Final Project: DevOps Pipeline for Microservices-based E-commerce Platform

Andres F. Camacho

March 11, 2026

Phase 1: Repository Setup and Version Control

I created the four repositories for the ecommerce app and set up the main and develop branches using this code:

```
REPOS=("ecomm_front_end" "ecomm_product_service" "ecomm_order_service" "ecomm_database")

for repo in "${REPOS[@]}"; do
  (cd "$repo" && \
    git branch -m master main 2>/dev/null || git branch -m main main 2>/dev/null; \
    git checkout -b develop 2>/dev/null || git checkout develop; \
    git branch -a)
  echo ""
done
```

The repositories are hosted on GitHub and can be accessed at the following URLs:

- Frontend Service: https://github.com/anfelipecb/ecomm_front_end
- Product Service: https://github.com/anfelipecb/ecomm_product_service
- Order Service: https://github.com/anfelipecb/ecomm_order_service
- Database: https://github.com/anfelipecb/ecomm_database

I also added a shared library to the project root to be used by the Jenkins pipeline. The shared library is hosted on GitHub and can be accessed at the following URL: https://github.com/anfelipecb/ecomm_jenkins-shared-library

Repository Structure

All four repositories follow a consistent structure, like this:

```
<repo-name>/
|-- src/           # Source code
|-- config/        # Configuration files (env, schema, init scripts)
|-- tests/         # Test files
```

```
|-- .gitignore
|-- README.md
\-- package.json   # (Node/React services only)
```

- **ecomm_front_end**: React app with `src/`, `public/`, `config/`, `tests/`
- **ecomm_product_service**: Express REST API for product catalog
- **ecomm_order_service**: Express REST API for order management
- **ecomm_database**: PostgreSQL schema and init scripts in `config/init.sql`

Git Workflow and Branching Strategy

I use the **Git Flow** branching model to support collaborative development, scheduled releases, and clear separation between production and development code.

Permanent Branches

Branch	Purpose
main	Production-ready code only. Every commit is a tagged release.
develop	Integration branch for ongoing development. Reflects the next release.

Supporting Branches (Temporary)

Branch	Branches from	Merges into	When to use
feature/*	develop	develop	New functionality
release/*	develop	main + develop	Preparing a release (bug fixes only, no new features)
hotfix/*	main	main + develop	Urgent production fixes

Branch Lifecycle

1. **Feature workflow**: `develop` → `feature/xyz` → merge to `develop`
2. **Release workflow**: `develop` → `release/v1.0` → merge to `main` and `develop`
3. **Hotfix workflow**: `main` → `hotfix/critical-fix` → merge to `main` and `develop`

Screenshots

1. Initial Setup of Main and Develop Branches This is what I get when running `git branch -a` in each repository:

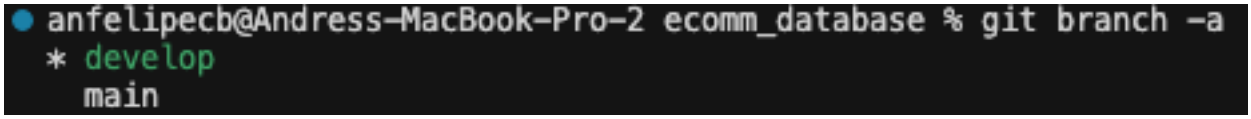


Figure 1: Git branch output

2. Complete Feature Development Workflow In Figures 2, 3, and 4, I show the steps from the creation of a feature branch to its merge into the develop branch.

```
anfelipecb@Andress-MacBook-Pro-2 ecomm_database % git add .
anfelipecb@Andress-MacBook-Pro-2 ecomm_database % git commit -m "Adding initial sctructure to the repo"
[feature/add_functionality 9f622b6] Adding initial sctructure to the repo
 4 files changed, 44 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 config/init.sql
 create mode 100644 src/.gitkeep
 create mode 100644 tests/.gitkeep
anfelipecb@Andress-MacBook-Pro-2 ecomm_database % git push
```

Figure 2: Feature branch pushed to remote

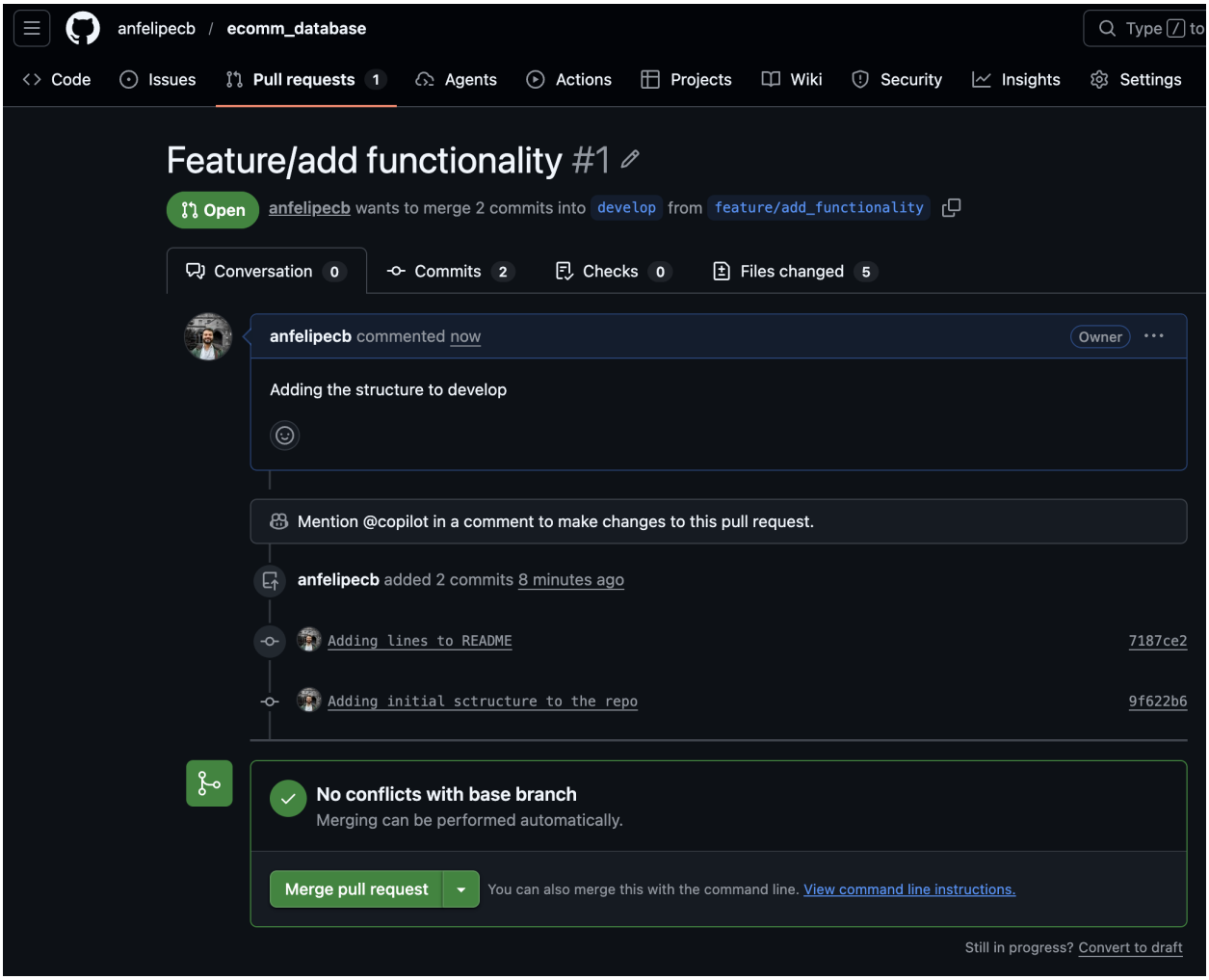


Figure 3: Opened Pull Request to Develop

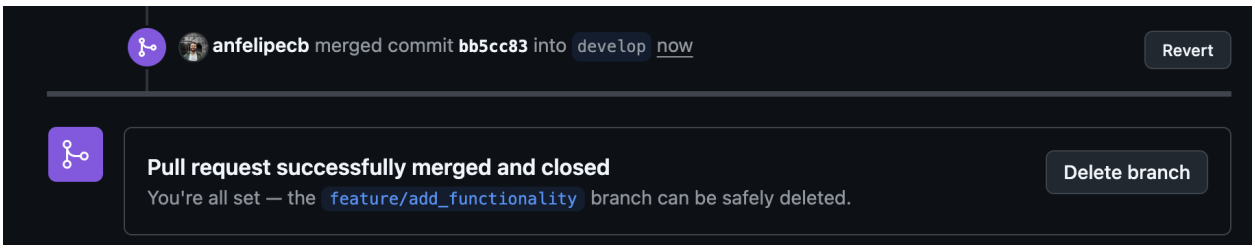


Figure 4: Feature branch Merged into Develop

Steps demonstrated: 1. After creating the feature branch from develop: `git checkout develop && git checkout -b feature/add_functionality` 2. Make changes, commit, push 3. Open Pull Request: feature → develop 4. Merge Pull Request

3. Complete Release Workflow Here in Figure 5, I show the creation and push of a release branch with a small bug fix.

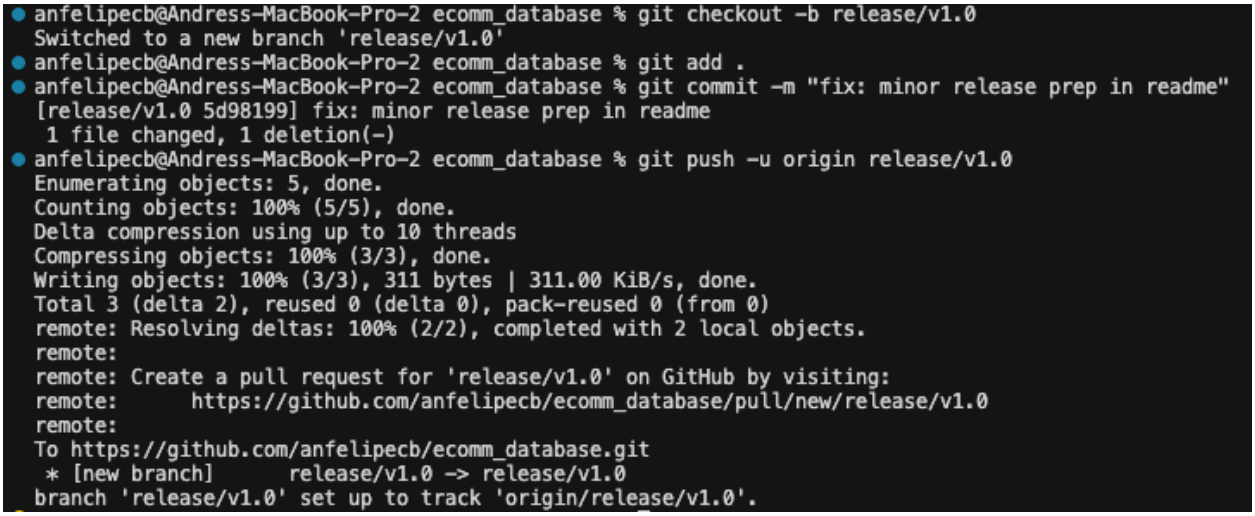


Figure 5: Release branch creation

And in Figure 6, I show the opened pull request merged into the main branch.

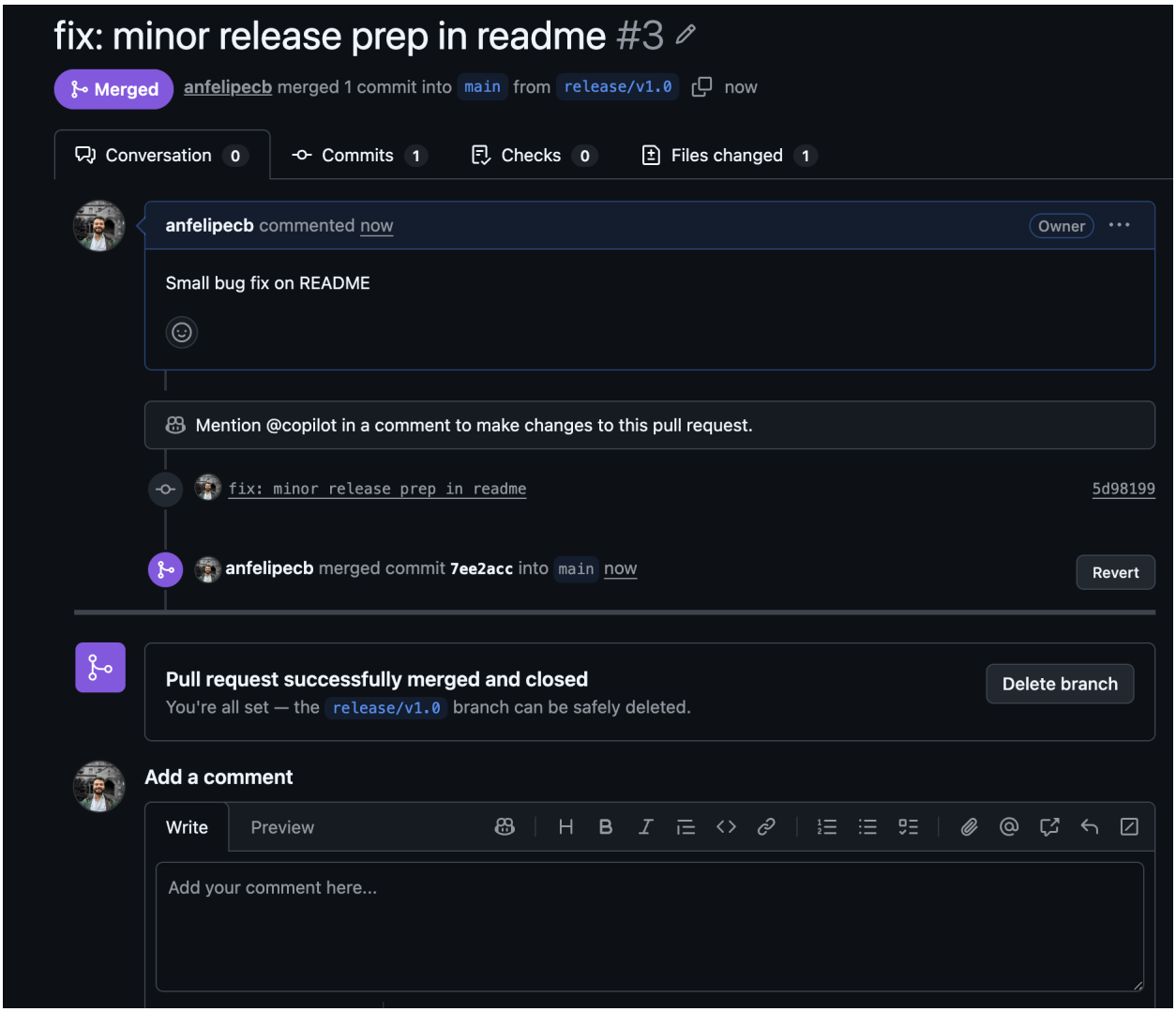


Figure 6: Opened Pull Request to Main

Figures 7 and 8 show the tag of the release and the merge of the release branch back into the develop branch.

