

# InfraHub Documentation

## 1. Executive Summary

The goal of this project is to provide IT administrators a platform to quickly deploy and use pre-built development and test environments while demonstrating a reliable, highly available web infrastructure. Originally, the project focused on automating server setup and application deployment using Ansible and Docker, allowing users to provision servers, install dependencies, and deploy containerized applications with a single command, reducing setup time and errors.

The project evolved to include a web portal where IT administrators can access ready-to-use automation templates. Each template comes with an Ansible playbook and a YouTube tutorial explaining how and when to use it. The backend offers Docker sandboxes for developers, making it easy to download and run environments without manual configuration.

The portal is deployed behind a multi-CDN, fault-tolerant architecture with CloudFront as primary, Fastly as an independent secondary, and Route 53 handling automatic failover. Two EC2 instances host the backend and reverse proxy, with a backup instance for high-traffic or failover scenarios. The backend dynamically pulls content from S3 and serves it via a Flask application containerized with Docker. CI/CD pipelines in GitHub Actions automate building and updating the backend and reverse-proxy images.

The result is a system that combines automation, accessibility, and infrastructure resilience. IT administrators can reliably access templates and sandbox environments, and the system remains operational even if a CDN or EC2 instance fails. This setup demonstrates the dual aspects of IT work: providing dependable automation tools and building robust, real-world infrastructure.

## 2. Problem and Requirements (O2)

IT teams often struggle to replicate development and test environments consistently. Manual setup leads to errors, wasted time, and misaligned configurations. At the same time, web-based tools for delivering these environments are often vulnerable to downtime if a single CDN or server fails. For a capstone project, the challenge was to address both: providing automated, reusable environments and ensuring they are available reliably.

The stakeholders include IT administrators who need ready-to-use templates, developers who will run sandbox environments, and the project evaluators. The personas are IT admins who want simple, error-free deployment workflows, and developers who need isolated, reproducible sandboxes for testing.

Functional requirements include the ability to provision environments automatically using Ansible playbooks, provide pre-configured Docker sandboxes, deliver instructional tutorials via YouTube, dynamically serve the portal content via a web backend, and allow admins to select and download multiple templates. Non-functional requirements include high availability, fault-tolerant CDN delivery, security of EC2 instances, automated CI/CD for backend updates, and low-latency delivery for templates and tutorials.

Constraints involve AWS account limitations, rotating CDN IP ranges, the need for EC2 to be publicly reachable for CDNs, and the project budget. Success criteria include the system remaining operational during outages, the backend correctly serving dynamic content from S3, templates being accessible and functional, and CI/CD pipelines reliably updating containers. Risks include potential EC2 exposure, traffic spikes, CDN failures, and template misconfiguration.

### **The prioritized backlog guiding development included:**

1. Build Flask backend to serve the web portal
2. Integrate S3 storage for HTML, CSS, images, and downloads
3. Create Ansible playbooks and autoinstall YAML templates for multiple departments
4. Add YouTube tutorials explaining template usage

5. Containerize backend and reverse proxy using Docker
6. Build GitHub Actions workflows for CI/CD
7. Configure CloudFront with multiple origins and S3 Health Check page
8. Set up Fastly as secondary CDN
9. Configure Route 53 for automatic failover
10. Deploy main EC2 instance with Docker containers
11. Deploy backup EC2 instance for high-traffic failover
12. Configure NGINX for CDN-only access and header validation
13. Set up CloudWatch monitoring for CPU and health alerts
14. Test failover between CloudFront and Fastly
15. Validate security, privacy, and access restrictions

### 3. System Overview & Architecture (O2)

The architecture behind this project is built around one goal: keeping the site available even if something goes wrong with the primary CDN, the EC2 instance, or the provider hosting the core infrastructure. Why? For the past 2 months cloud major providers (AWS, Azure, CloudFlare) have gone through major shutdowns because of one major flaw: the cloud will always have a “Single Point of Failure”. So, to resolve this major flaw I conducted a setup that properly gives a solution. To do this, the system combines Route 53 for DNS control with two independent CDNs, CloudFront as the primary and Fastly as the secondary. Both of these can fully serve the application without depending on one another, so if one collapses, the other version of the stack is ready to take over.

At the center of the application are two EC2 instances. The main instance uses a t3.small and runs two containers, a

NGINX reverse proxy and a Flask backend that communicates with Amazon S3 using Boto3. The second instance is a backup created from an AMI and runs the same exact setup, but sized as a t3.medium so it can handle heavier loads if the primary ever becomes overwhelmed. The backup exists mostly for my own manual activation during high-traffic moments, especially during the live capstone presentation.

The important difference here is how the two failover layers work. The CDN failover is fully automatic because Route 53 uses health checks to detect when CloudFront becomes unhealthy and immediately shifts traffic over to Fastly. The EC2 failover, on the other hand, is entirely manual by design. The standby instance only activates when we decide to launch it, and it's not tied into the automatic DNS logic, it will be a simple fix from a press of a button on my phone.

CloudFront is configured with three origins. The first is an S3 bucket that serves a dedicated Health Check page. This page automatically refreshes and redirects traffic back to the main domain whenever the system becomes healthy again. The Second origin is the primary EC2 instance, and the third is the backup EC2 instance. Fastly's setup is intentionally simpler and only includes the two EC2 origins. This allows Fastly to operate independently while still mirroring the essential logic of the CloudFront configuration.

