MiniLink - DevOps, Testing & Deployment Report

1. Introduction and Scope

This report intends to document the second phase of the MiniLink software, the minimal URL shortener built using FastAPI, SQLModel, and SQLite. The first assignment focused on designing and building the functional application: so users can sign up, create links, track their click counts, and view their private analytics through a modern interface.

For this assignment 2, we build on that foundation and shift towards the software quality and DevOps practices. Rather than adding new features for the system, the goal is to make the existing MiniLink more robust, testable, automatable and observable. In other words, this phase aims to introduce:

- Code refactoring → improved structure, readability, separation of concerns
- Automated tests → 81% line coverage and a coverage gate enforced by CI
- GitHub actions pipeline → runs tests, checks coverage, build Docker image, and publishes it to GHCR (GitHub Container Registry)
- Production-ready Dockerfile → which leads to deployment on Render using image
- Health and monitoring endpoints → /health, /metrics with Prometheus compatible metrics for requests, latency, and errors

This report will explain these improvements, show how everything is configured (pipeline and deployment) and reflect the evolution of MiniLink to a small but realistic, DevOps-ready service.

2. Code Quality and Refactoring

2.1 Refactoring objectives

The improvements to transform Minilink into a cleaner, more maintainable application:

- Reducing duplicated code across system's logic
- Separating responsibility so each module just handles a single concern
- Preparing codebase for CI/CD, testing, deployment
- Better readability for long-term maintainability
- Applying SOLID principles to keep components independent and reusable

So, make MiniLink less a monolithic script and more like a small production service

2.2 Main Refactor Changes

The application was reorganized into dedicated modules:

- **app/main.py** — routing, FastAPI configuration, middleware, metrics, dependency
- **app/models.py** — SQLModel definitions for User and Link
- **app/schemas.py** — request/response validation models
- **app/auth.py** — password hashing and verification logic
- **app/services.py** — URL validation, short-code generation, and utility functions
- **app/db.py** — database initialization and session management

Key changes for refactoring:

1. Authentication logic was isolated in auth.py
2. Short code generation and URL validation moved to services.py
3. All DB session management moved to db.py
4. Routes simplified and focused

2.3 Code Smells Removed

Hardcoded logic across route handlers → moved into reusable helper functions and service modules.

Mixed authentication and routing → split into get_current_user() helper + isolated auth.py module.

Duplicated code for link handling → link validation, code generation, and checks now live in shared functions.

Database logic embedded directly in routes → centralized in one place (db.py), making the app more structured and test-friendly.

Inconsistent error handling → unified FastAPI exceptions and HTTP statuses across the app.

3. Testing and Coverage

3.1 Testing Strategy

MiniLink uses pytests jointly with FastAPI's TestClient to validate functionality and stability. The goal of this phase is to ensure all critical features were covered, such as success and failure paths.

- **Health Check Tests**
  Verifies that /health returns a 200 status and correct JSON structure.
- **CRUD API Tests**
  Tests for creating links, retrieving them, updating fields, and deleting them.
  Ensures proper user scoping so one user can't access another's links.

- **Redirect & Analytics Tests**
  Tests that /r/{code} correctly redirects and increments:
    - click_count
    - last_accessed
- **Expiry Tests**
  Requests to expired links return **410 Gone** and do not redirect.
- **Error Case Tests**
    - Non-existent links → **404 Not Found**
    - Duplicate custom codes → **409 Conflict**
    - Invalid URLs (non-HTTP/HTTPS) → **422 Unprocessable Entity**

These test suite ensures both correctness and robustness

3.2 Coverage and Threshold

MiniLink achieves 81% line coverage, measured using:
python -m pytest --cov=app --cov-report=xml --cov-report=html

This generates:
- coverage.xml (used by CI)
- HTML coverage report inside htmlcov/

In the CI pipeline, a Python script parses coverage.xml and fails the pipeline automatically if coverage is below 70%. This guarantees that code with insufficient coverage cannot merge into main, the deployment only occurs when tests and coverage pass successfully, and that coverage stays constant over time to prevent regression.

