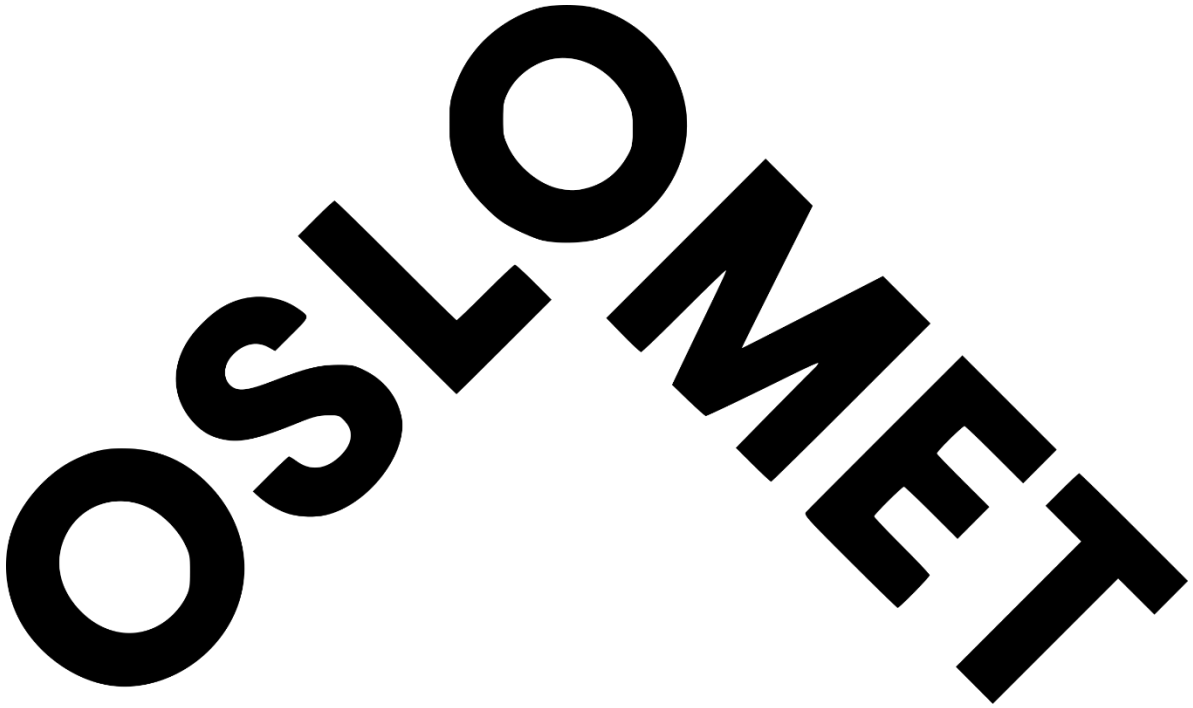
Contents


OSLOMET

ACIT- 4410 Agile Service Delivery and Developer Operations

Project Canario

1. Introduction

In today's digital landscape, companies depend on being able to always deliver stable and high-quality websites and applications. Even a short period of downtime can lead to financial loss, frustrated customers, and unnecessary pressure on the organization. A single overlooked error or a poorly handled deployment can quickly turn into several hours of outage, where users leave the site, revenue drops, and the team struggles to identify what went wrong. It is assumed that situations like this usually happen because the delivery process is outdated, automation is limited, and the infrastructure is not built for reliability. As a result, the company ends up with more risk, slower workflows, and far less control over its own services.

This is the situation facing the Canario's company, which specializes in building and hosting websites for its customers. They struggle with maintaining control over the issues that appear during development and when running customer websites in production. Its assumed that over time, these recurring failures have created instability and frustration across the organizations. The CEO eventually became concerned about how often these problems occurred and the impact they had on both customers and internal workflow. As a result, the company decided that it was necessary to identify the root causes behind these failures and understand what changes were needed to achieve a more reliable and structured way of working

2. Case description

In this case, the focus is to create a prototype solution that will help the Canario company reduce downtime as much as possible. It is assumed an agile approach, in this case based on DevOps principles, will give the company a more structured way of handling incidents and maintaining higher quality and efficiency. The goal is to demonstrate how and why this workflow can improve long-term reliability, and how developers can make changes without risking crashes or bringing down the website servers.

3. Case relevance

The challenges in the Canario case are not unique, and they represent a normal type of issues that are common in organizations. It is assumed that many organizations lack the designing visuals and the workflow and clear operational standards. It's also assumed that the core problem is not the technical incompetence but rather the how the standard workflow and standards processes automation and traceability. When deployments depend on manual routines and when no version control or testing workflow exists, even small changes can trigger outages, lost revenue and unsatisfied customers. Canario's current situation reflects this kind of pattern.

In larger industries, these patterns are widely recognized and represent a shift toward operational excellence as a competitive factor. Modern and successful organizations rely heavily on continuous delivery, high availability, and predictable release cycles to maintain customer trust and reduce operational risk. It's assumed that companies cannot maintain stability if they experience frequent downtime, as this results in higher operational costs and slower development cycles. The Canario case therefore serves as an example of how unstable manual processes can be and why it should be transformed through the introduction of DevOps practices such as automation, CI/CD pipelines, containerization, and orchestration.

For Canario, it's assumed that adopting this workflow is not simply a technical upgrade but a long-term strategic change. With a structured pipeline, controlled deployments, and automated orchestration, the company can guarantee higher uptime, reduce operational fragility, and scale more efficiently as new customers are added. In this way, the case highlights a widespread industry problem and demonstrates how a modern DevOps-based prototype can systematically address these issues through standardization and automation.

4. Problem analysis

After analysing the problems, the important methods for resolving them can be identified, as well as why they are necessary. First, one of the main issues is that each website is tied to a single server that cannot be moved, and if that server crashes, everything crashes along with it.