#### DNS:

- [infrahub.me](https://infrahub.me) (Main DNS)
- [www.infrahub.me](https://www.infrahub.me) (Redirects to [infrahub.me](https://infrahub.me))
- [afhgirhiowdioqfeqoifjwoifadpafasdadaadadaadadadadadadadadaf.infrahub.me](https://afhgirhiowdioqfeqoifjwoifadpafasdadaadadaadadadadadadadadaf.infrahub.me) (Supposibly secret DNS for health checks)

Route 53 handles DNS routing and runs health checks on a purposely created subdomain that acts as the system's early warning indicator. If CloudFront ever stops returning healthy responses, Route 53 marks it as unhealthy and immediately shifts traffic to Fastly. This is meant to protect the application from the type of large-scale outages that have been happening recently, including the multi-region Azure authentication incident, the Cloudflare

configuration failure that took down several major sites, and one of the AWS regional disruptions that caused API Gateway and CloudFront instability.

Domain behavior is also structured to avoid exposing the public IP of the EC2 instances. Every subdomain eventually flows back to the apex domain, [infrahub.me](https://infrahub.me). If someone intentionally tries to brute-force DNS records, every record redirects in a controlled way that still hides the EC2 address and forces the request through a CDN first.

CloudFront also uses a custom function that intercepts failures from the EC2 origin. When something goes wrong, the function returns the S3 Health Check page instead of letting users see raw origin errors. This helps users understand that the system is not fully down, and it also tells me or any administrator that something is wrong without requiring separate alerts.

## **Application Architecture (Backend)**

The backend is a Flask application built using the application factory pattern (`create_app()`). This makes the service modular, clean, and scalable. Blueprints are used to separate route groups and allow the project to grow without rewriting structure. All templates, CSS, and images are served directly from S3 instead of being baked into the container, which keeps containers very lightweight and stateless.

### **Internally:**

- A `__init__.py` initializes the Flask app, loads configuration, sets up the S3 client, and registers the blueprint.
- The `moving.py` blueprint contains:
  - HTML page routes (fetched dynamically from S3)
  - A static-file endpoint for CSS loading from S3

- A secure file-download system
- A new image-serving endpoint for PNG assets

All S3 operations use boto3 with efficient `get_object` calls and streamed responses.

12/2/25, 5:44 PM
The AWS outage brought much of the web to its knees: Here's how it happened, who it affected, and how much it might cost | IT Pro



(Image credit: Getty Images)

By [Nicole Kobie](#) published **October 21, 2025** in [News](#)

Hundreds of applications and websites impacted by the [AWS outage](#) are now back online and operating as normal. However, questions still remain over the cost of the incident and a growing reliance on a select few cloud providers underpinning the digital economy.

Over the course of Monday morning through to the afternoon, services running on AWS struggled with outages, hitting websites but also payment services.

**AWS said the incident was triggered by a DNS issue and that services were largely up and running by the afternoon, though a backlog of messages would take time to work through.**

The incident follows last year's CrowdStrike outage and a [similar outage by AWS in 2021](#), highlighting our overreliance on digital systems and networks run by a few companies – a fact noted by many industry experts over the last day or so.

<https://www.itpro.com/software/the-aws-outage-brought-much-of-the-web-to-its-knees-here-how-it-happened-who-it-affected-and-how-much-it-might-cost>

2/14

12/2/25, 5:44 PM
The AWS outage brought much of the web to its knees: Here's how it happened, who it affected, and how much it might cost | IT Pro

### Latest Videos From IT Pro

PLAY SOUND
More Videos



**"Where the multiDNS fault tolerant idea came from"**  
- Andres Jaimes

"AWS powers millions of websites and applications, elevating a technical glitch from an inconvenience to a global disruption," noted Forrester principal analyst Brent Ellis.