3.3 Example: Important Test Cases

test_redirect_and_analytics
Creates a link, performs a redirect, and checks that:

a. click_count increases
b. last_accessed is updated
c. redirect uses HTTP 307

test_redirect_expired_returns_410
Creates a link with an expiration in the past and ensures accessing it returns **410** instead of redirecting.

test_delete_link
Confirms that deleting a link permanently removes it and subsequent reads return **404**.

These tests help verify end-to-end functionality of the system together with test_smoke.py

4. CI/CD Pipeline (GitHub Actions + Docker + GHCR + Render)

This section explains how MiniLink implements a complete CI/CD workflow, covering automated testing, containerization, and cloud deployment.

4.1 CI Workflow Overview

The full CI pipeline is defined at:
.github/workflows/ci.yml

The workflow triggers automatically **on every push or pull request to the main branch**.

The main CI steps are:

1. **Checkout repository**
   Uses actions/checkout@v4.
2. **Set up Python 3.11**
   Ensures a consistent environment across all runs.
3. **Install dependencies**
   Installs project requirements and test dependencies (pytest, pytest-cov).
4. **Run tests + coverage**
   Executes: pytest --cov=app --cov-report=xml --cov-report=term-missing
5. **Coverage enforcement (≥70%)**
   A Python script reads coverage.xml.
   If coverage is *below 70%*, the pipeline **fails automatically**, preventing deployment.
6. **Build Docker image**
   The CI builds the application container: docker build -t minilink:${{github.sha }}
7. **Docker smoke test**
   Before deployment, a lightweight integration test ensures the container works:
   a. Run container in background
      docker run -d --name minilink_test -p 8000:8000 minilink:${{ github.sha }}
   b. Wait for startup
      sleep 5
   c. Hit the health check
      curl -f http://127.0.0.1:8000/health
   d. Output logs for debugging
   e. Stop + remove the container

This guarantees the Dockerized app works before deployment

4.2 Docker Image and GHCR Publishing

MiniLink includes a production DockerFile used in CI and deployment. It uses the GitHub's official docker/build-push-action to publish the container image

Key points:

- Image published to GHCR as: ghcr.io/javronich1/minilink:latest
- Authentication uses the built-in GitHub Actions GITHUB_TOKEN.
- The Docker build always happens *after*:
    - tests pass
    - coverage threshold is met
    - smoke test succeeds

This ensures that only stable images are published.

4.3 Deployment to Render (CD)

The Continuous Development (CD) is managed by Render, which pulls directly the Docker image from GitHub

Render is configured with:

- **Service type**: Docker-based web service
- **Source**: GitHub → repository javronich1/minilink
- **Branch**: main
- **Auto-deploy**: *Enabled only for the main branch*
- **Runtime**: Build and run using your uploaded Dockerfile

This means that only commits that successfully pass CI are deployed.

Environment Variables:
Render uses production runtime variables, including:

- COOKIE_SECRET
- ENV=production

These ensure secure authentication and stable environment behavior in production. It hence satisfies the configure secrets and triggers so only the main branch deploys automatically.

4.4 Full CI/CD Summary

1. **Developer pushes to main**
2. **GitHub Actions CI** runs:
    - installs dependencies
    - runs tests
    - enforces ≥70% coverage
    - builds Docker image
    - smoke-tests /health
    - pushes image to GHCR
3. **Render Deployment**

    - Detects new commit in main
    - Pulls repo
    - Builds using Dockerfile
    - Deploys final container
4. **Application goes live**
    - at https://minilink-9gdf.onrender.com/

5. Monitoring and Health Checks

This section covers the observability components added to MiniLink, satisfying the monitoring portion of the assignment requirements

5.1 Health Endpoint

Dedicated health check route at: GET /health
Returns simple JSON payload: {"status": "ok"}

This endpoint is used in several contexts:

- **Local development** to confirm the server is running
- **CI Docker smoke test**, where the container must return a successful /health response before the pipeline continues
- **Production readiness checks**, since platforms like Render or third-party uptime monitors can use this URL to verify availability

5.2 Prometheus Metrics

MiniLink implements prometheus_client library and custom FastAPI middleware to get the request-level analytics.

