Swarm, Kubernetes, or Amazon ECS can be used. These tools automate tasks such as deployment, scaling, load balancing, and service discovery.

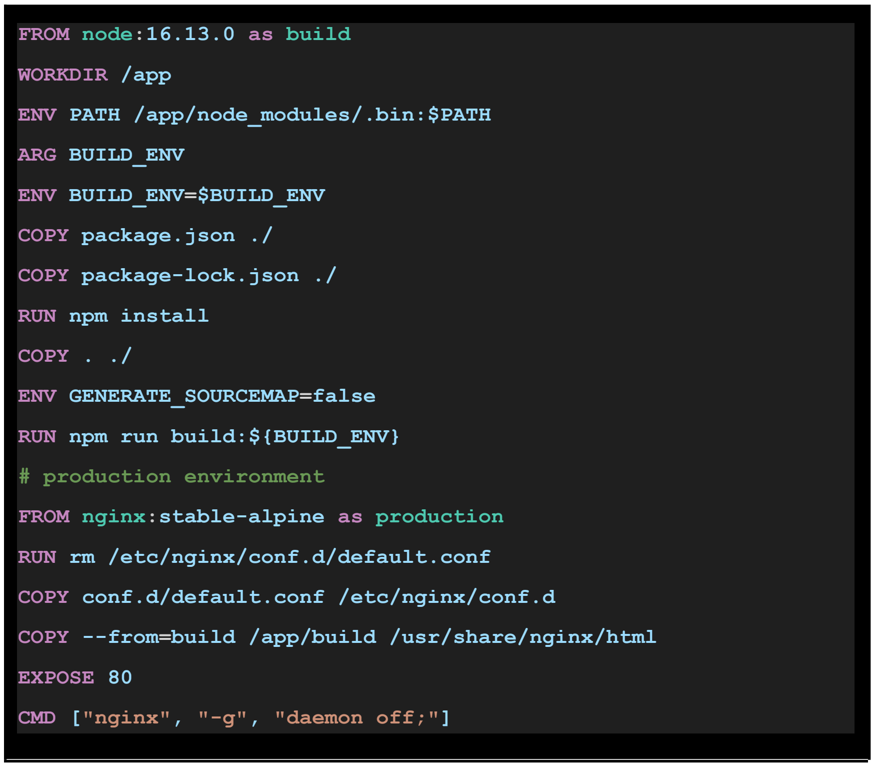
Benefits of Dockerization:

- Portability: Docker containers can run on any platform that supports Docker, making it easy to move applications between development, testing, and production environments.
- Isolation: Containers provide lightweight isolation, ensuring that applications run consistently across different environments without interference from other processes.

- Resource Efficiency: Containers share the host operating system's kernel, reducing overhead and resource usage compared to traditional virtual machines.
- Consistency: Dockerization ensures that all dependencies and configurations required for an application are bundled together, reducing the risk of compatibility issues and making deployments more predictable.

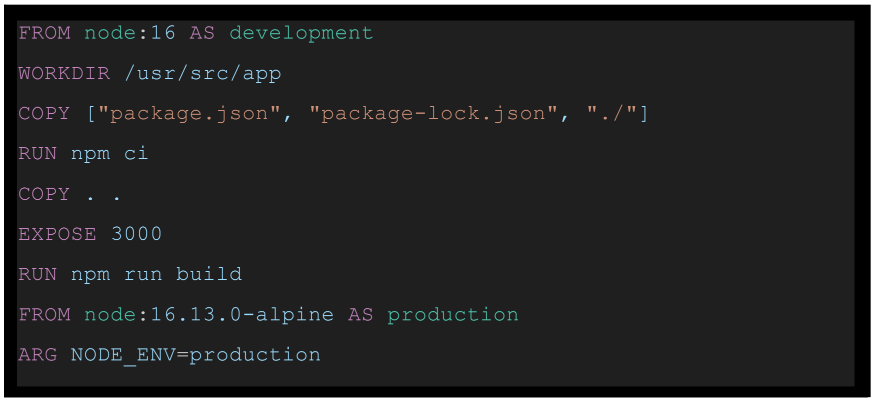
Dockerization simplifies application development, deployment, and management by providing a standardized and portable platform for containerization.