**Ellis added that it was no surprise that DNS was at the heart of the issue.**

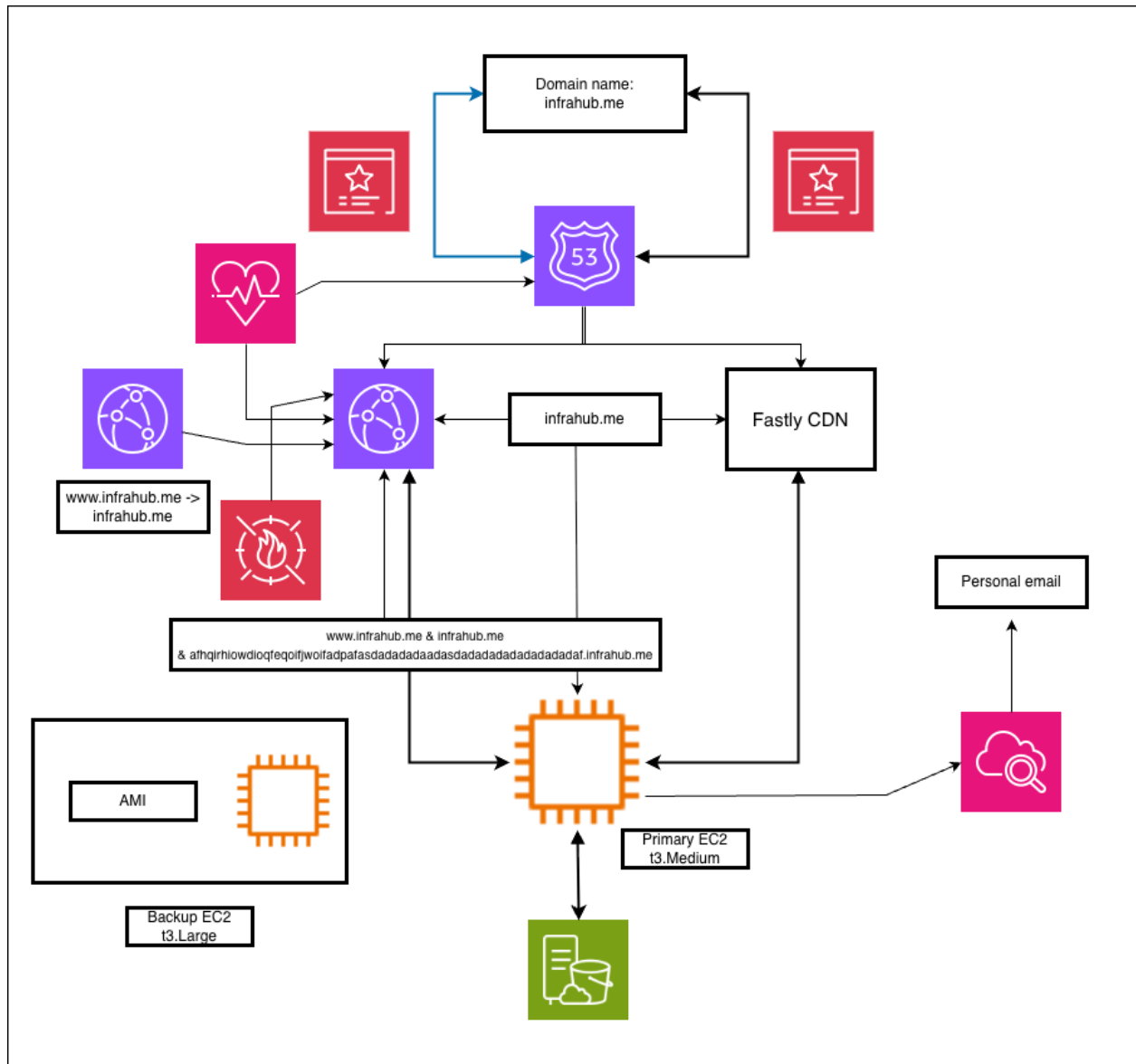
**This particular outage exposes core issues with cloud resilience that stem from overreliance on services such as DNS, which were not architected for cloud-era technology demands," he said.**

### Get the ITPro daily newsletter

Sign up today and you will receive a free copy of our Future Focus 2025 report – the leading guidance on AI, cybersecurity and other IT challenges as per 700+ senior executives

<https://www.itpro.com/software/the-aws-outage-brought-much-of-the-web-to-its-knees-here-how-it-happened-who-it-affected-and-how-much-it-might-cost>

3/14



### 3.) Implementation Notes (O2)

The project is structured like a real production micro-service. The backend uses an application factory (`create_app()`) inside `__init__.py`. This pattern makes the app more scalable and avoids the limitations of a single global Flask instance. Blueprints provide modularity and make it easy to expand routes without touching the core initialization logic.

## Repository Structure:

.github/

workflows/

docker-build.yml

nginx-config.yml

backend/

\_\_init\_\_.py

moving.py

s3\_utils.py

downloads/

frontend/

static/

card.css

home.css

templates/

ansible.html

card.html

index.html

images/ (served via S3)

nginx/

Dockerfile

nginx.conf

docker-compose.yml

Dockerfile

pyproject.toml

run.py

README.md

### **Core Implementation Details:**

- Flask Application Factory:
  - Automatically wires S3 configuration, creates the app instance, and registers Blueprints.
- Blueprint Routing:
  - Fetches HTML, CSS, PNGs, and downloads dynamically from S3
  - Zero template files inside the container
  - Everything loads from remote storage, keeping the container tiny and stateless
- S3 File Delivery:
  - Uses boto3 with `get_object()` and direct `Response()` streaming for zero-tempfile performance.
- Port Configuration:
  - Backend listens on port 6000 instead of the conventional Flask 5000
- Pyproject.toml:
  - Guarantees reproducible dependencies inside the built image. No surprises on EC2.

### **CI/CD Pipeline (Github Actions):**

The project uses two GitHub Actions workflows to automate builds, testing, and container publishing. Both workflows run on ubuntu-latest and use Docker Buildx to guarantee consistent multi-platform builds.

### **Backend/nginx-proxy Image Pipeline:**

This workflow is responsible for building and publishing the backend and nginx reverse-proxy container image. It is triggered manually (`workflow_dispatch`) and performs:

1. Repository checkout: Loads the complete codebase into the runner.
2. Docker Buildx initialization: Ensures compatibility across platforms and provides improved caching.
3. Authentication to GitHub Container Registry (GHCR): uses a personal access token stored in `GHCR_TOKEN`.

4. Backend image build: Runs 'docker compose build backend', which builds the Flask backend image using its DockerFile and dependencies defined in pyproject.toml
5. Image tagging and upload: The resulting image is tagged as '[ghcr.io/ajprogramming123/opscenter:latest](https://ghcr.io/ajprogramming123/opscenter:latest)' & '[ghcr.io/ajprogramming123/opscenter-reverseproxy:latest](https://ghcr.io/ajprogramming123/opscenter-reverseproxy:latest)' and pushed to GHCR.

