Proyecto Final

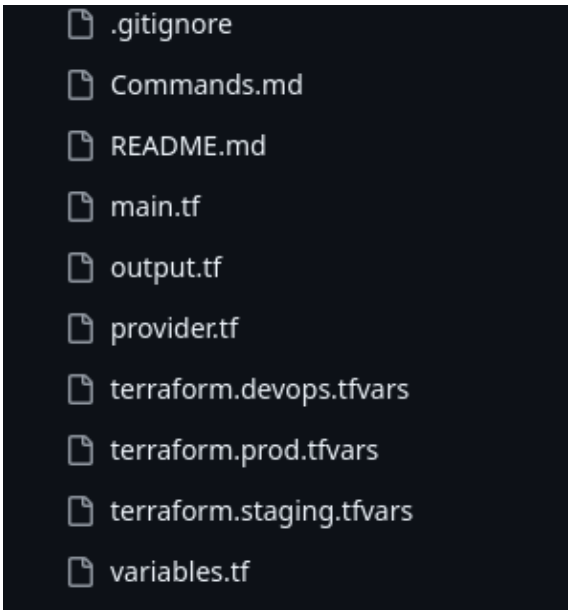
Geoffrey Pasaje Vidal - A00380495

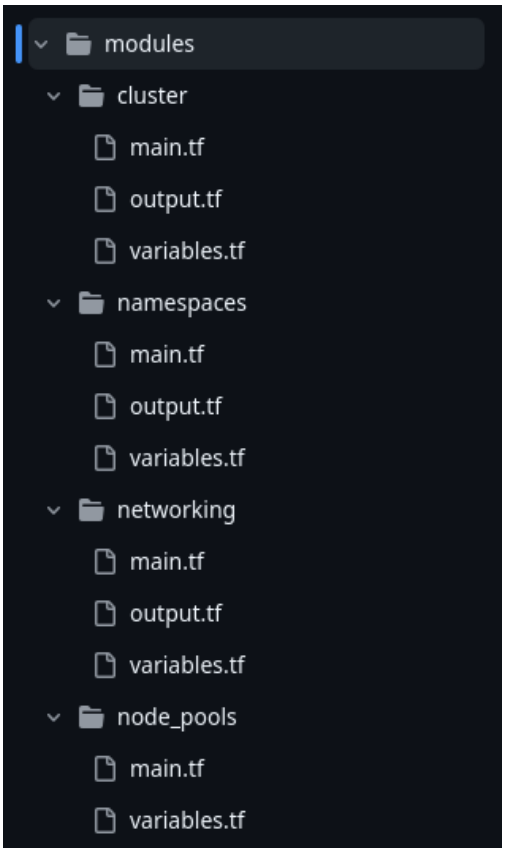
1. **(10%) Configurar Jenkins, Docker y Kubernetes para su uso.**

   Despliegue de clúster de Kubernetes (kS8) en la plataforma de Google Cloud (GCP) usando la herramienta Terraform para definir la infraestructura como código IaC.

- **Definimos la infraestructura en Terraform:**

  Se usó una distribución modular para la definición de esta infraestructura, con el fin de la organización y la reutilización de cada módulo, donde cada componente tiene su definición de main, output y variables. Esto nos sirve porque podemos usar el mismo módulo de "clúster" para crear otro entorno de staging y production con solo cambiar las variables de entrada. Precisamente, lo que se hace es que cuando se definen los archivos como *terraform.devops.tfvars, terraform.prod.tfvars y terraform.staging.tfvars*, se usan como piezas clave para gestionar múltiples entornos.





- Ejemplo de la estructura en Terraform para despliegue de k8s en Google Cloud plataforma

```hcl
# Get available GKE versions
data "google_container_engine_versions" "gke_version" {
  location       = var.region
  version_prefix = var.gke_version_prefix
}

# Create GKE cluster
resource "google_container_cluster" "primary" {
  name                     = var.name
  location                 = var.region
  node_locations           = var.node_locations
  remove_default_node_pool = true
  initial_node_count       = 1
  network    = var.vpc_name
  subnetwork = var.subnet_name

  # Set GKE version to avoid version issues
  min_master_version = data.google_container_engine_versions.gke_version.latest_master_version

  # Use unspecified release channel for better control
  release_channel {
    channel = "UNSPECIFIED"
  }

  # Private cluster configuration
  private_cluster_config {
    enable_private_nodes    = var.enable_private_nodes
    enable_private_endpoint = var.enable_private_endpoint
    master_ipv4_cidr_block  = var.master_ipv4_cidr_block
  }

  # IP allocation policy for pods and services
  ip_allocation_policy {
    cluster_secondary_range_name  = var.pods_range_name
    services_secondary_range_name = var.services_range_name
  }
```

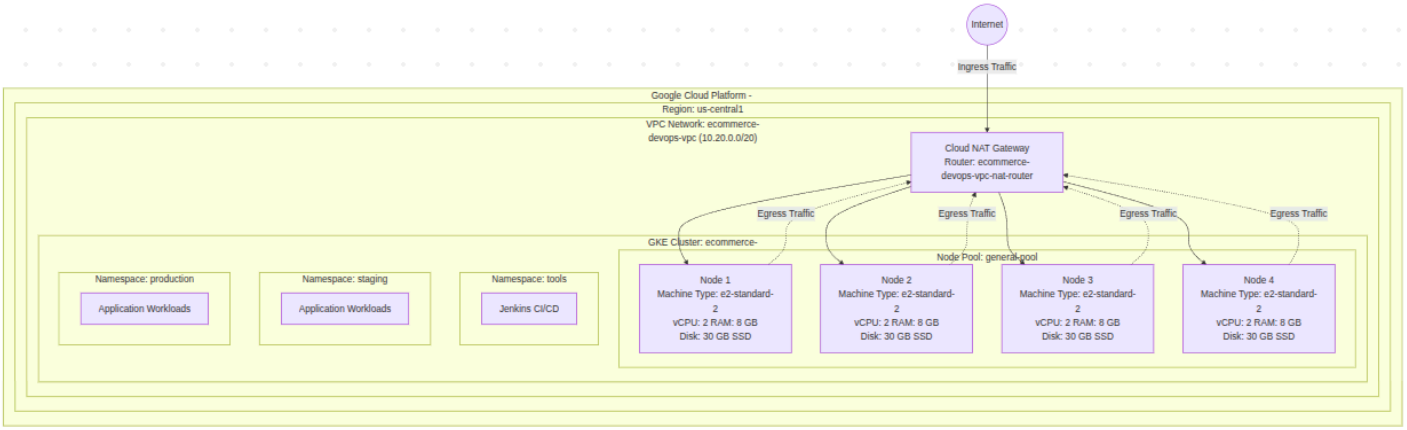
```hcl
# Private cluster configuration
private_cluster_config {
  enable_private_nodes    = var.enable_private_nodes
  enable_private_endpoint = var.enable_private_endpoint
  master_ipv4_cidr_block  = var.master_ipv4_cidr_block
}

# IP allocation policy for pods and services
ip_allocation_policy {
  cluster_secondary_range_name  = var.pods_range_name
  services_secondary_range_name = var.services_range_name
}

# Default node configuration
node_config {
  disk_type    = "pd-standard"
  disk_size_gb = 20
}

# Logging and monitoring
logging_service    = "logging.googleapis.com/kubernetes"
monitoring_service = "monitoring.googleapis.com/kubernetes"

# Workload Identity
workload_identity_config {
  workload_pool = "${var.project_id}.svc.id.goog"
}

# Network policy
network_policy {
  enabled = true
}
```

```
  # Addons
  addons_config {
    http_load_balancing {
      disabled = false
    }

    horizontal_pod_autoscaling {
      disabled = false
    }

    network_policy_config {
      disabled = false
    }
  }

  # Enable protection against accidental cluster deletion.
  # This will be temporarily disabled to allow the cluster to be recreated.
  deletion_protection = false
}
```

terraform.staging.tfvars > credentials_file

```
1   credentials_file = "terraform-sa-key.json"
2   project_id       = "ecommerce-backend-1760307199"
3   region           = "us-central1"
4   node_locations   = ["us-central1-b"]
5   repo_name        = "ecommerce-staging"
6   # CIDR ranges for staging environment
7   subnet_cidr      = "10.10.0.0/20"
8   pods_cidr        = "10.11.0.0/16"
9   services_cidr    = "10.12.0.0/20"
10
11  node_pools = {
12    core       = 1
13    backend    = 2
14    database   = 1
15    monitoring = 1
16  }
17
18  namespaces = [
19    "core",
20    "backend",
21    "database",
22    "monitoring"
23  ]
```

- **Arquitectura de la infraestructura:**



Esta infraestructura está desplegada en Google Cloud Platform y diseñada para ejecutar aplicaciones en contenedores de forma segura y escalable.

El componente central es un **Clúster de Google Kubernetes Engine (GKE)**, el cual está alojado dentro de una red privada virtual (**VPC Network**) y una subred (**Subnet**) con el rango de IPs 10.x.0.0/20. Esta configuración aísla los recursos de la red pública, mejorando la seguridad.

El clúster está compuesto por un único Node Pool, que es un conjunto de máquinas virtuales donde se ejecutan los Pods (las aplicaciones en contenedores). Esta segmentación permite optimizar costos y rendimiento asignando diferentes tipos de máquinas a distintas cargas de trabajo.

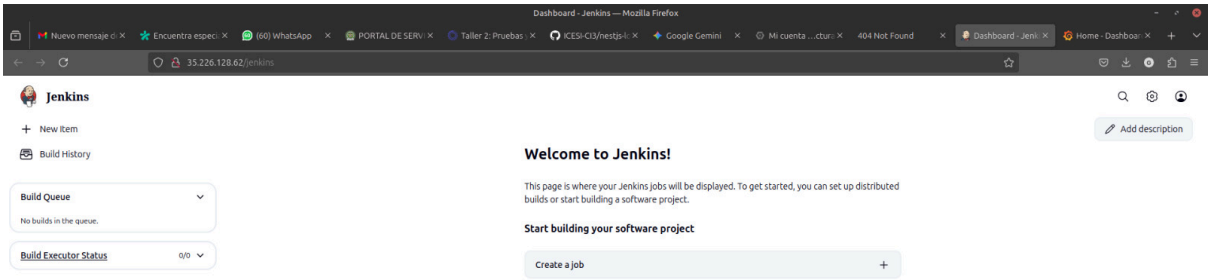
Para la conectividad externa, la arquitectura utiliza:

- Un **NAT Gateway**, que permite a los pods sin IP pública iniciar conexiones salientes a Internet (por ejemplo, para descargar dependencias o contactar APIs externas).
- Un **Cloud Router**, que gestiona el enrutamiento dinámico del tráfico entre la VPC y redes externas, asegura que la comunicación sea eficiente y esté correctamente dirigida.

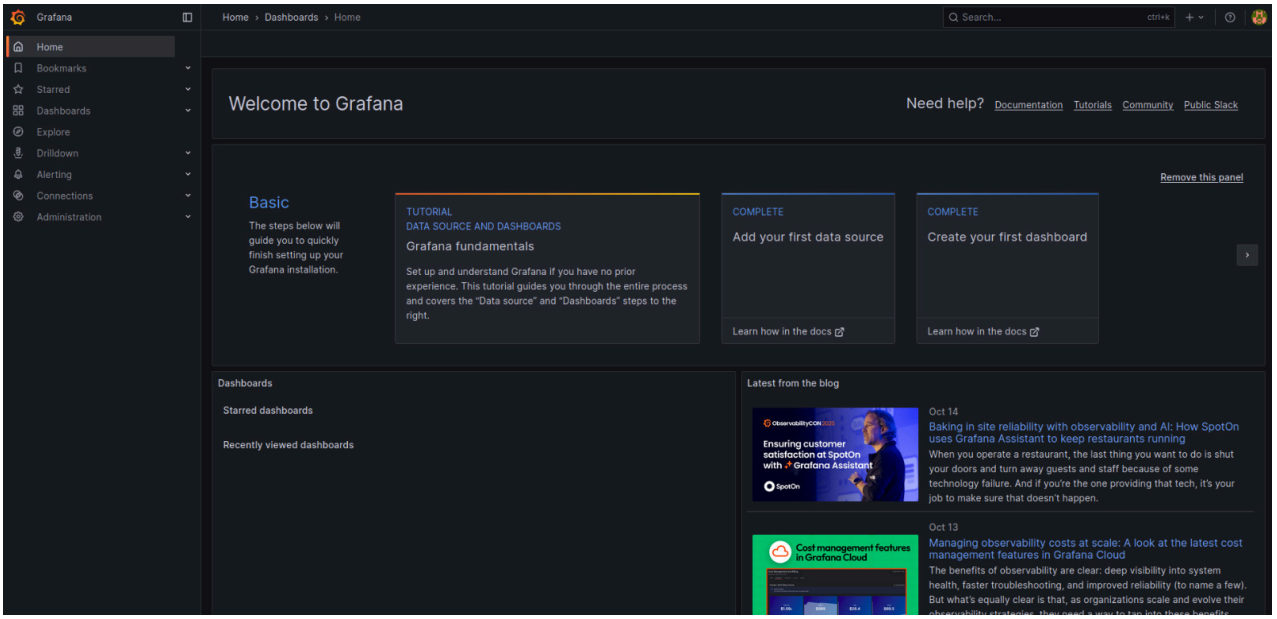
1. Despliegue del servidor de Jenkins dentro del clúster de Kubernetes para probarlo en el env de staging usando Helm y Helmfiles

```
$ cd /home/geoffrey0pv/Projects/2025-2/Ingeosft5/k8s-apps &&
kubectl get pods -n tools
NAME              READY   STATUS    RESTARTS   AGE
jenkins-0         2/2     Running   0          14h
redis-master-0    1/1     Running   0          15h
redis-replicas-0  1/1     Running   0          15h
redis-replicas-1  1/1     Running   0          15h
redis-replicas-2  1/1     Running   0          15h
```