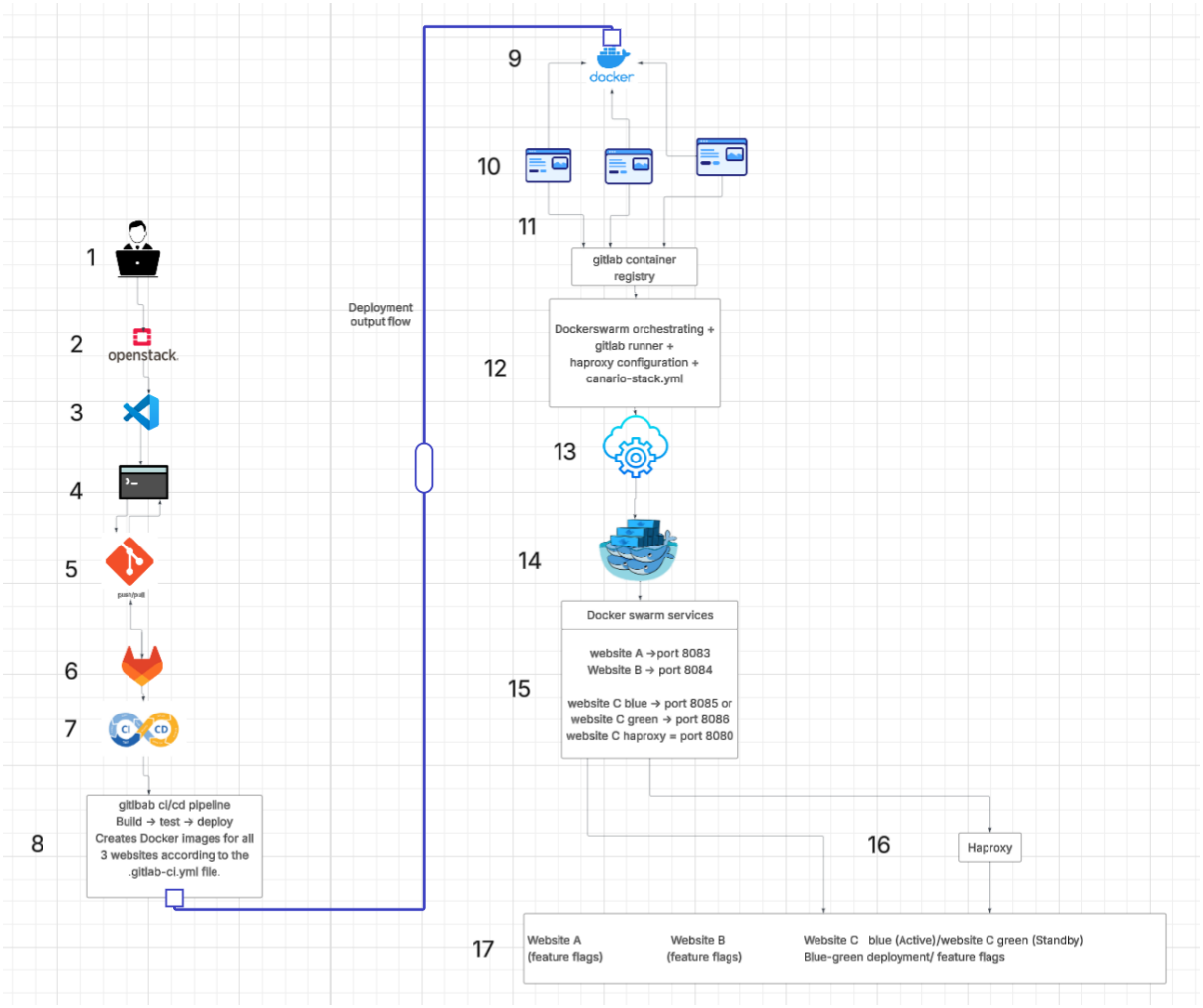
The best solution as assumed is to host the websites on a single server environment where multiple sites and a backup can run together, with the backup relying on a DevOps approach in which a GitLab server is used for continuous integration and continuous deployment. This approach requires creating a repository where all work is pushed to the GitLab server, which is beneficial because any lost work can always be retrieved by pulling it back to the local machine.

It's assumed that this DevOps methodology, the process of testing and deploying becomes much easier due to the systems implemented in GitLab. One of the most important advantages is that there is no need to shut down the website or websites to change the code, and the site continues running while the developer updates and fixes the code locally. When the changes are ready, they pass through the CI/CD pipeline and are deployed while the website remains operational. If a deployment fails, a HAProxy setup can maintain service availability by falling back to the previous version of the website, a process known as blue-green deployment.

Overall, the assumption of the main purpose of this method is to provide greater control over the workflow and reduce downtime. This approach helps save costs, makes the website more stable and faster, and supports a more structured and logical way of working.

5. Solution design

The current overall design:



(figure 1 created from lucidchart)

This diagram shows the full workflow of the Canario prototype from development to deployment.

Step 1-4 illustrate how the developer works locally in Visual Studio Code and connects to the cloud environment through OpenStack to access the server. The code is tested locally before being pushed to GitLab (steps 5-6).

At step 7-8, the GitLab CI/CD pipeline begins. The pipeline follows the .gitlab-ci.yml file and automatically builds, tests, and deploys all three websites. When the build stage finishes, Docker images are created and uploaded to the GitLab Container Registry (steps 9-11).

Next, Docker Swarm takes over (step 12). Swarm orchestrates the new images, updates the running services, removes old containers, and applies the configuration from canario-stack.yml.

Step 13-14 show Swarm deploying the updated services for Website A and B, and the blue/green versions of Website C. Website C is routed through HAProxy (step 16), which handles traffic and switches between blue (active) and green (standby) without downtime.