Metrics are exposed in text format by: GET /metrics

- **minilink_requests_total{method, path, status}**
  Counts every HTTP request, grouped by method, route, and status code.
- **minilink_request_latency_seconds (Histogram)**
  Measures response times, enabling latency distribution analysis.
- **minilink_request_errors_total**
  Counts all server-side (5xx) errors per route.

Also, Prometheus automatically includes standard runtime metrics such as: Python garbage collection statistics, process level CPU usage and the memory usage. Together, these provide an observability layer enabling the tracking of traffic, performance and unexpected errors.

5.3 Optional Local Prometheus Setup

A sample Prometheus config file is under: monitoring/prometheus.yml
Defining:

```
scrape_configs:
  - job_name: "minilink"
    static_configs:
      - targets: ["host.docker.internal:8000"]
```

6. Reflection and Lessons Learned

6.1 Improvements from Assignment 1

- **From local-only to fully deployed:** The project is now containerized with Docker, published to GHCR, and deployed to the cloud using Render's Docker runtime.
- **From manual testing to automation:** All tests now run automatically on every push through GitHub Actions, including a coverage gate and a Docker-based smoke test.
- **From zero observability to structured monitoring:** Health checks and Prometheus metrics provide visibility into traffic, latency, and failures.
- **From scattered logic to clean architecture:** Authentication, services, database logic, and routing are separated, improving maintainability and testability.

6.2 Future Work

- **Introduce a staging environment** to test deployments before releasing to production.
- **Add monitoring alerts**, such as error-rate thresholds or latency spikes using Prometheus Alertmanager.
- **Adopt Infrastructure as Code (Terraform)** to provision cloud infrastructure reproducibly instead of relying on UI-based setup.
- **Migrate to PostgreSQL** for stronger data integrity, concurrency, and scalability.
- **Enhance test coverage** with deeper unit tests, especially around authentication and expiration logic.

7. Appendices

Appendix A - AI Usage

AI tools (ChatGPT by OpenAI) were used selectively throughout this assignment to support development—not to replace it.

They assisted mainly with:

- Structuring the **CI/CD pipeline** and understanding best practices for coverage gates and job dependencies.
- Debugging **YAML syntax errors** in GitHub Actions.
- Suggesting patterns for **Docker smoke tests** and **Prometheus metrics** instrumentation.
- Providing guidance for organizing the report structure and ensuring no grading criteria were overlooked.

All final configurations, code decisions, refactoring steps, and deployment logic were fully reviewed, understood, and implemented by me.

Any AI-assisted suggestions were critically evaluated and adapted to fit the project's requirements.

Appendix B - Snippet of ci.yml

```
on:
  push:
    branches: ["main"]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: "3.11"
      - run: python -m pytest --cov=app --cov-report=xml

      # Coverage enforcement
      - run: |
          python - << 'EOF'
          import xml.etree.ElementTree as ET
          rate=float(ET.parse("coverage.xml").getroot().get("line-rate"))
          if rate < 0.70: raise SystemExit("Coverage < 70%")
          EOF
```