2. Como previamente configuramos una Nat Gateway, a través de esta nuestros pods son accesibles por internet, como podemos observar en la imagen, hemos accedido a nuestra pods donde se está ejecutando Jenkins a través de la siguiente dirección: http://jenkins.35.226.128.62.nip.io/login?from=%2F

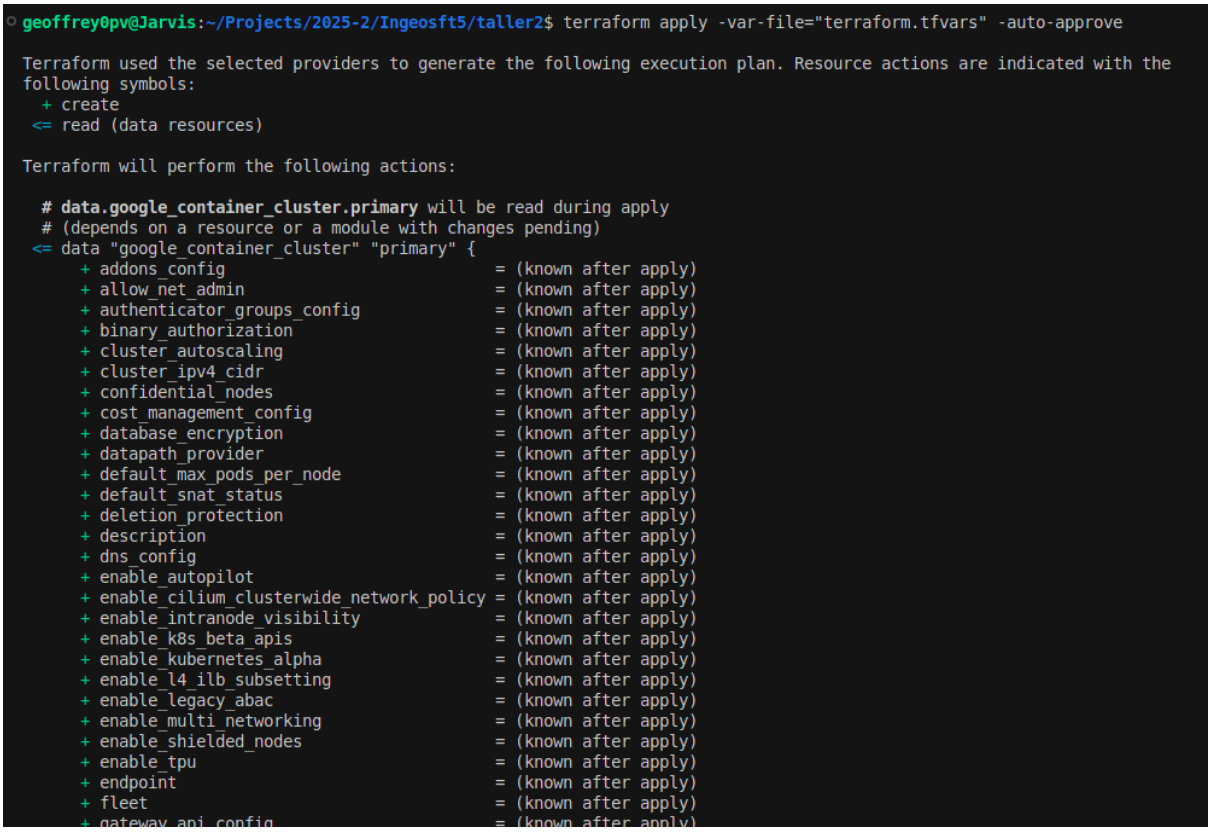


3. Igualmente a través de la Nat Gateway podemos acceder a nuestro servidor de grafana a través de http://grafana.35.226.128.62.nip.io/bookmarks.



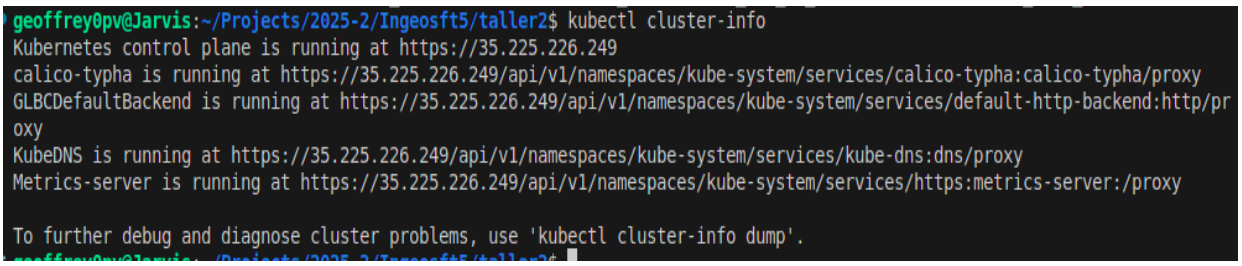
- **Montamos la infraestructura en GKE**

Usamos el comando terraform apply usando las tfvars definidas previamente



- **Verificamos el cluster:**

Se usa *kubectl cluster-info* para obtener información del clúster desplegado. Este comando te muestra información general del clúster y confirma que kubectl está correctamente configurado.



Aquí usamos kubectl get pods -n kube-system, este comando lista los pods que corren en el namespace kube-system, que son el cerebro de Kubernetes (servidor de API, DNS, etc.). Todos deberían estar en estado **Running**.

- **Pods del sistema.**

```
geoffrey0pv@Jarvis:~/Projects/2025-2/Ingeosft5/taller2$ kubectl get pods -n kube-system
NAME                                                              READY   STATUS    RESTARTS   AGE
calico-node-4lvtz                                                 1/1     Running   0          37m
calico-node-rnwfc                                                 1/1     Running   0          37m
calico-node-vertical-autoscaler-bdf57b85-cpvgt                    1/1     Running   0          43m
calico-node-x4n2p                                                 1/1     Running   0          37m
calico-node-zdt5l                                                 1/1     Running   0          37m
calico-typha-57c87b9485-gmgrv                                     1/1     Running   0          43m
calico-typha-57c87b9485-pbv5v                                     1/1     Running   0          36m
calico-typha-horizontal-autoscaler-75c45d987d-tggr6               1/1     Running   0          43m
calico-typha-vertical-autoscaler-b66f4686-pjqsn                   1/1     Running   0          43m
event-exporter-gke-6f5dbb5cc-gkj9n                                2/2     Running   0          43m
fluentbit-gke-67c9k                                               3/3     Running   0          37m
fluentbit-gke-n7ppc                                               3/3     Running   0          37m
fluentbit-gke-splxf                                               3/3     Running   0          37m
fluentbit-gke-spv9l                                               3/3     Running   0          37m
gke-metadata-server-gh928                                         1/1     Running   0          37m
gke-metadata-server-k2kd9                                         1/1     Running   0          37m
gke-metadata-server-wfsrs                                         1/1     Running   0          37m
gke-metadata-server-zbzdf                                         1/1     Running   0          37m
gke-metrics-agent-dtfkn                                           2/2     Running   0          37m
gke-metrics-agent-lhdb6                                           2/2     Running   0          37m
gke-metrics-agent-z2hc8                                           2/2     Running   0          37m
gke-metrics-agent-zrdnt                                           2/2     Running   0          37m
ip-masq-agent-g4hp5                                               1/1     Running   0          37m
ip-masq-agent-jq4tt                                               1/1     Running   0          37m
ip-masq-agent-qxww4                                               1/1     Running   0          37m
ip-masq-agent-x2chj                                               1/1     Running   0          37m
konnectivity-agent-autoscaler-587779c95c-7qmv9                    1/1     Running   0          43m
konnectivity-agent-c5cc4d449-7jzl9                                2/2     Running   0          43m
konnectivity-agent-c5cc4d449-8d6nm                                2/2     Running   0          36m
konnectivity-agent-c5cc4d449-8rjxw                                2/2     Running   0          36m
konnectivity-agent-c5cc4d449-dsn27                                2/2     Running   0          36m
kube-dns-autoscaler-9c49dc997-szvlq                               1/1     Running   0          43m
kube-dns-fc4984b87-bhkjd                                          4/4     Running   0          43m
kube-dns-fc4984b87-kpqxt                                          4/4     Running   0          36m
kube-proxy-gke-ecommerce-devops-general-pool-poo-d16e28bd-1wjg    1/1     Running   0          37m
kube-proxy-gke-ecommerce-devops-general-pool-poo-d16e28bd-825x    1/1     Running   0          37m
kube-proxy-gke-ecommerce-devops-general-pool-poo-d16e28bd-snl7    1/1     Running   0          37m
```

- **Nodos**

```
geoffrey0pv@Jarvis:~/Projects/2025-2/Ingeosft5/taller2$ kubectl get nodes -o wide
NAME                                                STATUS   ROLES    AGE   VERSION               INTERNAL-IP   EXTERNAL-
IP   OS-IMAGE                             KERNEL-VERSION   CONTAINER-RUNTIME
gke-ecommerce-devops-general-pool-poo-d16e28bd-1wjg   Ready    <none>   29m   v1.32.9-gke.1207000   10.20.0.3     <none>
     Container-Optimized OS from Google   6.6.105+        containerd://1.7.28
gke-ecommerce-devops-general-pool-poo-d16e28bd-825x   Ready    <none>   29m   v1.32.9-gke.1207000   10.20.0.5     <none>
     Container-Optimized OS from Google   6.6.105+        containerd://1.7.28
gke-ecommerce-devops-general-pool-poo-d16e28bd-snl7   Ready    <none>   29m   v1.32.9-gke.1207000   10.20.0.6     <none>
     Container-Optimized OS from Google   6.6.105+        containerd://1.7.28
gke-ecommerce-devops-general-pool-poo-d16e28bd-wc9g   Ready    <none>   29m   v1.32.9-gke.1207000   10.20.0.4     <none>
     Container-Optimized OS from Google   6.6.105+        containerd://1.7.28
geoffrey0pv@Jarvis:~/Projects/2025-2/Ingeosft5/taller2$
```

Nodos: 4 (todos en estado Ready)
Tipo de máquina: e2-standard-2
vCPU por nodo: 2 cores
RAM por nodo: 8 GB
Disco por nodo: 30 GB SSD
Total cluster: 8 vCPUs, 32 GB RAM, 120 GB storage

- **Namespaces**

```
geoffrey0pv@Jarvis:~/Projects/2025-2/Ingeosft5/taller2$ kubectl get namespaces
NAME                          STATUS   AGE
default                       Active   39m
gke-managed-cim               Active   37m
gke-managed-system            Active   37m
gke-managed-volumepopulator   Active   37m
gmp-public                    Active   37m
gmp-system                    Active   37m
kube-node-lease               Active   39m
kube-public                   Active   39m
kube-system                   Active   39m
production                    Active   89s
staging                       Active   89s
tools                         Active   89s
```

tools - Para Jenkins y CI/CD
staging - Para entorno de pruebas
production - Para entorno de producción

- **Resumen completo de servicios creados por terraform**

```
geoffrey0pv@Jarvis:~/Projects/2025-2/Ingeosft5/taller2$ terraform output
cluster_ca_certificate = <sensitive>
cluster_endpoint = <sensitive>
cluster_location = "us-central1"
cluster_name = "ecommerce-devops-cluster"
created_namespaces = [
  "production",
  "staging",
  "tools",
]
environment = "default"
kubectl_connection_command = "gcloud container clusters get-credentials ecommerce-devops-cluster --region us-central1 --pro
ject ecommerce-backend-1760307199"
network_id = "projects/ecommerce-backend-1760307199/global/networks/ecommerce-devops-vpc"
network_name = "ecommerce-devops-vpc"
project_id = "ecommerce-backend-1760307199"
region = "us-central1"
subnet_id = "projects/ecommerce-backend-1760307199/regions/us-central1/subnetworks/ecommerce-devops-subnet"
subnet_name = "ecommerce-devops-subnet"
geoffrey0pv@Jarvis:~/Projects/2025-2/Ingeosft5/taller2$
```

Nombre: ecommerce-devops-cluster
Versión: 1.32.9-gke.1207000
Master IP: 35.225.226.249
Región: us-central1
Zona: us-central1-c
Estado: RUNNING

- **Recursos de red**

```
geoffrey0pv@Jarvis:~/Projects/2025-2/Ingeosft5/taller2$ gcloud compute networks list --project=ecommerce-backend-1760307199
  | grep ecommerce
ecommerce-devops-vpc  CUSTOM       REGIONAL
geoffrey0pv@Jarvis:~/Projects/2025-2/Ingeosft5/taller2$ gcloud compute routers nats list --router=ecommerce-devops-vpc-nat-
router --router-region=us-central1 --project=ecommerce-backend-1760307199
NAME                          NAT_IP_ALLOCATE_OPTION  SOURCE_SUBNETWORK_IP_RANGES_TO_NAT  ENDPOINT_TYPES
ecommerce-devops-vpc-nat-gateway  AUTO_ONLY           ALL_SUBNETWORKS_ALL_IP_RANGES       ENDPOINT_TYPE_VM
```

VPC: ecommerce-devops-vpc (CUSTOM, REGIONAL)
Subnet: ecommerce-devops-subnet (10.20.0.0/20)
Pod Range: 10.21.0.0/16
Service Range: 10.22.0.0/20
NAT Gateway: ecommerce-devops-vpc-nat-gateway (AUTO_ONLY)

Router: ecommerce-devops-vpc-nat-router
Firewall Rules: 2 (allow-internal, allow-ssh)