Finally, step 17 shows the websites running with feature flags enabled, demonstrating a complete automated delivery pipeline with orchestration, routing, and safe deployment.

6. Prototype and implementation

When the project was initiated, it had to go through a list of requirements for implementation, as it was important to identify what was necessary so the prototype could focus on automation and robustness. It's assumed that this step was essential for understanding why certain components were needed and what had to be implemented in the prototype.

6.1 GitLab Repository Structure

The GitLab repository folder is called project-canario and consists of three websites, the GitLab YAML file, the README file, the Canario-stack file, and the HAProxy file. These files form the roots of the project folder, and each of them serves an important role.

Starting with the core components, the GitLab YAML file is one of the most important tools, as it enables the entire system to work automatically. The code inside the file contains the instructions that define how the system should operate, including starting the build, test, and deploy phases. It functions as the connecting point between developers and operations. Everything that happens in the pipeline must be included in this file, and it can be compared to giving a robot a step-by-step instruction list that it follows it exactly step by step. If something is wrong, the CI/CD pipeline notifies the user at the correct stage.

Next is the Canario-stack.yml file, which is the Docker Swarm configuration file. This file orchestrates and manages the Docker containers as required. For example, it can automatically delete old Docker images and create new ones without manual work. This becomes especially important when handling many containers simultaneously, as it prevents crashes and helps maintain a stable development environment. Through this file, Docker Swarm ensures that the system remains efficient and that container-related failures occur far less frequently.

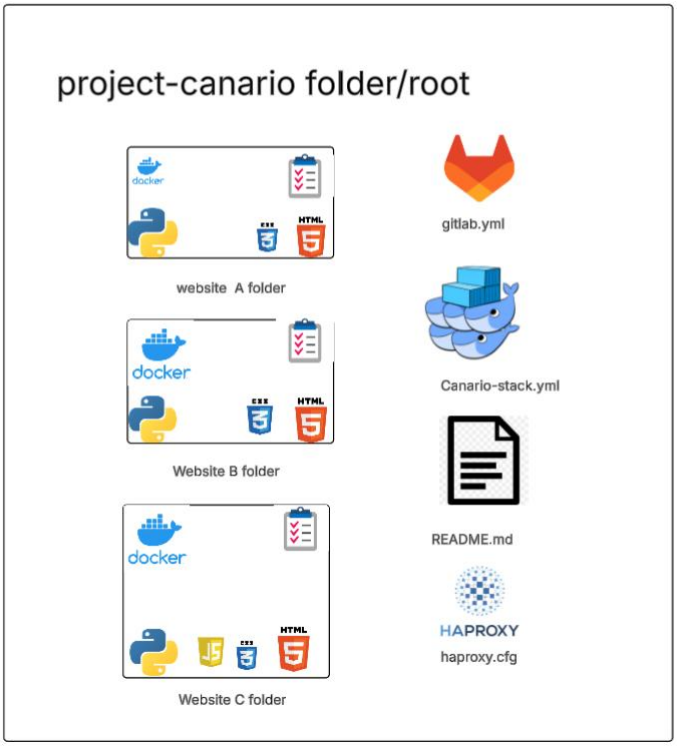
Another essential file is the haproxy.cfg file, which controls how traffic flows within the system. When updating the website or application, this file makes it possible to use blue-green deployment. This means that the server requiring an update can be taken offline, while another server using the same ports (common ports) continues to run. Once the update is ready, the server can be activated again, and the new changes will appear. If the updated version crashes, HAProxy can route traffic to the older, stable version. In this project, this technique is demonstrated specifically in Website C. It is an effective method to avoid losing customers and serves as a backup plan to prevent downtime.

The repository also contains a README file, which provides instructions explaining what happens in each file. This documentation is important for ensuring clarity and maintaining an overview of the system's components.

The three websites consist of HTML and CSS styling, and Website C additionally includes JavaScript. Everything displayed on each site is handled inside the index.html file located in the template folder. Inside each website folder, there is also a requirements.txt file, which is necessary for running all features on the site, such as feature flags or enabling Docker functionality. Each website includes a Dockerfile and a main.py file in its app folder. These

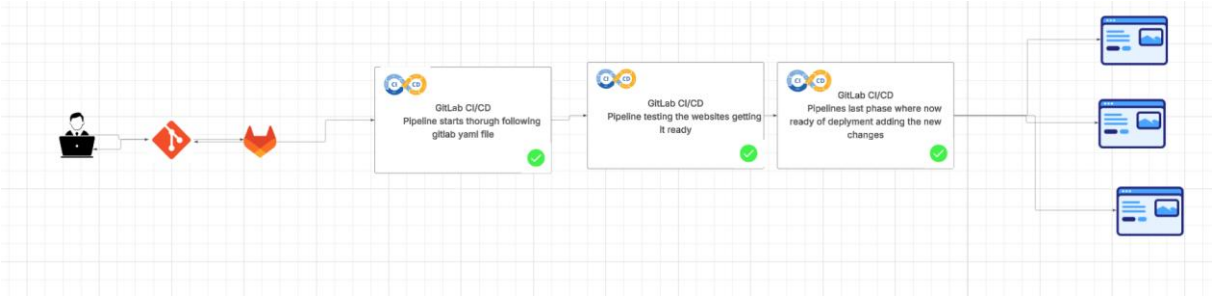
components are essential for making the website features work correctly and for communicating with the feature flag functions used in the project.

The structure appears in GitLab in the following way:



(figure 2 created using lucid chart)

6.2 CI/CD Pipeline



(figure 3 created using lucid chart)

The CI/CD pipeline is the most important phase in the project. This stage is activated after pushing the work that has been completed, for example fixing the YAML file or changing the appearance of the website. If the changes work correctly on the local machine, the push becomes the essential step that sends the code into the CI/CD pipeline.