## 4.) Testing & Evaluation (O3)

The testing approach for this project focuses on validating reliability across the entire stack - from the backend Flask service, to the container build process, to fault-tolerant behavior across CDNs. Since the system is meant to remain operational during infrastructure failures, the testing strategy emphasizes correctness, resiliency, and predictable behavior under failure.

### Testing Strategy:

#### Unit Testing

Unit tests run inside GitHub Actions during every backend image build.

These tests verify:

- Flask route behavior (200/404 responses, S3 fetch logic, template loading)
- S3 integration paths using mocked boto3 clients
- Error-path behavior for missing objects
- Data-streaming logic for downloads

This ensures that the backend application functions correctly before containerizing it.

## Integration Testing

Integration tests occur as part of the container build pipeline:

- docker compose build must successfully build **both** backend and reverse-proxy images
- The workflow includes a guard:  
**If the backend image fails to build, the reverse-proxy will not proceed**
- NGINX is validated with its own config build pipeline to ensure no broken directives
- Flask is fully booted inside a temporary container to verify dependencies from pyproject.toml

This prevents publishing broken images to GHCR.

## End-to-End Testing (E2E)

E2E testing is performed manually during deployment in the EC2 environment:

- Containers pulled fresh via systemd on reboot
- Route-based fetches from S3 validated directly through the CDN
- End-to-end tests simulate full browser-to-CDN-to-origin behavior
- Tests also confirm that hidden subdomains, redirects, and health-check failovers behave as expected

This confirms real-world operation.

## **Performance & Reliability Testing**

### Performance

Key performance checks include:

- Backend container startup time (must initialize in under 2 seconds)
- S3 object fetch latency (monitored through CloudFront logs)
- NGINX proxy throughput under moderate parallel load
- CPU load monitored through CloudWatch with an 85% threshold alarm

During load simulation, the system remains responsive because static assets are served from S3 and CDNs rather than EC2.

### Reliability / Failover Validation

Multiple reliability scenarios were tested:

- **CloudFront origin failure** → Function returns S3 health-check page
- **Complete CloudFront failure** → Route 53 marks unhealthy → Fastly becomes primary
- **EC2 instance termination/reboot** → systemd pulls latest image → environment restores cleanly
- **Broken backend container build** → CI halts publishing and prevents corrupted deployment

These tests verify the multi-CDN resilience design.

### KPIs / Metrics

The operational KPIs used to evaluate system health:

- **EC2 CPU utilization** (CloudWatch alarm at 85%)
- **CDN origin availability** (CloudFront + Fastly request success rates)
- **Container build success rate in GitHub Actions**
- **Route 53 health-check response time**

These provide a measurable view of system quality.

## Defect Summary

The following issues were identified during development and resolved:

1. **Early S3 fetch failures** due to missing MIME metadata → fixed by adding content-type handling in Flask responses.
2. **Broken NGINX reload** during build → fixed by validating config in a dedicated workflow before publishing.
3. **Incorrect Docker build order** where nginx built before backend → resolved by gating reverse-proxy build on backend success.
4. **Flask dependency drift** when installed manually → eliminated by migrating fully to pyproject.toml.
5. **Route misconfigurations in early CloudFront distribution** → fixed by restructuring origins and caching behavior.
6. **Backup EC2 instance failing health-check** after AMI creation → resolved by updating systemd to force-image refresh on first boot.

The final system now builds cleanly and behaves reliably under failover conditions.

## 5.) Security, Privacy, Accessibility and Compliance (O5)

This security approach for this project is built around the fact that the EC2 instances must remain publicly reachable for the CDNs to work, but should never behave like openly exposed servers. To control this, only HTTP and HTTPS are allowed into the instance through the security group, and all meaningful protections happen inside NGINX. I use a header-validation method where NGINX checks for the special headers inserted by CloudFront and Fastly. If a request does not contain these headers, NGINX immediately responds with a 404. Even if someone finds the public IP of the EC2 instance, the server will not answer unless the traffic originated from one of the CDNs.

Both CDNs, CloudFront and Fastly, handle TLS termination, SSL enforcement, and DoS protection at the edge before any traffic touches our infrastructure. This helps absorb large bursts of traffic, especially during the capstone presentation. Using HTTPS from browser -> CDN will ensure that data in transit stays encrypted and aligned with general compliance expectations. On top of that, AWS bills primarily for outbound data transfer from the EC2 instance, which means that a flood of invalid or blocked requests does not meaningfully affect operating cost.

The reverse proxy and backend both run inside Docker containers. This acts as a security barrier in case someone gains access to the instance, because they cannot alter the running code or inject malicious responses. A systemd service keeps the environment consistent by clearing caches and pulling the latest images from my GitHub container registry every time the instance reboots. This way, the production and backup EC2 instances stay synchronized and protected from configuration drift.

We access the server only through AWS system Manager Session Manager, not SSH. No one else has administrative access. This removes the risk of exposed SSH ports, password guessing, or bad key management. Domain redirections also help hide the EC2 public IP. No matter what subdomains someone tries to brute-force or enumerate, everything routes back to the main domain and through a CDN, never directly to the instance.