- Pusheamos las imágenes de Docker al Artifact Registry de Google Cloud

```
geoffrey0pv@Jarvis:~/Projects/2025-2/Ingeosft5/ecommerce-microservice-backend-app$ docker push us-central1-docker
.pkg.dev/ecommerce-backend-1760307199/ecommerce-microservices/user-service:v1.7 & docker push us-central1-docker.
pkg.dev/ecommerce-backend-1760307199/ecommerce-microservices/product-service:v1.7 & docker push us-central1-docke
r.pkg.dev/ecommerce-backend-1760307199/ecommerce-microservices/order-service:v1.7 & wait
[1] 15706
[2] 15707
[3] 15708
The push refers to repository [us-central1-docker.pkg.dev/ecommerce-backend-1760307199/ecommerce-microservices/or
der-service]
The push refers to repository [us-central1-docker.pkg.dev/ecommerce-backend-1760307199/ecommerce-microservices/pr
oduct-service]
The push refers to repository [us-central1-docker.pkg.dev/ecommerce-backend-1760307199/ecommerce-microservices/us
er-service]
ec38513fedde: Preparing
8e25251b50f1: Preparing
5f70bf18a086: Preparing
262ddcac4352: Preparing
7b7f3078e1db: Preparing
```

- Historial de imágenes de Docker de nuestro microservicio user-services, subidas al Pkg de Google Cloud cada que hagamos un cambio, GCP, cuando haya una versión de nuestras imágenes, por si tenemos que hacer rollback a una imagen anterior.

```
geoffrey0pv@Jarvis:~/Projects/2025-2/Ingeosft5/ecommerce-microservice-backend-app$ gcloud container images list-
tags us-central1-docker.pkg.dev/ecommerce-backend-1760307199/ecommerce-microservices/user-service --limit=10
DIGEST        TAGS  TIMESTAMP
745f919d1f6b  v1.7  2025-10-19T10:42:19
486c50e5c6ea  v1.5  2025-10-18T13:58:16
aeb822123189  v1.4  2025-10-18T11:06:38
b83df4e69371  v1.3  2025-10-17T22:10:29
050c8b173efa  v1.2  2025-10-17T21:53:04
b8527ca2f611  v1.1  2025-10-17T19:53:12
f60ac8cb3e49  v1.0  2025-10-17T10:43:25
d6cb1c5a2e37        2025-10-17T08:29:29
```

- Desplegar microservicios usando helm upgrade para aplicar todos los manifiestos empaquetados por nuestro gestor de paquetes manifiestos

```
geoffrey0pv@Jarvis:~/Projects/2025-2/Ingeosft5/ecommerce-microservice-backend-app$ helm upgrade shipping-
service manifests-gcp/shipping-service/ \
>     --namespace staging \
>     --set image.tag=latest-dev \
>     --wait --timeout=5m
Release "shipping-service" has been upgraded. Happy Helming!
NAME: shipping-service
LAST DEPLOYED: Sat Nov  1 21:47:43 2025
NAMESPACE: staging
STATUS: deployed
REVISION: 2
TEST SUITE: None
geoffrey0pv@Jarvis:~/Projects/2025-2/Ingeosft5/ecommerce-microservice-backend-app$ helm upgrade --install
 favourite-service manifests-gcp/favourite-service/ \
>     --namespace staging \
>     --set image.tag=latest-dev \
>     --wait --timeout=5m
Release "favourite-service" does not exist. Installing it now.
NAME: favourite-service
LAST DEPLOYED: Sat Nov  1 21:52:40 2025
NAMESPACE: staging
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

```
TEST SUITE: None
geoffrey0pv@Jarvis:~/Projects/2025-2/Ingeosft5/ecommerce-microservice-backend-app$ helm upgrade --install
 favourite-service manifests-gcp/favourite-service/ \
>     --namespace staging \
>     --set image.tag=latest-dev \
>     --wait --timeout=5m
Release "favourite-service" does not exist. Installing it now.
NAME: favourite-service
LAST DEPLOYED: Sat Nov  1 21:52:40 2025
NAMESPACE: staging
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

- Verificamos el estado de nuestros microservicios desplegados en el entorno de staging (namespace staging) visualizando los pods running.

```
geoffrey0pv@Jarvis:~/Projects/2025-2/Ingeosft5/ecommerce-microservice-backend-app$ kubectl get pods -n staging
NAME                                 READY   STATUS    RESTARTS   AGE
favourite-service-7d9cbf4ff8-bjqks   1/1     Running   0          22m
order-service-698df98bf5-hmhmc       1/1     Running   0          11m
payment-service-7965845bcd-75v8m     1/1     Running   0          29m
product-service-59f6c97bb5-ztps9     1/1     Running   0          6m9s
shipping-service-74454f7ccd-6nscw    1/1     Running   0          27m
user-service-788b7448cf-zmt45        1/1     Running   0          30h
geoffrey0pv@Jarvis:~/Projects/2025-2/Ingeosft5/ecommerce-microservice-backend-app$
```

Y vemos que tenemos todos nuestros microservicios desplegados en el entorno de staging correctamente funcionando en nuestro cluster de Google Cloud Services

- **Despliegue de microservicios en entorno local usando Minikubs**

```
geoffrey0pv@Jarvis:~/Projects/2025-2/Ingeosft5/ecommerce-microservice-backend-app$ kubectl get pods
NAME                               READY   STATUS    RESTARTS   AGE
discovery-74cbc54fcf-dgmkj         1/1     Running   0          21m
order-service-fb8c5cb75-cvck9      0/1     Running   0          22s
product-service-5cdfd89dc8-mq5nn   0/1     Running   0          31s
user-service-584cbd5b7b-7sh9m      0/1     Running   0          88s
zipkin-5444dd8464-xrb9q            1/1     Running   0          98s
geoffrey0pv@Jarvis:~/Projects/2025-2/Ingeosft5/ecommerce-microservice-backend-app$ sleep 60 && kubectl get pods
NAME                               READY   STATUS    RESTARTS   AGE
discovery-74cbc54fcf-dgmkj         1/1     Running   0          22m
order-service-fb8c5cb75-cvck9      1/1     Running   0          106s
product-service-5cdfd89dc8-mq5nn   1/1     Running   0          115s
user-service-584cbd5b7b-7sh9m      0/1     Running   0          2m52s
zipkin-5444dd8464-xrb9q            1/1     Running   0          3m2s
```

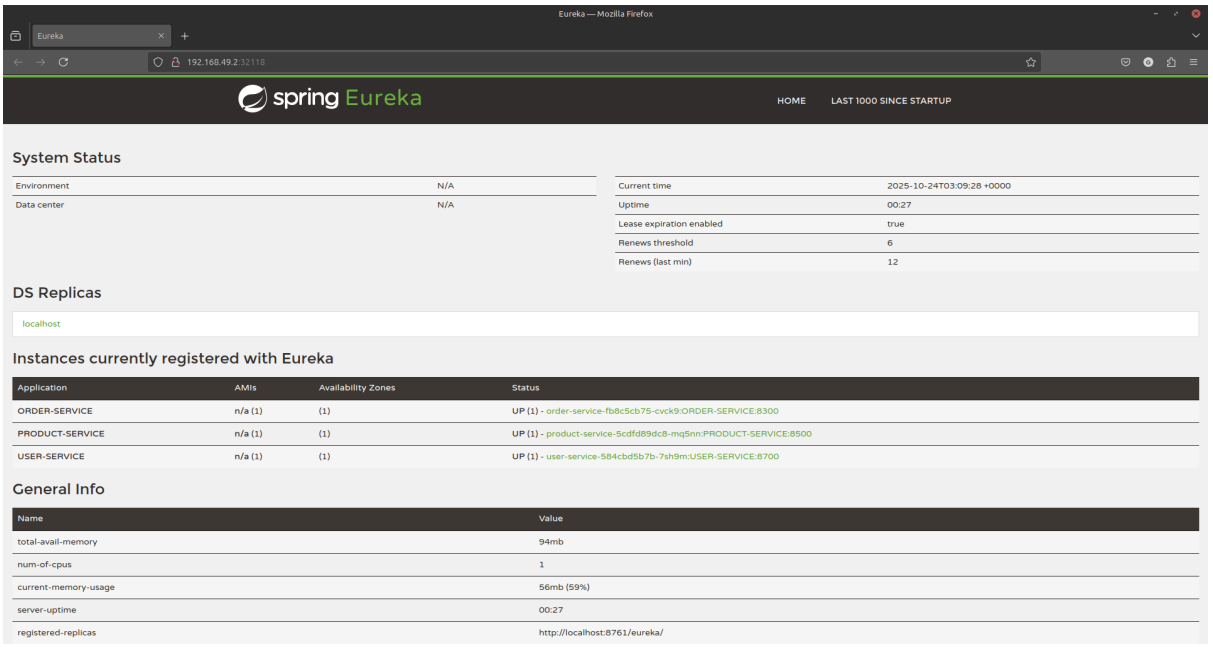
```
geoffrey0pv@Jarvis:~/Projects/2025-2/Ingeosft5/ecommerce-microservice-backend-app$ helm install user-service ./manifests/use
r-service
NAME: user-service
LAST DEPLOYED: Thu Oct 23 22:02:16 2025
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
geoffrey0pv@Jarvis:~/Projects/2025-2/Ingeosft5/ecommerce-microservice-backend-app$ helm install product-service ./manifests/
product-service
NAME: product-service
LAST DEPLOYED: Thu Oct 23 22:03:14 2025
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
geoffrey0pv@Jarvis:~/Projects/2025-2/Ingeosft5/ecommerce-microservice-backend-app$ helm install order-service ./manifests/or
der-service
NAME: order-service
LAST DEPLOYED: Thu Oct 23 22:03:23 2025
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Despliegue y verificación de clúster y contenedores Docker funcionando con Minikube en el local en este caso desplegamos el discovery de eureka y zipkin y 3 microservicios User, Product y Order.

```
geoffrey0pv@Jarvis:~/Projects/2025-2/Ingeosft5/ecommerce-microservice-backend-app$ sleep 60 && kubectl get ingres
s -n staging && echo -e "\n=== Testing endpoints ===" &&  kubectl get ingress -n staging -o json | jq -r '.items[
] | select(.status.loadBalancer.ingress != null) | .metadata.name + ": " + (.status.loadBalancer.ingress[0].ip //
 "pending")'
NAME                CLASS   HOSTS   ADDRESS         PORTS   AGE
favourite-service   gce     *       34.117.8.107    80      21m
order-service       gce     *                       80      15m
payment-service     gce     *       130.211.44.55   80      21m
product-service     gce     *                       80      15m
shipping-service    gce     *       34.120.34.46    80      21m
user-service        gce     *       34.110.151.96   80      15m
```

Ingress controller configurado

```
geoffrey0pv@Jarvis:~/Projects/2025-2/Ingeosft5/ecommerce-microservice-backend-app$ kubectl get svc,ingress -n
  staging
NAME                              TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
service/favourite-service         NodePort   10.22.10.242   <none>        8800:32482/TCP   13h
service/order-service             NodePort   10.22.3.186    <none>        8300:31369/TCP   13h
service/payment-service           NodePort   10.22.2.176    <none>        8400:32394/TCP   14h
service/product-service           NodePort   10.22.2.114    <none>        8500:31647/TCP   13h
service/shipping-service          NodePort   10.22.10.163   <none>        8600:30500/TCP   14h
service/user-service              NodePort   10.22.15.87    <none>        8700:32182/TCP   47h