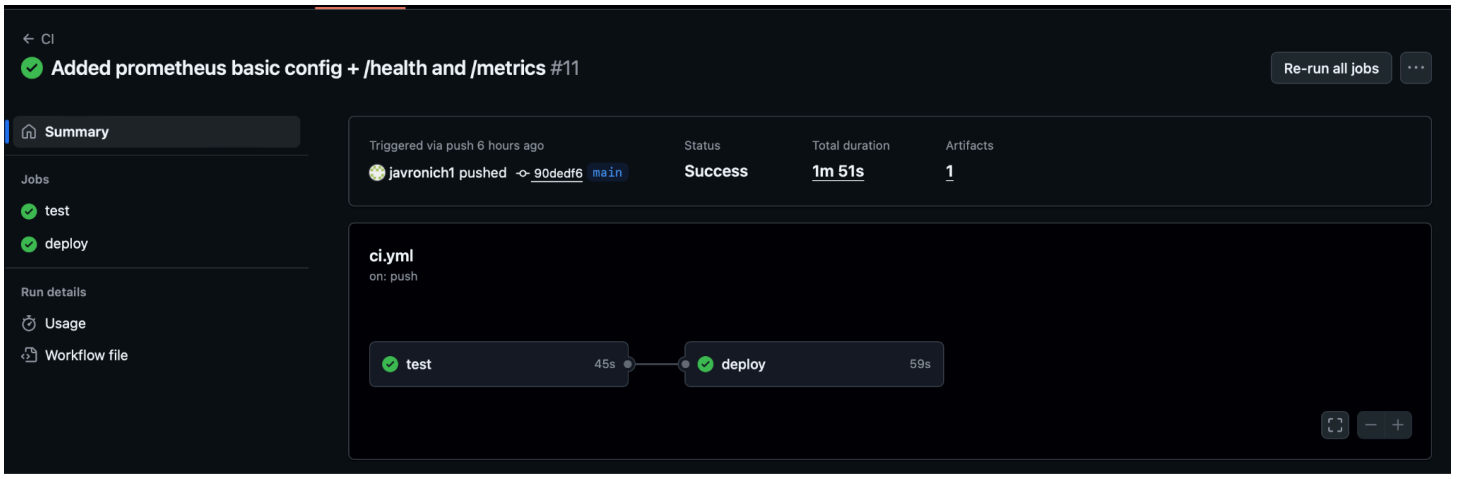
```
  deploy:
    needs: test
    if: github.ref == 'refs/heads/main'
```

