Domain Pulse

DevOps Excellence

Table of Contents

1. Introduction

2. Non-Functional Requirement Focus

3. CI/CD Tools and Usage

4. Observability Principles

5. Operational Practices

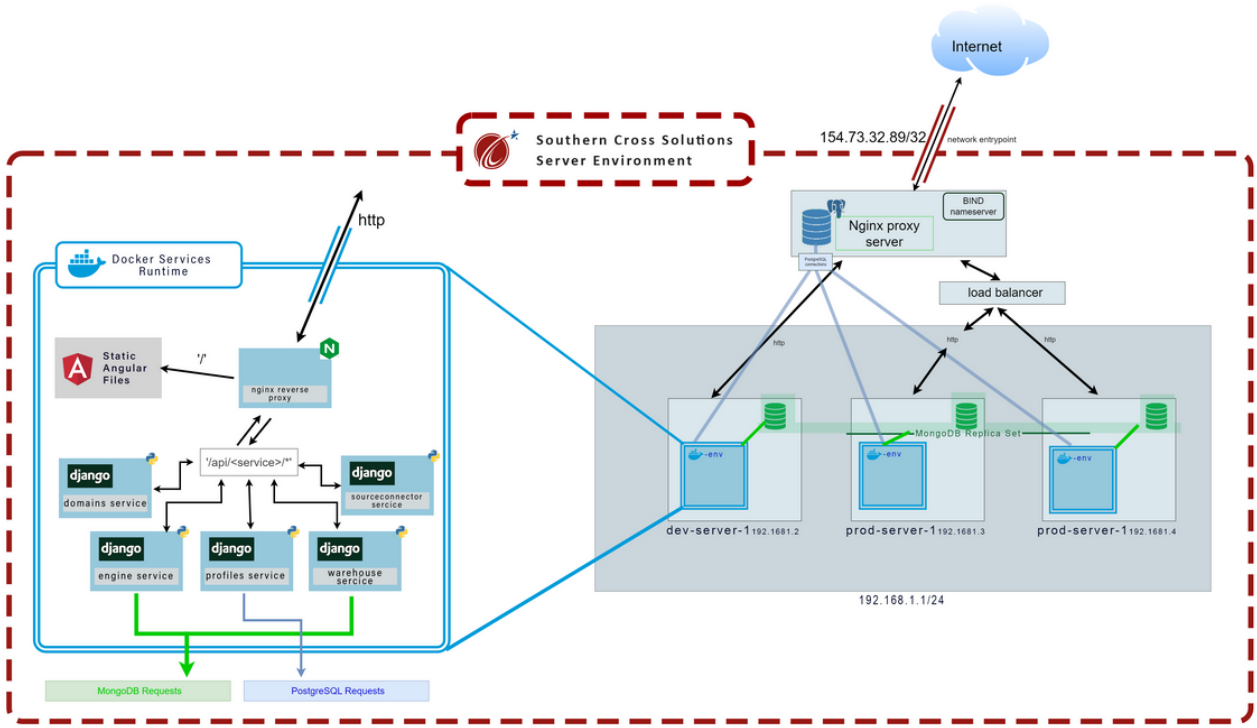
1 Introduction

Within Domain Pulse, DevOps was an integral piece of our project, that was highly focused on and perfected, so as to ensure smooth and easy deployments, high quality of code and observability of our important, intensive services.

2 Non-Functional Requirement Focus

Our team's commitment to excellence is evident in our meticulous attention to non-functional requirements. We didn't just focus on the project's features but also delved deep into its non-functional aspects, ensuring seamless performance, scalability, maintainability, usability and security. Rigorous testing became our mantra, where we subjected every component to exhaustive performance assessments, ensuring it could handle varied workloads flawlessly.

Scalability was achieved through careful architecture design. The implementation of a Service-Oriented Architecture, comprising five independently deployable services, allows our application services to be replicated on demand to handle higher loads, through the use of docker containers.



This scalability was then tested through the use of an external script that would run and make a variable number of requests, from 10 to 1000 simultaneous requests.

Security was paramount; We compared our security measures against the OWASP top ten vulnerabilities list.

Maintainability was tested thoroughly using the modifiability index and cyclomatic complexity, whereby we ensured scores that far and away exceeded what would be considered ‘good’.

Finally our usability was ensured and tested through aggregate and processing responses from our usability test surveys that we handed out to a large test audience. All of these tests of our non-functional requirements had strict minimum and maximum requirements we allowed and our dedication to these rigorous testing methods not only validated our non-functional requirements but elevated our project to an unparalleled level of reliability and resilience.