NAME                                                CLASS   HOSTS   ADDRESS         PORTS   AGE
ingress.networking.k8s.io/favourite-service         gce     *                       80      15m
ingress.networking.k8s.io/order-service             gce     *                       80      9m47s
ingress.networking.k8s.io/payment-service           gce     *       130.211.44.55   80      15m
ingress.networking.k8s.io/product-service           gce     *                       80      9m43s
ingress.networking.k8s.io/shipping-service          gce     *                       80      15m
ingress.networking.k8s.io/user-service              gce     *       34.110.151.96   80      9m51s
```

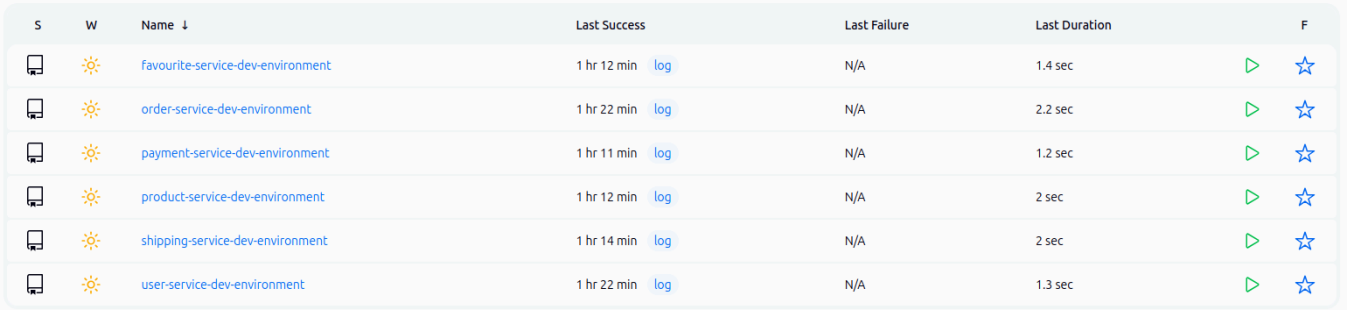


Servidor de eureka accesible y funcional

2. **(15%) Para los microservicios escogidos, debe definir los pipelines que permitan la construcción de la aplicación (dev environment)**

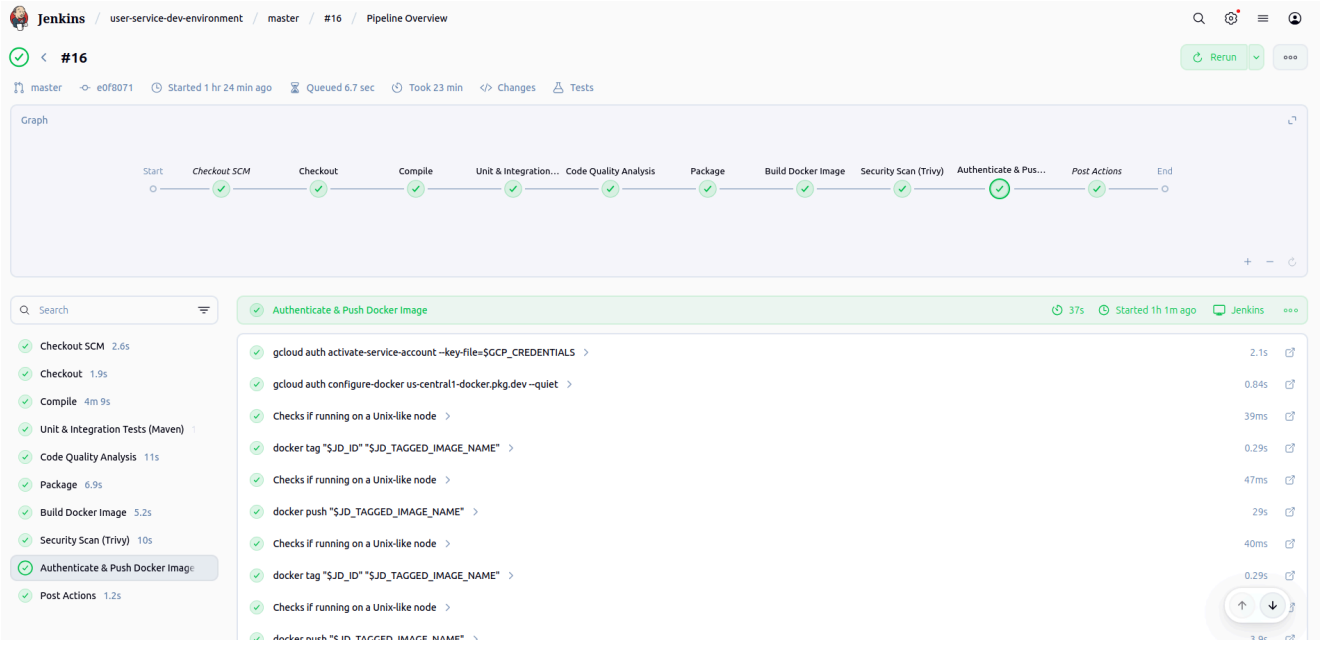
Se implementó una estrategia de CI/CD basada en el patrón '**Build Once, Deploy Many**', la cual garantiza que el mismo artefacto inmutable (imagen Docker) sea construido una sola vez y promovido a través de todos los ambientes sin reconstrucción. Esta estrategia se fundamenta en tres pipelines interconectados: el **Pipeline DEV** se dispara al hacer merge de **feature branches** a **develop** y ejecuta **compilación**, **pruebas unitarias/integración**, **análisis de código con SonarQube**, **escaneo de seguridad con Trivy**, y **construcción de la imagen Docker** etiquetada con el Git Commit SHA; el **Pipeline STAGE** se activa al hacer merge de develop a staging y despliega la **imagen pre-construida** en el clúster de Kubernetes de staging para ejecutar **pruebas dinámicas (E2E, rendimiento, smoke tests)** sin recompilar código; finalmente, el **Pipeline MASTER** se ejecuta al hacer merge de staging a main y promueve la imagen verificada a producción con **smoke tests y generación automática de Release Notes.** Esta arquitectura elimina el riesgo de inconsistencias entre ambientes **(el problema de 'funciona en staging pero falla en producción')**, asegura trazabilidad completa desde el commit hasta producción mediante tags inmutables, reduce el tiempo de despliegue al evitar compilaciones repetidas e implementa múltiples capas de seguridad y calidad que bloquean artefactos vulnerables antes de llegar a producción.

- Pipeline definidas para el entorno de dev.

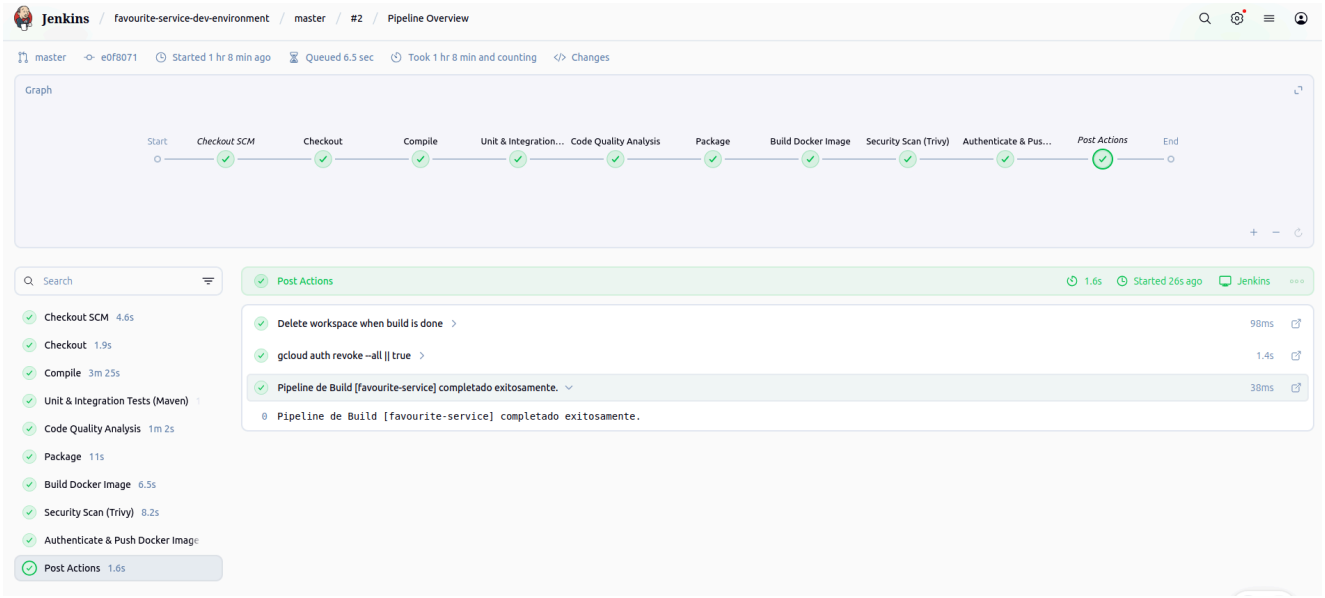


S	W	Name ↓	Last Success	Last Failure	Last Duration	F
		favourite-service-dev-environment	1 hr 12 min log	N/A	1.4 sec	
		order-service-dev-environment	1 hr 22 min log	N/A	2.2 sec	
		payment-service-dev-environment	1 hr 11 min log	N/A	1.2 sec	
		product-service-dev-environment	1 hr 12 min log	N/A	2 sec	
		shipping-service-dev-environment	1 hr 14 min log	N/A	2 sec	
		user-service-dev-environment	1 hr 22 min log	N/A	1.3 sec	

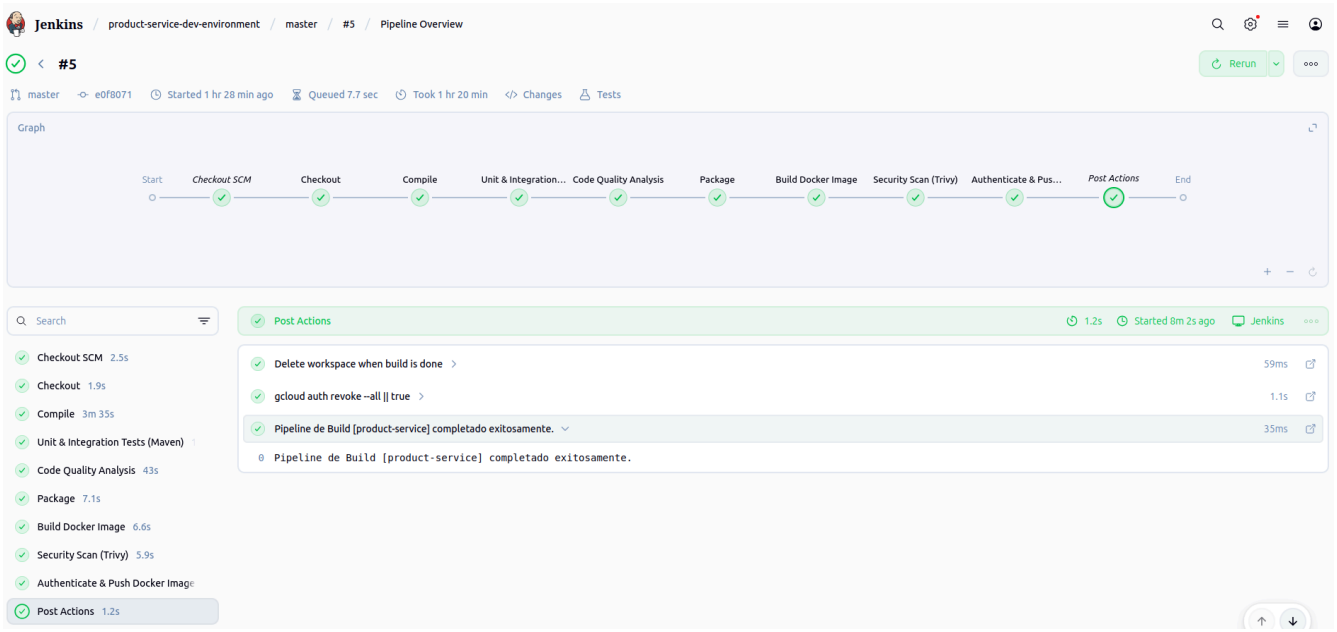
- User Service Pipeline dev environment



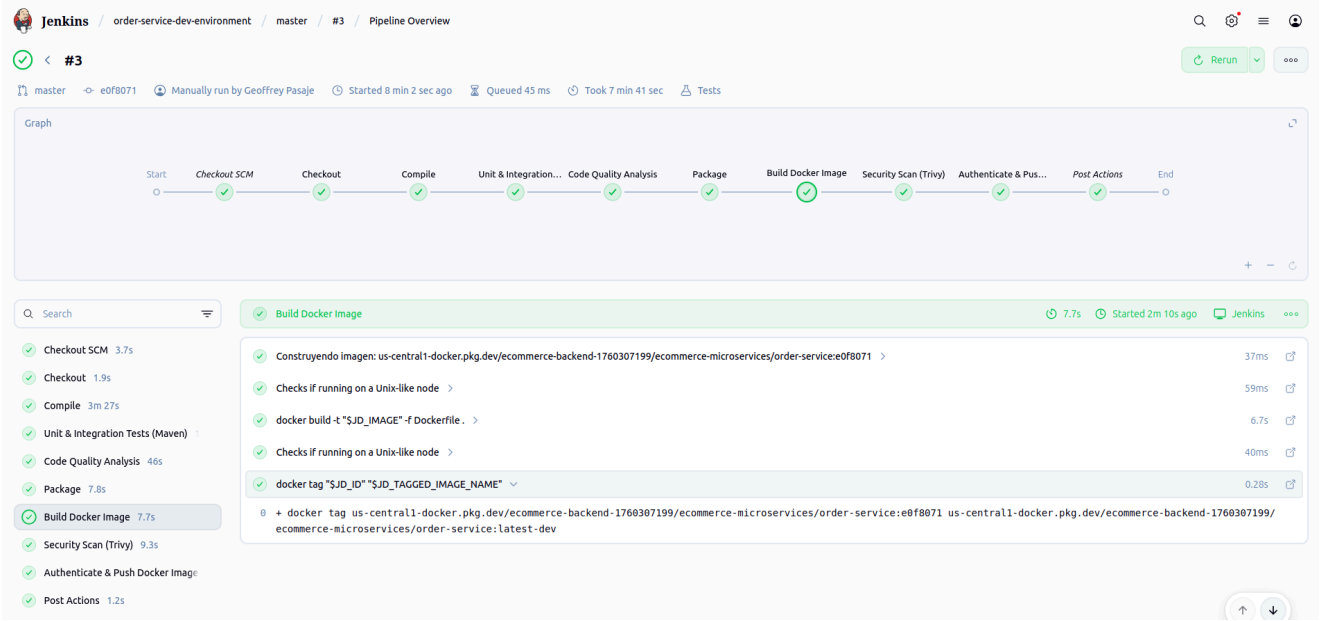
- Favourite Service Pipeline dev environment



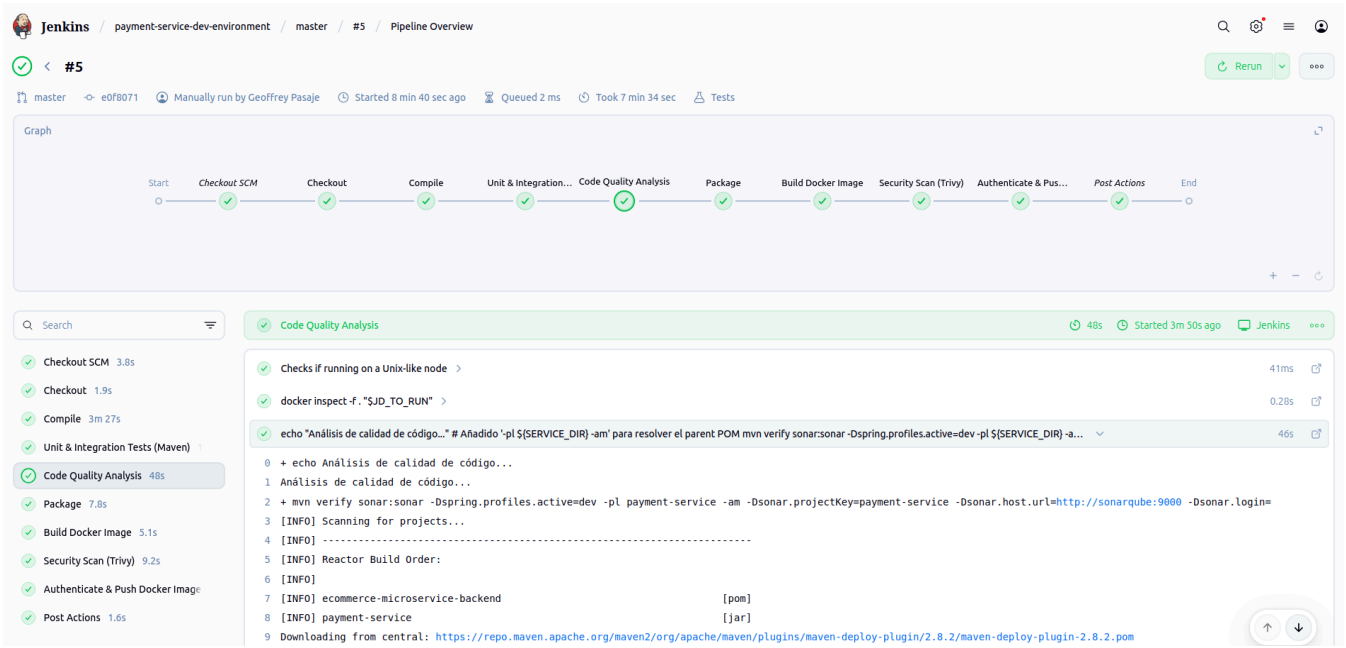
- Product Service Pipeline dev environment



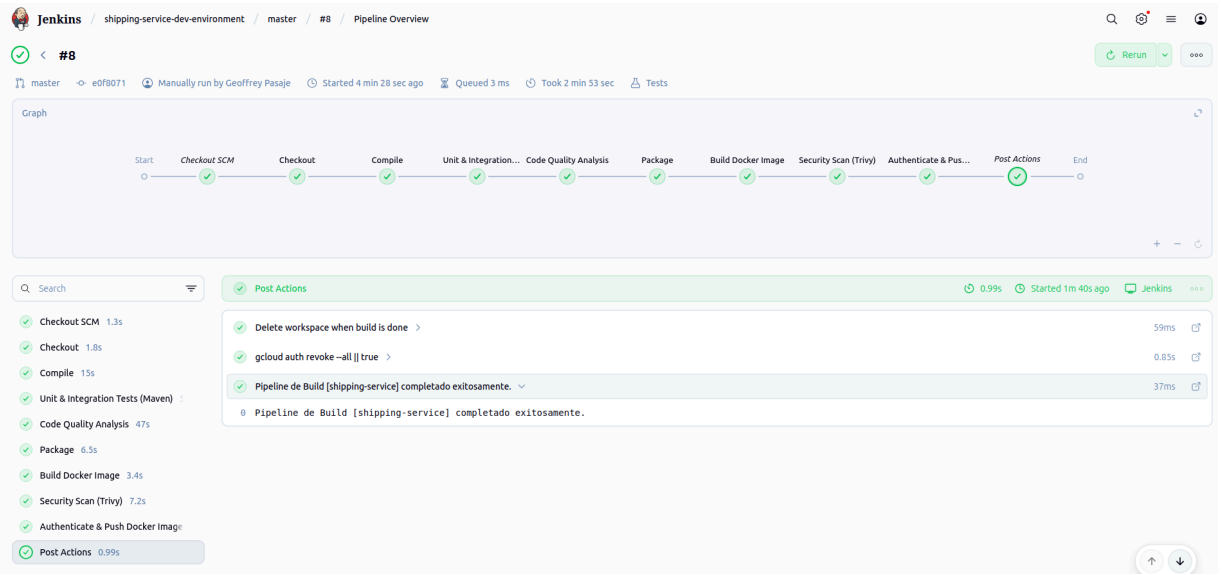
- Order Service Pipeline dev environment



- Payment Service Pipeline dev environment



- Shipping Service Pipeline dev environment



Para la definición de todas las pipelines del entorno de desarrollo usamos las mismas stages y los mismos pasos en cada una, solo cambia la variable de entorno de IMAGE_NAME dependiendo del nombre del servicio y Service Dir, cómo se mencionó en la estrategia anteriormente la idea es ejecutar pruebas de unitarias, de integración de chequeo de calidad de código (usando sonarqube) y de escaneo de seguridad (con Triby) cómo veremos a continuación, crear la imagen y irla promoviendo a través del entorno staging y luego production:

```
pipeline {
    agent any
    environment {
        IMAGE_NAME = "user-service"
        GCR_REGISTRY = "us-central1-docker.pkg.dev/ecommerce-backend-1760307199/ecommerce-microservices"
        SERVICE_DIR = "user-service"
        SPRING_PROFILES_ACTIVE = "dev"
        GCP_CREDENTIALS = credentials('gke-credentials')
    }

    stages {
        stage('Checkout') {
            steps {
                checkout scm
                script {
                    env.GIT_COMMIT_SHA = sh(script: "git rev-parse --short HEAD", returnStdout: true).trim()
                    env.FULL_IMAGE_NAME = "${GCR_REGISTRY}/${IMAGE_NAME}"
                    env.IMAGE_TAG = "${env.GIT_COMMIT_SHA}"
                }
                echo "Procesando Build para ${IMAGE_NAME} commit ${IMAGE_TAG}"
            }
        }

        stage('Compile') {
            steps {
                script {
                    docker.image('maven:3.8.4-openjdk-11').inside {
                        sh '''
                            mvn clean install -N -Dspring.profiles.active=dev
                            cd ${SERVICE_DIR}
                            mvn clean compile -Dspring.profiles.active=dev
                        '''
                    }
                }
            }
        }
```

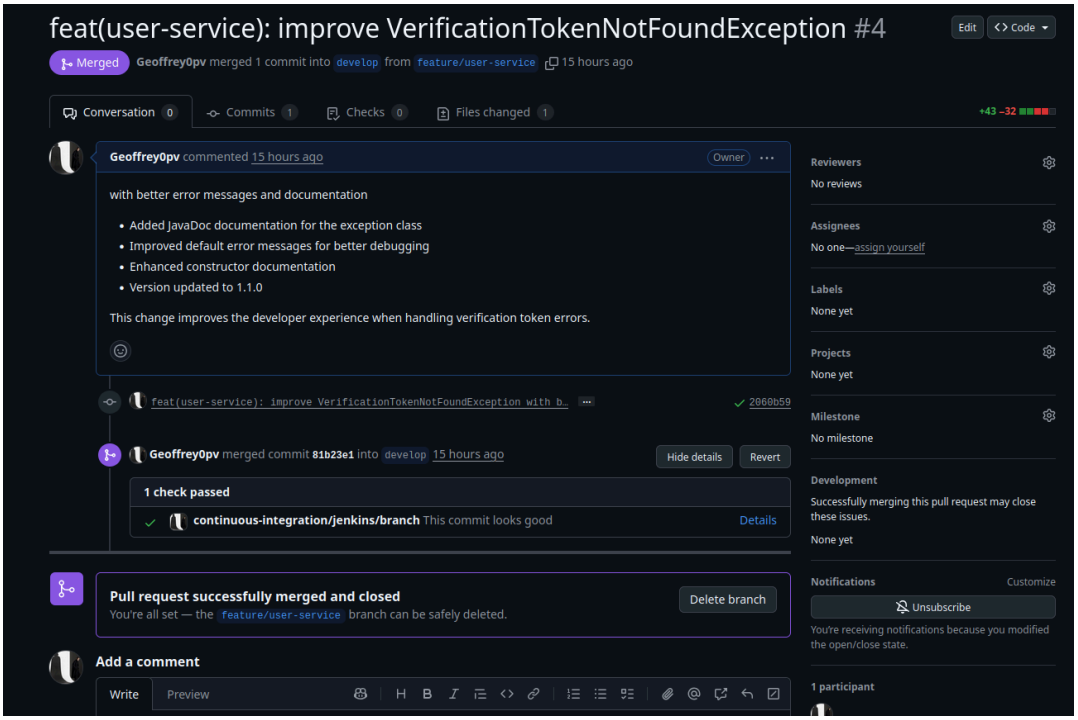
```
stage('Unit & Integration Tests (Maven)') {
    steps {
        script {
            docker.image('maven:3.8.4-openjdk-11').inside {
                // Ejecuta pruebas unitarias Y de integración (corren con Maven)
                sh "mvn verify -Dspring.profiles.active=dev -pl ${SERVICE_DIR} -am"
            }
        }
    }
    post {
        always {
            junit '**/target/surefire-reports/*.xml'
        }
    }
}

stage('Code Quality Analysis') {
    steps {
        dir("${SERVICE_DIR}") {
            script {
                docker.image('maven:3.8.4-openjdk-11').inside {
                    sh '''
                        echo "Análisis de calidad de código..."
                        mvn verify sonar:sonar \
                            -Dsonar.projectKey=${IMAGE_NAME} \
                            -Dsonar.host.url=http://sonarqube:9000 \
                            -Dsonar.login=${SONAR_TOKEN} || echo "SonarQube no configurado"
                    '''
                }
            }
        }
    }
}
```

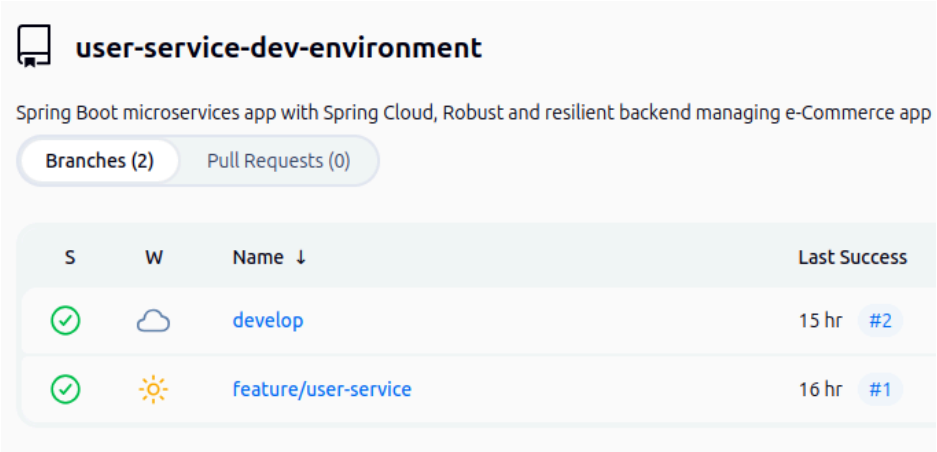
```
stage('Package') {
    steps {
        script {
            docker.image('maven:3.8.4-openjdk-11').inside {
                sh "mvn package -DskipTests=true -Dspring.profiles.active=dev -pl ${SERVICE_DIR} -am"
            }
        }
    }
}

stage('Build Docker Image') {
    steps {
        dir("${SERVICE_DIR}") {
            script {
                echo "Construyendo imagen: ${FULL_IMAGE_NAME}:${IMAGE_TAG}"

                def customImage = docker.build("${FULL_IMAGE_NAME}:${IMAGE_TAG}", "-f Dockerfile .")

                customImage.tag("latest-dev")
            }
        }
    }
}

stage('Security Scan (Trivy)') {
    steps {
        script {
            echo "Escaneando imagen ${FULL_IMAGE_NAME}:${IMAGE_TAG} para vulnerabilidades..."
            sh """
                mkdir -p \$HOME/.trivy/cache
                docker run --rm \
                    -v /var/run/docker.sock:/var/run/docker.sock \
                    -v \$HOME/.trivy/cache:/root/.cache/trivy \
                    aquasec/trivy:latest image \
                    --severity HIGH,CRITICAL \
                    --format table \
                    ${FULL_IMAGE_NAME}:${IMAGE_TAG} || echo "ADVERTENCIA: Vulnerabilidades encontradas pero se permite
            """
        }
    }
}
```

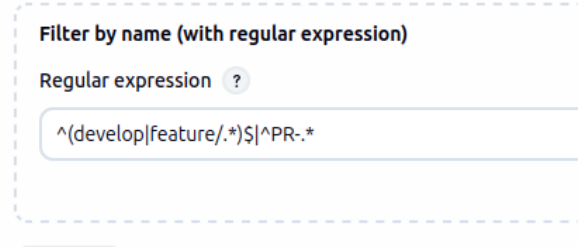
Por último, luego de construir la imagen y se hace el escaneo de seguridad a la imagen, y por último se pushea la imagen al Google Cloud Artifact Registry para que esté disponible para usar en nuestro entorno de staging en el clúster de Kubernetes que teníamos previamente desplegado.