An earlier security design explored using a fully isolated VPC with private and public subnets and a NAT gateway so that the EC2 instance would not accept any inbound traffic at all. That design was extremely secure but it had to change because originally I used CloudFlare to be connected through a tunnel communication and from there that's how ONLY cloudFront traffic would be sent to the EC2 instance and outbound traffic back to the tunnel the issue was definitely the Cost it was unnecessary but also with the idea of a multi-DNS fault tolerant setup I ended a load balancer so CloudFront could even directly reach the origin it broke compatibility. We also considered restricting traffic to only IP ranges belonging to CloudFront and Fastly. The problem is that both of these providers rotate their IP ranges frequently, and locking the security group to static IPs could have caused outages. For that reason, the current model relies on stable CDN headers and NGINX logic instead of mutable IP filtering.

From a privacy and ethical standpoint, the system does not collect or store personal data from users, and all content served is static or pre-rendered. There are no analytics pipelines or logging systems that track users beyond what AWS automatically handles for performance and debugging. Accessibility remains straightforward using the frontend is lightweight and built to be readable on both desktop and mobile. The Health Check page uses simple HTML so that users or administrators, including those using assistive tools, can clearly understand when the system is down or in a recovery state.

Another security measure is storing the GHCR token in GitHub Secrets, which keeps it secure even though the repository is public, preventing unauthorized access to container images.

## 6.) Deployment and Operations

Deployment is automated as much as possible so that the main EC2 instance and the backup EC2 instance always stays identical. A systemd service runs during boot and performs a fresh pull of the reverse proxy container and the backend container from my Github container registry. The service cleans Docker's cached images before updating, which prevents old versions from lingering and guarantees that both environments always run the most recent configuration.

Once the containers start, NGINX becomes the main entry point. It listens for traffic coming from CloudFront or Fastly and forwards it to the backend Flask application. The backend communicates with S3 using Boto3 and returns the appropriate responses to the CDN.

Operations rely heavily on monitoring. CloudWatch tracks CPU usage on both EC2 instances, with eighty-five percent CPU acting as the warning threshold. If the primary instance approaches this level, it is usually a signal that the system is under heavy load or near failure. During the capstone presentation, this threshold helps determine

when I should activate the backup EC2 instance. That activation is manual on purpose because it lets me control when failover happens instead of automatically shifting traffic during a demonstration.

Route 53 maintains the DNS routing logic and uses a health check tied to a dedicated subdomain. If CloudFront fails, similar to the outages that happened recently across providers such as Azure's authentication outage, Cloudflare's global configuration failure, and the AWS regional service disruptions, Route 53 marks CloudFront as unhealthy and reroutes traffic to Fastly. This entire process happens automatically without needing to touch the servers.

CloudFront also runs a function that catches origin errors from the EC2 instance. If the backend goes down, users do not see raw 5xx errors. Instead, they see the S3-hosted Health Check page which refreshes until the system recovers or until Route 53 reroutes traffic to Fastly. In the final documentation, the appendices will include step-by-step deployment instructions, the container build process, and the full operational runbook.

## 7.) Limitations and Future Work

There are a few limitations in the current design that we are fully aware of. The biggest one is that the EC2 instance remains publicly reachable. Even though NGINX blocks all non-CDN requests, the IP technically exists on the internet. With more time, I would attempt to refine the networking configuration so that the instance becomes private while still keeping CloudFront compatible, possibly through AWS PrivateLink, a CloudFront origin access approach, or a small load balancer that can sit in a public subnet without exposing the EC2 directly.

Another limitation is that failover to the backup EC2 instance is manual. It works perfectly for the presentation, but a more advanced setup would include automatic scaling, automatic failover, or a load-balanced approach that can distribute traffic intelligently without user involvement.

We also wanted to implement IP-restricted security groups that only accept CloudFront and Fastly traffic. The problem is that both providers rotate their IP ranges frequently, and relying on static IP lists can break the entire system if their networks change. A more sophisticated solution might involve periodic API polling combined with automation that updates security groups in real time.

The architecture does not include deep observability features such as logs, traces, or distributed metrics across the CDNs. Adding structured dashboards, alarms for more metrics, and error-budget tracking would make the system more production-grade.

### **Prototype #3 – ECS-based architecture:**

- Instead of running everything on EC2, the goal is to migrate the entire workflow into ECS with automated scaling.
- The plan is to run a small cluster of NGINX containers acting as reverse proxies, and have them route traffic to multiple backend containers. If one backend container gets overloaded or fails, ECS would automatically create a new one using the same backend image.
- Each backend container would also have its own S3 bucket for quick data retrieval and to avoid depending on local EC2 storage.
- This approach would be cheaper, recover faster, and eliminate the single-instance bottleneck entirely.
- Other improvements include revisiting a proper load balancer setup once AWS account restrictions are lifted, restoring a private–public subnet VPC model for stronger isolation, and expanding DoS protections with CDN-level rate-limiting.
- Route 53 is already handling the automatic failover between CloudFront and Fastly, but there's still room to enhance the health-check logic and potentially add additional CDNs to make the system even more fault-tolerant.