When the code is pushed, it is processed directly by the pipeline, which is managed by the GitLab Runner. The runner follows the instructions written in the GitLab YAML file. The first step is the build phase, where everything required for the application is prepared. When the build succeeds, the pipeline continues to the testing phase. If the test results are successful, the pipeline moves to the final phase called deploy. In this phase the application is ready to be

launched, and it becomes possible to see how it performs. After deployment the application is released publicly. This sequence describes the flow of the CI/CD pipeline.

Failures can occur during this process. If the pipeline stops or fails to progress through a phase, it indicates that something is wrong. It then becomes necessary to identify the exact point of failure. For example, if the failure happens in the build phase, the issue might be caused by a conflict with another Docker container, especially if Docker Swarm has not been implemented yet.

Understanding how the pipeline works is important because it reduces the time needed for troubleshooting. Instead of spending many hours investigating the issue manually, the pipeline shows exactly where the failure happened, which creates a faster and more efficient problem-solving process.

6.3 Docker Swarm as Orchestrator

Before Docker Swarm was introduced, the pipelines often encountered problems caused by multiple Docker containers running at the same time. Older containers had to be deleted manually, which slowed down progress and created unnecessary delays. This manual work also contributed to repeated failures in the CI/CD pipeline.

Implementing Docker Swarm on the server addressed these issues. Docker Swarm does not only manage several containers but also follows specific instructions that automate important tasks. It can remove old containers, create new ones, and rebuild the required environments without manual intervention. As a result, the number of pipeline failures decreased, and the workflow became more consistent.

Docker Swarm is an effective choice for managing large numbers of containers. It also supports health checks, which provide clear insight into which containers are running correctly and which are not. This contributes to a more stable and predictable operational environment.

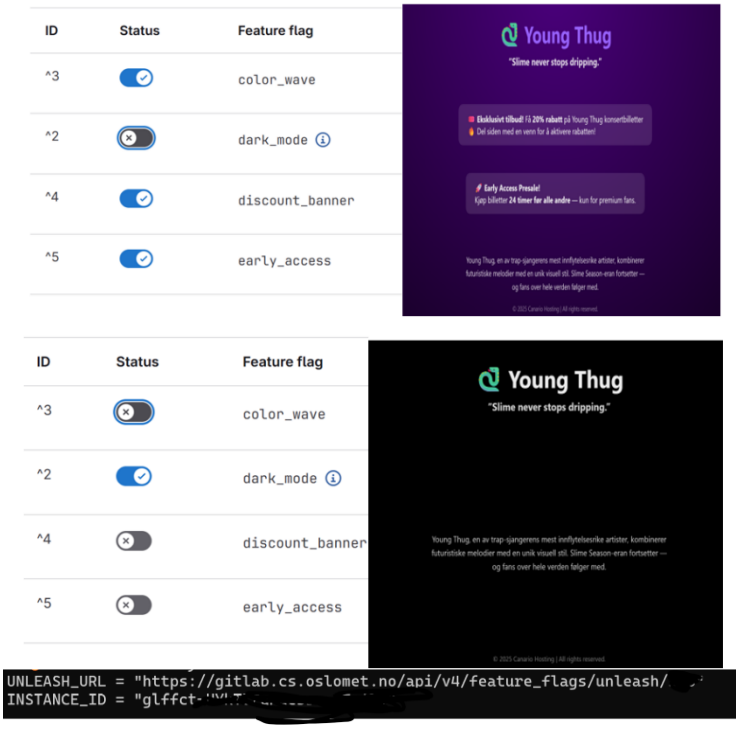
6.4 Blue- Green Deployment/haproxy

Blue green deployment is demonstrated in website C through the use of HAProxy. HAProxy is configured to expose a common port, 8080, which acts as the public entry point. When visiting this port, traffic is routed either to port 8085, which represents the blue version of the website and serves as the primary environment, or to port 8086, which represents the green version.

If the server is running the blue version and it suddenlty requires an update, it can be taken offline without shutting down the entire service. In this case, traffic is automatically directed to the green version on port 8086. The green version remains active until the new release on port 8085 is fully ready. Once the updated version is verified, HAProxy switches traffic back to port 8085 automatically.

This ensures that customers can continue using the website without interruption while updates are deployed.

6.5 Feature Flags via GitLab Unleash



(figure 4 and 5 from the prototype project folder)

Feature flags are implemented through GitLab Unleash, where each flag can be toggled on or off to immediately enable or disable specific feature on the website. This allows visual changes or features to be activated without redeploying the application through the CI/CD pipeline. The mechanism works because each feature flag provides an ID and token, which are configured inside the app folder where the main.py file is located. The Unleash client uses these values to check the status of each flag, while the website's index.html includes the necessary flag references to conditionally display or hide features.

In this project, four feature flags were created. One flag, "dark mode", applies globally across all websites. Website A includes an additional unique flag that controls a visual styling effect called color wave. Websites A and B both include two commercial flags that demonstrate elements such as promotions or discount banners. These flags allow changes to the user interface without requiring code modifications or deployment.

6.6 Website Themes (A, B, C)

The prototype includes three different websites which are Website A, Website B and Website C. These demonstrate how the delivery platform supports multiple independent customer environments. Each site provides its own visual style, but the main purpose is to function as a controlled test case for validating the infrastructure, the workflow and the DevOps capabilities implemented in the solution.

Website A and Website B represent the baseline scenario. They are simple websites used to verify containerization, continuous integration and continuous deployment automation, and the connection to the GitLab registry. They are also used to test several feature flags such as dark mode and promotional elements like discount pop up messages. Website A has one

additional visual feature flag that controls whether the site name appears in a single colour or in a wave of colours depending on the configuration that is active.

Website C contains the same general features but is mainly used to demonstrate the more advanced parts of the system. This website is where blue green deployment is shown, together with traffic control through HAProxy. It can also run two versions of the website on different ports while presenting everything through one single public port. This makes it possible to demonstrate controlled failover, traffic switching and instant rollback without affecting the customers.