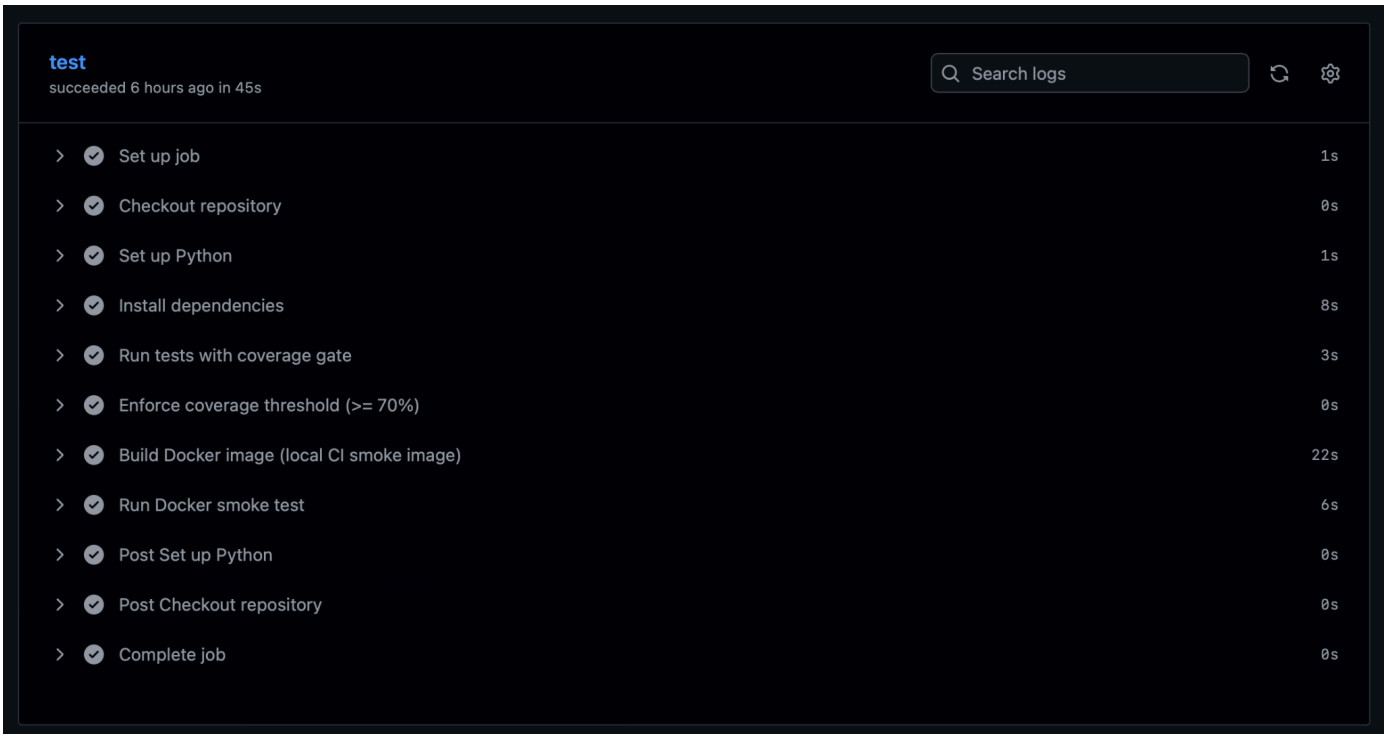
Appendix C - Snippet of DockerFile

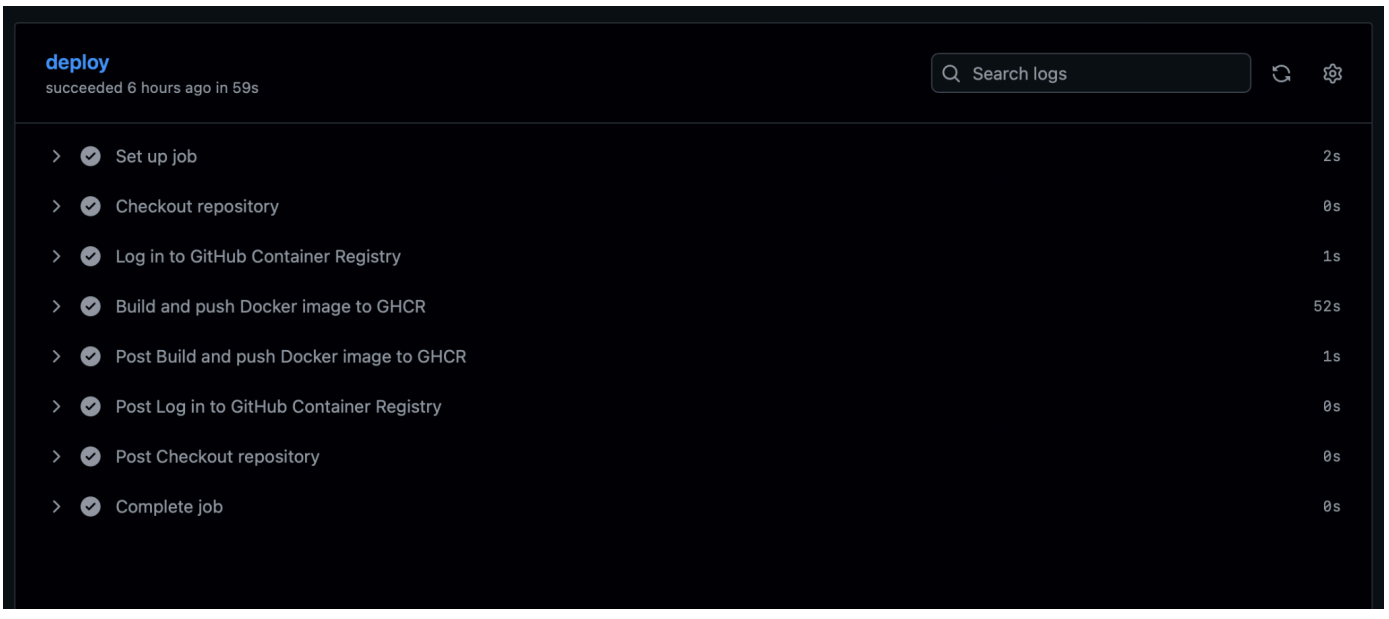
```
# Use a small Python base image
FROM python:3.11-slim

# Do not write .pyc files and flush stdout/stderr immediately
ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1

# Set working directory inside the container
WORKDIR /app

# Install system dependencies (for SQLite etc.)
RUN apt-get update && apt-get install -y --no-install-recommends \
    build-essential \
    && rm -rf /var/lib/apt/lists/*

# Install Python dependencies
# 1) Copy only requirements first (better for Docker layer caching)
COPY requirements.txt .

RUN pip install --upgrade pip && \