```
    stage('Authenticate & Push Docker Image') {
        steps {
            script {
                // Autenticarse en GCR/Artifact Registry
                sh 'gcloud auth activate-service-account --key-file=$GCP_CREDENTIALS'
                sh 'gcloud auth configure-docker us-central1-docker.pkg.dev --quiet'

                // Subir la imagen
                docker.image("${env.FULL_IMAGE_NAME}:${env.IMAGE_TAG}").push()
                docker.image("${env.FULL_IMAGE_NAME}:latest-dev").push()

                echo "Imagen publicada: ${env.FULL_IMAGE_NAME}:${env.IMAGE_TAG}"
            }
        }
    }
}

post {
    success {
        echo "Pipeline de Build [${IMAGE_NAME}] completado exitosamente."
    }
    always {
        cleanWs()
        // Limpiar credenciales de GCloud
        sh 'gcloud auth revoke --all || true'
    }
    failure {
        echo "Build falló para ${IMAGE_NAME}"
    }
}
```

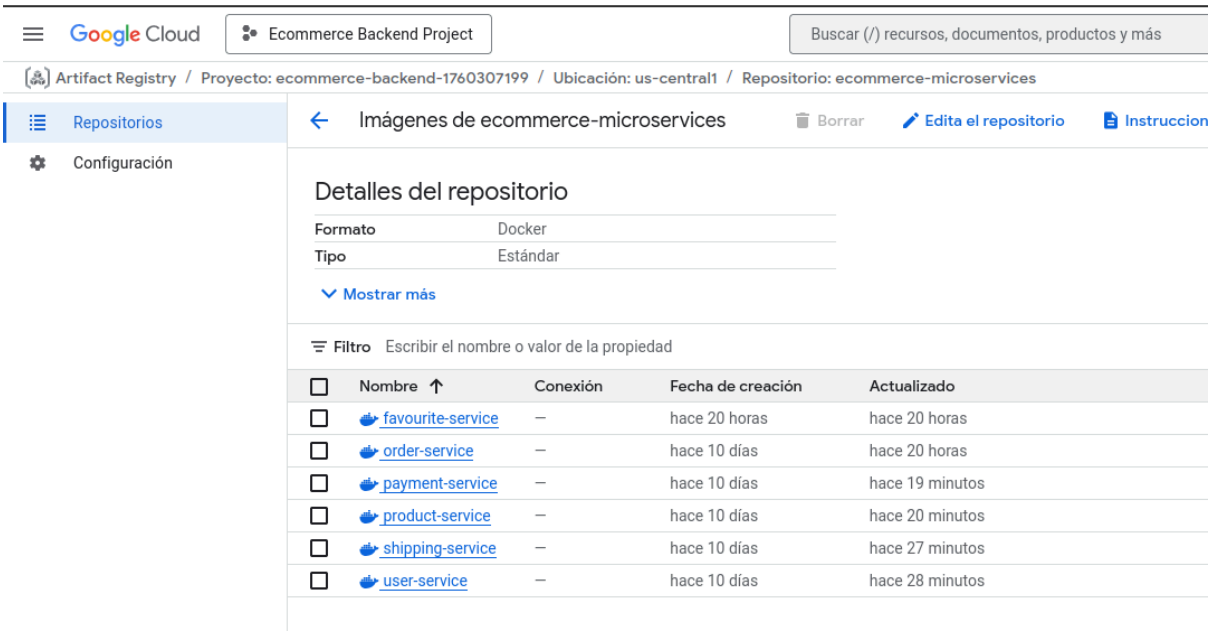
Aquí podemos observar cómo la pipeline de desarrollo paso y podemos hacer el merge luego de hacer el pull request de feature -> develop, al darle a details nos vas a redirigir al servidor de jenkins especificamente al overview de la pipeline de user-service que fue la tomada cómo prueba para este caso







En Jenkins podemos visualizar en la sección de la multibranch-pipeline cómo la rama feature/user-service que hicimos merge con develop está en un estado aceptado (el check en verde), y para que esta pipeline se active cada vez que una feature hace PR a develop, agregamos un behavior de filtro por nombre específicamente una expresión regular.



Aquí podemos visualizar cómo todas nuestras imágenes para cada microservicio fueron cargadas correctamente en el Google Artifact Registry para ser usadas para el entorno de Staging que tenemos definido en uno de los namespaces del cluster de Kubernetes.

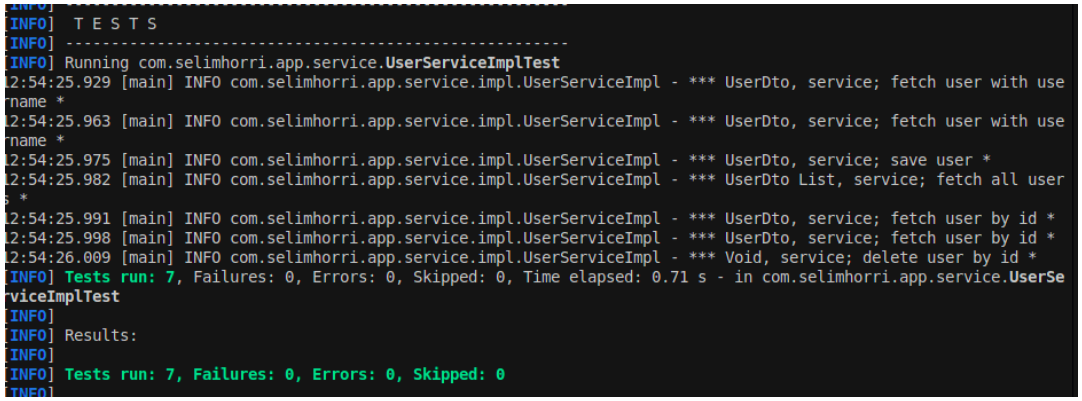
3. **(30%) En algunos de los microservicios, debe definir pruebas unitarias, integración, E2E y rendimiento que involucren los microservicios**

   Aquí primero se van a ejecutar las pruebas de forma local para asegurarnos de que funcionen y posteriormente definirlas en la stage de testing de las pipelines de dev y posteriormente utilizarlas en el entorno de staging.

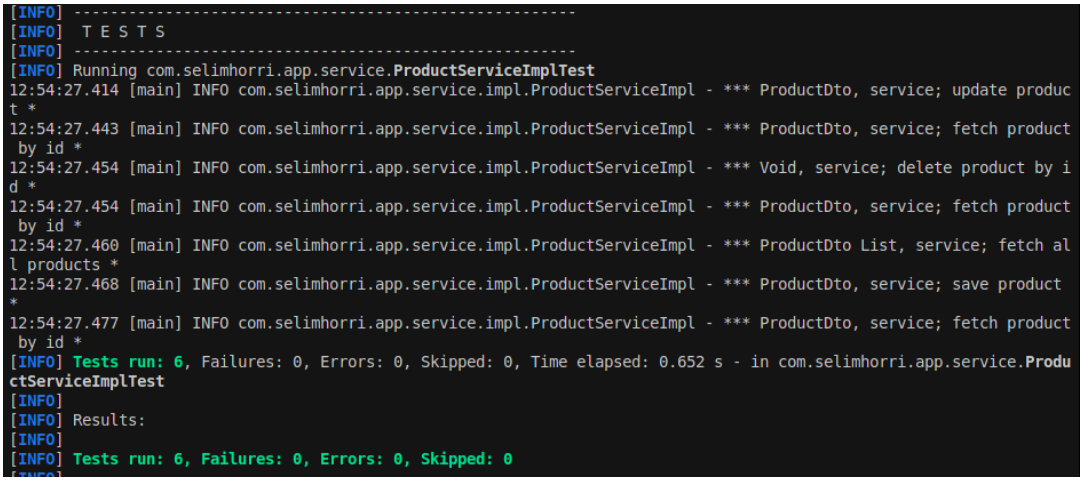
- **Al menos cinco nuevas pruebas unitarias que validen componentes individuales**

  A continuación el comando para ejecutar todas las pruebas unitarias: ***./mvnw test -Dtest="UserServiceImplTest,ProductServiceImplTest,OrderServiceImplTest,PaymentServiceImplTest,OrderItemServiceImplTest,FavouriteServiceImplTest" -pl user-service,product-service,order-service,payment-service,shipping-service,favourite-service***

- *User-Service Unit tests*

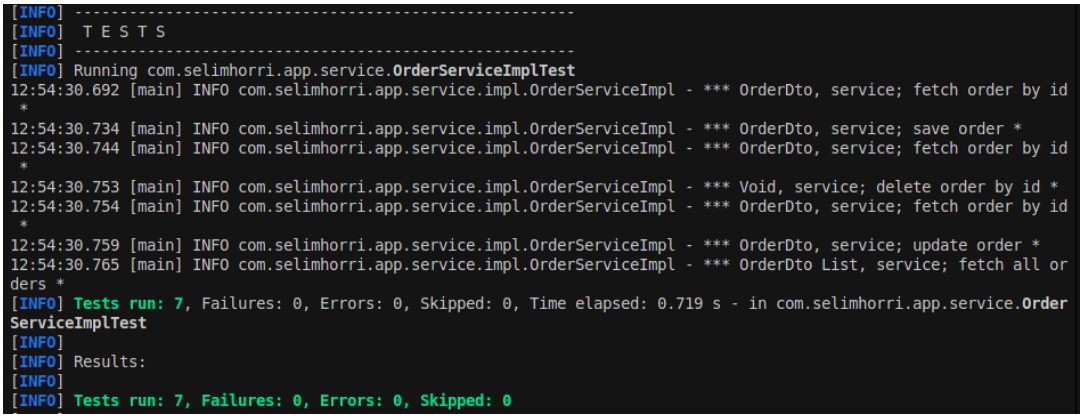


- *Product-Service Unit tests*

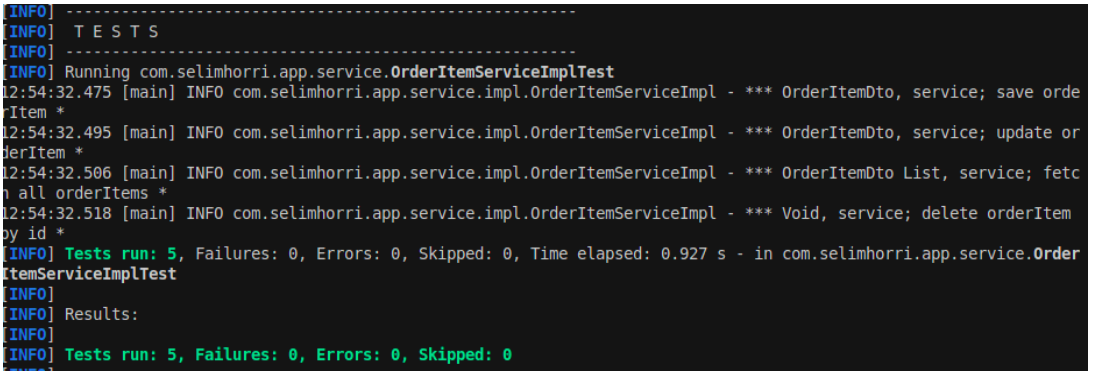


- *Favourite-Service Unit tests*

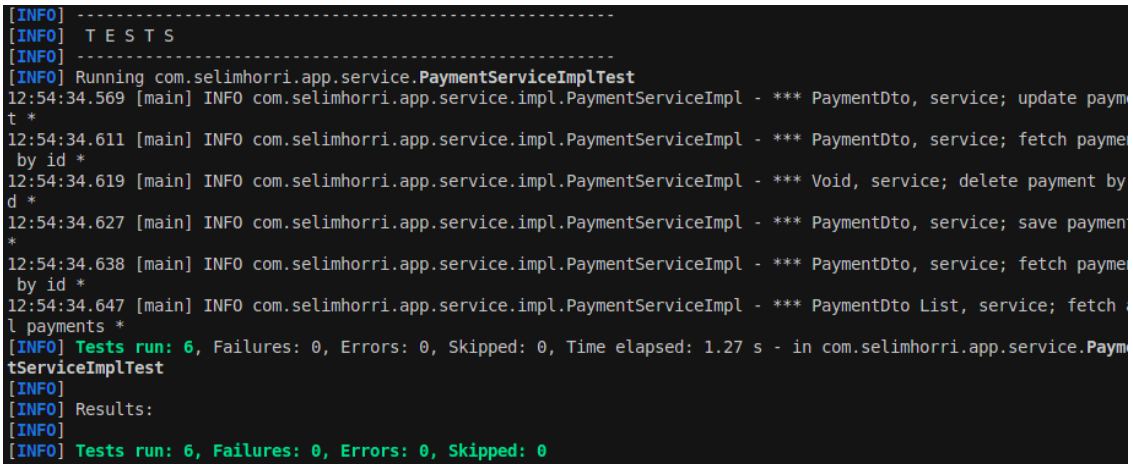
- *Order-Service Unit tests*



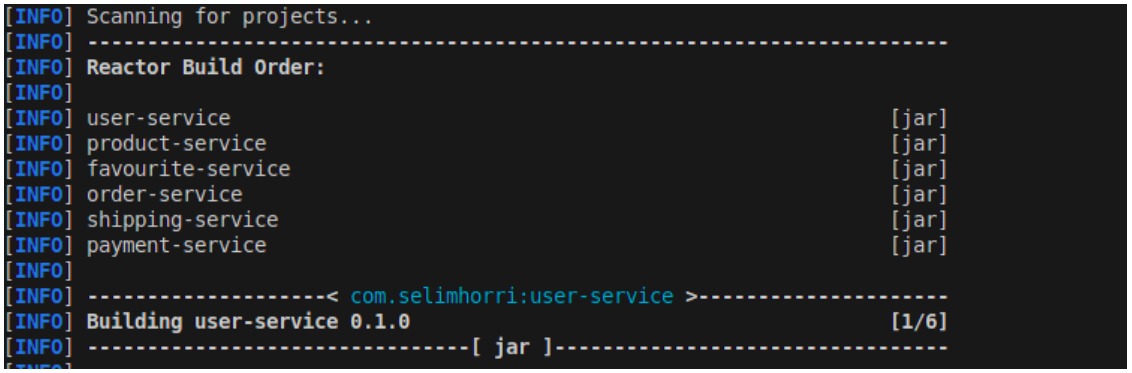
- *Shipping-Service Unit tests*

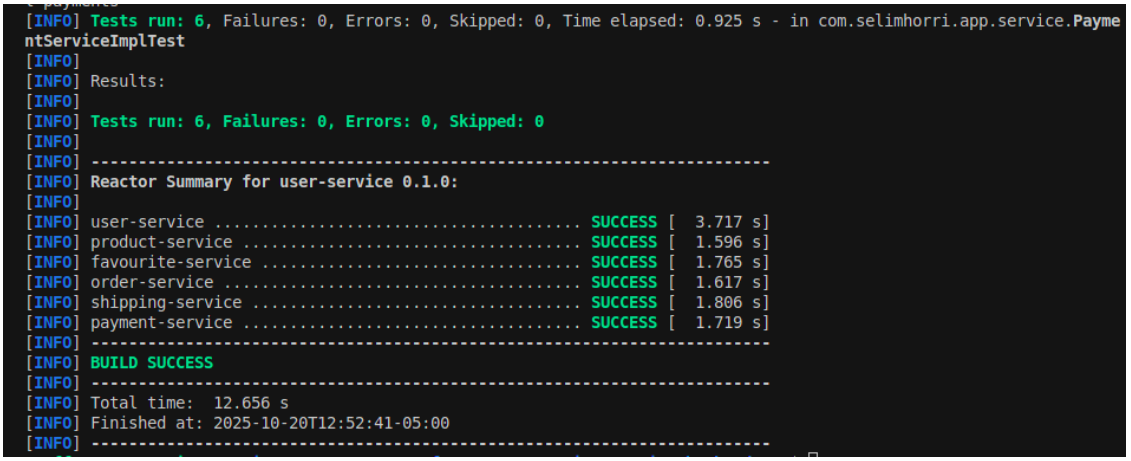


- *Payment-Service Unit tests*



- Resumen general de unit testing para los 6 microservicios.

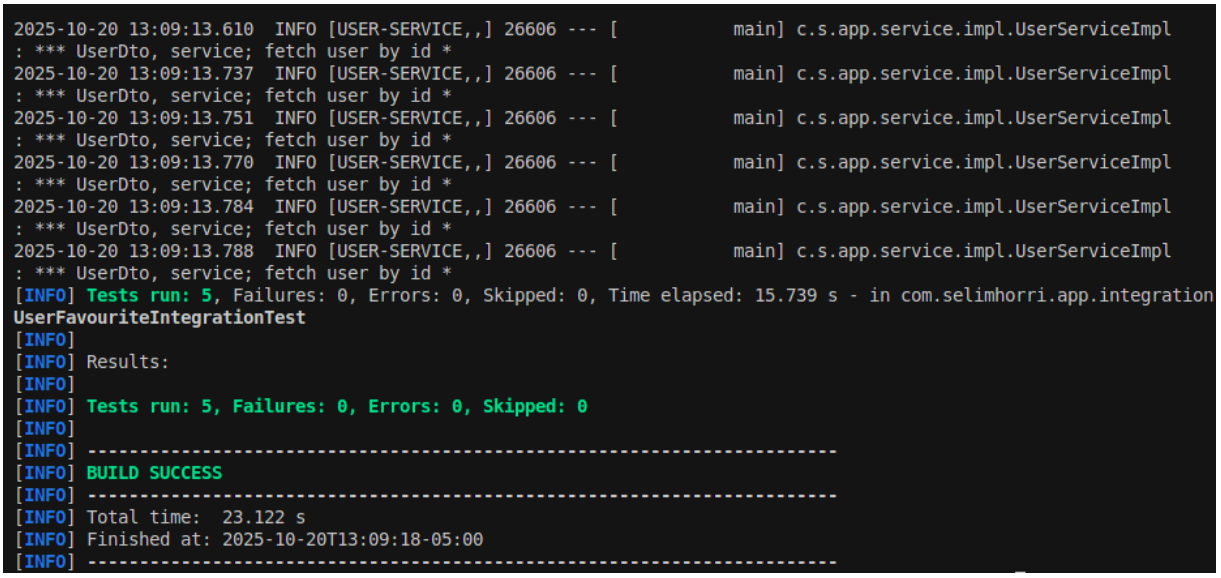




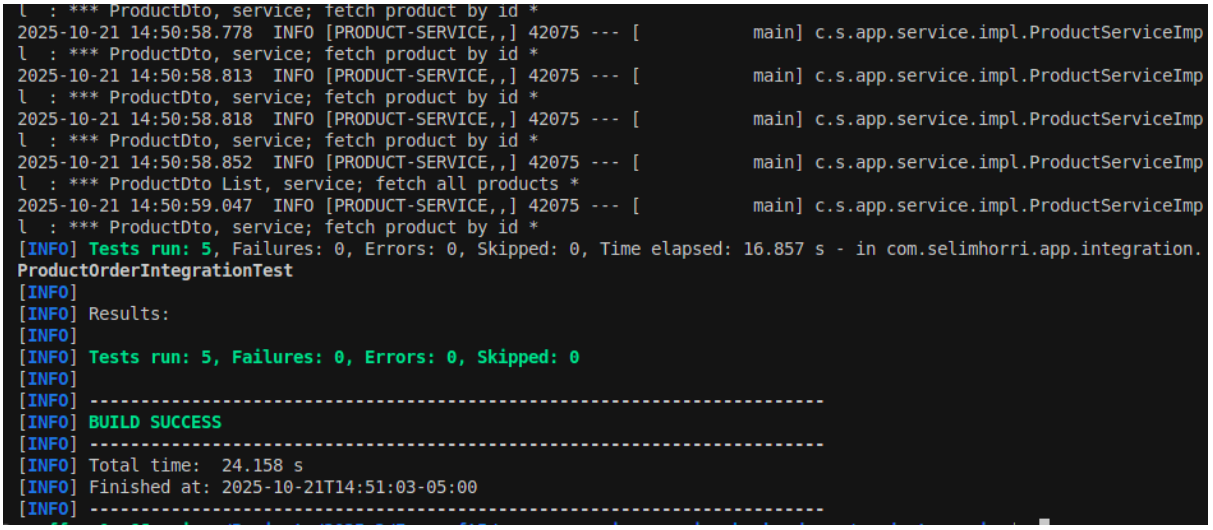
- Al menos cinco nuevas pruebas de integración que validen la comunicación entre servicios

En este caso ejecutamos para visualizar el resultado de todos los test de intgración **./mvnw test -Dtest="*IntegrationTest" -Dsurefire.failIfNoSpecifiedTests=false -pl user-service,product-service,order-service,payment-service,shipping-service,favourite-servicepara obtener los resultados de los test de integración en todos nuestros microservicios**

- *User-Service Integration tests*



- *Product-Service Integration tests*



- *Favourite-Service Integration tests*

