Task

You are a DevOps expert tasked with setting up a platform. A platform consists of a set of projects grouped together within an infrastructure. Initially, you are provided with a project, including the code and a Docker file, for which you need to provision infrastructure and deploy the project.

Your responsibilities include:

1. Implementing the Continuous Integration/Continuous Deployment (CI/CD) pipeline: This must be set up as the first step.
2. Gathering logs from all existing services: This will be required in the subsequent stages of the project.
3. Conducting a performance test of the implemented platform: This includes making any necessary changes to the infrastructure and services to achieve high performance.

Important Features of the Project:

• The Docker file must be optimized for efficiency.
• The CI/CD pipeline should be highly optimized and fault-tolerant.
• Application logs must be collected using any suitable tool, with a detailed justification for the chosen tool included in the documentation.
• The infrastructure configuration must be designed to ensure optimal performance.

Additional Requirements:

• For each decision made during the implementation of the infrastructure, a detailed explanation must be provided in a PDF or Markdown file.
• The final project files must be submitted in a ZIP format. Do not upload them to a public repository.
• The use of tools, approaches, or solutions is optional, but the final evaluation of the project will be based on the decisions made and

Solutions and Tools

Tools

- Infrastructure:	Kubernetes
- Continuous Integration:	GitlabCI
- Continuous Deployment:	GitOps | FluxCD
- Log:	ELK
- Performance Test:	k6

Kubernetes as the Running Infrastructure

Kubernetes provides a scalable, resilient, and automated environment for running your FastAPI app. By abstracting infrastructure complexity, it ensures consistent deployments across environments and can handle traffic spikes seamlessly, making it well-suited for modern CI/CD workflows.

K8s delivers high availability, load balancing, self-healing, and automatic scaling, ensuring reliable performance. Its robust resource management, service discovery, and configuration management further enhance operational stability. Kubernetes also integrates naturally with CI/CD pipelines, including GitOps-based CD, enabling automated, version-controlled deployments from Git repositories.

GitLab CI for Automated Builds and Tests (CI)

GitLab CI offers a fully integrated solution for building, testing, and validating code before deployment. Its tight integration with Git allows pipelines to run automatically on commits or merge requests, ensuring early detection of issues and consistent code quality.

Pipelines are defined as code, supporting repeatable, version-controlled workflows, parallel execution, and containerized environments. GitLab CI pairs seamlessly with Kubernetes and GitOps tools like FluxCD, creating a reliable CI foundation ready to trigger automated, auditable CD.

Continuous Deployment with GitOps and FluxCD

GitOps defines deployments declaratively, storing the desired state of applications and infrastructure in Git. This approach enables full automation, traceability, and easy rollback, ensuring consistency and reducing human error across environments.

FluxCD implements GitOps on Kubernetes by continuously monitoring Git repositories and reconciling cluster state to match the desired configuration. Combined with CI, it provides a fully automated, auditable deployment workflow where CI validates changes and FluxCD enforces the correct production state.

ELK as Logging Tools

There are lots of log collection tools for Kubernetes environments and apps running on it.

We are going to use ELK Stack. The **ELK stack** (Elasticsearch, Logstash, Kibana) is a robust solution for log collection, storage, and visualization. Logstash gathers logs, Elasticsearch indexes them for fast search, and Kibana provides dashboards.

It offers **enterprise-grade logging and analysis**, ideal for monitoring all services and supporting future performance insights.

Performance Test

A complete performance-evaluation strategy can include load testing, stress testing, soak/endurance testing, and raw API benchmarking; however, given the scope and complexity of this project, **load testing and stress testing are sufficient to validate system behavior under expected and peak traffic conditions**. These two test categories provide the necessary insight into throughput, latency, scaling behavior, and failure thresholds without requiring the overhead of long-duration or micro-benchmark testing.

k6 is selected as the performance-testing tool because it is lightweight, modern, and fully cloud-native, making it ideal for Kubernetes environments. It supports expressive JavaScript-based test scenarios, integrates seamlessly into GitLab CI pipelines, and scales easily when executed inside Kubernetes. This makes k6 a highly efficient and maintainable choice for conducting both load and stress tests in an automated and production-like environment.