7. Transitional Path (How Canario Should Migrate)

When it comes to migrating Canario from its current unstructured workflow to the DevOps model, it is important to understand that this change cannot happen overnight. The transition must go through learning and adapting to its new way of working, as well as adapting to how the tools and technology are going to be used. Therefore, a roadmap outlines a phased and low risk approach that allows Canario to adapt to the new workflow model efficiently. During this process, it will go through the phases described on how to approach and adapt step by step.

7.1 Phase 1 version control and standardisation

The first important establishment should be using GitLab as the primary source for all the projects. The focus should be on placing every website project that is made on live systems. After that, the company should adapt to repository structure, branching practices, and coding conventions. The goal of phase 1 is mostly based on the cultural aspect, where developers must learn that all changes need to flow through Gitlab so they can have more control, remain traceable, and be easy to review during development.

7.2 Phase 2 the introduction to CI/CD pipelines

Once everything is completed in phase 1 and the code is centralised, it is assumed that the introduction of CI/CD pipelines can begin. This phase focuses on automating build, test, deploy, and packaging workflows. Developers should no longer need to deploy manually but instead rely on pipelines to automatically validate their work. The shift from manual testing to automated pipelines is the single most important step in reducing Canario’s operational instability.

7.3 Phase 3 containers and local reproducibility

After the pipelines are operational, all websites should be containerised using docker. Containerisation gives Canario a reproducible environment that eliminates the “it works on my machine” problem and separates application behaviour from the server state. Developers also gain the ability to run identical versions of the site locally.

7.4 Phase 4 the introducing of orchestration with docker swarm

With multiple containers in place, the focus in this phase is to gain control over the number of containers and manage them automatically, including handling deployments. Swarm

automates scaling and ensures a consistent container lifecycle, while also reducing manual operational overhead. This establishes the foundation for high availability and structured releases.

7.5 Phase 5 add HAProxy and blue-green deployments

Once Swarm is fully functioning and stable, and the developers understand how to use it, introducing HAProxy becomes necessary. HAProxy can be used to manage traffic and provide solutions for traffic handling, and one of the first features to implement is blue green deployment. This is a critical step that eliminates website downtime entirely. New versions can be deployed to a secondary environment and switched live only when they are verified. Rollback becomes instant and safe.

7.6 Phase 6 introducing feature flag

With deployment automated, the next step is introducing feature flags, where Canario can adapt to using feature flags through GitLab Unleash. This allows website configuration rollouts and experimentation without needing to modify the deployment, unless the feature flag itself is being created, where deployment is required only once. After that, the feature can simply be toggled on or off in the GitLab feature flags menu, which can also be accessed by designers or marketing teams without involving the developers.

7.7 Phase 7 organisational adoption and training

The tools and technology alone will not deliver operational excellence until the organisation evolves alongside them. Developers, designers, and managers will require training in DevOps practices, pipeline usage, debugging workflows, and versioned releases. A governance model should also be introduced.

7.8 Phase 8 full production migration

With the new platform validated and adopted, Canario can migrate each customer website into the new pipeline and orchestrated infrastructure. Over time, the company benefits from higher stability, fewer outages, and faster feedback loops. With this in place, operational risk is significantly reduced, allowing the organisation to focus more on development, improving quality, and reaching higher milestones both now and in the future.

8. ARC method

Throughout this project, it is important to understand that following a method capable of creating a robust and efficient workflow is necessary to recognise the value of how tools are used within the organisation. Each tool provides a specific function that is tied to certain principles, and these principles generate organisational value. This approach is referred to as the ARC method. The method is encouraged because it reduces the communication gap between technical and non-technical workers in the organisation and is highly beneficial for achieving excellence in their work. Examples of the ARC method used in the prototype include, for example:

Gitlab-runner is an important tool because its function is to use and run CI/CD pipeline to lower downtime which creates efficient work and have control of the code. which in this case showcases the value that is control and stability.

With Docker as the tool, its function is to isolate the application environment. The principle behind this is the need for controlled and reproducible testing, and the value it provides is the ability to deliver efficient releases without the risk of unintended reversals.

And Docker Swarm as the tool has the function of managing the containers. The principle behind it is to orchestrate the containers so the system becomes more automated, and the value this provides is improved operational stability.

Blue-green deployment as a tool has the function of shifting between web servers. Its principle is to eliminate downtime, and its value is preventing customer loss and enabling instant rollback.

Another tool is feature flags, where the function is to toggle features on or off when needed. Its principle is giving the application the right appearance or promotion at the right time, and its value is greater customer experience and a stronger relationship with customers or clients.

9 Principles of operational excellence

Operational excellence in this context is understood as a collection of underlying practices that help digital systems remain stable, adaptable, and predictable as they evolve. It provides for a evaluating how well the platform supports reliability and structured growth. These principles have evolved across several technologies eras but remain relevant today as way to help organizations redundancies, improve stability and work more predictably. the principles are in form of the foundation for evaluating modern systems.

9.1 Principles related to Canario prototype

Policy-based governance through important rules is reflected in how GitLab CI/CD, Dockerfiles, and stack configurations consistently enforce expected system behaviour.

appropriateness of resources can be seen in how Docker and Swarm handle the containers. They remove old ones, start new ones when needed, and make sure the system only uses the resources it requires.

Responsible empowerment is shown in how developers use pipelines and containers to deploy safely without risking downtime.

Automations as pipelines appears in the build-test-deploy flow, which replaces manual work and ensures predictable results.

multimodal automation appears through how GitLab Runner, Docker, Swarm, HAProxy and feature flags work together.

continuous mission metrics introduced through pipeline results and swarm service status, although full monitoring is noted as future work.