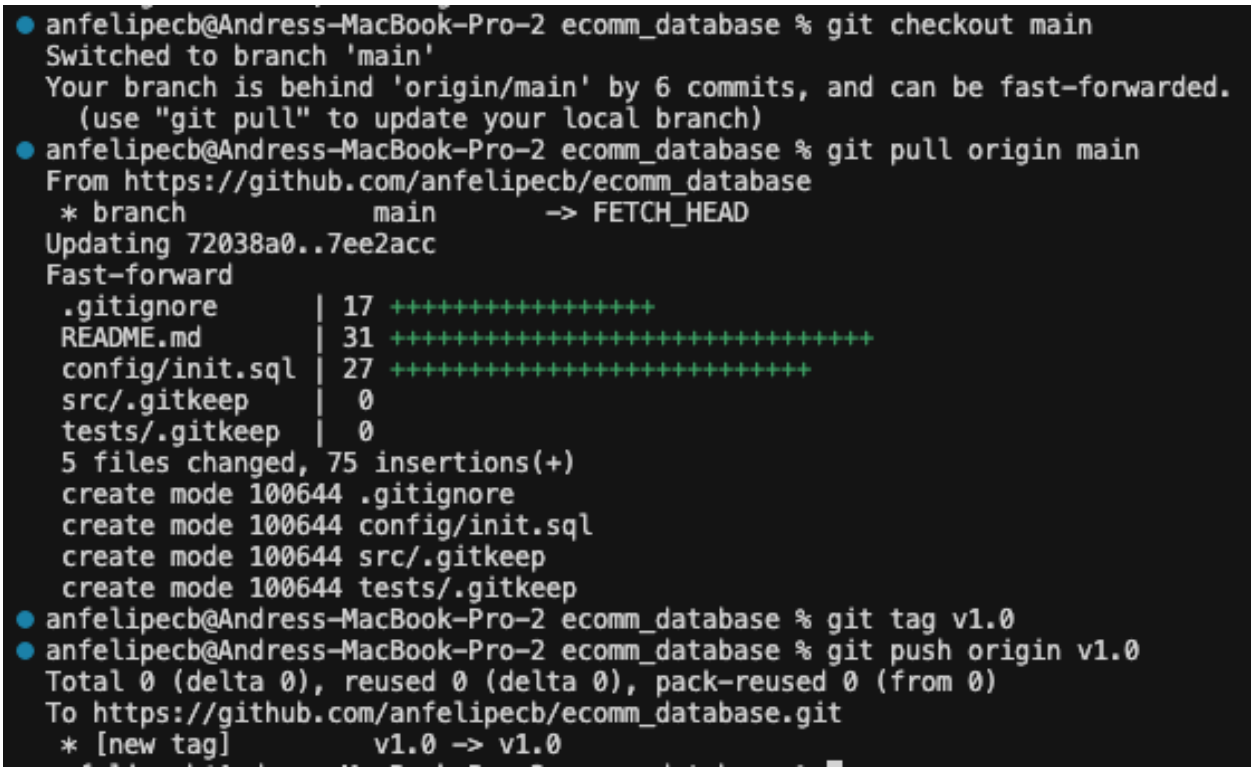


Figure 7: Tag Release on Main

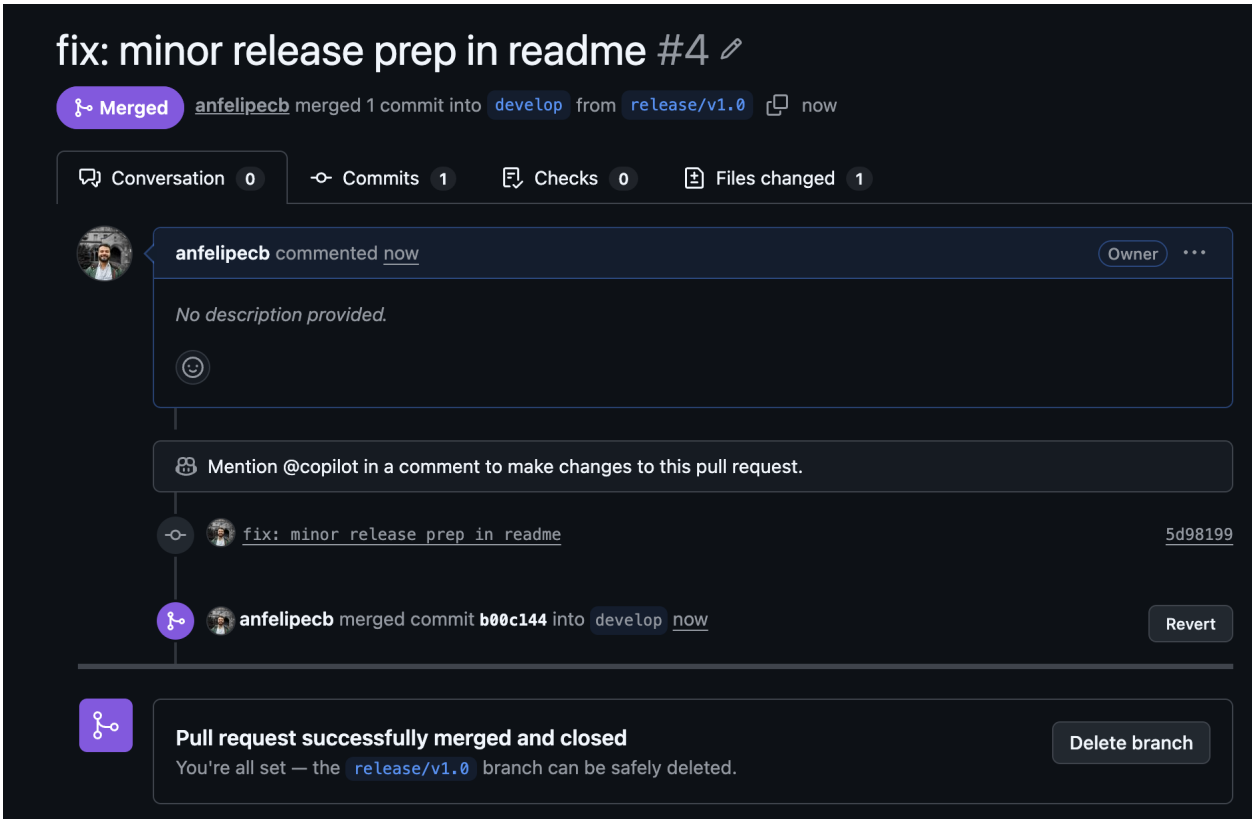


Figure 8: Merged Release Back to Develop

Steps demonstrated: 1. Create release branch: `git checkout develop && git checkout -b release/v1.0` 2. Bug fixes only, no new features 3. PR: release/v1.0 → main 4. Tag release on main: `git tag v1.0` 5. Merge release back to develop

4. Complete Hotfix Workflow After detecting a security issue in the production environment, I create a hotfix branch to fix the issue as shown in Figure 9.

```
anfelipecb@Andress-MacBook-Pro-2 ecomm_database % git checkout -b hotfix/security-patch
Switched to a new branch 'hotfix/security-patch'
anfelipecb@Andress-MacBook-Pro-2 ecomm_database % git add .
anfelipecb@Andress-MacBook-Pro-2 ecomm_database % git commit -m "fix: critical securitu patch"
[hotfix/security-patch 7b07efd] fix: critical securitu patch
 1 file changed, 1 insertion(+)
anfelipecb@Andress-MacBook-Pro-2 ecomm_database % git push -u origin hotfix/security-patch
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 10 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 355 bytes | 355.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
remote:
remote: Create a pull request for 'hotfix/security-patch' on GitHub by visiting:
remote:      https://github.com/anfelipecb/ecomm_database/pull/new/hotfix/security-patch
remote:
To https://github.com/anfelipecb/ecomm_database.git
 * [new branch]      hotfix/security-patch -> hotfix/security-patch
branch 'hotfix/security-patch' set up to track 'origin/hotfix/security-patch'.
```

Figure 9: Hotfix branch creation

Then in Figures 10 and 11, I create the pull request and merge it into the main and develop branches.

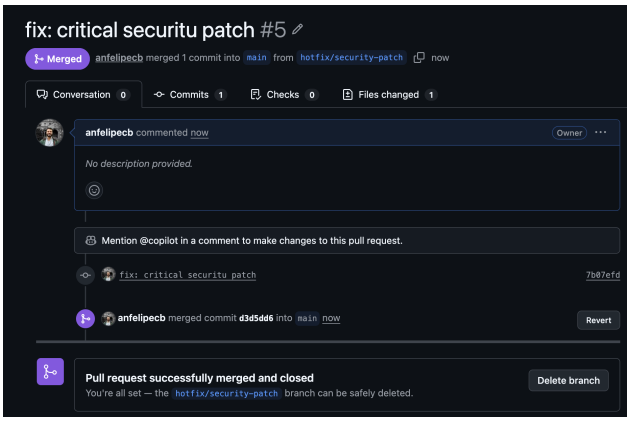


Figure 10: Opened Pull Request to Main

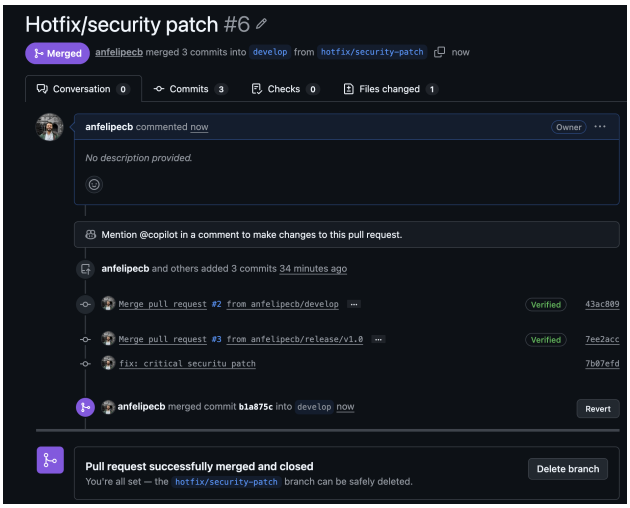


Figure 11: Opened Pull Request to Develop

Steps demonstrated: 1. Create hotfix from main: `git checkout main && git checkout -b hotfix/security-patch` 2. Fix, commit, push 3. PR: hotfix → main 4. Merge to main, then merge main → develop (or PR hotfix → develop)

---

5. How All Branch Types Interact (Git Flow Diagram) The git flow graph showing how all branch types interact is shown in Figure 12.

```
anfelipecb@Andress-MacBook-Pro-2 ecomm_database % git log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr)' --abbrev-comm
l
*   b1a875c - (origin/develop) Merge pull request #6 from anfelipecb/hotfix/security-patch (5 minutes ago)
|\
* \   b00c144 - Merge pull request #4 from anfelipecb/release/v1.0 (18 minutes ago)
|\ \
| | | *   d3d5dd6 - (origin/main, origin/HEAD) Merge pull request #5 from anfelipecb/hotfix/security-patch (7 minutes ago)
| | | |\
| | | |/
| | |/|
| | * | 7b07efd - (HEAD -> hotfix/security-patch, origin/hotfix/security-patch) fix: critical securitu patch (10 minutes ago)
| | |/
| | *   7ee2acc - (tag: v1.0, main) Merge pull request #3 from anfelipecb/release/v1.0 (27 minutes ago)
| | |\
| | |/
| |/|
| * | 5d98199 - (origin/release/v1.0, release/v1.0) fix: minor release prep in readme (32 minutes ago)
|/ /
| *   43ac809 - Merge pull request #2 from anfelipecb/develop (39 minutes ago)
| |\
| |/
|/|
* |   bb5cc83 - (develop) Merge pull request #1 from anfelipecb/feature/add_functionality (63 minutes ago)
|\ \
| |/
|/|
| * 9f622b6 - (origin/feature/add_functionality, feature/add_functionality) Adding initial sctructure to the repo (70 minutes ago)
| * 7187ce2 - Adding lines to README (74 minutes ago)
|/
* 72038a0 - Initial commit (3 hours ago)
```

Figure 12: Git Flow Diagram

Design Decisions and Rationale

1. Separate Repositories, One per Microservice I chose to split the application into four separate repositories—one for each microservice—rather than keeping everything in a single monorepo. The assignment calls for each service to be independently buildable and deployable, and separate repos support that directly. Each team (or developer) can own a service, version it on its own schedule, and run its own CI/CD pipeline without stepping on others. This also keeps merge conflicts low and matches the microservices idea of small, autonomous units.

2. Consistent Folder Structure (src, config, tests) I standardized the same folder layout across all repos: `src/` for application code, `config/` for configuration and schema, and `tests/` for tests. This keeps things predictable—anyone opening a repo knows where to find code, config, and tests. It also separates concerns: code lives in `src/`, env files and init scripts in `config/`, and tests in `tests/`, which makes it easier to wire up CI and tooling later.