Continuous Integration (CI)

Optimizing dockerFile

The Docker file is optimized for higher efficiency.

Dockerfile

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 8000
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Changes and Optimizations

1- **Use a slimmer base image** – Image python:3.9-slim instead of python:3.9 is used. This reduces the image size significantly, decreasing build time, storage, and network transfer.

2- **Leverage layer caching** – Install dependencies before copying the entire source code.

3- **Reduce unnecessary cache** – Adding --no-cache-dir to pip install prevents pip from storing package caches in the image, keeping it leaner.

4- **Minimize layers** – Combining commands where possible reduces the number of intermediate layers, which further decreases the final image size.

Continuous Integration (CI)

This following pipeline is **fast, reliable, and production-ready**, with realistic testing using Redis, efficient parallel execution, and fault-tolerant CI steps.

.gitlab-ci.yml

```
variables:
  IMAGE_NAME: "$CI_REGISTRY_IMAGE/fastapi-app"
  PIP_CACHE_DIR: "$CI_PROJECT_DIR/.cache/pip"

image: docker:24.0.2

services:
  - docker:24.0.2-dind
  - name: redis:7-alpine
    alias: redis

stages:
  - lint
  - test
  - build

# -----------------------------------
# Lint Stage (Parallel for multiple linters)
# -----------------------------------
lint:
  stage: lint
  image: python:3.9-slim
  parallel: 2  # could run multiple lint configs in parallel
  script:
    - python -m pip install --upgrade pip
    - pip install flake8 black
    - flake8 . --max-line-length=120
    - black --check .
  rules:
    - if: '$CI_COMMIT_BRANCH'
  allow_failure: false
```

```
# ----------------------------------
# Test Stage (Parallel for fast execution)
# ----------------------------------
test:
  stage: test
  image: python:3.9-slim
  services:
    - name: redis:7-alpine
      alias: redis
  cache:
    key: pip-cache
    paths:
      - .cache/pip
  parallel: 2  # run tests in two shards if needed
  script:
    - python -m pip install --upgrade pip
    - pip install -r requirements.txt pytest
    - pytest --maxfail=1 --disable-warnings -v
  rules:
    - if: '$CI_COMMIT_BRANCH'
  allow_failure: false

# ----------------------------------
# Docker Build Stage
# ----------------------------------
build:
  stage: build
  image: docker:24.0.2
  services:
    - docker:24.0.2-dind
  before_script:
    - echo "$CI_REGISTRY_PASSWORD" | docker login -u "$CI_REGISTRY_USER" --password-stdin $CI_REGISTRY
  script:
    - docker build -t $IMAGE_NAME:$CI_COMMIT_SHA .
    - docker push $IMAGE_NAME:$CI_COMMIT_SHA
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
  allow_failure: false
```

Key Enhancements & Optimizations

1. **Parallel Jobs**
    - Lint and test stages run in parallel shards, reducing pipeline runtime.
2. **Redis Service for Tests**
    - Redis container included in test stage so integration tests can run realistically.
    - Ensures CI reflects actual runtime environment, improving reliability.
3. **Caching**
    - Pip cache for dependencies, avoiding repeated downloads and speeding up rebuilds.
4. **Fail-Fast & Fault-Tolerant**
    - Tests stop immediately on failure.
    - Parallel execution isolates failures to specific shards without breaking unrelated jobs.
5. **Dynamic Docker Builds**
    - Builds Docker images only on main branch.
    - Tagged with commit SHA for traceability and easy rollback.

Continuous Deployment (CD)

The Continuous Deployment of the **FastAPI** app is handled by **FluxCD** using the **Kustomization** methodology, where all Kubernetes manifests are stored in a Git repository and applied automatically to the cluster. This approach ensures that the cluster state always mirrors the Git repository, enabling fully automated, auditable, and easily rollbackable deployments. **Kustomization** is chosen over Helm for simplicity and transparency.