Lastly, a problem came up just before the official presentation. We realized that enabling a fully automatic failover for EC2s with CloudFront would have required a \$1,000 per month subscription. This conflicted with my existing

backup EC2 setup, because CloudFront only allows one origin to be active at a time. As a solution, We removed the automatic behavior and documented the procedure so that if the production EC2 ever goes down, I can quickly set up the backup EC2 as a functioning origin for both CloudFront and Fastly.

## 8.) References

### Autoinstaller/Subiquity:

1. Ubuntu, "Subiquity 24.04.1 Released to the Stable Channel," *Ubuntu Discourse*, 2024. [Online]. Available:  
<https://discourse.ubuntu.com/t/subiquity-24-04-1-has-been-released-to-the-stable-channel/44493>
2. SSH Communications Security, "SSH Command," *SSH Academy*. [Online]. Available:  
<https://www.ssh.com/academy/ssh/command>
3. Ubuntu, "Firewall Configuration," *Ubuntu Documentation*. [Online]. Available:  
<https://documentation.ubuntu.com/server/how-to/security/firewalls/>
4. Canonical, "Autoinstall Reference," *Subiquity Documentation*. [Online]. Available:  
<https://canonical-subiquity.readthedocs-hosted.com/en/latest/reference/autoinstall-reference.html>
5. SSH Communications Security, "SSH Keygen," *SSH Academy*. [Online]. Available:  
<https://www.ssh.com/academy/ssh/keygen>
6. Canonical, "Autoinstall Quick Start," *Subiquity Documentation*. [Online]. Available:  
<https://canonical-subiquity.readthedocs-hosted.com/en/latest/howto/autoinstall-quickstart.html>

### Cloudflared:

7. Cloudflare, "Arbitrary TCP Authentication Using Cloudflared," *Cloudflare One Documentation*. [Online]. Available:

<https://developers.cloudflare.com/cloudflare-one/access-controls/applications/non-http/cloudflared-authentication/arbitrary-tcp/>

## AWS:

8. Amazon Web Services, "AWS Documentation," 2024. [Online]. Available:  
<https://docs.aws.amazon.com/>
9. Amazon Web Services, "Amazon S3 Documentation." [Online]. Available:  
<https://docs.aws.amazon.com/s3/>
10. Amazon Web Services, "VPC Documentation." [Online]. Available:  
<https://docs.aws.amazon.com/vpc/>
11. Amazon Web Services, "CloudFront Documentation." [Online]. Available:  
<https://docs.aws.amazon.com/cloudfront/>
12. Amazon Web Services, "TLS Inspection Configurations," *AWS Network Firewall Docs*. [Online]. Available:  
<https://docs.aws.amazon.com/network-firewall/latest/developerguide/tls-inspection-configurations.html>
13. Amazon Web Services, "Amazon EC2 Documentation." [Online]. Available:  
<https://docs.aws.amazon.com/ec2/>
14. Amazon Web Services, "Amazon Route 53 Developer Guide." [Online]. Available:  
<https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/Welcome.html>
15. Amazon Web Services, "CloudFront Functions," *Developer Guide*. [Online]. Available:  
<https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/cloudfront-functions.html>
16. Amazon Web Services, "Amazon Linux 2023 Overview," *AL2023 User Guide*. [Online]. Available:  
<https://docs.aws.amazon.com/linux/al2023/ug/what-is-amazon-linux.html>
17. Amazon Web Services, "Python on Amazon Linux 2023," *AL2023 User Guide*. [Online]. Available:  
<https://docs.aws.amazon.com/linux/al2023/ug/python.html>

18. Amazon Web Services, "AWS CLI User Guide." [Online]. Available:  
<https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-welcome.html>
19. Amazon Web Services, "Amazon S3 Service Authorization Reference." [Online]. Available:  
[https://docs.aws.amazon.com/service-authorization/latest/reference/list\\_amazons3.html](https://docs.aws.amazon.com/service-authorization/latest/reference/list_amazons3.html)
20. Amazon Web Services, "NAT Gateway Documentation." [Online]. Available:  
<https://docs.aws.amazon.com/vpc/latest/userguide/nat-gateway-working-with.html>
21. Amazon Web Services, "Cross-Account Access via IAM Roles." [Online]. Available:  
[https://docs.aws.amazon.com/IAM/latest/UserGuide/tutorial\\_cross-account-with-roles.html](https://docs.aws.amazon.com/IAM/latest/UserGuide/tutorial_cross-account-with-roles.html)
22. Amazon Web Services, "S3 Buckets Cross-Account Access." [Online]. Available:  
<https://docs.aws.amazon.com/res/latest/ug/S3-buckets-cross-account-access.html>
23. Amazon Web Services, "AWS CloudFormation Overview." [Online]. Available:  
<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cloudformation-overview.html>
24. Amazon Web Services, "IAM Managed Policies Tutorial." [Online]. Available:  
[https://docs.aws.amazon.com/IAM/latest/UserGuide/tutorial\\_managed-policies.html](https://docs.aws.amazon.com/IAM/latest/UserGuide/tutorial_managed-policies.html)
25. Amazon Web Services, "Install SSM Plugin on Windows." [Online]. Available:  
<https://docs.aws.amazon.com/systems-manager/latest/userguide/install-plugin-windows.html>

#### Fastly:

26. Fastly, "Fastly Documentation Guides." [Online]. Available:  
<https://www.fastly.com/documentation/guides/>

#### Youtube Videos:

27. [https://youtu.be/ol7Vvg4-RyA?si=sL5x\\_PSCpFA4tBN3](https://youtu.be/ol7Vvg4-RyA?si=sL5x_PSCpFA4tBN3)
28. <https://youtu.be/eaicwmnSdCs>
29. <https://youtu.be/9t9Mp0BGnyl>

### Books:

30. M. Wittig and A. Wittig, *Amazon Web Services in Action*, Manning Publications.
31. B. Holder et al., *AWS Certified Solutions Architect Study Guide*, Wiley.