Example of Dockerfile ready for React.js:



```
FROM node:16.13.0 as build
WORKDIR /app
ENV PATH /app/node_modules/.bin:$PATH
ARG BUILD_ENV
ENV BUILD_ENV=$BUILD_ENV
COPY package.json ./
COPY package-lock.json ./
RUN npm install
COPY . ./
ENV GENERATE_SOURCEMAP=false
RUN npm run build:${BUILD_ENV}
# production environment
FROM nginx:stable-alpine as production
RUN rm /etc/nginx/conf.d/default.conf
COPY conf.d/default.conf /etc/nginx/conf.d
COPY --from=build /app/build /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Example of Dockerfile ready for Node.js:



```
FROM node:16 AS development
WORKDIR /usr/src/app
COPY ["package.json", "package-lock.json", "./"]
RUN npm ci
COPY . .
EXPOSE 3000
RUN npm run build
FROM node:16.13.0-alpine AS production
ARG NODE_ENV=production
```

```dockerfile
ENV NODE_ENV=${NODE_ENV}

USER node

RUN mkdir /home/node/app

WORKDIR /home/node/app

COPY ["package.json", "package-lock.json", "./"]

RUN npm ci --only=production && npm cache clean --force

COPY --chown=node:node . .

COPY --from=development --chown=node:node /usr/src/app/dist ./dist

EXPOSE 3000

CMD ["node", "dist/main.js"]
```

Continuous Integration and Continuous Deployment

GitHub Actions is a powerful CI/CD platform provided by GitHub. It allows you to automate workflows directly within your GitHub repository. You can set up continuous integration using GitHub Actions, as shown:

- Create a Workflow File: In your GitHub repository, create a directory named .github/workflows. Inside this directory, create one or more YAML files to define your workflows. For example, you might create a file named ci.yml.
- Define Workflow: In the YAML file, define the workflow steps using the GitHub Actions syntax. You'll specify the events that trigger the workflow, the jobs to be executed, and the steps within each job.