GitOps Repository Structure

```
gitops-repo/
├── apps/
│   └── fastapi-app/
│       ├── Dockerfile
│       ├── .gitlab-ci.yml
│       ├── main.py
│       ├── requirements.txt
│       ├── deployment.yaml
│       ├── service.yaml
│       ├── kustomization.yaml
├── infrastructure/
│   └── redis/
│       ├── deployment.yaml
│       ├── service.yaml
│       ├── kustomization.yaml
│   └── elk/
│       ├── elasticsearch-deployment.yaml
│       ├── elasticsearch-service.yaml
│       ├── logstash-deployment.yaml
│       ├── logstash-configmap.yaml
│       ├── logstash-service.yaml
│       ├── kibana-deployment.yaml
│       ├── kibana-service.yaml
│       └── kustomization.yaml
│   └── k6/
│       ├── load-test.js
│       ├── stress-test.js
│       ├── load-test-job.yaml
│       ├── stress-test-job.yaml
│       ├── k6-scripts-configmap.yaml
│       └── kustomization.yaml
├── flux/
│   ├── gotk-sync.yaml        # bootstrap FluxCD sync
│   └── kustomization.yaml    # optional cluster-level resources
└── README.md
```

NOTE:

apps/fastapi-app/	All manifests for your FastAPI application.
infrastructure/redis/	Redis deployment & service manifests (so CD can manage dependencies)
infrastructure/elk/	Implement ELK as log aggregation tools (explained in log section)
infrastructure/k6/	Implement k6 as performance test tools (explained in perf test section)
flux/	FluxCD bootstrap and cluster-level configuration.

Files:

--- FastAPI App---

Requirements.txt

```
fastapi==0.101.0
uvicorn==0.23.0
redis==5.3.0
```

main.py

```
from fastapi import FastAPI, HTTPException
import redis
app = FastAPI()
redis_client = redis.Redis(host='redis', port=6379, db=0)
@app.get("/")
def read_root():
value = redis_client.get("example_key")
if value is None:
raise HTTPException(status_code=404, detail="Key not found")
return {"message": value.decode()}
@app.post("/write/{key}")
def write_to_redis(key: str, value: str):
redis_client.set(key, value)
return {"message": f"Key '{key}' set to '{value}'"}
```

deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fastapi-app
  labels:
    app: fastapi-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: fastapi-app
  template:
    metadata:
      labels:
        app: fastapi-app
    spec:
      containers:
        - name: fastapi-app
          image: <CI_REGISTRY_IMAGE>/fastapi-app:latest  # Image pushed by CI
          ports:
            - containerPort: 8000
          env:
            - name: REDIS_HOST
              value: "redis"
            - name: REDIS_PORT
              value: "6379"
```

service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: fastapi-app
spec:
  type: ClusterIP
  selector:
    app: fastapi-app
  ports:
    - port: 80
      targetPort: 8000
```

kustomization.yaml

```
resources:
  - deployment.yaml
  - service.yaml
```

--- Redis Manifests ---

deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
        - name: redis
          image: redis:7-alpine
          ports:
            - containerPort: 6379
```

service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: redis
spec:
  type: ClusterIP
  selector:
    app: redis
  ports:
    - port: 6379
      targetPort: 6379
```

kustomization.yaml

```
resources:
  - deployment.yaml
  - service.yaml
```

--- FluxCD Configuration ---

gotk-sync.yaml

```
apiVersion: source.toolkit.fluxcd.io/v1beta2
kind: GitRepository
metadata:
  name: gitops-repo
  namespace: flux-system
spec:
  interval: 1m
  url: https://gitlab.com/your-org/gitops-repo.git
  branch: main
  secretRef:
    name: flux-git-auth
```

kustomization.yaml

```
apiVersion: kustomize.toolkit.fluxcd.io/v1beta2
kind: Kustomization
metadata:
  name: fastapi-app
  namespace: flux-system
spec:
  interval: 1m
  path: "./apps/fastapi-app"
  prune: true
  sourceRef:
    kind: GitRepository
    name: gitops-repo
```

```
---
apiVersion: kustomize.toolkit.fluxcd.io/v1beta2
kind: Kustomization
metadata:
  name: elk-stack
  namespace: flux-system
spec:
  interval: 1m
  path: "./infrastructure/elk"
  prune: true
  sourceRef:
    kind: GitRepository
    name: gitops-repo
---
apiVersion: kustomize.toolkit.fluxcd.io/v1beta2
kind: Kustomization
metadata:
  name: k6-tests
  namespace: flux-system
spec:
  interval: 1m
  path: "./infrastructure/k6"
  prune: true
  sourceRef:
    kind: GitRepository
    name: gitops-repo
```