### Other:

32. Microsoft, "Install WSL," *Microsoft Learn*. [Online]. Available:

<https://learn.microsoft.com/en-us/windows/wsl/install>

## Appendices (Evidence & How-To)

The appendices are organized through video tutorials hosted on the project website, alongside GitHub repositories that include their own Wikis with step-by-step installation guides for the infrastructure. Each repository also contains detailed README files explaining how the RESTful Flask backend communicates with other services (S3), providing clear guidance on both setup and functionality.

### Youtube tutorials of templates link:

- [https://youtu.be/QGrkrmuJgJI?si=piJAYpa8kmO\\_wNvY](https://youtu.be/QGrkrmuJgJI?si=piJAYpa8kmO_wNvY)
- <https://youtu.be/9x3ZkkiX2xU?si=AbMXH90zzaH6FsVZ>
- <https://youtu.be/-jZqG2jlkC0?si=3RqvYRX2MMUSzG-O>
- <https://youtu.be/4KiVoIDbvlU>

Instances (2) Info

Refresh

Connect

Instance state

Actions

Launch instances

Find Instance by attribute or tag (case-sensitive)

All states

< 1 >

Settings

| <input type="checkbox"/> | Name           | Instance ID         | Instance state | Instance type | Status check      | Alarm status  | Availability Zone | Pub  |
|--------------------------|----------------|---------------------|----------------|---------------|-------------------|---------------|-------------------|------|
| <input type="checkbox"/> | PRODUCTION-... | i-07f2b320fb64398d6 | Stopped        | t3.small      | -                 | View alarms + | us-east-1a        | ec2- |
| <input type="checkbox"/> | BACKUP-ORIGIN  | i-0cdeb8dd3c982578f | Running        | t3.micro      | 3/3 checks passed | View alarms + | us-east-1a        | ec2- |

Route 53 > Hosted zones > infrahub.me

Route 53

Dashboard

Hosted zones

Health checks

Profiles

Global Resolver

Global resolvers

VPC Resolver

VPCs

Inbound endpoints

Outbound endpoints

Rules

Query logging

Outposts

Domains

Registered domains

Requests

Public infrahub.me Info

Delete zone

Test record

Configure query logging

Hosted zone details

Edit hosted zone

Records (13)

Accelerated recovery

DNSSEC signing

Hosted zone

Records (13) Info

Refresh

Delete record

Import zone file

Create record

Automatic mode is the current search behavior optimized for best filter results. [To change modes go to settings.](#)

Filter records by property or value

Type

Routing p...

Alias

< 1 >

Settings

| <input type="checkbox"/> | Record ...  | Type | Routin... | Differ... | Alias |
|--------------------------|-------------|------|-----------|-----------|-------|
| <input type="checkbox"/> | infrahub.me | A    | Failover  | Primary   | Yes   |

CloudFront > Distributions > ERRRECT4CW6WJ

CloudFront

Distributions

Policies

Functions

Static IPs

VPC origins

SaaS

Multi-tenant distributions

Distribution tenants

Telemetry

Monitoring

Alarms

Logs

Reports & analytics

Cache statistics

Popular objects

Top referrers

CDN-Capstone Free plan

Details

Distribution domain name

 d2htztcybfxeo1.cloudfront.net

Billing

Free plan (\$0/month)

Manage plan

General

Security

Origins

Behaviors

Error pages

Invalid requests

Settings

Name

CDN-Capstone 

Description

This is the primary CDN in route 53 for capstone: infrahub.me

Price class

Use all edge locations (best performance)

Alternate domain names

[infrahub.me](#) 

[afhqirhiowdioqfegoifjwoifad](#)

[dadadadadadadadaf.infra](#)

Route domains to CloudFront

Custom SSL certificate

 [infrahub.me](#) 

