URL SHORTENER

Javier Domínguez Segura || 2nd Assignment
Github repository | Documentation

RECAP

IDEA EXPLANATION

A URL shortener is a service that converts long web addresses (URLs) into shorter, more manageable links. It works by creating an alias that redirects users to the original, longer URL. This makes links easier to share on social media, in text messages, and in print. We provide this service with several different encoding algorithms.

PRIOR PROGRESS

In the initial phase of this project, I developed a fully functional URL shortener service with comprehensive full-stack capabilities. The implementation includes CRUD operations across 10 RESTful API endpoints, persistent storage using MySQL and S3 buckets, and a minimal web-based UI. The development environment features hot-reload capabilities through Docker Compose, significantly accelerating iteration speed.

Following an Agile methodology, I incrementally built the system from basic URL shortening functionality to a production-ready application with authentication, click analytics, and multi-environment deployment support (dev, staging, production). The infrastructure leverages Terraform for IaC, Ansible for configuration management, and automated scripts for environment setup and tunneling. Design patterns including Factory, Singleton, and Template Method were implemented to ensure maintainability and extensibility. This foundation prioritized learning core software engineering principles such as architectural design, DevOps pipelines, and infrastructure management, over complex business logic.

UPDATES

IMPROVEMENTS

URL Shortener has a large series of upgrades added since the first assignment. These improvements have mostly concentrated around the dev-ops pipeline,cloud-services security, monitoring, tests and developer experience (connection scripts, modularity, health checks, gemini).

Dev-Ops pipeline

I have integrated a complete CI/CD that deploys our code changes into our cloud environment automatically. It goes through an extensive continuous integration pipeline that among other things, checks the modifications to the repo do not break existing functionality and adhere to the repository coding standards (more info about this later). For continuous delivery I have replicated the prior AWS cloud environment into Azure. This makes the project multi-cloud, bringing resilience and minimal downtime at the expense of higher technical overhead and costs. I have also rolled back from a gitflow repository strategy to trunk-based model. This decision is due to the significant unnecessary overhead that brings the former approach.
I also switched to *uv* as our package-installation provider (contrary to pip). In addition to adding a cacheable layer in the dockerfile images I saw a decrease in build time of 87.5% (from 4 minutes to 30 seconds)

Cloud-services

Besides the integration of the multi-cloud strategy previously mentioned, I have also maintained AWS's staging environment which allows us to debug in an environment that mirrors production more closely, thus capturing errors earlier in the process. However, this comes at the expenditure of increased infrastructure costs and longer development time. Regarding container registry, I have decided to stick to Docker's standard Docker Hub due to some recurrent issues with Azure's service.

Security

In order to make sure that deployment from local host to production is minimally exposed I have integrated several security best practices. I have used environmental variables and github secrets to make sure credentials are never exposed to malicious actors. Besides that, I have a pre-commit hook that asserts no private keys are never committed (i.e., the hook fails the commit if the private key is committed). I also integrated a custom environmental variable script that checks that all the environmental variables needed for the selected cloud provider and environment (e.g., aws locally vs azure in prod).
In the CI pipeline I also have static security checkers that make sure security best practices are implemented. For example, in one failed CI, I saw how the analyzer demanded to use

the more secure *cryptography* library for random number generation (used in one of the strategies of the shortening service) instead of *random* library.
Finally, setting most patterns for secrets naming into gitignore is a simple but effective strategy that minimizes uncareful commits of secrets.

Tests

The tests suite has been significantly expanded. I currently have several different types of tests used both in the pre-commit as well as in the CI pipeline. I added integration, load (with k6), unit (expanded), smoke and regression tests. These expanded suites of tests have already proven to be extremely useful. They have allowed me to capture early subtle errors that could have a serious impact in a production setting. I currently have over 85% test coverage. Coverage is a requirement of the pipeline which fails when having less than 70% of total test coverage.
While not strictly a test, we have added another mechanism to assert the uptime of our app. I have created two endpoints under the *health* router: /ping and /dependencies. These capture basic information about the status of the app and allow a deployment if the health checks fail. I have also added health checks into most docker services. This allows to make sure that poisonous services do not get deployed and provide the false impression of a successful deployment.
Overall, we have tried to provide an ecosystem of tests and health checks that try to bring bugs into observation as soon as possible. We believe adopting a fail-fast approach, allows us to minimize the probabilities of having unexpected production failures.