10. Evaluation of Prototype

The prototype shows how important modern DevOps delivery is, with it being highly advantageous for Canario. During the development, several technical challenges occurred. These issues were sometimes related to Docker container conflicts, Docker images not being properly pushed to GitLab, and occasional CI/CD pipeline failures. These problems appeared mostly during the early to middle phases of the project. Failures are an important part of the learning curve and contribute to progress in development.

One of the most beneficial aspects was the introduction of Docker Swarm, which demonstrated automated orchestration. Before this, Docker containers would often crash and required manual cleanup. With Docker Swarm, these processes happened automatically, reducing such failures. This also made deployments easier and reduced the time spent on unsuccessful deployment attempts, which highly contributes to operational excellence.

The introduction of HAProxy and a working blue and green deployment reduced risk and showed how to manage traffic and switch between two servers without downtime. This demonstrates the value of maintaining control over releases. The ability to intentionally shut down one server and watch the system automatically transition to another while still satisfying customers is important and reflects the type of solution Canario needs.

During the creation of the prototype, feature flags also proved to be important in validating its flexibility at runtime. By allowing features to be toggled on and off without redeployment, the prototype demonstrated that Canario can use this mechanism more effectively than before. This aligns well with the DevOps method, as developers can quickly enable or disable features without exposing the company to unnecessary instability.

There are also limitations, as the prototype focuses mainly on automation, CI/CD pipelines, orchestration, routing and feature management. It does not include full monitoring or advanced multi-node scheduling. These would also be necessary in a real production environment, but the prototype is based on a single-node Swarm.

In total, the prototype successfully supports the Canario platform in its environment. Releases can be automated, downtime can be eliminated, and developers can work in a structured and predictable environment. This shift from manual processes to automated workflows is not only beneficial but also highly effective. Overall, the prototype serves as a realistic and valuable solution for the company.

11. Conclusion

The Canario case has illustrated how outdated manual delivery practices can create slow workflow progress, recurring outages and unnecessary financial risks. Through the development of the prototype, it has become clear that these challenges are not caused by poor technical skills, but rather by the lack of structured workflows, automation and traceability. The prototype demonstrates how DevOps methods implemented through CI/CD pipelines, together with tools such as blue-green deployment, automated orchestration and feature flags, can transform an unstable environment into a predictable and robust platform.

The solution allows Canario to move from a mechanical problem solving to controlled and repeatable processes. GitLab provides efficiency and better control, Docker creates reproducible environments, and Docker Swarm removes the need to manage containers manually. HAProxy manages traffic and enables safe releases and instant rollback, while feature flags give the organization the ability to adjust customer-facing functionality without redeployment. Together, these components create a foundation for operational excellence.

While the prototype focuses on automation and stability, it also highlights clear opportunities for future expansion, such as multi-node orchestration, monitoring and improved observability. Even in its current form, the solution provides a strong and realistic foundation for reducing downtime and increasing reliability. It assumes that DevOps is not only a technical upgrade but also a strategic shift toward controlled delivery. It is assumed that by adopting this workflow, Canario can move from outdated practices to a professional, automated and robust platform, positioning the company for future growth.

APPENDIX

- *Appendix A: gitlab-ci.yml*

ubuntu@canario-server:~/project-canario$ cat .gitlab-ci.yml

```
stages:
  - build
  - test
  - deploy

variables:
  DOCKER_HOST: "unix:///var/run/docker.sock"

# BUILD
build:
  stage: build
  script:
    - docker login omcr.cs.oslomet.no:443 -u "$CI_REGISTRY_USER" -p
"$CI_REGISTRY_PASSWORD"

    - docker build -t omcr.cs.oslomet.no:443/user/project-canario/website-a:latest ./website-a
    - docker build -t omcr.cs.oslomet.no:443/user/project-canario/website-b:latest ./website-b
    - docker build -t omcr.cs.oslomet.no:443/user/project-canario/website-c:latest ./website-c

    # all pushes now use correct registry URL
    - docker push omcr.cs.oslomet.no:443/user/project-canario/website-a:latest
    - docker push omcr.cs.oslomet.no:443/users/project-canario/website-b:latest
    - docker push omcr.cs.oslomet.no:443/users/project-canario/website-c:latest
  tags: [canario]

# TEST

test:
  stage: test
  script:
    - docker rm -f test-a test-b test-c || true

    # Test Website A
    - docker run -d --name test-a -p 8089:80 website-a:latest
    - sleep 8
    - curl -f http://localhost:8089

    # Test Website B
    - docker run -d --name test-b -p 8090:80 website-b:latest
    - sleep 8
    - curl -f http://localhost:8090

    # Test Website C
    - docker run -d --name test-c -p 8091:80 website-c:latest
    - sleep 8
```

```
    - curl -f http://localhost:8091

  tags: [canario]

# DEPLOY (USING DOCKER SWARM)
deploy:
  stage: deploy
  script:
    - echo "Updating Website A..."
    - docker service update --force
        --image omcr.cs.oslomet.no:443/users/project-canario/website-a:latest
        canario_website-a

    - echo "Updating Website B..."
    - docker service update --force
        --image omcr.cs.oslomet.no:443users/project-canario/website-b:latest
        canario_website-b

    - echo "Updating Website C (BLUE)..."
    - docker service update --force
        --image omcr.cs.oslomet.no:443users/project-canario/website-c:latest
        canario_website-c-blue

  tags: [canario]
  when: on_success

ubuntu@canario-server:~/project-canario$
```