Key Features

1. **Fully GitOps-Driven CD**
   - Everything declarative; cluster state mirrors Git repo.
2. **Custom Manifests**
   - No Helm; everything is fully transparent and configurable.
3. **Fault-Tolerant**
   - Flux automatically retries failed reconciliations.
   - Rollbacks are trivial by reverting Git commits.
4. **Scalable**
   - App replicas and services defined per environment; Flux can manage multiple clusters via multi-tenancy.
5. **CI/CD Integration**
   - Docker image built & pushed by GitLab CI.
   - Updating image: in deployment.yaml triggers Flux to deploy new version.

Log Aggregation

ELK Architecture for Kubernetes

1- Log Collection:
   - Filebeat or Logstash agents run as DaemonSets on each node, collecting logs from applications (FastAPI + Redis) and Kubernetes components (API server, kubelet, etc.).

2- Log Processing:
   - Logstash parses, filters, and enriches logs (e.g., adds Kubernetes metadata like namespace, pod, labels).

3- Log Storage & Indexing:

   - Elasticsearch cluster stores and indexes logs for fast search and analytics.

4- Visualization & Monitoring:
   - Kibana provides dashboards, search, and alerts for both app-level and infrastructure-level logs.

Flow: Logs Filebeat/Logstash → Elasticsearch → Kibana

This setup ensures centralized, searchable, and visualized logs for both infrastructure and application components in Kubernetes.

The ELK stack also will be run by FLUX CD, so (as mentioned previously) it is a part of project repo

```
gitops-repo/
├── apps/
│   └── ...
├── infrastructure/
│   └...
│   └── elk/
│       ├── elasticsearch-deployment.yaml
│       ├── elasticsearch-service.yaml
│       ├── logstash-deployment.yaml
│       ├── logstash-configmap.yaml
│       ├── logstash-service.yaml
│       ├── kibana-deployment.yaml
│       ├── kibana-service.yaml
│       └── kustomization.yaml
├── flux/
│   ├── ...
└── README.md
```

Files

elasticsearch-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: elasticsearch
  labels:
    app: elasticsearch
spec:
  replicas: 1
  selector:
    matchLabels:
      app: elasticsearch
  template:
    metadata:
      labels:
        app: elasticsearch
    spec:
      containers:
        - name: elasticsearch
          image: docker.elastic.co/elasticsearch/elasticsearch:8.10.0
          ports:
            - containerPort: 9200
          env:
            - name: discovery.type
              value: single-node
```

elasticsearch-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: elasticsearch
spec:
  type: ClusterIP
  selector:
    app: elasticsearch
  ports:
    - port: 9200
      targetPort: 9200
```

logstash-configmap.yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: logstash-config
data:
  logstash.conf: |
    input {
      beats {
        port => 5044
      }
    }
    filter {
      if [kubernetes] {
        mutate { add_field => { "namespace" => "%{[kubernetes][namespace]}" } }
      }
    }
    output {
      elasticsearch {
        hosts => ["http://elasticsearch:9200"]
        index => "k8s-logs-%{+YYYY.MM.dd}"
      }
    }
```

logstash-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: logstash
  labels:
    app: logstash
spec:
  replicas: 1
  selector:
    matchLabels:
      app: logstash
  template:
    metadata:
      labels:
        app: logstash
    spec:
      containers:
        - name: logstash
          image: docker.elastic.co/logstash/logstash:8.10.0
          ports:
            - containerPort: 5044
          volumeMounts:
            - name: config
              mountPath: /usr/share/logstash/pipeline/logstash.conf
              subPath: logstash.conf
      volumes:
        - name: config
          configMap:
            name: logstash-config
```