```
name: CI
on:
push:
branches:
- main
pull_request:
branches:
- main
jobs:
build:
runs-on: ubuntu-latest
steps:
- name: Checkout code
uses: actions/checkout@v2
- name: Setup Node.js
uses: actions/setup-node@v2
with:
node-version: '14'
- name: Install dependencies
run: npm install
- name: Run tests
run: npm test
```

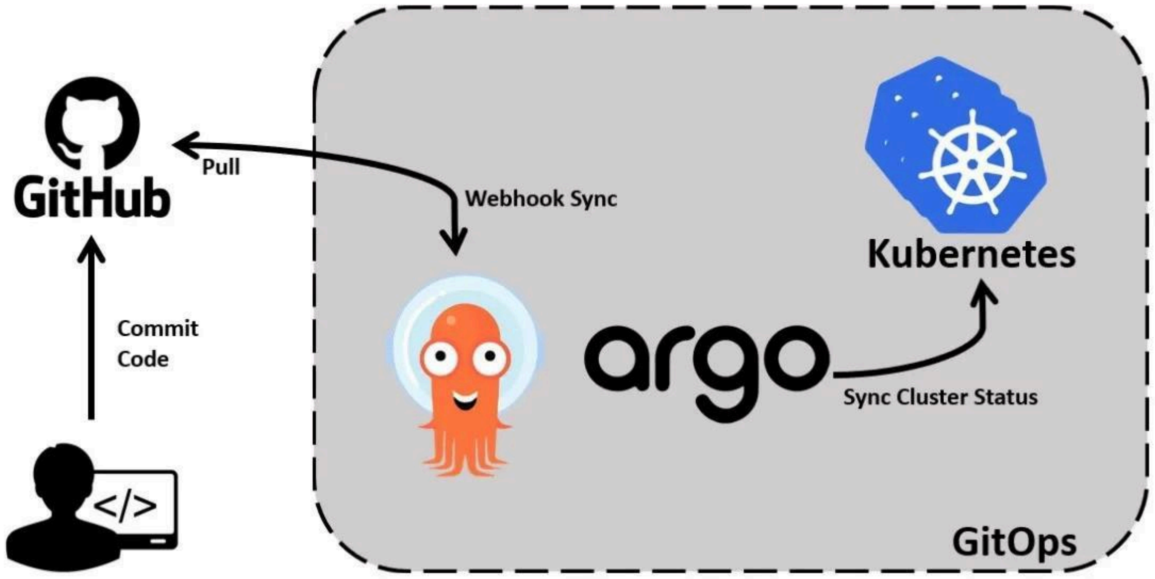
In this example, the workflow triggers push to the main branch and pull requests targeting the main branch. It consists of a single job (build) that runs on the latest version of Ubuntu. The steps check out the code, set up Node.js, install dependencies, and run tests.

- Commit and Push: Save the workflow file and commit it to your repository. Push the changes to GitHub, and GitHub Actions will automatically start executing the workflow based on the trigger events specified.
- Monitor Execution: You can monitor the execution of your workflows on the "Actions" tab of your GitHub repository. GitHub provides logs and status indicators to help you track the progress and outcome of each workflow run.
- Receive Feedback: GitHub Actions will provide feedback on the build status, indicating whether the workflow was completed successfully or encountered any errors. You can review the logs to debug any issues that arise.
- Customize and Expand: GitHub Actions supports a wide range of actions and customization options. You can integrate with external services, deploy to various platforms, and automate additional tasks as needed to suit your specific CI requirements.

By leveraging GitHub Actions for Continuous Integration, you can automate the testing and validation of your code changes, streamlining the development process and ensuring code quality and reliability.

GitOps

ArgoCD is a popular tool used for implementing GitOps practices in Kubernetes environments. GitOps is a methodology for managing infrastructure and applications using Git as the single source of truth. ArgoCD ensures that the desired state of your Kubernetes clusters defined in Git repositories is continuously reconciled with the actual state in the cluster.



Release procedure

The best practices for release procedures in GitHub are the following:

- Push to **develop** branch will trigger build to deploy on the **development** environment
- Creating a tag with the name **UAT-1.0.0** will trigger build to deploy on **uat** environment
- Creating tag with name PROD-1.0.0 will trigger build to deploy on **production** environment

Monitoring

The Grafana Prometheus Loki stack is a combination of open-source monitoring and logging tools commonly used for observability in cloud-native environments. Let's break down each component:

- Prometheus
    - Prometheus is a monitoring and alerting toolkit designed for reliability and scalability. It collects metrics from monitored targets by scraping HTTP endpoints on those targets. These metrics are then stored in a time-series database where they can be queried and aggregated in real time.
    - Prometheus uses a pull-based model, meaning it periodically scrapes metrics from configured targets.
    - It supports a flexible query language called PromQL for querying and aggregating collected metrics data.
- Grafana
    - Grafana is a popular open-source analytics and visualization platform. It allows users to query, visualize, and alert on metrics data from multiple sources, including Prometheus.
    - With Grafana, users can create customizable dashboards with various panels such as graphs, tables, heatmaps, and more to visualize their metrics data.
    - It supports a wide range of data sources, including Prometheus, Graphite, InfluxDB, Elasticsearch, and more.
- Loki:
    - Loki is a horizontally scalable, multi-tenant log aggregation system inspired by Prometheus. It is designed to handle large volumes of log data with high availability and low cost.
    - Unlike traditional log aggregators, Loki does not index log lines. Instead, it indexes metadata about logs (such as labels) and stores the log lines themselves in chunks in object storage like Amazon S3 or Google Cloud Storage.
    - Loki utilizes labels and PromQL-style queries to filter and retrieve log data efficiently. This makes it particularly well-suited for Kubernetes environments where logs are often highly structured and tagged with metadata.
    - Loki also integrates tightly with Grafana, allowing users to query and visualize log data directly in Grafana dashboards using the Loki data source.

Grafana-Prometheus-Loki stack provides a comprehensive solution for monitoring, visualization, and logging in cloud-native environments. Prometheus collects metrics data, Grafana provides visualization and alerting capabilities for that data, and Loki aggregates and indexes log data for querying and visualization alongside metrics in Grafana.

This combination offers powerful observability capabilities for monitoring and troubleshooting applications and infrastructure.