3. Git Flow over Trunk-Based Development I adopted Git Flow (main, develop, feature, release, hotfix). The project has distinct environments (dev, staging, prod) and planned releases, and Git Flow fits that model. It clearly separates production (`main`) from ongoing work (`develop`), uses release branches for stabilization, and provides a hotfix path for production issues. The workflow is well known and easy for new team members to follow.

4. Protected Branches I protect `main` and `develop` so that no one can push directly to them. All changes must go through Pull Requests. That way, code is reviewed before it lands, CI must pass before merge (once Jenkins is set up), and force pushes or accidental overwrites are blocked. It also keeps a clear record of who changed what and when.

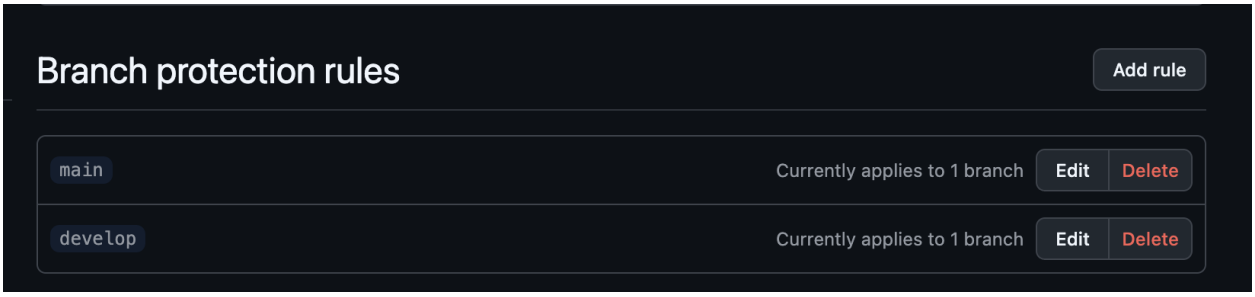


Figure 13: rules for protected branches

Phase 2: Containerization with Docker

For this phase, I created the Dockerfiles for each microservice and the docker-compose.yml file to run the application locally.

The docker files for each microservice can be found directly inside the repositories referenced at the beginning of this document.

The docker-compose.yml configuration implemented is the following:

```yaml
services:
  database:
    build: ./ecomm_database
    image: ecomm-database
    container_name: ecomm-database
    environment:
      POSTGRES_DB: ecommerce
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: password
    ports:
      - "5432:5432"

  product-service:
    build: ./ecomm_product_service
    image: ecomm-product-service
    container_name: ecomm-product-service
    environment:
      DB_HOST: database
      DB_NAME: ecommerce
      DB_USER: postgres
      DB_PASSWORD: password
    ports:
      - "3001:3001"
    depends_on:
      - database

  order-service:
    build: ./ecomm_order_service
    image: ecomm-order-service
    container_name: ecomm-order-service
    environment:
      DB_HOST: database
      DB_NAME: ecommerce
      DB_USER: postgres
      DB_PASSWORD: password
      PRODUCT_SERVICE_URL: http://product-service:3001
    ports:
      - "3002:3002"
    depends_on:
      - database
      - product-service

  frontend:
```

```yaml
build: ./ecomm_front_end
image: ecomm-frontend
container_name: ecomm-frontend
ports:
  - "3000:80"
depends_on:
  - product-service
  - order-service
```

The **security scan reports** for each microservices are found in the Figures 14, 15, 16, and 17 using the following command: `docker scan <image_name>`.

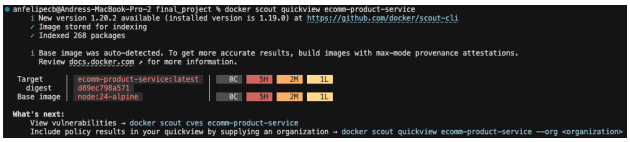


Figure 14: Security scan report for product service

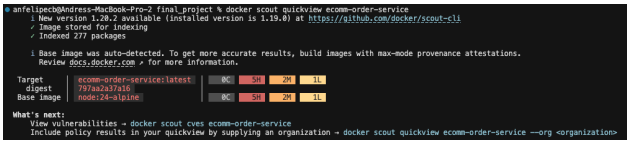


Figure 15: Security scan report for order service

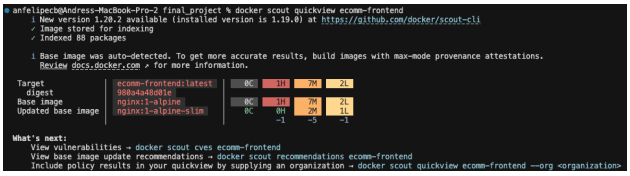


Figure 16: Security scan report for frontend

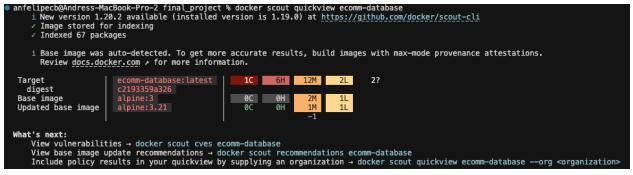


Figure 17: Security scan report for database

After that I created the repositories in Docker Hub and pushed the images to them, using this code:

```
docker login
DOCKERHUB_USER=afcamachob
docker tag ecomm-product-service $DOCKERHUB_USER/ecomm-product-service:latest
docker tag ecomm-order-service $DOCKERHUB_USER/ecomm-order-service:latest
docker tag ecomm-frontend $DOCKERHUB_USER/ecomm-frontend:latest
docker tag ecomm-database $DOCKERHUB_USER/ecomm-database:latest

docker push $DOCKERHUB_USER/ecomm-product-service:latest
docker push $DOCKERHUB_USER/ecomm-order-service:latest
docker push $DOCKERHUB_USER/ecomm-frontend:latest
docker push $DOCKERHUB_USER/ecomm-database:latest
```

And the images successfully pushed to Docker Hub can be seen in Figure 18:

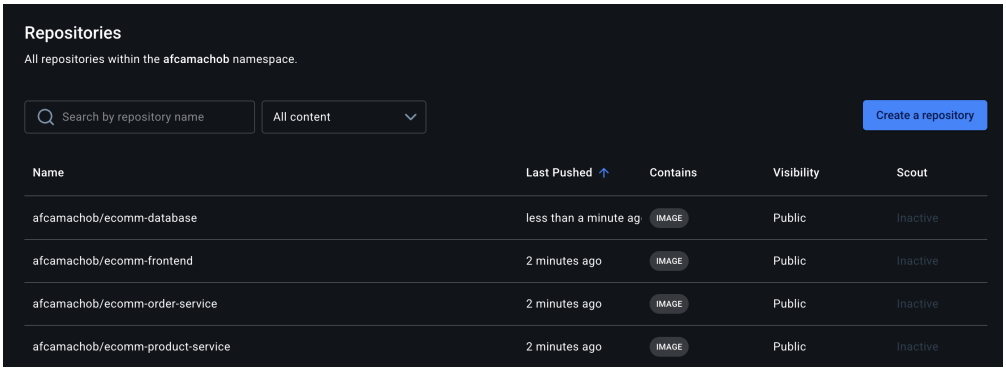


Figure 18: Images successfully pushed to Docker Hub

The Docker Hub repositories for each microservice can be accessed at the following URLs:

- Product Service: https://hub.docker.com/repository/docker/afcamachob/ecomm-product-service
- Order Service: https://hub.docker.com/repository/docker/afcamachob/ecomm-order-service
- Frontend: https://hub.docker.com/repository/docker/afcamachob/ecomm-frontend
- Database: https://hub.docker.com/repository/docker/afcamachob/ecomm-database

Phase 3: CI/CD Pipeline with Jenkins

For this phase, I created the Jenkinsfiles for each microservice and the Jenkins shared library.

The Jenkinsfiles for each microservice can be found directly inside the repositories referenced at the beginning of this document.

The Jenkins shared library can be found at the following URL: https://github.com/anfelipecb/ecomm_jenkins-shared-library

The pipelines configured can be seen in figure 19, and figure 20 shows the branches recognized by the pipeline:

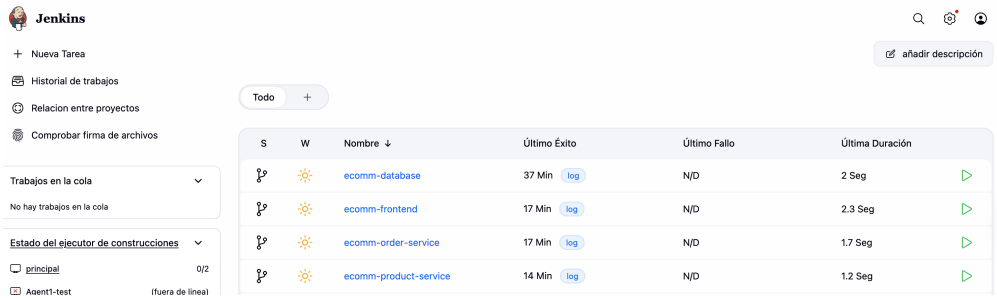


Figure 19: Pipelines configured

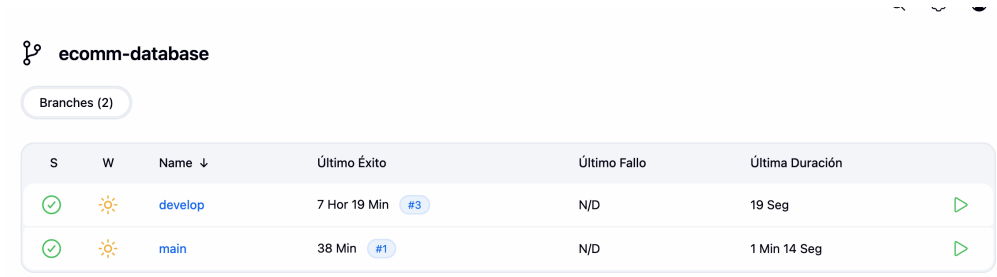


Figure 20: Branches recognized by the pipeline

I also verified the pipeline execution for each environment; in figure 21 I present the build pipeline execution for the database service, after creating a PR from the develop branch to the main branch and the console output:

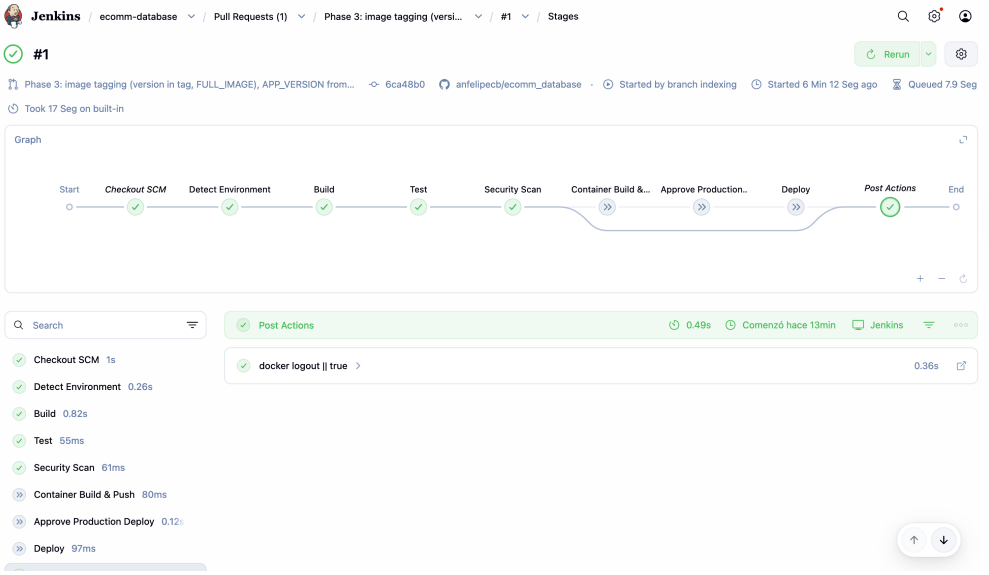
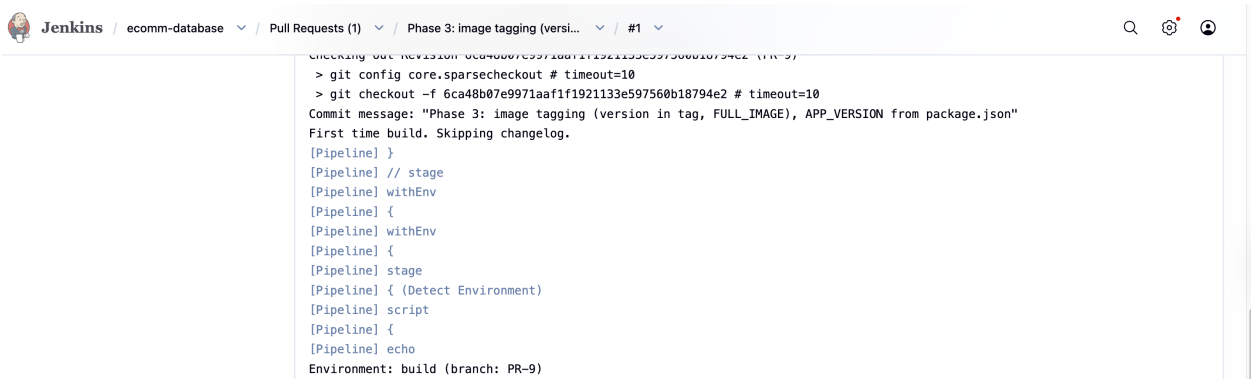


Figure 21: Build pipeline execution for the develop branch in the database service



In figure 22 I show the pipeline execution of the develop branch in the database service, after pushing a new commit to the develop branch, which the GitHub webhook triggers the build in Jenkins and this is the result:

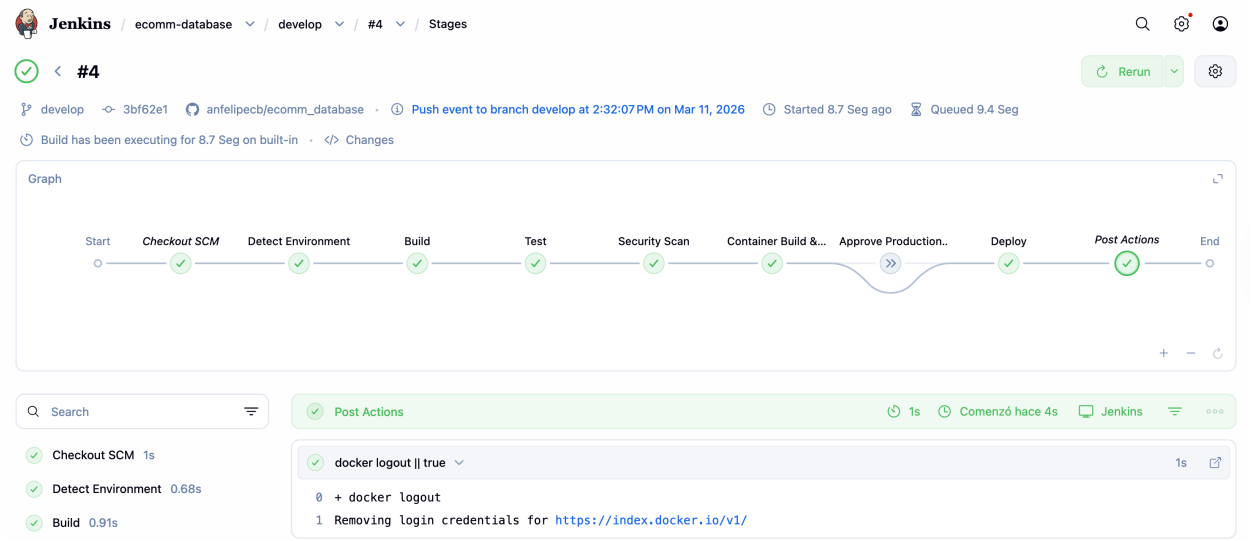


Figure 22: Stages in the Development pipeline execution

Now, to show the staging pipeline execution, I created a release branch (v2.0) and pushed a new commit to it, which the GitHub webhook triggers the build in Jenkins and this is the result:

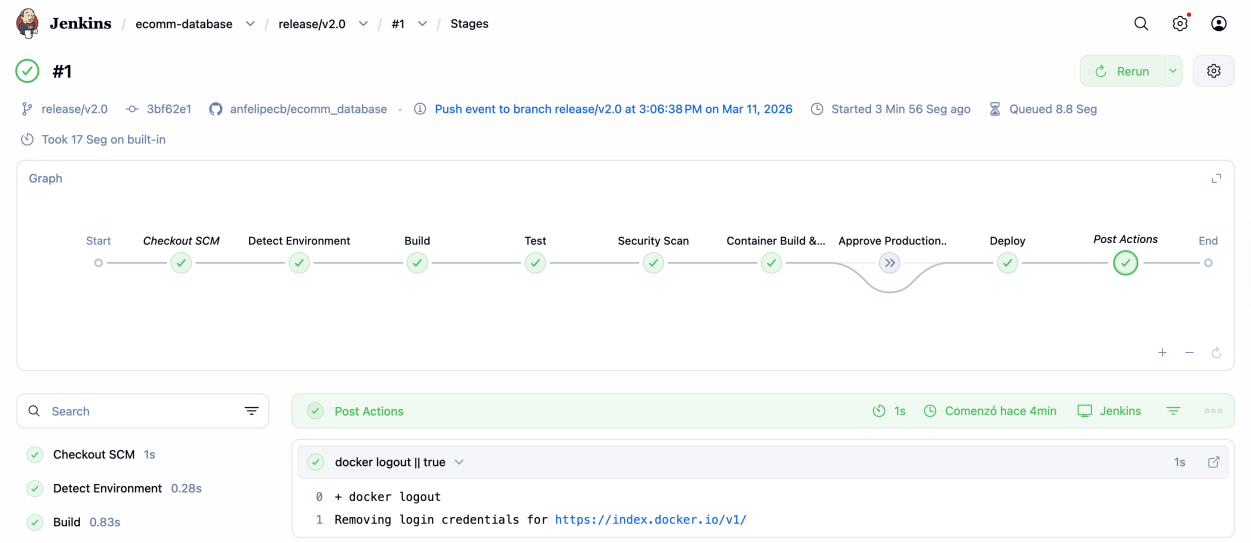


Figure 23: Staging pipeline execution

And finally, to show the production pipeline execution, I created a pull request from the release branch (v2.0) to the main branch, which the GitHub webhook triggers the build in Jenkins, and it prompts for an approval to deploy to the production environment:

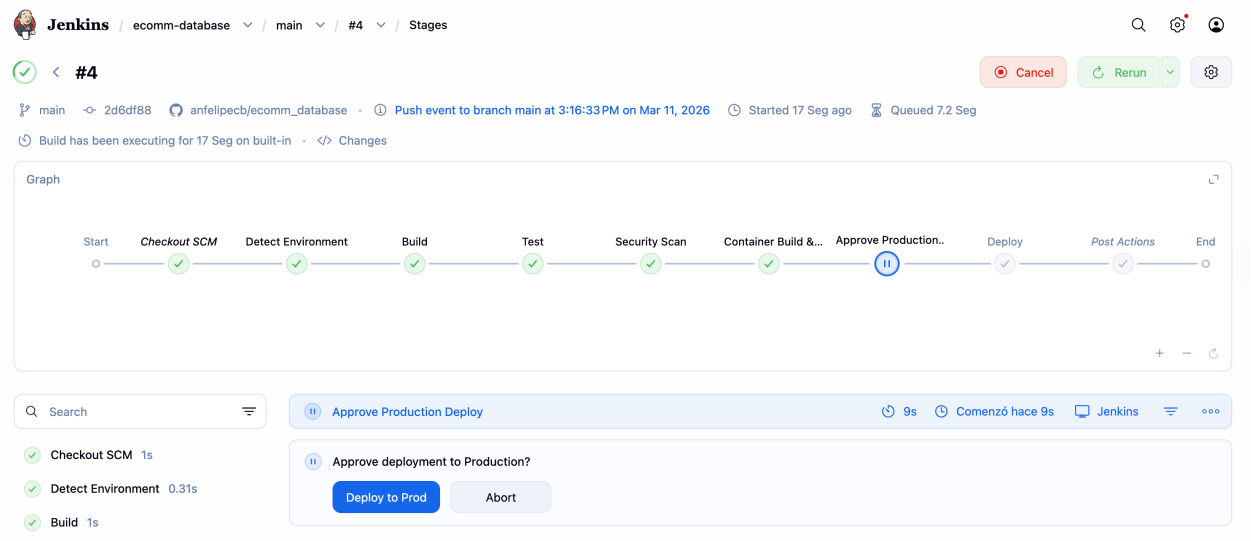


Figure 24: Production pipeline execution

After approved, the Deploy stage is executed and the database service is deployed to the production environment as shown in figure 25:

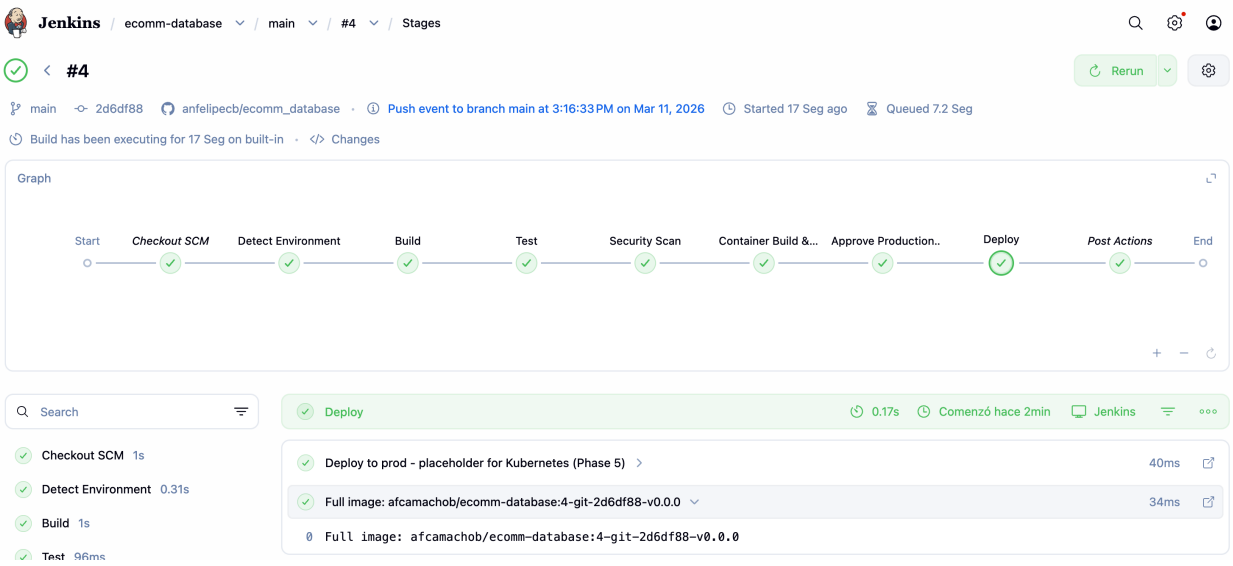


Figure 25: Deploy to production environment

Phase 4: Infrastructure as Code with Terraform

For this phase, I created Terraform modules to provision local infrastructure: a Docker network for container discovery, Minikube clusters, PostgreSQL containers, and Jenkins servers. The configuration lives in the `terraform/` directory at the project root.

1. Terraform Modules

I implemented four modules:

Local Development Network Module Creates a Docker network per environment so containers can discover each other by name (e.g., `ecomm-postgres-dev`). Useful when running app services locally with Docker Compose or when connecting tools to PostgreSQL/Jenkins.

```
# terraform/modules/local-dev/main.tf
resource "docker_network" "ecomm" {
  name = "ecomm-${var.environment}-network"
}

output "network_name" {
  value = docker_network.ecomm.name
}
```

Minikube Module Starts a local Kubernetes cluster using Minikube. Uses `null_resource` with `local-exec` provisioners to run `minikube start` and `minikube delete` on destroy.

```
# terraform/modules/minikube/main.tf
resource "null_resource" "minikube_start" {
```

```
  triggers = {
    profile = var.profile
    cpus    = var.cpus
    memory  = var.memory
  }

  provisioner "local-exec" {
    command = <<-EOT
      minikube start --profile=${var.profile} --cpus=${var.cpus} --memory=${var.memory}
    EOT
  }

  provisioner "local-exec" {
    when    = destroy
    command = "minikube delete --profile=${self.triggers.profile} || true"
  }
}
```

PostgreSQL Module Provisions a PostgreSQL 15 Alpine container using the Docker provider. Used for local development and testing.

```
# terraform/modules/postgresql/main.tf (excerpt)
resource "docker_image" "postgres" {
  name = "postgres:15-alpine"
}

resource "docker_container" "postgres" {
  name  = "ecomm-postgres-${var.environment}"
  image = docker_image.postgres.image_id
  env   = ["POSTGRES_DB=ecommerce", "POSTGRES_USER=postgres", "POSTGRES_PASSWORD=${var.p
  ports { internal = 5432; external = var.port }
}
```

Jenkins Module Provisions a Jenkins LTS container with a persistent volume for `/var/jenkins_home`. Uses ports 9080 (dev), 9081 (staging), 9082 (prod) to avoid conflict with an existing Jenkins on 8080.

```
# terraform/modules/jenkins/main.tf (excerpt)
resource "docker_volume" "jenkins_home" {
  name = "jenkins_home_${var.environment}"
}

resource "docker_container" "jenkins" {
  name  = "ecomm-jenkins-${var.environment}"
  image = docker_image.jenkins.image_id
```

```
  ports { internal = 8080; external = var.port }
  volumes {
    volume_name    = docker_volume.jenkins_home.name
    container_path = "/var/jenkins_home"
  }
}
```

Root Configuration The root `main.tf` wires the modules together and passes variables from the workspace-specific `.tfvars` files.

```
# terraform/main.tf
module "local_dev" {
  source      = "./modules/local-dev"
  environment = var.environment
}

module "minikube" {
  source  = "./modules/minikube"
  profile = var.minikube_profile
  cpus    = var.minikube_cpus
  memory  = var.minikube_memory
}

module "postgresql" {
  source = "./modules/postgresql"
  environment = var.environment
  port        = var.postgres_port
}

module "jenkins" {
  source = "./modules/jenkins"
  environment = var.environment
  port        = var.jenkins_port
}
```

2. Terraform Workspaces

I use workspaces to manage dev, staging, and prod from the same configuration. Each workspace has its own state file to prevent environment conflicts.

Workspace	Minikube Profile	PostgreSQL Port	Jenkins Port
dev	ecomm-dev	5435	9080
staging	ecomm-staging	5433	9081
prod	ecomm-prod	5434	9082

```
terraform workspace new dev
terraform workspace new staging
terraform workspace new prod
terraform workspace select dev
```

3. State Management

I use the local backend with per-workspace state files. Terraform uses file-level locking during operations.

Backend configuration (`terraform/backend.tf`):

```
terraform {
  backend "local" {
    path = "terraform.tfstate"
  }
}
```

State file locations: - `terraform.tfstate.d/dev/terraform.tfstate` - `terraform.tfstate.d/stagin`
- `terraform.tfstate.d/prod/terraform.tfstate`

Backup procedure: Before major changes, copy the state directory:

```
cp -r terraform.tfstate.d terraform.tfstate.d.backup.$(date +%Y%m%d)
```

Full state management documentation is in `terraform/README.md` and `terraform/STATE_MANAGEMENT.md`.