Developer Experience

In order to make the iterative development earlier I have adopted several best practices. First, I have registered most ideas for improvement into the issues tab of github. This provides a central place of ground truth data that can be used to provide the clear steps ahead.
For debugging in production, besides telemetry, I have developed a series of scripts that seamlessly create ssh tunnels from localhost into the server. This proved to be super useful when debugging staging environments, database and the docker compose as a whole.
The repository has significantly grown in length. Currently, it is around 13k lines of code. Thus the need to have substantial modularity is imperative. With respect to the prior version, we have created the /deployment directory. Here are all the configurations of the different docker compose (both base + environment-specific) as well as the telemetry and reverse proxy configuration.
Finally, the usage of pre-commits allowed us to run unit tests locally, as well to check formatting, long files and debug statements before making any commit. This ensures commits adhere to some minimal level of standarization.

Monitoring

I have integrated Grafana alongside the prior existing log functionality. In a grafana dashboard (see appendix) I am using a FastAPI template that brings a large amount of functionality out of the box. I am currently using Prometheus + Grafana docker images in order to access the telemetry functionality. I am also using a custom logger that provides

enhanced event metadata for faster debugging processes. Given the nature of the sensitive information exposed in this service, security is paramount. I have created two layers of security to prevent unauthorised access. The service suspended the production service, due to nginx not providing a route toward Grafana port. Thus, the only way to access this is by creating a ssh tunnel into the machine that maps our 3000 port to the server's 3000 port (where Grafana resides). In order to create this ssh tunnel you need the ssh pem key generated by terraform, which is not publicly shared. After that, you also need the grafana admin password which is another secret.
This telemetry service is extremely useful for debugging in production. It captures most of the information needed to troubleshoot user sessions, making it an invaluable tool.
You can find the service configuration here: docker compose, prometheus, grafana dashboard

CI/CD PIPELINE

My CI/CD implementation represents a comprehensive automation strategy that handles everything from code quality checks to production deployment. The pipeline is split into continuous integration (triggered on every push) and continuous delivery (triggered by semantic version tags).

Continuous Delivery

The CD workflow activates when I push a semantic version tag like *v1.2.3.* It starts by comparing the current commit against the previous tag to detect infrastructure changes in the infra/terraform or infra/ansible directories. This change detection is crucial because it determines whether we need a full infrastructure deployment (around 10 minutes) or just an application update (around 3 minutes).

The pipeline handles secrets through GitHub Actions secrets and environment variables. I create production environment files dynamically during the workflow, injecting sensitive values like Azure storage keys and Firebase credentials that never touch the repository. The SSH private key gets written to *~/.ssh/ssh_key.pem* with restricted permissions for Ansible to use. We also write the corresponding firebase credentials that are needed for the backend (written to *backend/src/url_shortener/core/clients/secret.url-shortener-abadb-firebase-adminsdk-fbsvc-48d38c91f0.json*)

After installing dependencies (with *uv* and caching aggressively for speed), the pipeline builds the frontend, pushes a multi-platform Docker image tagged with both the version and environment, then executes the deployment through a series of Makefile targets that orchestrate Terraform, environment synchronization, and Ansible.

Infrastructure as Code

My Terraform setup provisions Azure resources (primary choice, equivalent AWS services is also present) including a Virtual Network with public and private subnets, Network Security Groups controlling HTTP/HTTPS/SSH access, a MySQL Flexible Server in a private subnet, Key Vault for credential storage, and the application VM. The state is stored remotely in Azure Storage for security and remote access by the github runner [1].

What makes this particularly elegant is the environment synchronization system. I wrote a Python script that extracts Terraform outputs and injects them into three targets: backend dotenv files, frontend dotenv files, and Ansible inventory files. The script uses Pydantic models to validate outputs per environment, ensuring type safety. It also supports injecting outputs from an environment variable (TERRAFORM_OUTPUTS_JSON) when Terraform credentials aren't available in the pipeline [1].

As expressed before, terraform has support for both azure and aws for the corresponding services for the different dev, staging and production environments.

Deployment with Ansible

The Ansible playbook implements progressive rollout with automatic rollback. It deploys the new Docker Compose stack, performs health checks against the /api/health/dependencies endpoint with configurable retries, and if the checks fail, it automatically rolls back to the previous version stored in a tag file on the server. The playbook also captures Docker logs from failed deployments for debugging.

Continuous Integration

I have engineered a Continuous Integration pipeline that prioritizes a "fail-fast" philosophy to respect developer time while ensuring code integrity. Leveraging GitHub Actions, I split the backend pipeline into three distinct stages. The first stage focuses on speed: I utilized ruff for instantaneous linting and formatting. Simultaneously, bandit performs static security analysis to catch vulnerabilities before they leave the branch.

Only upon the success of these checks does the pipeline commit to Stage 2. This stage spins up a complete Dockerized environment to run a comprehensive test suite unit tests for logic, smoke tests for basic health. If making a PR to main and the Stage 2 ran successfully, then Stage 3 is executed. Here you have heavy integration/regression tests that simulate real-world usage against the Azure and AWS configurations (using mock API providers, though. We never hit the real cloud servers when testing). I also plan to add load tests into this third and final stage.