3 CI/CD Tools and Usage

Our streamlined development process incorporates robust CI/CD practices, enhancing our agility and efficiency. Within our repositories, we've crafted custom shell scripts designed to expedite the setup of new machines, accelerating our scaling capabilities significantly.

These scripts serve as the backbone of our infrastructure, ensuring swift and consistent deployment. Leveraging GitHub Actions, these scripts are seamlessly integrated into our pipeline.

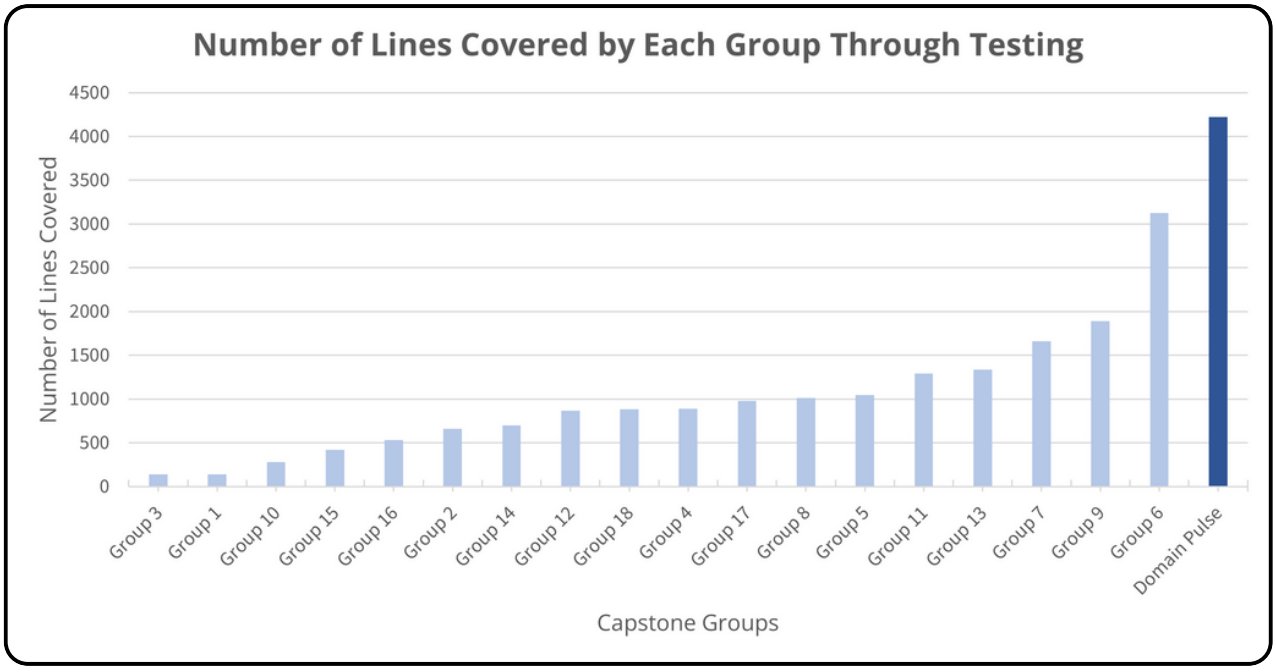
Upon pushing to our main or dev/stage branches, these scripts are triggered automatically by GitHub Action runners. Our pipeline begins with a pull request to the main or stage branch, followed by rigorous code tests to guarantee quality (and ensure that the code coverage, calculated by codecov, of both the patch code and project codebase does not decrease).

At time of writing, the code coverage of the project sits comfortably above 90%, with over 4000 lines covered (excluding testing and boilerplate files).

Once the tests pass, the pull request is approved, initiating our custom scripts. These scripts, accessed via SSH, effortlessly fetch the latest service images and execute them on our host machines. This automated deployment process not only ensures reliability but also reflects our commitment to efficient operations, enabling us to deliver high-quality services with unparalleled speed and precision.

Early on during our development, we realised the importance of rigorous testing. We utlised Codecov to its full potential and followed strict development guidelines for our Pull Requests and test coverage, in which the diff coverage had to be above a certain matching percentage and the PR could not decrease the overall coverage.

With consistent effort we were able to cover the most lines of code out of all the capstone groups. Below you can see a graph of a comparison of the number of lines covered, which is calculated as follows: ***Nr. of lines tracked x Code coverage %.*** This calculation is important, since a high coverage with low nr. of lines tracked would mean that little testing was done.