4. Variable Files per Environment

I use `.tfvars` files to avoid hardcoding values. Each environment has its own file:

dev.tfvars:

```
environment      = "dev"
minikube_profile = "ecomm-dev"
minikube_cpus    = 2
minikube_memory  = 4096
postgres_port    = 5435
jenkins_port     = 9080
docker_registry  = "index.docker.io"
```

staging.tfvars and **prod.tfvars** follow the same pattern with environment-specific ports and profile names.

5. Terraform Outputs for CI/CD

Outputs expose values the Jenkins pipeline needs:

```
# terraform/outputs.tf
output "docker_network" { value = module.local_dev.network_name; description = "Docker n
output "minikube_profile" { value = var.minikube_profile }
```

```
output "minikube_kubeconfig_command" { value = "minikube kubectl --profile=${var.minikub
output "postgres_host" { value = "localhost" }
output "postgres_port" { value = var.postgres_port }
output "jenkins_url" { value = "http://localhost:${var.jenkins_port}" }
output "docker_registry" { value = var.docker_registry }
```

Retrieving outputs for pipeline configuration:

```
terraform workspace select dev
terraform output
```

Screenshots

Terraform plan output (for each environment) Figure 26 shows the output of `terraform plan -var-file=dev.tfvars` for the dev workspace (divided in two parts). The plan lists the seven resources to be created: Docker network, Minikube cluster, PostgreSQL container and image, Jenkins container, image, and volume.

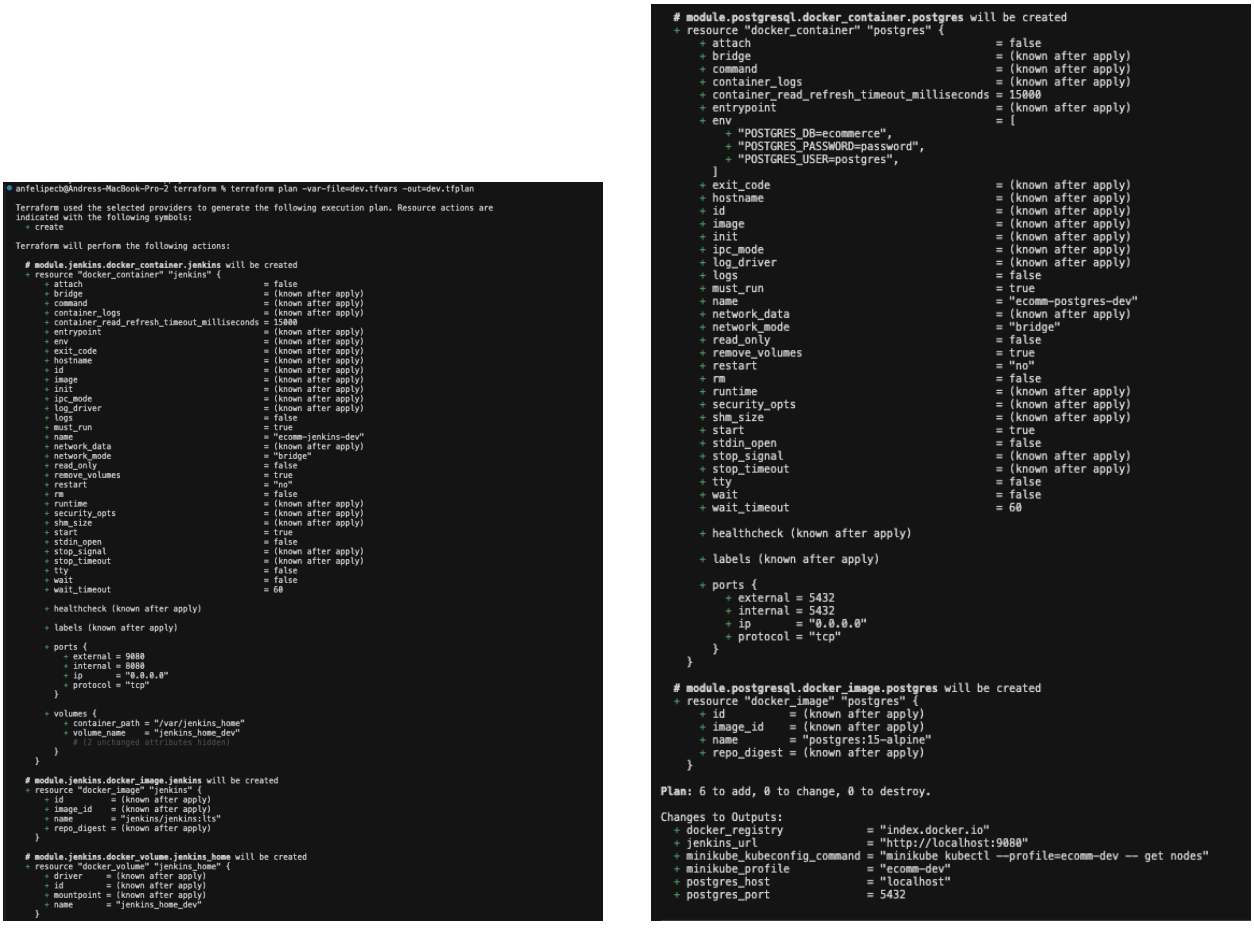


Figure 26: Output of `terraform plan -var-file=dev.tfvars` for the dev workspace (part 1 and part 2). The plan lists the seven resources to be created: Docker network, Minikube cluster, PostgreSQL container and image, Jenkins container, image, and volume.

In Figure 27, I show the output of `terraform plan -var-file=staging.tfvars` for the staging workspace; as the first part of the output is long, from now on I only show the second part for the other environments. The plan lists the six resources to be created:

Figure 27: Terraform plan output for staging environment

In Figure 28 I show the output of `terraform plan -var-file=prod.tfvars` for the prod workspace:

Figure 28: Terraform plan output for prod environment

Successful terraform apply (dev environment) Figure 29 shows the result of `terraform apply -var-file=dev.tfvars -auto-approve`. All seven resources were created successfully.

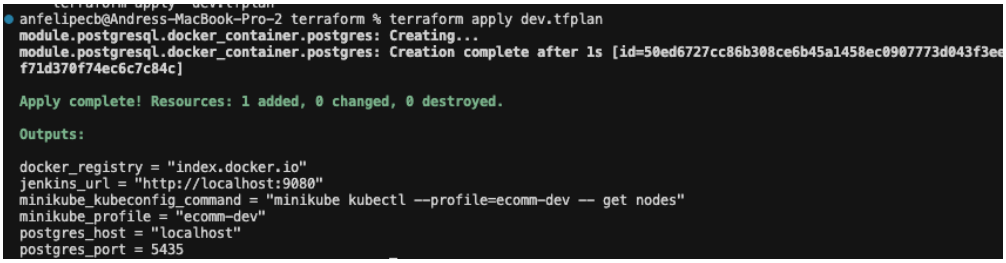


Figure 29: Terraform apply success for dev environment

The same was verified for the other environments with a similar output.

Running Minikube cluster and local resources Figure 30 shows the Minikube cluster and Docker containers after apply. The output of `minikube status` and `docker ps` confirms the ecomm-prod cluster is running and the PostgreSQL and Jenkins containers are up.

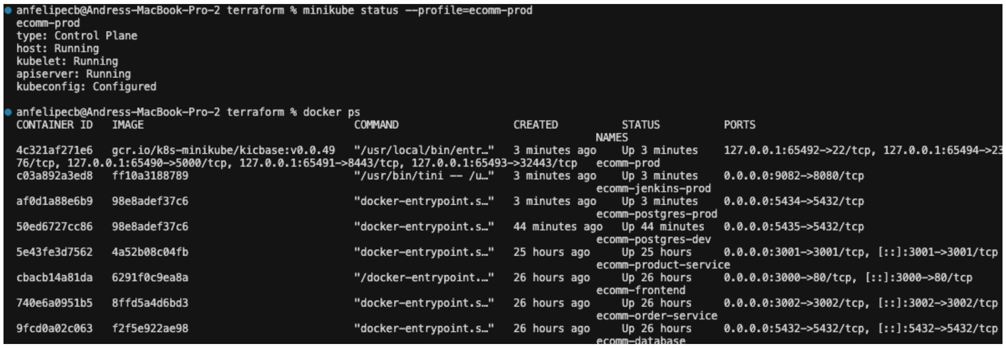


Figure 30: Minikube and Docker resources after apply

Terraform outputs for Jenkins Figure 31 shows the output of `terraform output` for the prod workspace. These values (docker_network, minikube_profile, postgres_host, postgres_port, jenkins_url, docker_registry) will be used by the Jenkins pipeline and local development tooling.

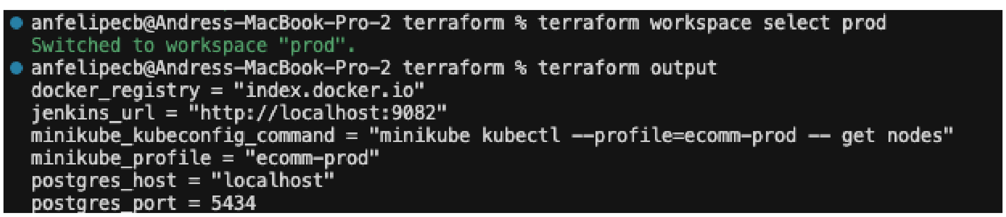


Figure 31: Terraform outputs used by Jenkins pipeline

Phase 5: Kubernetes Deployment

For this phase, I created Kubernetes manifests for each service and wired the Jenkins Deploy stage to apply them to Minikube. Manifests live in the `kubernetes/` directory inside each service repository, with separate folders for dev, staging, and prod.

1. Kubernetes Manifests

Each service has Deployments, Services, and (for the database) ConfigMaps, Secrets, and PersistentVolumes. I use the image tags produced by the Jenkins pipeline (e.g. `7-git-abc1234-v1.0.0`) so deployments track versions correctly.

Database (`ecomm_database/kubernetes/{dev,staging,prod}/`): namespace, secret, configmap, pv-database, deployment, service. The deployment mounts a PersistentVolumeClaim so data survives pod restarts.

Product, Order, Frontend (`ecomm_*_service/kubernetes/{dev,staging,prod}/`): deployment and service. Each deployment uses `strategy.type: RollingUpdate` with `maxSurge: 1` and `maxUnavailable: 0`.

Services use ClusterIP for internal communication and NodePort for external access. NodePorts are assigned per environment (dev: 30000–30002, staging: 30100–30102, prod: 30200–30202).

2. Namespaces for Environments

I created namespaces `ecomm-dev`, `ecomm-staging`, and `ecomm-prod` with ResourceQuotas so each environment has limits (e.g. dev: 2 CPU / 2Gi memory requests, prod: 8 CPU / 8Gi). The database pipeline applies the namespace first; other services deploy into it.

3. Persistent Volumes for Database

The database uses a hostPath PersistentVolume and a PersistentVolumeClaim. The deployment mounts the PVC at `/var/lib/postgresql/data`, so data persists across pod restarts. Paths are environment-specific (e.g. `/data/ecomm-db-dev`).

4. Jenkins Deploy Integration

I added a shared library function `deployToK8s` that applies manifests and sets the deployment image. Jenkins runs locally, so I use `minikube kubectl --profile=ecomm-{dev|staging|prod} --` to target the right cluster. Each pipeline's Deploy stage calls `deployToK8s` with the deployment name, container name, and list of manifest files. The function replaces `PLACEHOLDER_IMAGE` with `env.FULL_IMAGE` before apply and runs `kubectl set image` so the built image is deployed.

5. Manual Apply Script

For local testing without Jenkins, I use `./apply-k8s-dev.sh` from the project root. It applies all manifests in order using specific versioned image tags from Docker Hub.

Screenshots

Running pods in ecomm-dev namespace Figure 32 shows the output of `kubectl get pods -n ecomm-dev` after applying the manifests. All four services (database, product-service, order-service, frontend) are Running and Ready.

```
anfelipecb@Andress-MacBook-Pro-2 ecomm_order_service % kubectl get pods -n ecomm-dev
NAME                                  READY   STATUS    RESTARTS   AGE
ecomm-database-5476d9974c-m44zv       1/1     Running   0          52m
ecomm-frontend-86d47fc947-6wxtd       1/1     Running   0          51m
ecomm-order-service-5db4fd4f95-hx6d5  1/1     Running   0          51m
ecomm-product-service-77cdbfbbd9-zhx4c 1/1    Running   0          51m
```

Figure 32: Running pods in ecomm-dev namespace

Services configuration Figure 33 shows the output of `kubectl get svc -n ecomm-dev`. ClusterIP services enable internal communication; NodePort services expose the frontend (30000), product (30001), and order (30002) for browser access.

```
anfelipecb@Andress-MacBook-Pro-2 ecomm_order_service % kubectl get svc -n ecomm-dev
NAME                    TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
ecomm-database          ClusterIP   10.106.14.62     <none>        5432/TCP         52m
ecomm-frontend          NodePort    10.109.51.90     <none>        80:30000/TCP     52m
ecomm-order-service     NodePort    10.105.127.222   <none>        3002:30002/TCP   52m
ecomm-product-service   NodePort    10.104.5.35      <none>        3001:30001/TCP   52m
```

Figure 33: Services in ecomm-dev namespace

Application in browser Figure 34 shows the e-commerce frontend loaded via `minikube service ecomm-frontend -n ecomm-dev --url --profile=ecomm-dev`. Products load from the product service, and orders can be created through the order service.