This tiered approach ensures we catch syntax errors in seconds and logic errors in minutes, preventing broken code from ever reaching the merge queue.

Finally, while not strictly part of the github action, we have added multiple AI PR-reviewers into the iterative development process. This simplifies dramatically the PR review process as well as providing direct feedback to the code owner for introduced bugs/deviation of code standards.

Makefiles

The entire system is orchestrated through hierarchical Makefiles that provide clean abstractions. Running make deploy-start from the root triggers make -C infra terraform-apply, make -C infra sync_all, make -C frontend build, make -C backend push_docker, and finally make -C infra ansible-start. Each component is independently testable and the color-coded output makes debugging straightforward.

This Makefile-centric approach solves a critical problem: keeping deployment logic consistent between CI/CD pipelines and manual operations. Without this abstraction, I'd maintain duplicate deployment logic in both GitHub Actions workflows and shell scripts for local/emergency deployments. This duplication inevitably leads to drift where the CI pipeline succeeds but manual deployments fail (or vice versa) because they're executing subtly different commands. By having GitHub Actions call the same Makefile targets that I use locally, I guarantee that make deploy-start behaves identically whether executed by a developer, system administrator, or automated pipeline. This single source of truth has already prevented numerous issues where environment-specific flags or dependency installation steps would have diverged between automation and manual intervention.

Conclusion

This CI/CD architecture has dramatically improved deployment confidence. The combination of comprehensive testing, intelligent change detection, health checks, and automatic rollback means I can deploy to production knowing the system will either succeed completely or fail safely without leaving the application in a broken state.

Currently the services are deployed to IE’s provided azure services.