    pip install --no-cache-dir -r requirements.txt

# 2) Now copy the actual application code
COPY app ./app
COPY tests ./tests

# Expose the application port
EXPOSE 8000

# Default command: run uvicorn
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Appendix D - Screenshots

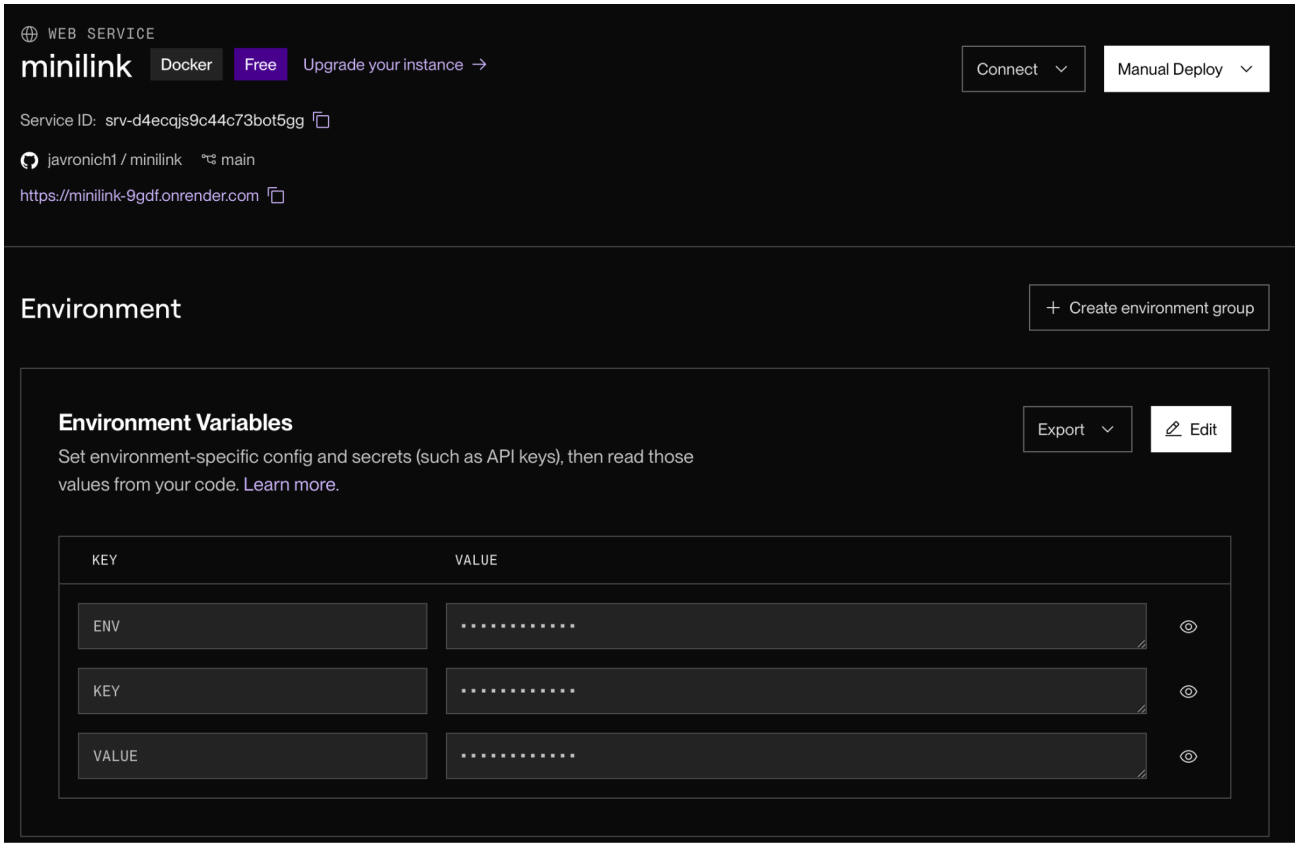
C.1 GitHub Actions (Successful run)


CI
Added prometheus basic config + /health and /metrics #11
Re-run all jobs
Summary
Jobs
test
deploy
Run details
Usage
Workflow file
Triggered via push 6 hours ago
javronich1 pushed 90dedf6 main
Status
Success
Total duration
1m 51s
Artifacts
1
ci.yml
on: push
test 45s
deploy 59s


test
succeeded 6 hours ago in 45s
Search logs
Set up job 1s