```
MockWebServiceServerTestExecutionListener@2f8b373f]
2025-10-21 15:42:03.206  INFO [FAVOURITE-SERVICE,,] 12507 --- [           main] c.s.a.service.impl.FavouriteServiceI
mpl  : *** FavouriteDto, service; fetch favourite by id *
2025-10-21 15:42:03.210  INFO [FAVOURITE-SERVICE,,] 12507 --- [           main] c.s.a.service.impl.FavouriteServiceI
mpl  : *** FavouriteDto, service; fetch favourite by id *
2025-10-21 15:42:03.239  INFO [FAVOURITE-SERVICE,,] 12507 --- [           main] c.s.a.service.impl.FavouriteServiceI
mpl  : *** FavouriteDto List, service; fetch all favourites *
2025-10-21 15:42:03.269  INFO [FAVOURITE-SERVICE,,] 12507 --- [           main] c.s.a.service.impl.FavouriteServiceI
mpl  : *** FavouriteDto, service; fetch favourite by id *
2025-10-21 15:42:03.294  INFO [FAVOURITE-SERVICE,,] 12507 --- [           main] c.s.a.service.impl.FavouriteServiceI
mpl  : *** FavouriteDto, service; fetch favourite by id *
2025-10-21 15:42:03.318  INFO [FAVOURITE-SERVICE,,] 12507 --- [           main] c.s.a.service.impl.FavouriteServiceI
mpl  : *** FavouriteDto, service; fetch favourite by id *
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.149 s - in com.selimhorri.app.integration.F
avouriteUserIntegrationTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 10, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
```

- *Shipping-Service Integration tests*