logstash-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: logstash
spec:
  type: ClusterIP
  selector:
    app: logstash
  ports:
    - port: 5044
      targetPort: 5044
```

kibana-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kibana
  labels:
    app: kibana
spec:
  replicas: 1
  selector:
    matchLabels:
      app: kibana
  template:
    metadata:
      labels:
        app: kibana
    spec:
      containers:
        - name: kibana
          image: docker.elastic.co/kibana/kibana:8.10.0
          ports:
            - containerPort: 5601
          env:
            - name: ELASTICSEARCH_HOSTS
              value: "http://elasticsearch:9200"
```

kibana-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: kibana
spec:
  type: ClusterIP
  selector:
    app: kibana
  ports:
    - port: 5601
      targetPort: 5601
```

kustomization.yaml (in infrastructure/elk/)

```
resources:
  - elasticsearch-deployment.yaml
  - elasticsearch-service.yaml
  - logstash-deployment.yaml
  - logstash-configmap.yaml
  - logstash-service.yaml
  - kibana-deployment.yaml
  - kibana-service.yaml
```

FluxCD kustomization.yaml

kustomization.yaml

```
apiVersion: kustomize.toolkit.fluxcd.io/v1beta2
kind: Kustomization
metadata:
  name: fastapi-app
```

```
...
---
apiVersion: kustomize.toolkit.fluxcd.io/v1beta2
kind: Kustomization
metadata:
  name: elk-stack
  namespace: flux-system
spec:
  interval: 1m
  path: "./infrastructure/elk"
  prune: true
  sourceRef:
    kind: GitRepository
    name: gitops-repo
```

NOTE:
Both **fastapi-app** and **elk-stack** are treated as **separate FluxCD Kustomization units**.
FluxCD watches the *same Git repository* but applies resources from *different directories*.
This separation improves:

- **Fault isolation** (ELK issues do not affect the app)
- **Granular reconciliation**
- **Independent rollouts**
- **Cleaner GitOps structure (apps vs infrastructure)**

Performance Test

As mentioned, given the scope and complexity of this project, **load testing** and **stress** testing are used to validate system behavior under expected and peak traffic conditions. And Also **k6** is selected as the performance-testing tool because it is lightweight, modern and fully cloud-native.

k6 Test Design Overview

- **Test Types**
  - **Load Test:** Measure normal performance (throughput, latency, error rate) under expected traffic.
  - **Stress Test:** Determine system limits and behavior under peak/excessive traffic.
- **Target**
  - FastAPI endpoints: / (GET) and /write/{key} (POST)
  - Validate response times, requests per second, and error rates.
- **Implementation**
  - Scripts written in **JavaScript**, stored in tests/k6/ in Git repo.
  - Runs inside **Kubernetes** via Jobs or k6-operator for realistic network and cluster conditions.
  - Logs output to **stdout** and optionally to **ELK** for metrics correlation.
- **Execution & Automation**
  - k6 tests run as Kubernetes Jobs after FluxCD deploys the app to staging.
  - Ensures tests reflect real cluster behavior, including replicas, Redis, and networking.
  - Results are logged and can be collected via ELK for analysis.

- **NOTE:** The **load** and **stress tests** are fully automated as part of the CD workflow, executed in Kubernetes after FluxCD deploys the app, and are **not integrated** into GitLab CI.

Test Implementation

Implement of **k6 load** and **stress tests** fully in **Kubernetes**, managed by **FluxCD**, and integrated into your existing **GitOps** repo.

GitOps Repo Structure

```
gitops-repo/
├── apps/
│   └── fastapi-app/
│       ...
├── infrastructure/
│   └── redis/
│       ...
│   └── elk/
│       ...
│   └── k6/
│       ├── load-test.js
│       ├── stress-test.js
│       ├── load-test-job.yaml
│       ├── stress-test-job.yaml
│       ├── k6-scripts-configmap.yaml
│       └── kustomization.yaml
├── flux/
│   ...
└── README.md
```

k6 Test Scripts

load-test.js