Checkout repository 0s
Set up Python 1s
Install dependencies 8s
Run tests with coverage gate 3s
Enforce coverage threshold (>= 70%) 0s
Build Docker image (local CI smoke image) 22s
Run Docker smoke test 6s
Post Set up Python 0s
Post Checkout repository 0s
Complete job 0s

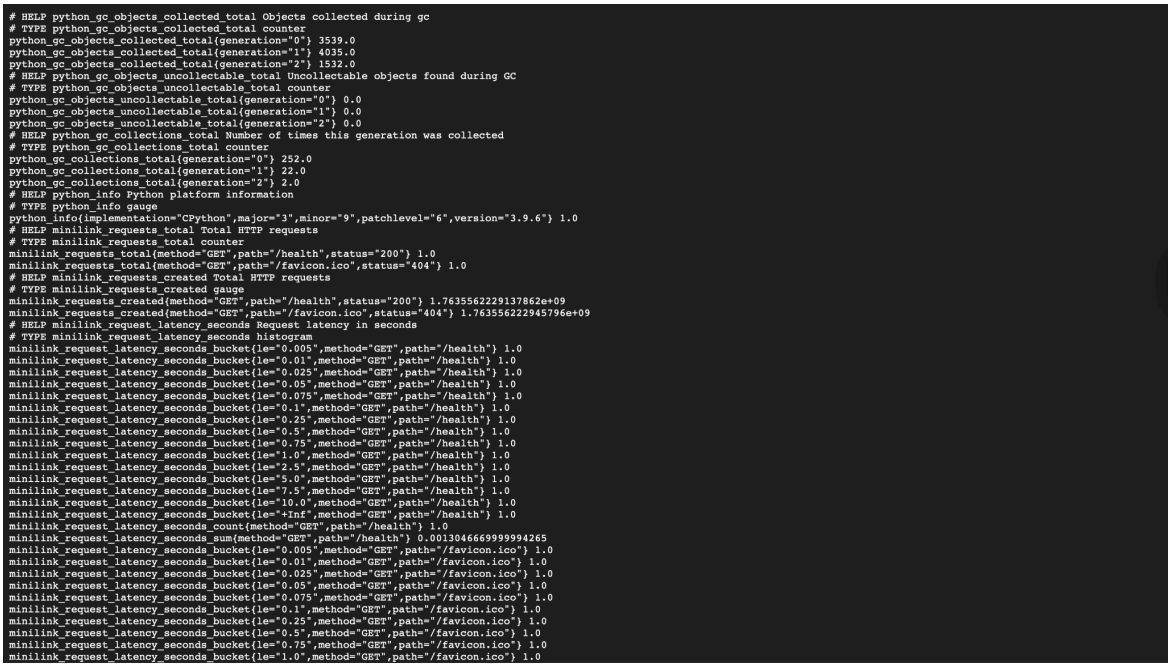

deploy
succeeded 6 hours ago in 59s
Search logs
Set up job 2s
Checkout repository 0s
Log in to GitHub Container Registry 1s
Build and push Docker image to GHCR 52s
Post Build and push Docker image to GHCR 1s
Post Log in to GitHub Container Registry 0s
Post Checkout repository 0s
Complete job 0s

C2. Render Deployment Dashboard



Link: https://minilink-9gdf.onrender.com

C3. Metrics Output