- *Appendix B: canario-stack.yml*

```
ubuntu@canario-server:~/project-canario$ cat canario-stack.yml
version: "3.8"

services:
  website-a:
    image: omcr.cs.oslomet.no:443/user/project-canario/website-a:latest
    ports:
      - "8083:80"
    deploy:
      replicas: 1
      restart_policy:
        condition: any
      update_config:
        parallelism: 1
        delay: 3s
      rollback_config:
        parallelism: 1
    networks:
      - canario-net
```

```
 website-b:
  image: omcr.cs.oslomet.no:443/user/project-canario/website-b:latest
  ports:
    - "8084:80"
  deploy:
    replicas: 1
    restart_policy:
      condition: any
    update_config:
      parallelism: 1
      delay: 3s
    rollback_config:
      parallelism: 1
  networks:
    - canario-net

 website-c-blue:
  image: omcr.cs.oslomet.no:443/user/project-canario/website-c:blue
  ports:
    - "8085:80"
  deploy:
    replicas: 1
    restart_policy:
      condition: any
    update_config:
      parallelism: 1
      delay: 3s
    rollback_config:
      parallelism: 1
  networks:
    - canario-net

 # GREEN VERSION (inactive by default)
 website-c-green:
  image: omcr.cs.oslomet.no:443/ilcha8023/project-canario/website-c:green
  deploy:
    replicas: 0
    restart_policy:
      condition: any
  networks:
    - canario-net

 haproxy:
  image: haproxy:latest
  ports:
    - "8080:8080"
  volumes:
    - ./haproxy.cfg:/usr/local/etc/haproxy/haproxy.cfg
  deploy:
```

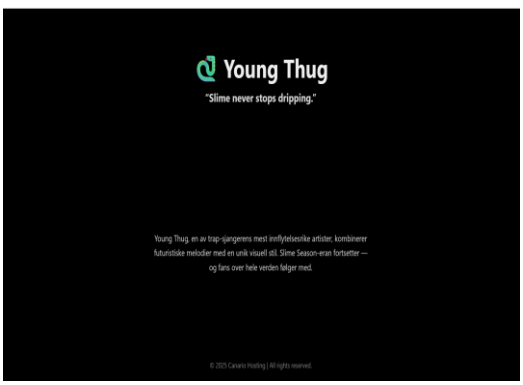
```
    replicas: 1
    placement:
      constraints: [node.role == manager]
  networks:
    - canario-net

networks:
  canario-net:
    driver: overlay
```

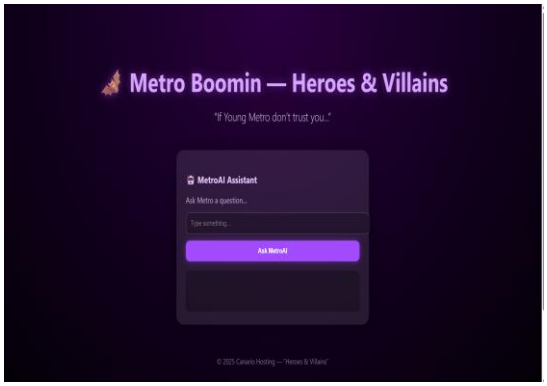
- *Appendix C: Docker service logs*

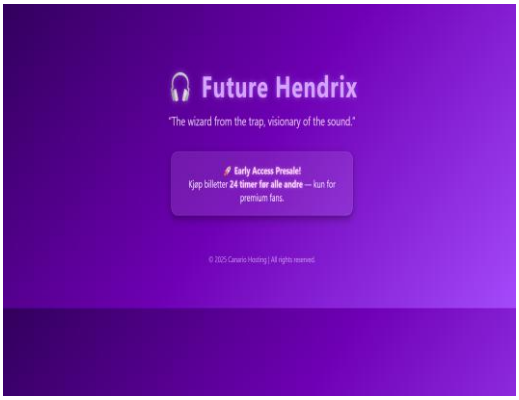
- *Appendix D: haproxy.cfg*

```
ubuntu@canario-server:~/project-canario$ cat haproxy.cfg
global
   log stdout format raw local0

defaults
   log     global
   mode    http
   timeout connect 5000ms
   timeout client  50000ms
   timeout server  50000ms
   option  httpchk

# ---------------------------
#   FRONTEND
# ---------------------------
frontend websites
   bind *:8080
   default_backend active_env

# ---------------------------
#   BACKEND MED AUTO FAILOVER
# ---------------------------
backend active_env
   option httpchk GET /
   server blue1 website-c-blue:80  check fall 3 rise 2
   server green1 website-c-green:80 check backup fall 3 rise 2
```

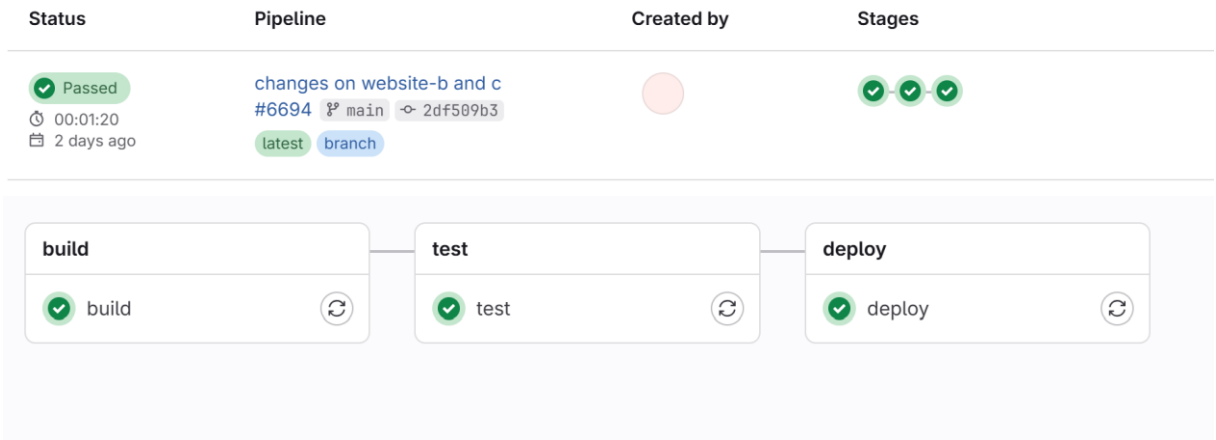
- *Appendix E: screenshots from the websites a,b, c*