Amazon S3

Amazon S3

Buckets

General purpose buckets

Directory buckets

Table buckets

Vector buckets New

Access management and security

Access Points

Access Points for FSx

Access Grants

IAM Access Analyzer

Storage management and insights

Storage Lens

Batch Operations

Account and organization settings

General purpose buckets All AWS Regions

Directory buckets

General purpose buckets (2) Info



 Copy ARN

Empty

Delete

Create bucket

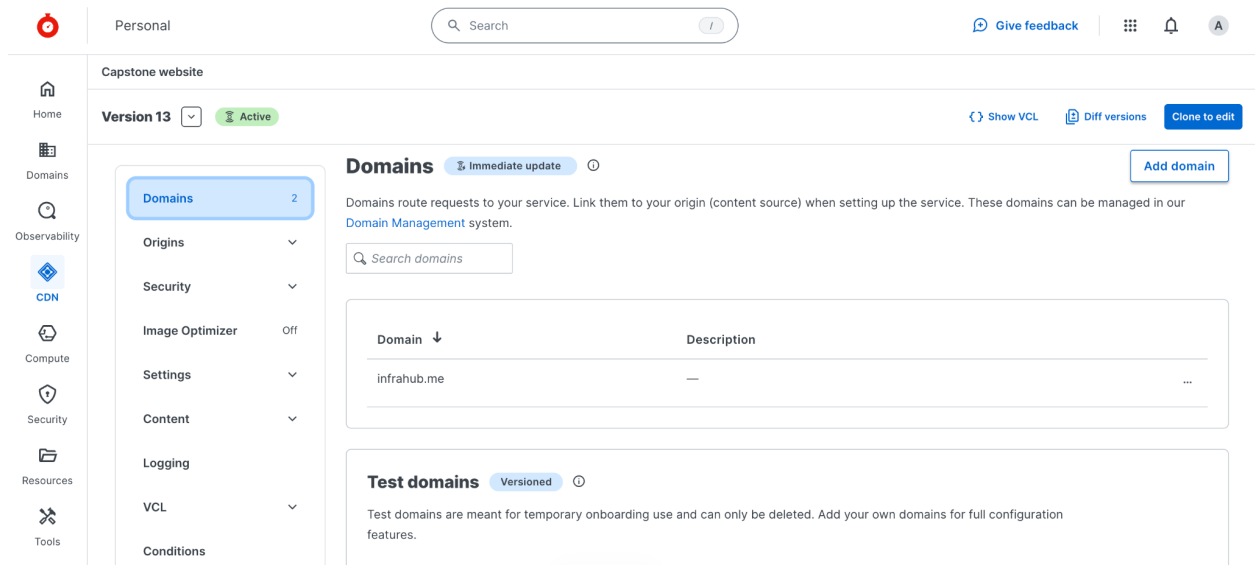
Buckets are containers for data stored in S3.

 Find buckets by name

< 1 >



| Name                                                                                                                    | AWS Region                      | Creation date                           |
|-------------------------------------------------------------------------------------------------------------------------|---------------------------------|-----------------------------------------|
|  <a href="#">capstone-ajaimes</a>    | US East (N. Virginia) us-east-1 | October 8, 2025, 22:26:59 (UTC-04:00)   |
|  <a href="#">cdn-capstone-health</a> | US East (N. Virginia) us-east-1 | November 26, 2025, 10:27:35 (UTC-05:00) |



## Discipline Specific Depth (O6)

### Architecture & Integration:

The system is structured so the entire setup avoids a single point of failure while still staying simple to operate. Route 53 serves as the central control layer since it hosts the DNS and manages how traffic flows across the infrastructure. It uses multiple A and AAAA records that point to both CDNs—CloudFront and Fastly—and each record includes an attached health check. This allows Route 53 to automatically redirect clients to the healthy CDN if one becomes unavailable. Route 53 also includes the required CNAME records for certificate validation and handles all TLS and SSL associations so the certificates remain valid regardless of which CDN is serving traffic.

CloudFront and Fastly operate independently. Each CDN has its own origin configuration that points directly to S3, so neither depends on the other to deliver content. S3 stores all static assets including HTML, CSS, and media files, allowing the backend to remain stateless and reducing the load on EC2.

Behind the CDN layer, the architecture includes two EC2 instances: the primary production instance and a secondary standby instance. Both instances run the same container stack consisting of a Flask application inside Docker behind an NGINX reverse proxy. The standby EC2 is not running continuously and is powered on only if the primary instance

fails. Since all application code runs inside containers and static files live in S3, the standby EC2 can take over immediately without requiring any synchronization.

The overall flow is straightforward. Traffic enters through Route 53, is routed to whichever CDN is healthy, the CDN retrieves static content from S3, and dynamic requests are sent to the EC2 instance that is currently active. This structure keeps the system resilient while maintaining a clean and manageable architecture.

### DevOps & SRE:

For DevOps and SRE, everything is automated through GitHub Actions using Docker Buildx and a pair of workflows that handle both backend and reverse-proxy image creation. The pipelines build, validate and publish the containers to GHCR so the EC2 instances always pull fresh images without manual work. The IaC side of the project lives inside the Ansible templates that IT admins can download directly from the web portal. These templates automatically provision development or testing environments, install dependencies and create reproducible developer sandboxes. Observability comes from CloudWatch alarms on CPU usage and health checks managed by Route 53. The system effectively has its own SLO by design, which is to remain available even when one provider breaks. The entire setup follows an error-budget type philosophy where the system should continue functioning during partial outages rather than waiting for full failure.

### Operations & Security:

Operationally and from a security standpoint, the environment is hardened to behave like a public-facing service without actually exposing the EC2 instances. Only HTTP and HTTPS are allowed in the security group, and NGINX immediately rejects any request that does not include the required CloudFront or Fastly headers. Even if someone finds the EC2 public IP, the instance will return a 404 unless the request came through an approved CDN. The containerized architecture gives an additional layer of protection because the proxy and backend are isolated from the host. Backup and restore are handled through AMI snapshots so we can re-create the entire environment instantly, and both EC2 instances refresh containers automatically using systemd on every reboot to avoid configuration drift. Change management happens through GitHub Actions since every update must pass through the CI pipeline before it can be deployed. Configuration hardening is done through NGINX rules, blocked direct access, restricted AWS IAM roles, and the use of AWS Systems Manager instead of SSH. All of this keeps the platform stable for end users and safe for the administrators running it.