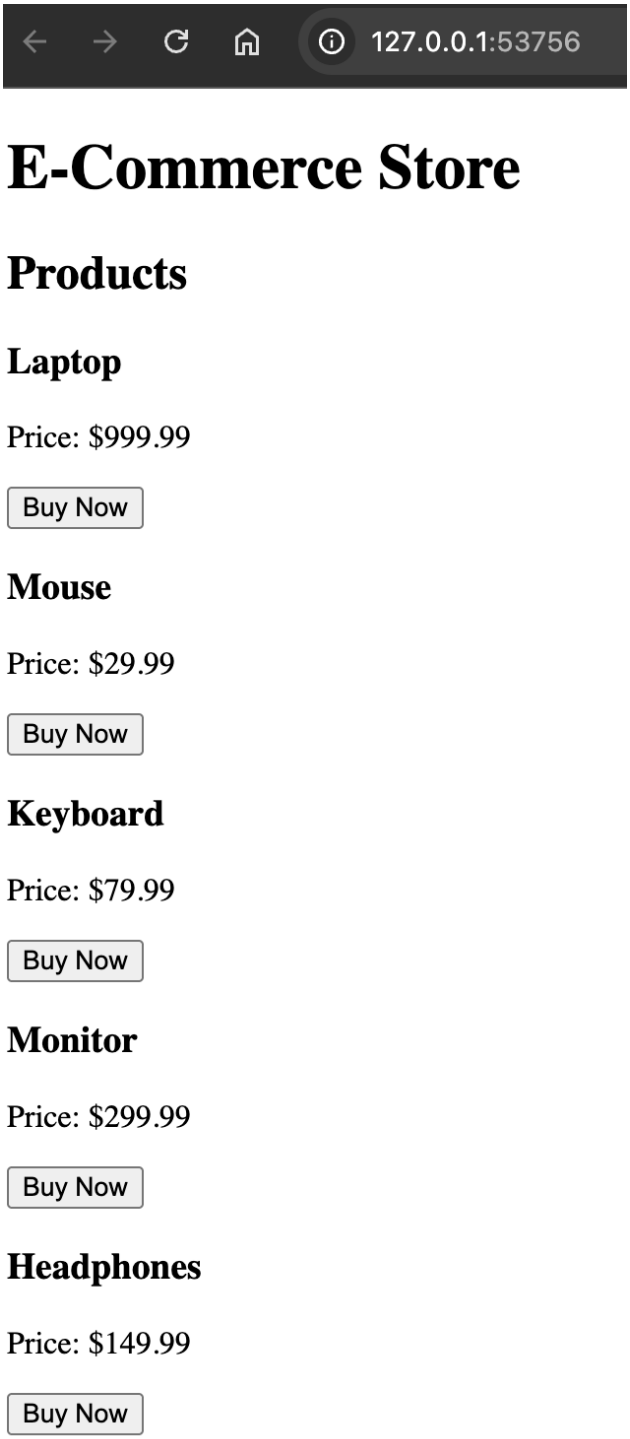


Figure 34: E-commerce app in browser

Jenkins Deploy stage Figure 35 shows the Deploy stage in a Jenkins pipeline run after pushing the Jenkinsfile to the develop branch inside the ecomm_product_service repository. The shared library applies the manifests and sets the deployment image to the versioned tag built by the pipeline.

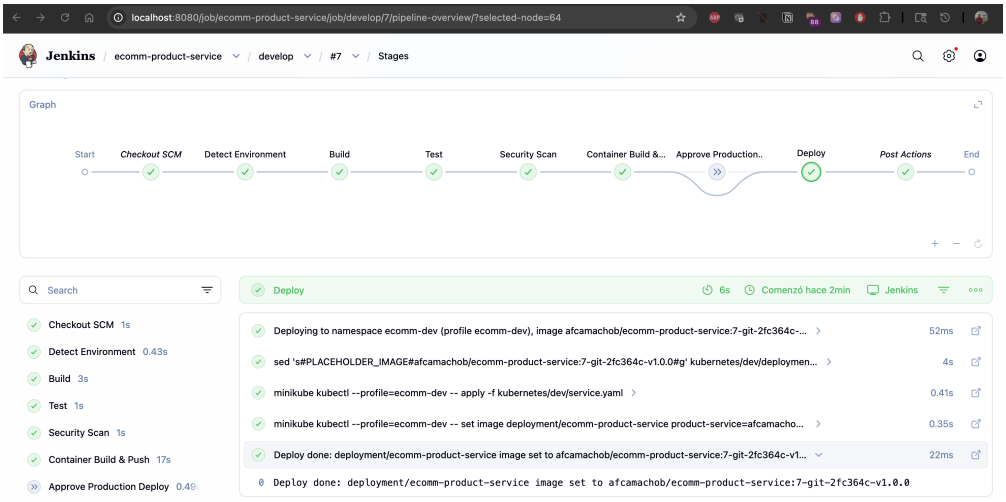


Figure 35: Jenkins Deploy stage success

Successful deployment rollout Figure 36 shows the output of `kubectl rollout status deployment/ecomm-product-service -n ecomm-dev` after a pipeline deploy, confirming the rollout completed successfully.

```
anfelipecb@Andress-MacBook-Pro-2 final_project % minikube kubectl --profile=ecomm-dev -- rollout status deployment/ecomm-product-service -n ecomm-dev
deployment "ecomm-product-service" successfully rolled out
```

Figure 36: Successful deployment rollout

Deployment Strategy

I use **rolling updates** for the application services (product, order, frontend). Each Deployment has `strategy.type: RollingUpdate` with `maxSurge: 1` and `maxUnavailable: 0`, so Kubernetes replaces pods gradually: it starts one new pod, waits for it to be ready, then terminates an old one. This keeps the service available during updates and requires no extra infrastructure.

For the database, I use `strategy.type: Recreate` because it runs a single replica with a PersistentVolume. Recreate stops the existing pod before starting the new one, which is appropriate for stateful workloads that cannot run multiple instances against the same volume.

I did not use blue-green or canary deployments. Blue-green would require two full environments and instant traffic switching, which adds complexity and resource usage for this project scope. Canary would require traffic splitting (e.g. via a service mesh or ingress rules) to roll out to a small percentage of users first. Rolling updates are the standard Kubernetes approach, require no additional tooling, and fit my Minikube-based setup. For production at scale, blue-green or canary could be considered for faster rollback or lower-risk releases.

Phase 6: Integration Validation

After running the dev, staging and prod clusters with `minikube start --profile=ecomm-dev`, `minikube start --profile=ecomm-staging` and `minikube start --profile=ecomm-prod` respectively, I made a code change in the ecomm_product_service repository by adding a new feature to the product service.

After pushing the change to the develop branch, the Jenkins pipeline is triggered and the build is successful. The image is built and pushed to the Docker Hub repository. The deployment is successful and the new feature is deployed to the dev environment. Here in Figure 37 I show the jenkins pipeline execution for the develop branch:

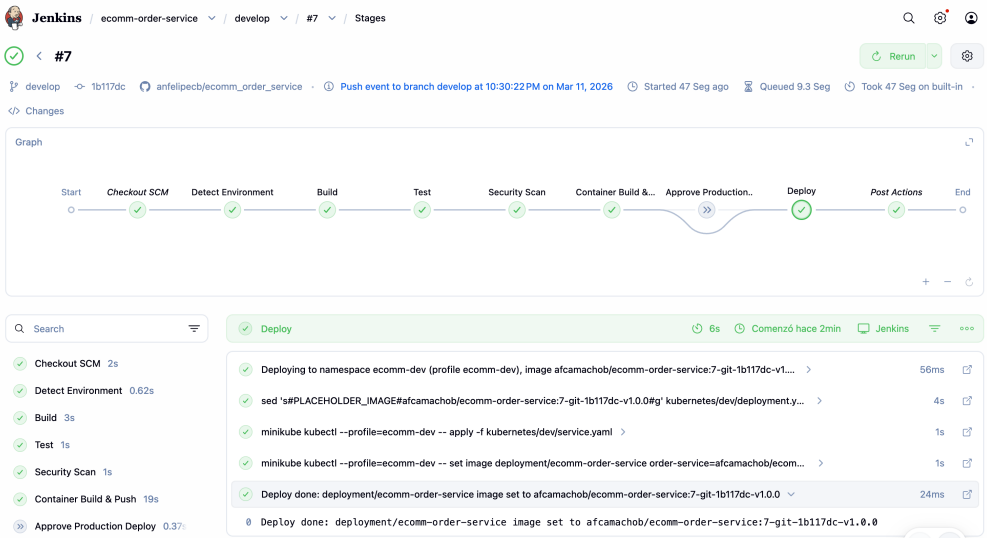


Figure 37: Jenkins pipeline execution for the develop branch

And in Figure 38 I show the result of `minikube kubectl --profile=ecomm-dev -- get pods -n ecomm-dev` after the deployment is successful:

```
anfelipecb@Andress-MacBook-Pro-2 ecomm_order_service % minikube kubectl --profile=ecomm-dev -- get pods -n ecomm-dev
NAME                                READY   STATUS    RESTARTS   AGE
ecomm-database-5d4f9d5d56-6fdnv     1/1     Running   0          117m
ecomm-frontend-78497456b7-qsd8l     1/1     Running   0          117m
ecomm-order-service-8fd96d669-q4vwn 1/1     Running   0          3m7s
ecomm-product-service-7786c4f849-mrlt2 1/1  Running   0          118m
```

Figure 38: Result of kubectl output after the deployment is successful

Now I will show the staging flow after creating a release branch (v2.0) and pushing a new commit to it. The Jenkins pipeline is triggered and the build is successful. The Figure 39 shows the commit made, Figure 40 shows the Jenkins pipeline execution for the release branch and Figure 41 shows the result of `minikube kubectl --profile=ecomm-staging -- get pods -n ecomm-staging` after the deployment is successful:

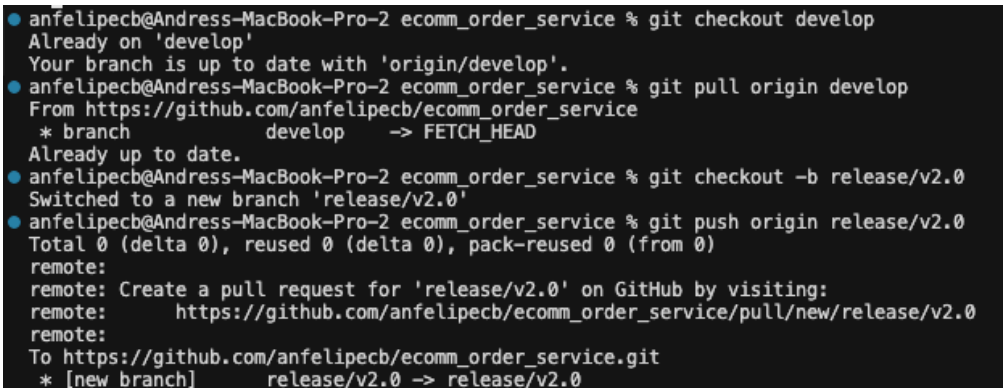


Figure 39: Commit made

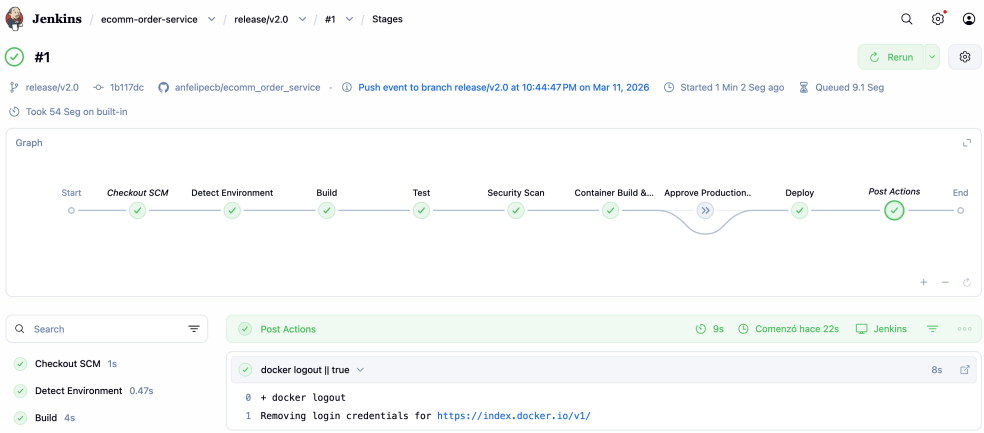


Figure 40: Jenkins pipeline execution for the release branch

Figure 41: Kubectl output after the deployment is successful

Finally, I will show the production flow after creating a pull request from the release branch (v2.0) to the main branch. The Jenkins pipeline is triggered and the build is successful. The Figure 42 shows the pull request made, Figure 43 shows the Jenkins pipeline execution for the pull request and Figure 44 shows the result of `minikube kubectl --profile=ecomm-prod -- get pods -n ecomm-prod` after the deployment is successful:

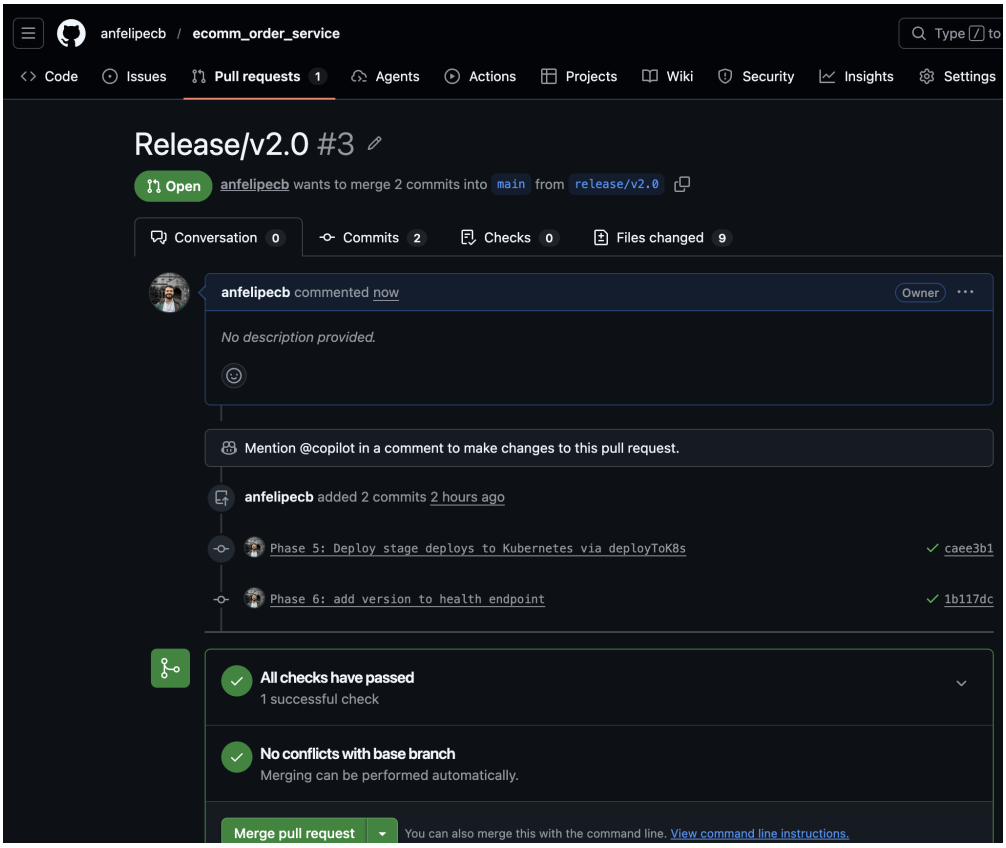


Figure 42: Pull request made

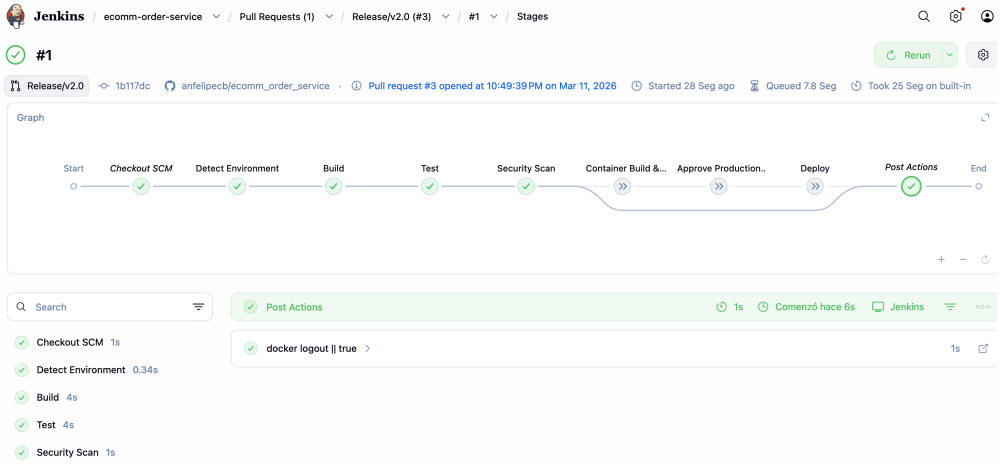


Figure 43: Jenkins pipeline execution for the pull request

```
anfelipecb@Andress-MacBook-Pro-2 ecomm_order_service % minikube kubectl --profile=ecomm-prod -- get pods -n ecomm-prod
NAME                                READY   STATUS    RESTARTS   AGE
ecomm-database-97bb7c8c8-sh8lg      1/1     Running   0          53m
ecomm-frontend-d867cbc5d-9xrhq      1/1     Running   0          53m
ecomm-frontend-d867cbc5d-gqhkk      1/1     Running   0          53m
ecomm-order-service-5857785df-25wkw 1/1     Running   0          53m
ecomm-order-service-5857785df-wv72t 1/1     Running   0          53m
ecomm-product-service-764ffdb659-g6b6m 1/1  Running   0          53m
ecomm-product-service-764ffdb659-hmxv7 1/1  Running   0          53m
```

Figure 44: Result of kubectl output after the deployment is successful

After merging the pull request, the main branch is updated and the approval is requested and given to deploy to the production environment. In Figure 45 I show the jenkins pipeline successful deployment to the production environment:

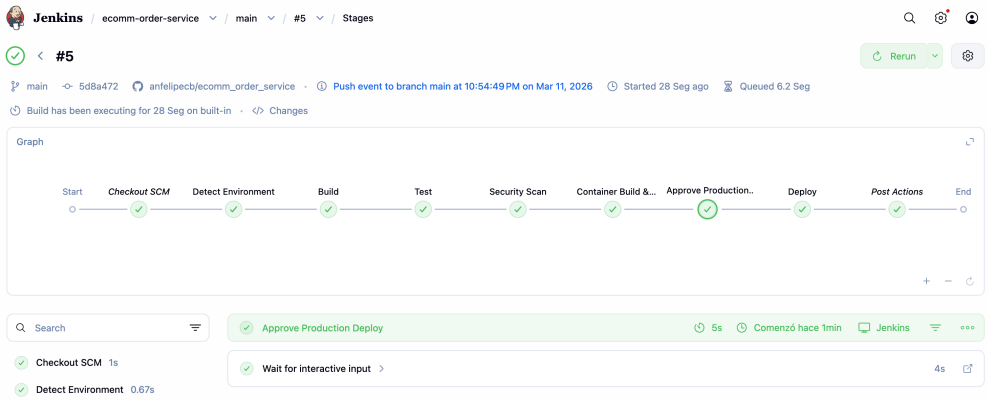


Figure 45: Jenkins pipeline successful deployment to the production environment

To show the application in the browser I ran this commands from three different terminals:

```
minikube kubectl --profile=ecomm-dev -- port-forward svc/ecomm-frontend 3000:80 -n ecomm
minikube kubectl --profile=ecomm-dev -- port-forward svc/ecomm-product-service 30001:300
minikube kubectl --profile=ecomm-dev -- port-forward svc/ecomm-order-service 30002:3002
```

And I got the following results:

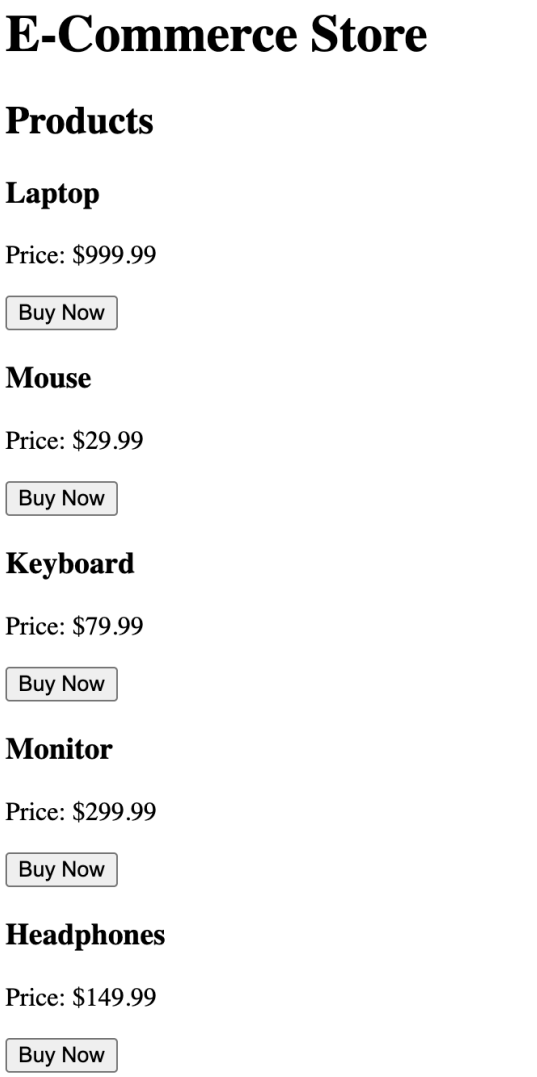


Figure 46: Application in the browser: products visible

After clicking on Buy Now for a laptop, I got:

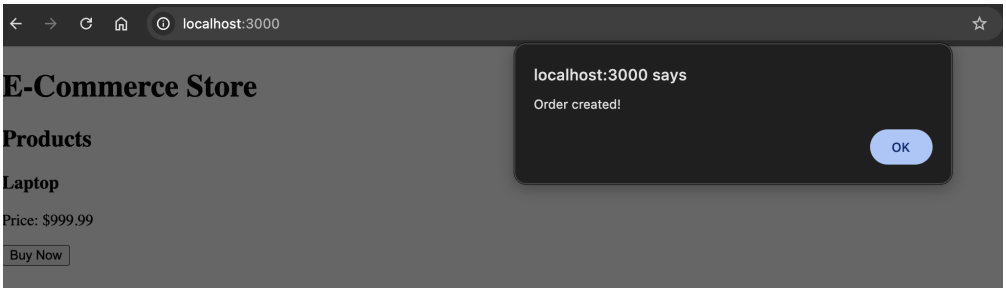


Figure 47: Order created

And I see the orders via `curl http://localhost:30002/orders`, which can be seen in Figure 48:

Figure 48: Orders in the terminal

Image tags in each environment To verify that each environment runs the correct versioned image, I ran the following command to show deployment images across dev, staging, and prod:

```
for env in dev staging prod; do
  echo "=== ecomm-$env ==="
  minikube kubectl --profile=ecomm-$env -- get deployments -n ecomm-$env -o custom-colum
  echo ""
done
```

Figure 49 shows the output. Each environment uses the image tag produced by its pipeline (e.g. `anfelipecb/ecomm-product-service:7-git-abc1234-v1.0.0`), confirming that dev, staging, and prod run distinct versions as expected.

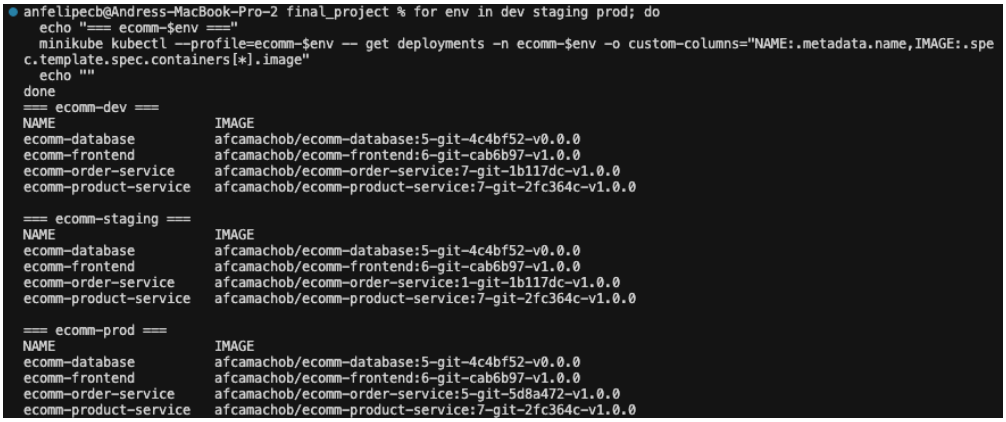


Figure 49: Image tags in dev, staging, and prod environments

Docker image build and push The Jenkins pipeline builds the Docker image with a versioned tag (e.g. `${BUILD_NUMBER}-git-${GIT_COMMIT}-v${VERSION}`) and pushes it to Docker Hub. Figure 50 shows the Container Build and Container Push stages in a Jenkins pipeline run, including the image tag used and the successful push to the registry.

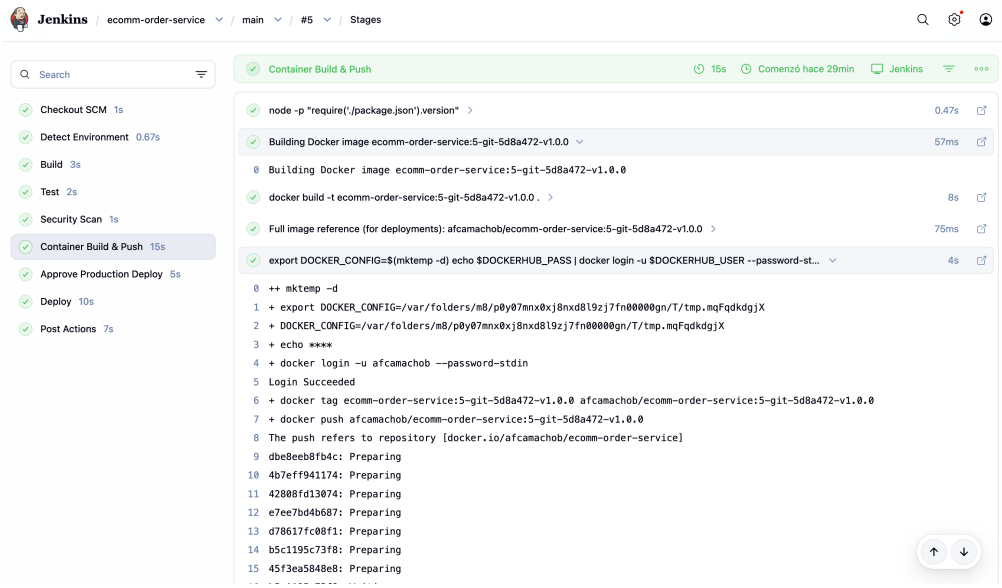


Figure 50: Docker image build and push in Jenkins pipeline

Challenges and How I Resolved Them

1. Wrong Minikube profile / kubectl context

When using multiple Minikube profiles (ecomm-dev, ecomm-staging, ecomm-prod), commands default to the wrong cluster. `minikube service --url` without `--profile=ecomm-dev` returned "control-plane node minikube host is not running" because it used the default profile. Similarly, `kubectl` does not have a `--profile` flag; using `kubectl --profile=ecomm-staging` caused "unknown profile" errors. I resolved this by always using `minikube kubectl --profile=ecomm-{env} --` for all kubectl commands and `minikube service ... --profile=ecomm-{env}` when exposing services.

2. Products not loading in the browser

After port-forwarding only the frontend to localhost:3000, the page loaded but products were empty. The frontend fetches products from localhost:30001 and orders from localhost:30002. I had to run three port-forwards in parallel: frontend (3000:80), product-service (30001:3001), and order-service (30002:3002). All three must be running for the app to work.