Security integrations

Integrating security practices into the DevOps pipeline is crucial for ensuring that software is developed, tested, and deployed with security in mind from the outset. Here’s the breakdown of the key measures to incorporate automated security scans, code analysis, and other security practices into the DevOps pipeline:

Static Application Security Testing (SAST)

- Integrate SAST tools into the CI/CD pipeline to analyze the source code for security vulnerabilities.
- Tools like Checkmarx, Fortify, and SonarQube can be configured to automatically scan code repositories for potential vulnerabilities, such as SQL injection, cross-site scripting (XSS), and insecure authentication mechanisms.

Dynamic Application Security Testing (DAST)

- Implement DAST tools to assess running applications for security vulnerabilities.
- Tools such as OWASP ZAP and Burp Suite can be automated to simulate attacks on deployed applications and identify vulnerabilities like injection flaws, broken authentication, and insecure configurations.

Dependency Scanning

- Use dependency scanning tools to identify vulnerable third-party libraries and components.
- Tools like OWASP Dependency-Check and Snyk can be integrated into the CI/CD pipeline to analyze dependencies and alert developers to known vulnerabilities in the libraries they're using.

Container Security Scanning

- Incorporate container security scanning to identify vulnerabilities in Docker images and other containerized applications.
- Tools like Anchore and Clair can automatically scan container images for known vulnerabilities in the base image or installed packages.

Infrastructure as Code (IaC) Security

- Apply security best practices to Infrastructure as Code templates (e.g., CloudFormation, Terraform) to ensure secure configurations.
- Use tools like Terraform Sentinel or AWS Config Rules to enforce security policies and detect misconfigurations in cloud environments.

Automated Compliance Checks

- Implement automated compliance checks to ensure that the code adheres to relevant security standards and regulatory requirements.
- Tools like Chef InSpec or Open Policy Agent (OPA) can be used to define and enforce compliance policies as code.

Security Training and Awareness

- Provide ongoing security training and awareness programs for development and operations teams to educate them about secure coding practices and emerging threats.

Incident Response Automation

- Integrate automated incident response workflows to quickly identify and respond to security incidents during development or deployment.
- Implementing tools like security orchestration, automation, and response (SOAR) platforms can help streamline incident response processes.

By incorporating these security practices into the DevOps pipeline, teams can identify and address vulnerabilities early in the development process, reducing the risk of security breaches and ensuring that software is delivered securely and efficiently.

Key Performance Indicators (KPIs)

Defining and tracking Key Performance Indicators (KPIs) and metrics related to DevOps processes is crucial for ensuring the efficiency, effectiveness, and continuous improvement of your software development and operations workflows. Here are some common KPIs and metrics related to DevOps:

- **Deployment Frequency**: This measures how often new code changes are deployed into production. It reflects the speed of your development and deployment processes.
- **Lead Time for Changes**: This measures the time it takes for a code change to go from a commit to being deployed in production. It helps in understanding the efficiency of your development pipeline.
- **Mean Time to Recover (MTTR):** MTTR measures the average time it takes to recover from a system failure. It reflects the resilience and reliability of your infrastructure and deployment processes.
- **Change Failure Rate:** This metric measures the percentage of changes that fail when deployed to production. A high failure rate indicates instability in your deployment processes or code quality issues.

- **Deployment Success Rate:** This measures the percentage of deployments that are successful without any issues. It reflects the reliability and effectiveness of your deployment process.
- **Infrastructure as Code (IaC) Coverage**: This metric measures the percentage of your infrastructure managed as code (e.g., using tools like Terraform or CloudFormation). Higher coverage indicates better automation and consistency in infrastructure provisioning.
- **Code Quality Metrics:** Metrics such as code complexity, code duplication, and code coverage can help assess the overall quality of your codebase. Tools like SonarQube or CodeClimate can provide insights into these metrics.
- **Mean Time Between Failures (MTBF):** MTBF measures the average time elapsed between two consecutive failures. It helps in assessing the reliability and stability of your system.