```
import http from 'k6/http';
import { check, sleep } from 'k6';

export let options = {
  stages: [
    { duration: '2m', target: 20 }, // ramp-up to 20 users
    { duration: '5m', target: 20 }, // sustain load
    { duration: '2m', target: 0 },  // ramp-down
  ],
};

export default function () {
  let res = http.get('http://fastapi-app:80/');
  check(res, { 'status is 200': (r) => r.status === 200 });
  http.post(`http://fastapi-app:80/write/key1`, JSON.stringify({ value: 'test' }), {
    headers: { 'Content-Type': 'application/json' },
  });
  sleep(1);
}
```

stress-test.js

```
import http from 'k6/http';
import { check, sleep } from 'k6';

export let options = {
  stages: [
    { duration: '1m', target: 50 }, // ramp-up quickly to 50 users
    { duration: '3m', target: 100 }, // push beyond normal load
    { duration: '2m', target: 0 },   // ramp-down
  ],
};
export default function () {
  let res = http.get('http://fastapi-app:80/');
  check(res, { 'status is 200': (r) => r.status === 200 });
  http.post(`http://fastapi-app:80/write/key2`, JSON.stringify({ value: 'stress' }), {
    headers: { 'Content-Type': 'application/json' },
  });
  sleep(0.5);
}
```

Kubernetes Job Manifests

load-test-job.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: k6-load-test
spec:
  template:
    spec:
      containers:
        - name: k6
          image: loadimpact/k6:0.45.0
          command: ["k6", "run", "/scripts/load-test.js"]
          volumeMounts:
            - name: scripts
              mountPath: /scripts
      restartPolicy: Never
      volumes:
        - name: scripts
          configMap:
            name: k6-scripts
```

stress-test-job.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: k6-stress-test
spec:
  template:
    spec:
      containers:
        - name: k6
          image: loadimpact/k6:0.45.0
          command: ["k6", "run", "/scripts/stress-test.js"]
          volumeMounts:
            - name: scripts
              mountPath: /scripts
      restartPolicy: Never
      volumes:
        - name: scripts
          configMap:
            name: k6-scripts
```

k6-scripts-configmap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: k6-scripts
data:
  load-test.js: |
    import http from 'k6/http';
    import { check, sleep } from 'k6';

    export let options = {
      stages: [
        { duration: '2m', target: 20 },
        { duration: '5m', target: 20 },
        { duration: '2m', target: 0 },
      ],
    };

    export default function () {
      let res = http.get('http://fastapi-app:80/');
      check(res, { 'status is 200': (r) => r.status === 200 });
      http.post(`http://fastapi-app:80/write/key1`, JSON.stringify({ value: 'test' }), {
        headers: { 'Content-Type': 'application/json' },
      });
      sleep(1);
    }

  stress-test.js: |
    import http from 'k6/http';
    import { check, sleep } from 'k6';

    export let options = {
      stages: [
        { duration: '1m', target: 50 },
        { duration: '3m', target: 100 },
        { duration: '2m', target: 0 },
      ],
    };

    export default function () {
      let res = http.get('http://fastapi-app:80/');
      check(res, { 'status is 200': (r) => r.status === 200 });
      http.post(`http://fastapi-app:80/write/key2`, JSON.stringify({ value: 'stress' }), {
        headers: { 'Content-Type': 'application/json' },
      });
      sleep(0.5);
    }
```

infrastructure/k6/kustomization.yaml

```
resources:
  - k6-scripts-configmap.yaml
  - load-test-job.yaml
  - stress-test-job.yaml
```