- *Appendix F: screenshots fra pipeline success*



- *Appendix G: command history logs and blue green deployment test*

```
1452  docker service update --publish-rm 8085:80 canario_website-c-green 2>/dev/null
1453  docker service update --publish-add 8086:80 canario_website-c-green
1454  docker service ls
1455  docker service scale canario_website-c-blue=1
1456  docker service scale canario_website-c-green=0
1457  docker service scale canario_website-c-green=1
1458  ls
1459  cd templates
1460  ls
1461  nano index.html
1462  cd ..
1463  git add .
1464  git commit -m "Test: Update Blue version"
1465  git push origin main
1466  docker service ls
1467  docker service scale canario_website-c-blue=0
1468  docker service scale canario_website-c-green=1
1469  git add .
1470  git commit -m "Test: Update Blue version turning it off"
1471  git push origin main
1472  docker service scale canario_website-c-blue=1
1473  nano haproxy.cfg
1474  server blue1 canario_website-c-blue:80 check
1475  nano haproxy.cfg
1476  nano canario-stack.yml
1477  docker stack deploy -c canario-stack.yml canario
1478  nano canario-stack.yml
1479  docker stack deploy -c canario-stack.yml canario
1480  ls
1481  cd website-c
1482  ls
1483  cd templates
1484  nano index.html
1485  cd ..
1486  git add .
1487  git commit -m "Test: Update haproxy"
1488  git push origin main
1489  docker service ls
1490  docker service scale canario_website-c-green=1
1491  docker service ps canario_website-c-blue
1492  docker service scale canario_website-c-blue=0
1493  docker logs $(docker ps -q -f name=haproxy)
1494  nano haproxy.cfg
```

- *Appendix H: openstack cloud ports and VM instance*

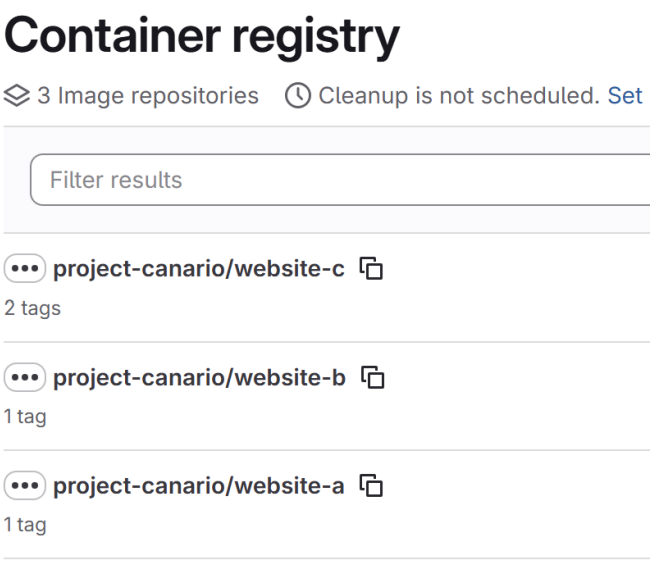
Displaying 7 items

Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone	Task	Power State	Age	Actions
canario-server	Ubuntu 24.04-LTS (Noble Numbat)	[redacted] 2001:700:740:a101:f816:3eff:fe57:af04	aem.2c2r.50g	manager	Active	nova	None	Running	1 month, 1 week	Create Snaps

Ingress	IPv4	TCP	8083	0.0.0.0/0	
Ingress	IPv4	TCP	8084	0.0.0.0/0	
Ingress	IPv4	TCP	8085	0.0.0.0/0	
Ingress	IPv4	TCP	8086	0.0.0.0/0	
Ingress	IPv4	TCP	8080	0.0.0.0/0	

- *Appendix I: gitlab container image registry*

```
ubuntu@canario-server:~$ sudo docker images
REPOSITORY                                                  TAG       IMAGE ID       CREATED         SIZE
omcr.cs.oslomet.no:443/[redacted]/project-canario/website-c  latest    37b3bbb885cd   2 days ago      159MB
omcr.cs.oslomet.no:443/[redacted]/project-canario/website-b  latest    80d1f1cf07f9   2 days ago      159MB
omcr.cs.oslomet.no:443/[redacted]/project-canario/website-a  latest    b47d76c38e9d   2 days ago      159MB
omcr.cs.oslomet.no:443/[redacted]/project-canario/website-c  <none>    0af7f399b096   2 days ago      159MB
omcr.cs.oslomet.no:443/[redacted]/project-canario/website-c  <none>    274d0051ff61   2 days ago      159MB
omcr.cs.oslomet.no:443/[redacted]/project-canario/website-c  blue      76567f79ce02   3 days ago      159MB
omcr.cs.oslomet.no:443/[redacted]/project-canario/website-c  green     76567f79ce02   3 days ago      159MB
omcr.cs.oslomet.no:443/[redacted]/project-canario/website-c  <none>    28c35785941b   3 days ago      159MB
```


Container registry
3 Image repositories
Cleanup is not scheduled. Set
Filter results
project-canario/website-c
2 tags
project-canario/website-b
1 tag
project-canario/website-a
1 tag

AI usage declaration

All code, configuration, and written content in this report was created manually by me. According to the course rules, ai tool were only used in allowed ways, such as technical support and for understanding concepts. AI was used to clarify error messages, debug issues in yaml, python and frontend files. And to understand why certain configuration failed.

in few cases, AI suggested small corrections like syntax fixes. But no complete code, text or design was generated by AI. This aligns with the course guideline that AI may support debugging but cannot replace the students own work. All project decisions, implementation and the full report writing where done by me.

The AI tool i used in this project was ChatGPT.