3. Namespace not found

Running `kubectl port-forward svc/ecomm-frontend 3000:80 -n ecomm-dev` returned "namespaces ecomm-dev not found" because the default kubectl context pointed to a different cluster. I used `minikube kubectl --profile=ecomm-dev -- port-forward ...` so the command targeted the correct cluster.

4. Docker driver on macOS

With the Minikube Docker driver on macOS, `minikube service --url` sometimes produced tunnel-related messages or unreliable URLs. Port-forwarding with `minikube kubectl`

`--profile=... -- port-forward` was more reliable for local browser and API access.

Phase 7: DevOps Extensions

I implemented two DevOps practices from the approved topics: (1) Infrastructure testing and validation, and (2) Metrics collection and visualization (Prometheus/Grafana). Both are integrated into the existing pipeline and deployment flow.

Topic 1: Infrastructure Testing and Validation

Design **Problem statement:** Kubernetes manifests can have schema errors, wrong API versions, or typos that only surface at deploy time. Validating manifests before deploy catches these early and prevents failed rollouts.

Architecture: I added a validation stage that runs `kubectl apply --dry-run=client -f` on all manifests before the Deploy stage. If validation fails, the pipeline stops and no deployment is attempted.

Acceptance criteria:

- Pipeline runs manifest validation before every Deploy
- Invalid manifests cause pipeline failure with a clear error
- The same manifest list as Deploy is used per service
- Validation works for dev, staging, and prod (uses `PIPELINE_ENV`)

Integration points: Jenkins shared library (jenkins-shared-library/vars/validateK8sManifests.groovy), and the Deploy stage in all four Jenkinsfiles (database, product-service, order-service, frontend).

Environment behavior: The same validation logic runs in all environments; only the target profile and manifest path differ (`kubernetes/{dev,staging,prod}/`).

Implementation I created a shared library function `validateK8sManifests(List<String> manifestFiles, String placeholderImage)` that:

- Uses `minikube kubectl --profile=ecomm-{env} --` to target the correct cluster
- For each manifest file: if it is a deployment, substitutes `PLACEHOLDER_IMAGE` with a placeholder (e.g. `busybox:latest`) and pipes to `kubectl apply --dry-run=client -f -`; otherwise runs `kubectl apply --dry-run=client -f <path>`
- Fails the build if any manifest is invalid

Each Jenkinsfile now has a **Validate Manifests** stage (when `PIPELINE_ENV != 'build'`) immediately before the Deploy stage, calling `validateK8sManifests` with the same list of manifest files used by `deployToK8s`.

Execution Figure 51 shows the Jenkins pipeline with the Validate Manifests stage succeeding before Deploy.

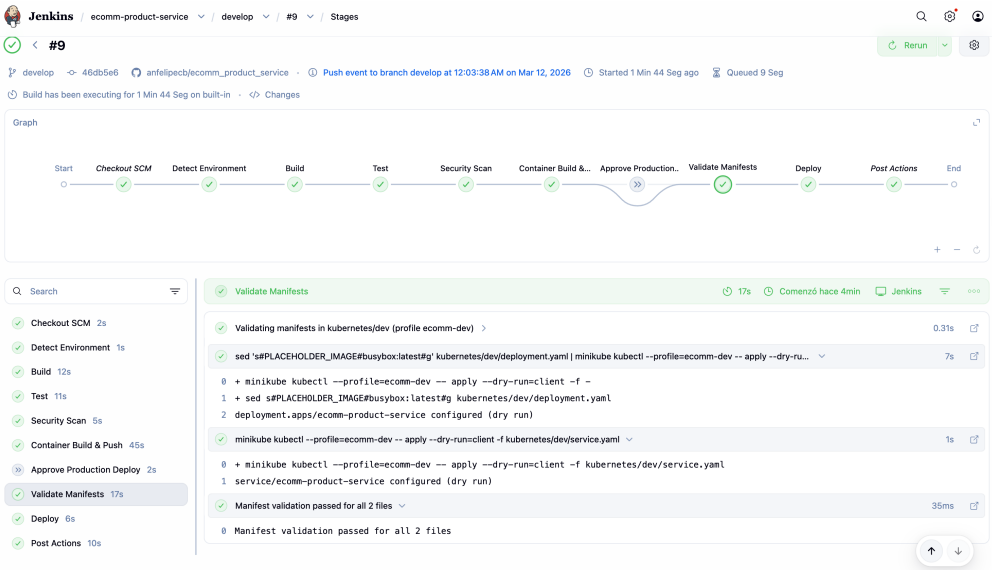


Figure 51: Validate Manifests stage in Jenkins pipeline

Topic 2: Metrics Collection and Visualization (Prometheus/Grafana)

Design Problem statement: Without metrics, we cannot observe service health, request rates, or latency. Prometheus collects metrics; Grafana visualizes them for dashboards and alerting.

Architecture: I added the `prom-client` library to the product and order services to expose a `/metrics` endpoint in Prometheus format. I deployed Prometheus and Grafana to the `ecomm-dev` namespace. Prometheus scrapes the product and order services at `ecomm-product-service:3001/metrics` and `ecomm-order-service:3002/metrics`. Grafana uses Prometheus as a datasource so we can build dashboards (e.g. `up` status, default Node.js metrics).

Acceptance criteria:

- Product and order services expose `/metrics` in Prometheus format
- Prometheus scrapes both services and shows them as "up" in the targets page
- Grafana dashboard shows at least one panel (e.g. `up` or `http_requests_total`)

Integration points: Application code (Node.js) in ecomm_product_service and ecomm_order_service, Kubernetes manifests in the project-root `monitoring/` directory, and the apply script `apply-monitoring-dev.sh`.

Environment behavior: Monitoring is deployed to the dev namespace for this project. The same manifests could be applied to staging or prod with a different namespace if needed.

Implementation

- **Application metrics:** In both product and order services I added the `prom-client` dependency and in `src/index.js`: a Registry, `collectDefaultMetrics({ register`

`}`), and `GET /metrics` returning `register.metrics()`.

- **Monitoring manifests:** In `monitoring/` I added:
  - `prometheus-configmap.yaml`: ConfigMap with `prometheus.yml` defining scrape configs for product-service (port 3001) and order-service (port 3002) in `ecomm-dev`
  - `prometheus-deployment.yaml`: Prometheus Deployment (image `prom/prometheus:v2.47.0`) mounting the config, and a ClusterIP Service on port 9090
  - `grafana-deployment.yaml`: Grafana Deployment (image `grafana/grafana:10.2.0`) and Service on port 3000, with admin/admin credentials for the demo
- **Apply script:** `apply-monitoring-dev.sh` applies the monitoring manifests using `minikube kubectl --profile=ecomm-dev --`. Run it after the app services are deployed (e.g. after `apply-k8s-dev.sh`).

Execution Figure 52 shows the Prometheus targets page with both product-service and order-service scraped successfully (State: UP).

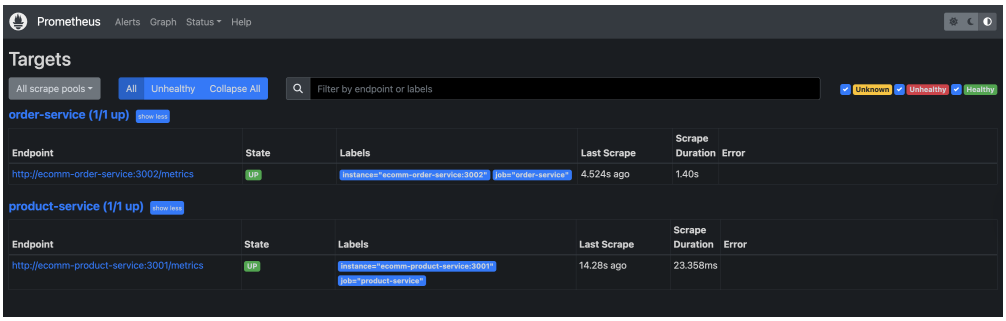


Figure 52: Prometheus targets showing product and order services

Figure 53 shows a Grafana dashboard with metrics from the services.

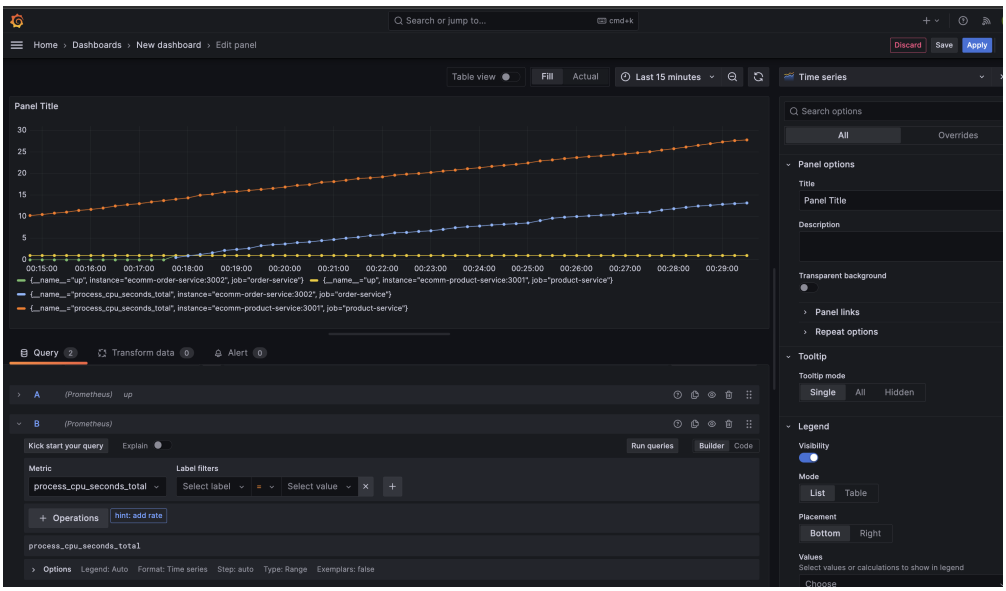


Figure 53: Grafana dashboard with Prometheus metrics

Figure 54 shows the raw `/metrics` output from the product service. This is obtained by running `curl http://localhost:30001/metrics` in a terminal after running `minikube kubectl --profile=ecomm-dev -- port-forward svc/ecomm-product-service 30001:3001 -n ecomm-dev`.

```
nodejs_heap_space_size_used_bytes{space="trusted_large_object"} 0

# HELP nodejs_heap_space_size_available_bytes Process heap space size available from Node.js in bytes.
# TYPE nodejs_heap_space_size_available_bytes gauge
nodejs_heap_space_size_available_bytes{space="read_only"} 0
nodejs_heap_space_size_available_bytes{space="new"} 404936
nodejs_heap_space_size_available_bytes{space="old"} 431744
nodejs_heap_space_size_available_bytes{space="code"} 290496
nodejs_heap_space_size_available_bytes{space="shared"} 0
nodejs_heap_space_size_available_bytes{space="trusted"} 89664
nodejs_heap_space_size_available_bytes{space="shared_trusted"} 0
nodejs_heap_space_size_available_bytes{space="new_large_object"} 1048576
nodejs_heap_space_size_available_bytes{space="large_object"} 0
nodejs_heap_space_size_available_bytes{space="code_large_object"} 0
nodejs_heap_space_size_available_bytes{space="shared_large_object"} 0
nodejs_heap_space_size_available_bytes{space="shared_trusted_large_object"} 0
nodejs_heap_space_size_available_bytes{space="trusted_large_object"} 0

# HELP nodejs_version_info Node.js version info.
# TYPE nodejs_version_info gauge
nodejs_version_info{version="v24.14.0",major="24",minor="14",patch="0"} 1

# HELP nodejs_gc_duration_seconds Garbage collection duration by kind, one of major, minor, incremental or weakcb.
# TYPE nodejs_gc_duration_seconds histogram
nodejs_gc_duration_seconds_bucket{le="0.001",kind="minor"} 1
nodejs_gc_duration_seconds_bucket{le="0.01",kind="minor"} 19
nodejs_gc_duration_seconds_bucket{le="0.1",kind="minor"} 39
nodejs_gc_duration_seconds_bucket{le="1",kind="minor"} 45
nodejs_gc_duration_seconds_bucket{le="2",kind="minor"} 45
nodejs_gc_duration_seconds_bucket{le="5",kind="minor"} 45
nodejs_gc_duration_seconds_bucket{le="+Inf",kind="minor"} 45
nodejs_gc_duration_seconds_sum{kind="minor"} 3.6252429159581667
nodejs_gc_duration_seconds_count{kind="minor"} 45
nodejs_gc_duration_seconds_bucket{le="0.001",kind="incremental"} 0
nodejs_gc_duration_seconds_bucket{le="0.01",kind="incremental"} 0
nodejs_gc_duration_seconds_bucket{le="0.1",kind="incremental"} 1
nodejs_gc_duration_seconds_bucket{le="1",kind="incremental"} 2
nodejs_gc_duration_seconds_bucket{le="2",kind="incremental"} 2
nodejs_gc_duration_seconds_bucket{le="5",kind="incremental"} 2
nodejs_gc_duration_seconds_bucket{le="+Inf",kind="incremental"} 2
nodejs_gc_duration_seconds_sum{kind="incremental"} 0.32280541698634624
nodejs_gc_duration_seconds_count{kind="incremental"} 2
nodejs_gc_duration_seconds_bucket{le="0.001",kind="major"} 0
nodejs_gc_duration_seconds_bucket{le="0.01",kind="major"} 0
nodejs_gc_duration_seconds_bucket{le="0.1",kind="major"} 0
nodejs_gc_duration_seconds_bucket{le="1",kind="major"} 2
nodejs_gc_duration_seconds_bucket{le="2",kind="major"} 2
nodejs_gc_duration_seconds_bucket{le="5",kind="major"} 2
nodejs_gc_duration_seconds_bucket{le="+Inf",kind="major"} 2
nodejs_gc_duration_seconds_sum{kind="major"} 0.49766712500154975
nodejs_gc_duration_seconds_count{kind="major"} 2
```

Figure 54: Product service /metrics endpoint output