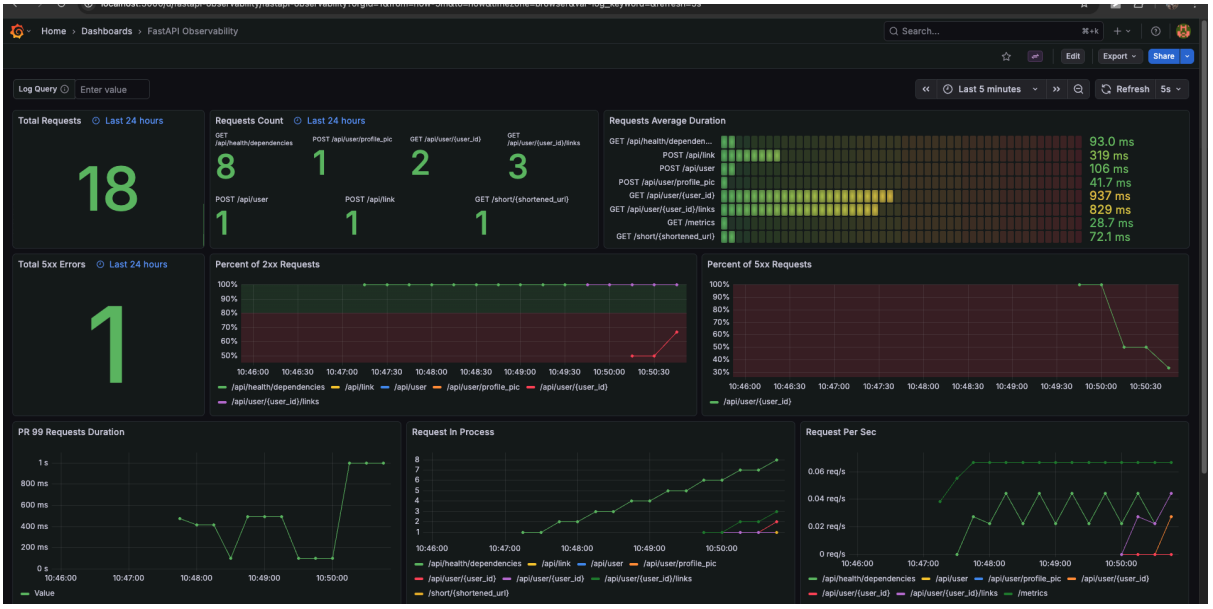
APPENDIX

REMARKS

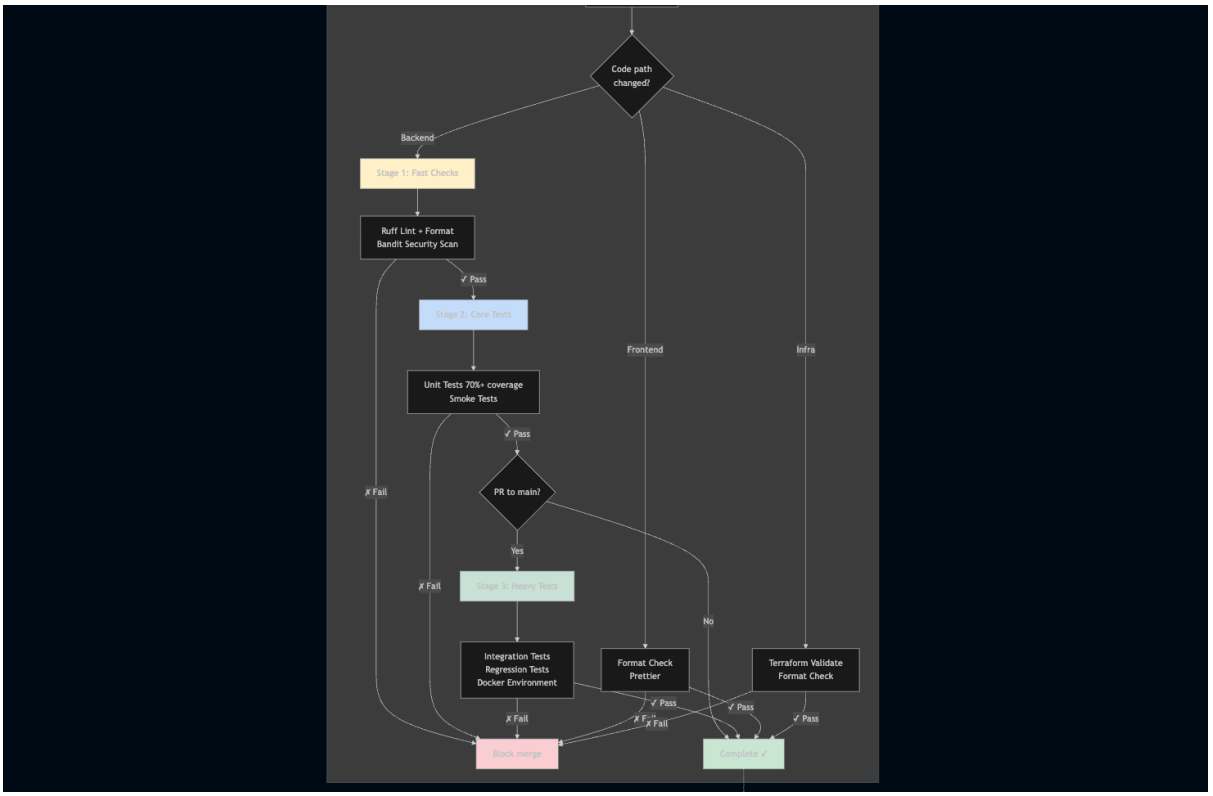
[1] Currently, the runners do not have permission to query the azure ecosystem for the corresponding container name. Thus, the permissions to run any terraform command when using a remote state is not allowed. While I wait for this to be solved for the group project, the workaround I did was to deploy the infra locally in my desktop (after doing *az log in*), then get the terraform outputs json and add it as a environmental variable for the scripts that syncs the base environmental variable and ansible inventory based on tf outputs.

DIAGRAMS AND DASHBOARDS

GRAFANA

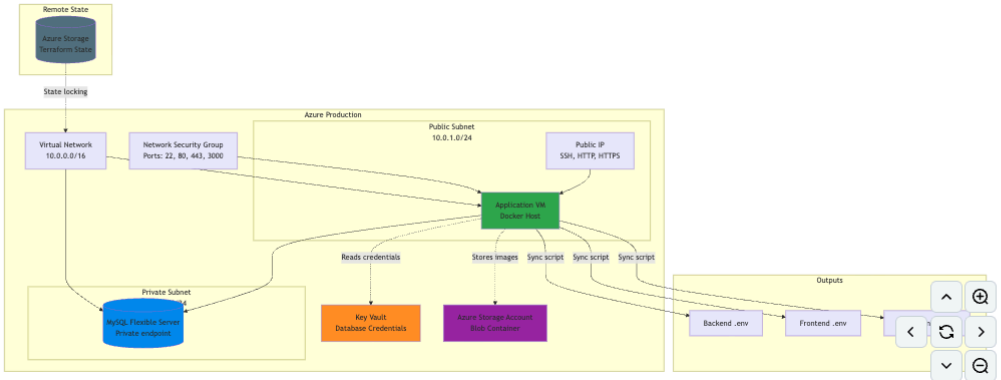


CI/CD PIPELINE



See [image here]() for better visualization

INFRASTRUCTURE



See [image here]() for better visualization

LINKS

[docker compose](), [prometheus](), [grafana dashboard]()