- **Environment Stability:** This measures the stability of various environments (e.g., development, testing, staging) over time. It helps in identifying environmental issues impacting the deployment process.
- **Resource Utilization**: Metrics related to CPU, memory, disk, and network usage can help in optimizing resource allocation and cost management in your infrastructure.
- **Customer Impact Metrics:** Metrics like customer-reported incidents, customer satisfaction scores, or Net Promoter Score (NPS) can help in understanding the real-world impact of your DevOps practices on end-users.
- **Security Metrics:** Metrics such as number of vulnerabilities identified, time to patch vulnerabilities, and compliance with security standards (e.g., CIS benchmarks) help in assessing the security posture of your applications and infrastructure.
- **Feedback Loop Metrics**: Metrics related to feedback loops, such as time to resolve customer-reported issues or time to incorporate feedback from stakeholders, help in improving collaboration and responsiveness.

It's essential to select KPIs and metrics that align with your organization's goals and objectives, and regularly analyze and iterate on them to drive continuous improvement in your DevOps processes. Additionally, visualizing these metrics through dashboards or reports can help in monitoring progress and identifying areas for optimization.

Conclusion

In this document the architecture and tool selection are laid out having in mind quick prototyping ability for small teams with budget constraints and without dedicated DevOps engineers. Tool selection and solutions are profiled accordingly. The next step is to implement such a system and it can grow naturally, in coexisting or evolving phases, ranging from initial quick docker-compose-based setup for fast prototyping up to full-fledged k8s with more elaborate ArgoCD setup.

SCOPE

Milestone 1

- Current State Assessment:
    - Evaluate the existing development and operations processes to identify strengths, weaknesses, and areas for improvement. This assessment provides a baseline for measuring progress.
- Stakeholder Alignment:
    - Ensure alignment and understanding of DevOps goals among key stakeholders, including development, operations, and leadership teams. Obtain buy-in and support for the DevOps initiative.
- Toolchain Evaluation and Selection:
    - Assess current tools and identify gaps. Select and define a toolchain that aligns with DevOps principles, addressing areas such as version control, continuous integration, deployment automation, and monitoring.
- Infrastructure as Code (IaC)
    - Introduce IaC practices to automate infrastructure provisioning and management. This could involve adopting tools like Terraform or Ansible and establishing version-controlled infrastructure configurations.
    - Make a comparison and add a rationale as to why certain tools are the best choice, compared to the other option.
- Continuous Integration (CI)
    - Define CI pipelines to automate the building, testing, and integration of code changes. Ensure that developers receive rapid feedback on the quality of their code.
    - Create a simple how-to tutorial style for developers for single server deployment
- Containerization and Orchestration
    - Introduce containerization (e.g., Docker) and container orchestration (e.g., Kubernetes) to enhance scalability, portability, and manageability of applications.
- Continuous Deployment (CD)
    - Define automation to the deployment process. Define CD pipelines to enable automated and reliable software releases, ensuring consistency across environments.
- Security Integration
    - Define security practices in the DevOps pipeline. Define automated security scans, code analysis, and other measures to identify and address vulnerabilities early in the development process.
- Monitoring and Logging

    - Define robust monitoring and logging practices to gain insights into system performance and detect issues proactively. Define tools for real-time visibility into application and infrastructure health.
- Feedback Loop Optimization:
    - Define feedback loops across the development lifecycle. This includes feedback on code quality, user experience, and operational performance, fostering a culture of continuous improvement.
- Documentation and Knowledge
    - Develop comprehensive documentation and knowledge-sharing mechanisms to ensure that best practices, processes, and learning are well-documented and accessible to all team members.
- Performance Metrics and KPIs:
    - Define and track key performance indicators (KPIs) and metrics related to the DevOps processes.

Milestone 2

- Present and discuss the strategy document, allowing customers to leave reviews and comments.
- Incorporate feedback from customers and iterate on the strategy.
- After all feedback is received and incorporated the supplier should seek final sign-off from the client