```
# HELP python_gc_objects_collected_total Objects collected during gc
# TYPE python_gc_objects_collected_total counter
python_gc_objects_collected_total{generation="0"} 3539.0
python_gc_objects_collected_total{generation="1"} 4035.0
python_gc_objects_collected_total{generation="2"} 1532.0
# HELP python_gc_objects_uncollectable_total Uncollectable objects found during GC
# TYPE python_gc_objects_uncollectable_total counter
python_gc_objects_uncollectable_total{generation="0"} 0.0
python_gc_objects_uncollectable_total{generation="1"} 0.0
python_gc_objects_uncollectable_total{generation="2"} 0.0
# HELP python_gc_collections_total Number of times this generation was collected
# TYPE python_gc_collections_total counter
python_gc_collections_total{generation="0"} 252.0
python_gc_collections_total{generation="1"} 22.0
python_gc_collections_total{generation="2"} 2.0
# HELP python_info Python platform information
# TYPE python_info gauge
python_info{implementation="CPython",major="3",minor="9",patchlevel="6",version="3.9.6"} 1.0
# HELP minilink_requests_total Total HTTP requests
# TYPE minilink_requests_total counter
minilink_requests_total{method="GET",path="/health",status="200"} 1.0
minilink_requests_total{method="GET",path="/favicon.ico",status="404"} 1.0
# HELP minilink_requests_created Total HTTP requests
# TYPE minilink_requests_created gauge
minilink_requests_created{method="GET",path="/health",status="200"} 1.7635562229137862e+09
minilink_requests_created{method="GET",path="/favicon.ico",status="404"} 1.7635562229457796e+09
# HELP minilink_request_latency_seconds Request latency in seconds
# TYPE minilink_request_latency_seconds histogram
minilink_request_latency_seconds_bucket{le="0.005",method="GET",path="/health"} 1.0
minilink_request_latency_seconds_bucket{le="0.01",method="GET",path="/health"} 1.0
minilink_request_latency_seconds_bucket{le="0.025",method="GET",path="/health"} 1.0
minilink_request_latency_seconds_bucket{le="0.05",method="GET",path="/health"} 1.0
minilink_request_latency_seconds_bucket{le="0.075",method="GET",path="/health"} 1.0
minilink_request_latency_seconds_bucket{le="0.1",method="GET",path="/health"} 1.0
minilink_request_latency_seconds_bucket{le="0.25",method="GET",path="/health"} 1.0
minilink_request_latency_seconds_bucket{le="0.5",method="GET",path="/health"} 1.0
minilink_request_latency_seconds_bucket{le="0.75",method="GET",path="/health"} 1.0
minilink_request_latency_seconds_bucket{le="1.0",method="GET",path="/health"} 1.0
minilink_request_latency_seconds_bucket{le="2.5",method="GET",path="/health"} 1.0
minilink_request_latency_seconds_bucket{le="5.0",method="GET",path="/health"} 1.0
minilink_request_latency_seconds_bucket{le="7.5",method="GET",path="/health"} 1.0
minilink_request_latency_seconds_bucket{le="10.0",method="GET",path="/health"} 1.0
minilink_request_latency_seconds_bucket{le="+Inf",method="GET",path="/health"} 1.0
minilink_request_latency_seconds_count{method="GET",path="/health"} 1.0
minilink_request_latency_seconds_sum{method="GET",path="/health"} 0.0013046669999994265
minilink_request_latency_seconds_bucket{le="0.005",method="GET",path="/favicon.ico"} 1.0
minilink_request_latency_seconds_bucket{le="0.01",method="GET",path="/favicon.ico"} 1.0
minilink_request_latency_seconds_bucket{le="0.025",method="GET",path="/favicon.ico"} 1.0
minilink_request_latency_seconds_bucket{le="0.05",method="GET",path="/favicon.ico"} 1.0
minilink_request_latency_seconds_bucket{le="0.075",method="GET",path="/favicon.ico"} 1.0
minilink_request_latency_seconds_bucket{le="0.1",method="GET",path="/favicon.ico"} 1.0
minilink_request_latency_seconds_bucket{le="0.25",method="GET",path="/favicon.ico"} 1.0
minilink_request_latency_seconds_bucket{le="0.5",method="GET",path="/favicon.ico"} 1.0
minilink_request_latency_seconds_bucket{le="0.75",method="GET",path="/favicon.ico"} 1.0
minilink_request_latency_seconds_bucket{le="1.0",method="GET",path="/favicon.ico"} 1.0
```