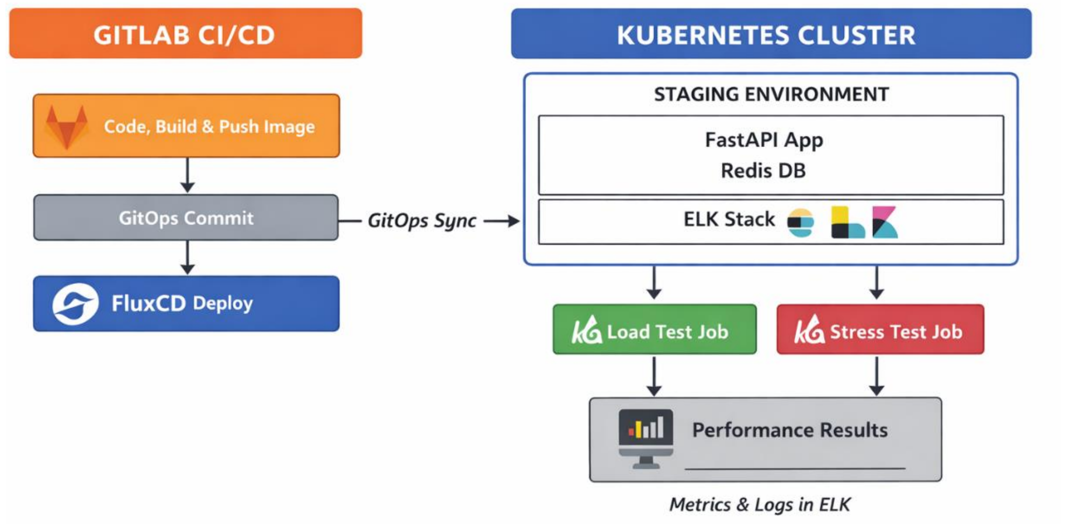
flux/kustomization.yaml

```
apiVersion: kustomize.toolkit.fluxcd.io/v1beta2
kind: Kustomization
metadata:
  name: fastapi-app
  ...
---
apiVersion: kustomize.toolkit.fluxcd.io/v1beta2
kind: Kustomization
metadata:
  name: elk-stack
  ...
---
apiVersion: kustomize.toolkit.fluxcd.io/v1beta2
kind: Kustomization
metadata:
  name: k6-tests
  namespace: flux-system
spec:
  interval: 1m
  path: "./infrastructure/k6"
  prune: true
  sourceRef:
    kind: GitRepository
    name: gitops-repo
```

How the Test Works
After deployment of **FastAP** and **ELK**, FluxCD automatically deploys the **k6 Jobs** for load and stress tests. Jobs run inside the cluster, sending traffic to the **internal FastAPI service**. Results are output to logs and can be collected in **ELK dashboards**.

Final Note

Project Work Flow
1. Continuous Integration (CI) – GitLab CI
   - Code is committed and version-controlled in GitLab.
   - CI pipeline runs unit tests and code validation.
   - Application is built, dockerized, and pushed to the container registry.
2. Continuous Deployment (CD) – FluxCD / GitOps
   - FluxCD monitors the Git repository for changes.
   - Automatically deploys FastAPI app, Redis, and ELK stack to the Kubernetes cluster.
   - Ensures deployments are declarative, consistent, and fully automated.
3. Observability – ELK Stack
   - Logs from FastAPI, Redis, and Kubernetes infrastructure are collected.
   - Elasticsearch stores and indexes logs.
   - Kibana visualizes metrics and performance insights.
4. Performance Testing – k6
   - Load testing: measures throughput and latency under normal traffic.
   - Stress testing: evaluates system behavior under peak/excessive traffic.
   - Tests run as Kubernetes Jobs in the staging cluster after FluxCD deploys the app.
   - Results are captured in ELK dashboards for monitoring and analysis.
5. Automation & GitOps Integration
   - All steps from deployment to testing are fully automated in the CD workflow.
   - GitLab CI handles only code testing, build, and image push, not cluster performance tests.



Optimal Performance

This solution ensures **optimal performance** by combining a **Kubernetes-based scalable infrastructure**, **automated CI/CD pipelines** with GitLab CI and FluxCD, **centralized observability via ELK**, and **targeted load and stress testing with k6**.

Each component is configured for efficiency:

- Kubernetes provides high availability, autoscaling, and resource management
- GitLab CI ensures fast, reliable builds
- FluxCD guarantees consistent, declarative deployments
- ELK enables real-time log analysis
- k6 validates system behavior under peak loads. Together, they create a **robust, efficient, and performance-optimized platform**.

--- Arian Ramezani ---

--- Feb 2026 ---

Arian Ramezani
+98-9355131142
linkedin.com/in/arian-rmz
arian-rmz.ir
arian.ramezani@gmail.com