A comparative breakdown of all the capstone projects' codecov can be found here

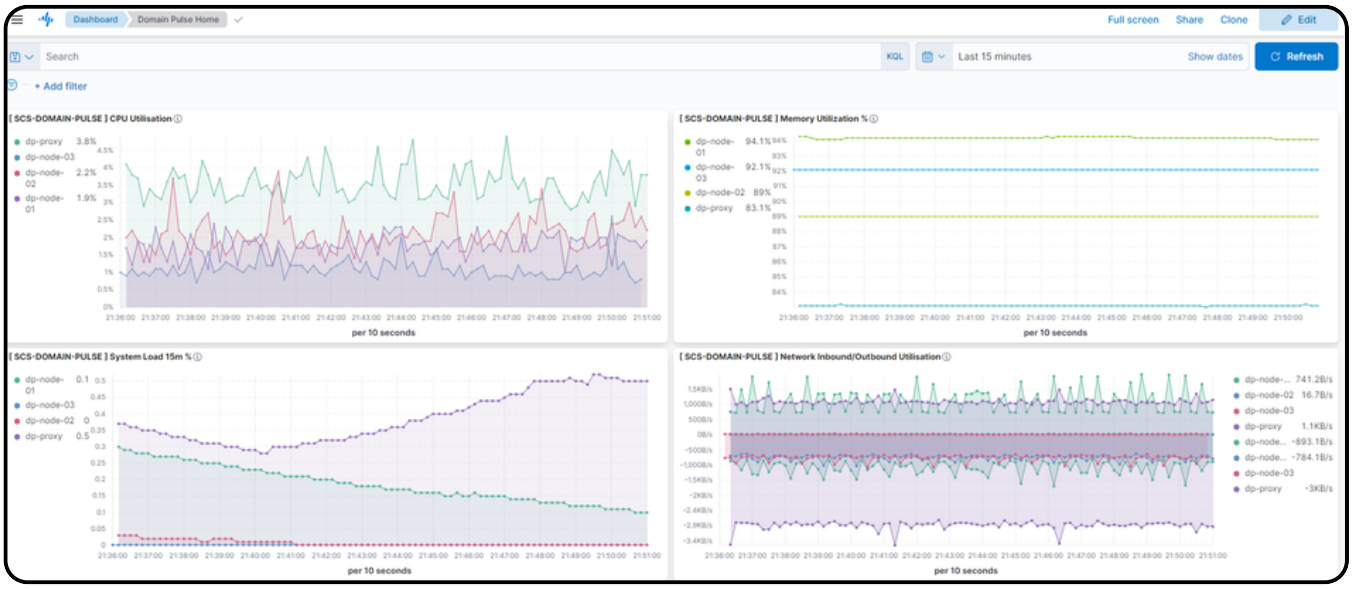
4 Observability Principles

Embracing observability principles is integral to our approach, ensuring that every facet of our infrastructure is transparent, understandable, and readily monitored.

We use APM agents across our Django backends, gaining us insights into our application's performance, allowing us to proactively identify bottlenecks and optimise code pathways. Additionally, our commitment to availability monitoring is exemplified by our custom endpoints that are systematically pinged, validating the reliability of our services.

Leveraging the powerful tools provided by Azure and Digital Ocean, we enhance our observability further. Azure's Application Insights grants us unparalleled visibility into our application's behaviour and helps us analyse telemetry data, enabling us to quickly identify and fix issues that arise.

Meanwhile, Digital Ocean's monitoring tools offer real-time metrics, ensuring our infrastructure's health. By diligently applying observability principles and harnessing these advanced tools, we empower our team to maintain the highest standards of performance, reliability, and user experience.



5 Operational Practices

Our operational practices prioritise redundancy and security. Databases are redundant across Gauteng and London, ensuring continuous service.

We have set up the backup replicas such that the data is 1 hour old. This is such that if something happens to our data, from a bad actor gaining access to our systems, that if there is corruption/deletion of our data, we have a point snapshot from which to restore the application . GitGuardian protects sensitive information from leaks.

Additionally, all our services are containerized, enhancing portability. These practices guarantee reliability and data security, showcasing our commitment to efficiency and robust protection. As well as the aforementioned practices, we also employ a rigid and well structured branching strategy and commit structure in our GitHub Repository.

During our development, we followed a branching strategy, of feature branching, with a development branch off of the master branch. Feature branches would be branched off from dev, and have code specific to the branches name (which follows a format of <*Section*>/<*feature*> such as backend/reportgenerator) developed on said branch.

To utilise this branching strategy, we then permitted developers to only create pull requests from feature branches to dev, to have a single branch for feature branches to merge, and only dev could be merged into the master branch, protecting the master branch from being affected by unchecked code.

All commits to the branches of our repository followed a format of ‘emoji’ (Section)/<Change> (e.g. 🐛(backend) Fixed Profile Bug) to allow for developers to easily track and understand changes made to the codebase.