```
2025-10-21 15:37:09.429  INFO [SHIPPING-SERVICE,,] 10019 --- [           main] c.s.a.service.impl.OrderItemServiceIm
pl  : *** OrderItemDto, service; fetch orderItem by id *
2025-10-21 15:37:09.465  INFO [SHIPPING-SERVICE,,] 10019 --- [           main] c.s.a.service.impl.OrderItemServiceIm
pl  : *** OrderItemDto, service; fetch orderItem by id *
2025-10-21 15:37:09.493  INFO [SHIPPING-SERVICE,,] 10019 --- [           main] c.s.a.service.impl.OrderItemServiceIm
pl  : *** OrderItemDto, service; fetch orderItem by id *
2025-10-21 15:37:09.519  INFO [SHIPPING-SERVICE,,] 10019 --- [           main] c.s.a.service.impl.OrderItemServiceIm
pl  : *** OrderItemDto, service; fetch orderItem by id *
2025-10-21 15:37:09.544  INFO [SHIPPING-SERVICE,,] 10019 --- [           main] c.s.a.service.impl.OrderItemServiceIm
pl  : *** OrderItemDto, service; fetch orderItem by id *
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 6.547 s - in com.selimhorri.app.integration.S
hippingOrderIntegrationTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 10, Failures: 0, Errors: 0, Skipped: 0
```

- *Payment-Service Integration tests*

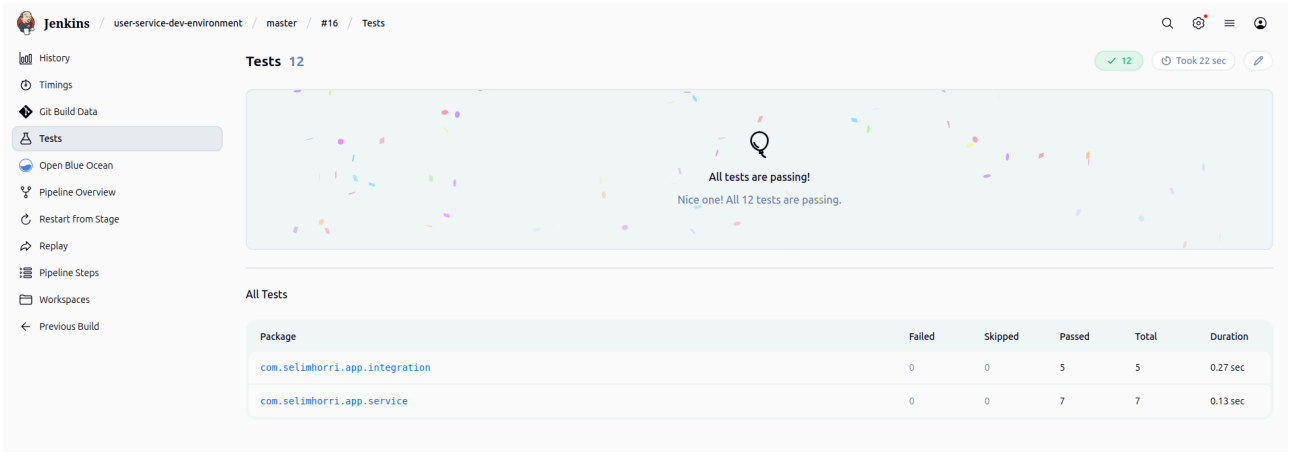
```
   : HTTP POST http://localhost:9411/api/v2/spans
2025-10-21 15:51:11.756 DEBUG [PAYMENT-SERVICE,,] 16209 --- [/spans} check()] org.springframework.web.HttpLogging
   : Accept=[text/plain, application/json, application/*+json, */*]
2025-10-21 15:51:11.756 DEBUG [PAYMENT-SERVICE,,] 16209 --- [/spans} check()] org.springframework.web.HttpLogging
   : Writing [[B@39bc6b0a] as "application/json"
2025-10-21 15:51:11.757  WARN [PAYMENT-SERVICE,,] 16209 --- [           main] o.s.c.s.a.z.ZipkinAutoConfiguration
   : Check result of the [RestTemplateSender{http://localhost:9411/api/v2/spans}] contains an error [CheckResult{ok=
false, error=org.springframework.web.client.ResourceAccessException: I/O error on POST request for "http://localhost
:9411/api/v2/spans": Connection refused; nested exception is java.net.ConnectException: Connection refused}]
2025-10-21 15:51:12.144  WARN [PAYMENT-SERVICE,,] 16209 --- [           main] iguration$LoadBalancerCaffeineWarnLogg
er : Spring Cloud LoadBalancer is currently working with the default cache. You can switch to using Caffeine cache,
by adding it and org.springframework.cache.caffeine.CaffeineCacheManager to the classpath.
2025-10-21 15:51:12.228  INFO [PAYMENT-SERVICE,,] 16209 --- [           main] o.s.b.a.e.web.EndpointLinksResolver
   : Exposing 24 endpoint(s) beneath base path '/actuator'
2025-10-21 15:51:12.294  INFO [PAYMENT-SERVICE,,] 16209 --- [           main] c.s.a.i.PaymentShippingIntegrationTest
   : Started PaymentShippingIntegrationTest in 4.738 seconds (JVM running for 21.892)
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 4.82 s - in com.selimhorri.app.integration.Pa
ymentShippingIntegrationTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 10, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  29.645 s
[INFO] Finished at: 2025-10-21T15:51:17-05:00
[INFO] ------------------------------------------------------------------------
```

- Resumen general de integration testing para los 6 microservicios.

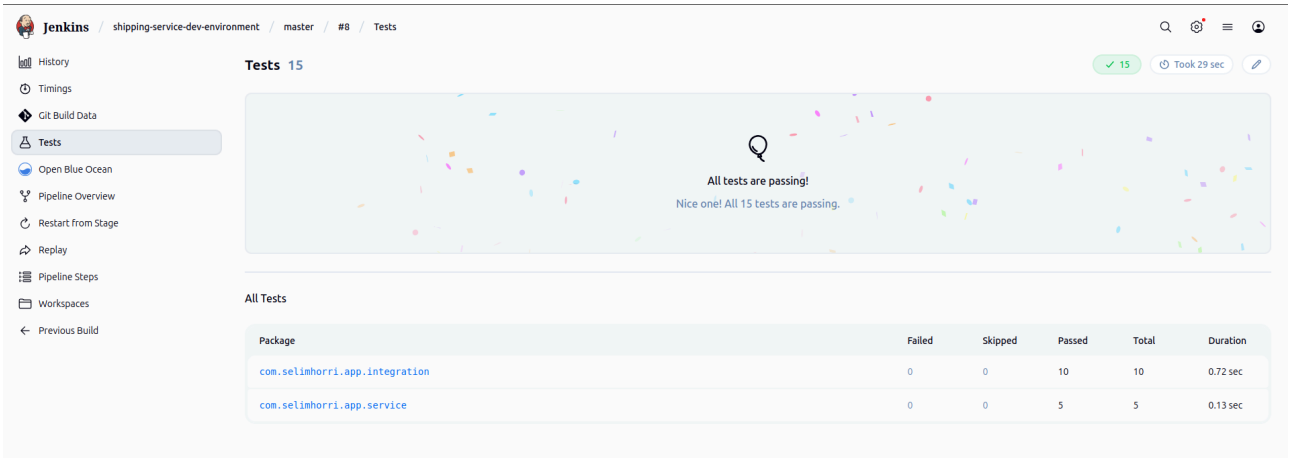
```
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 10, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary for user-service 0.1.0:
[INFO]
[INFO] user-service ....................................... SUCCESS [ 24.401 s]
[INFO] product-service .................................... SUCCESS [ 22.419 s]
[INFO] favourite-service .................................. SUCCESS [ 23.659 s]
[INFO] order-service ...................................... SUCCESS [  0.128 s]
[INFO] shipping-service ................................... SUCCESS [ 28.924 s]
[INFO] payment-service .................................... SUCCESS [ 28.980 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  02:09 min
[INFO] Finished at: 2025-10-21T15:37:43-05:00
[INFO] ------------------------------------------------------------------------
```

- Tests pasados en los pipelines.

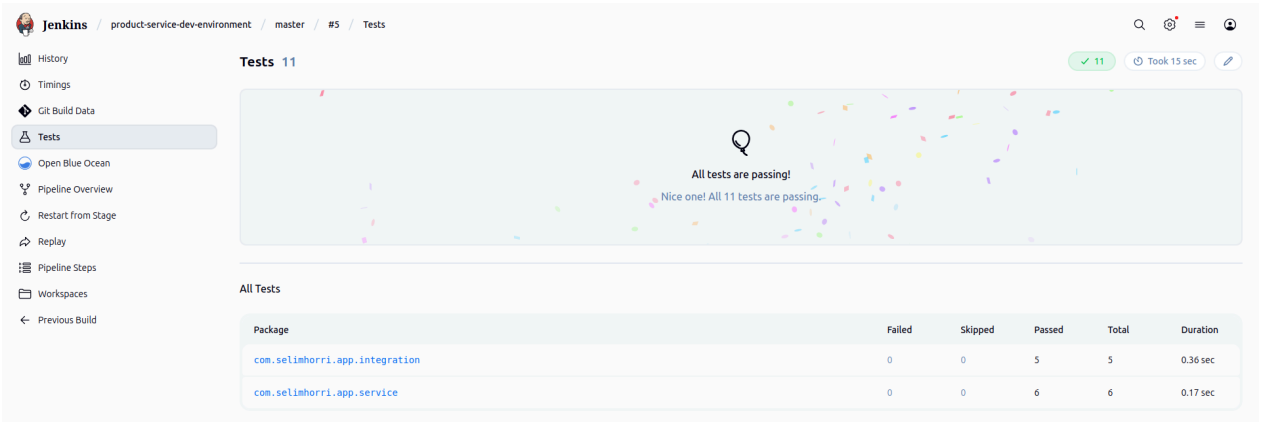
User-service



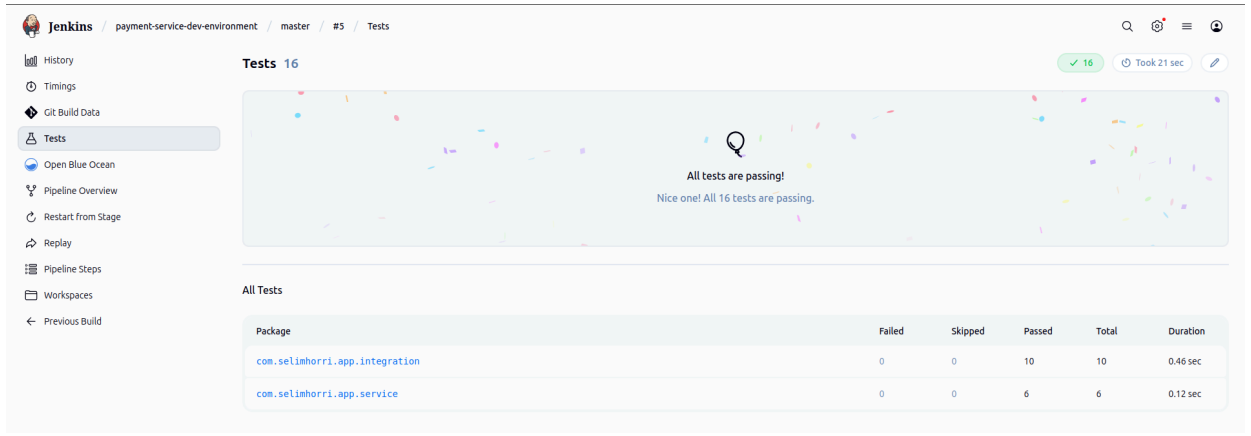
Shipping-service



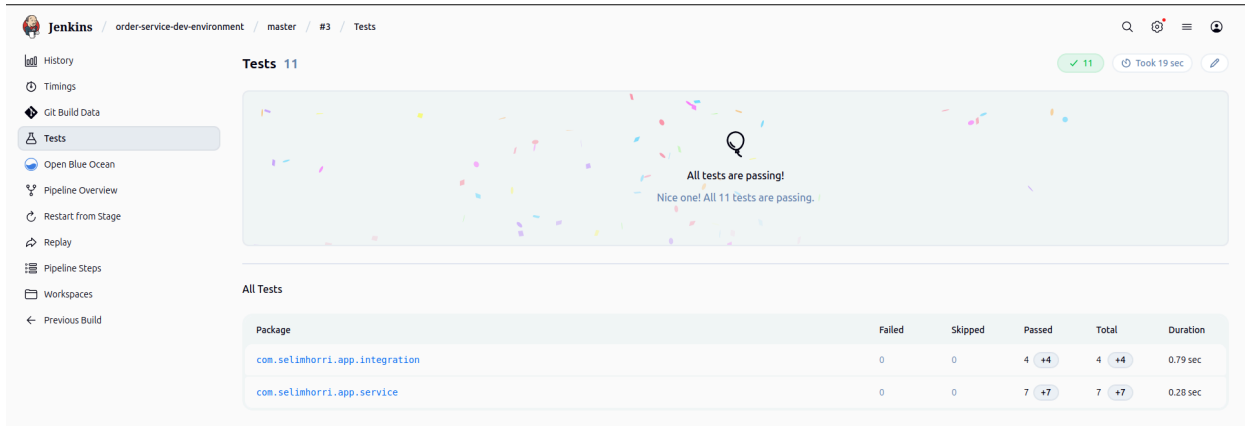
Product-service



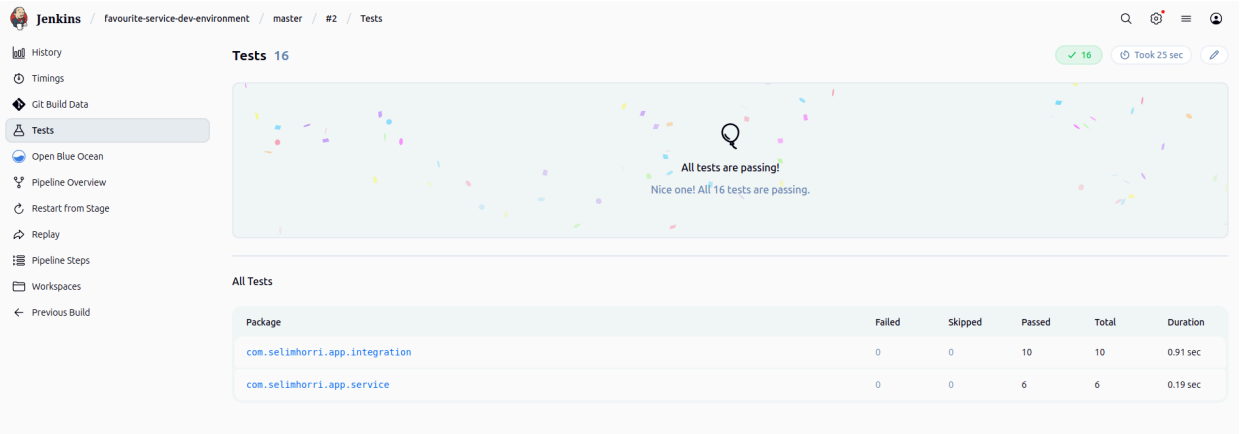
Payments-service



Order-service

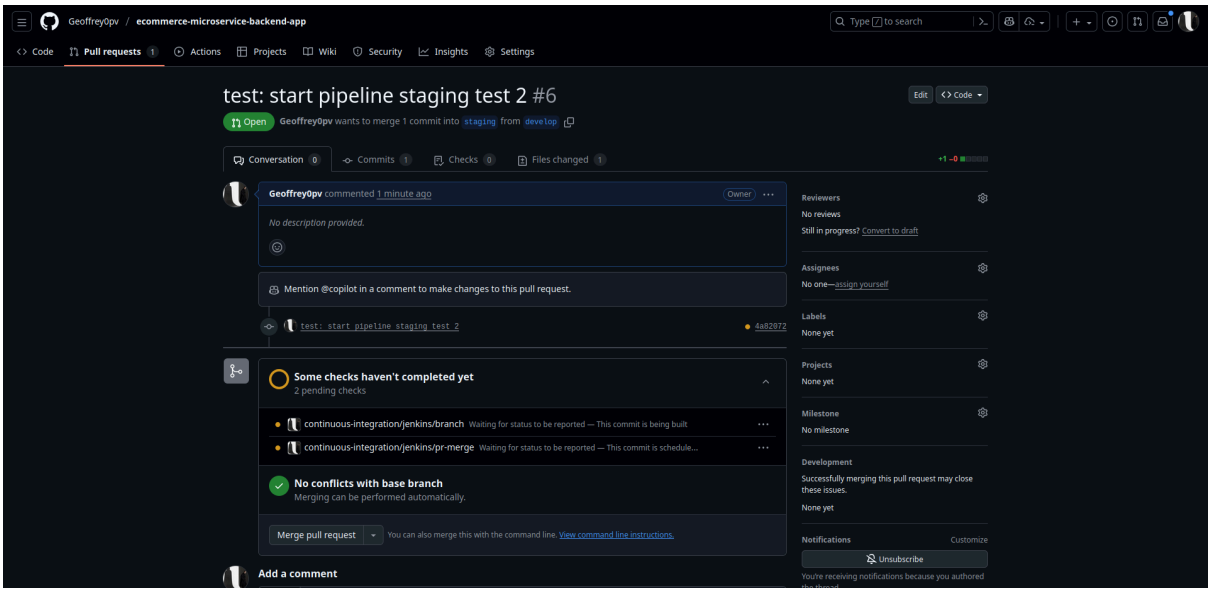


Favourite-service



4. **(15%) Para los microservicios escogidos, debe definir los pipelines que permitan la construcción incluyendo las pruebas de la aplicación desplegada en Kubernetes (stage environment).**

- **User-service Staging**

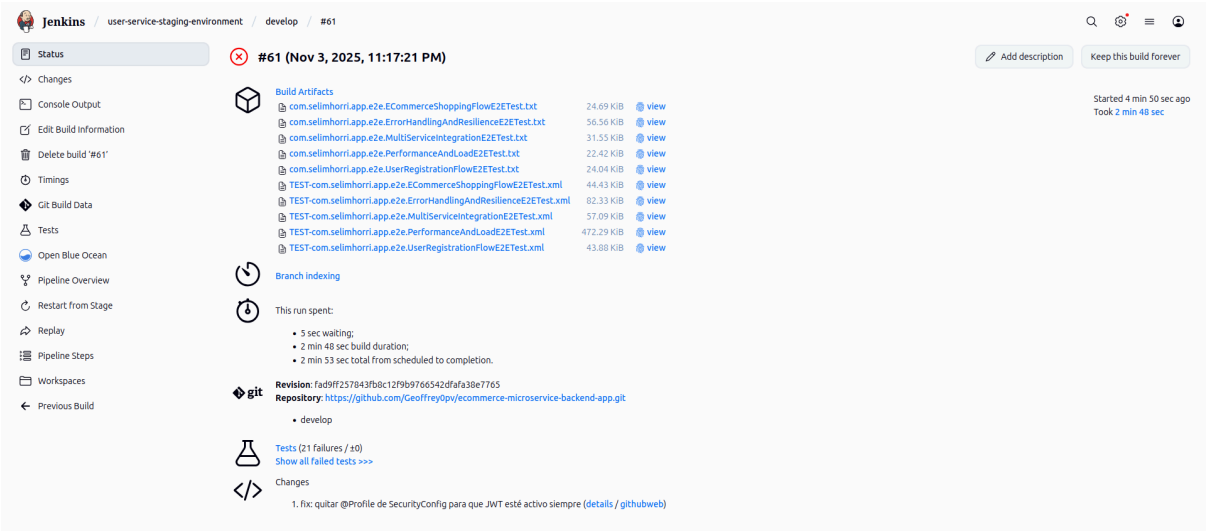


Pipeline de staging ejecución, ejemplo de la estrategia de branching se hace un PR desde develop a staging y se activa la ejecución de nuestra pipeline de staging que se puede visualizar a continuación.

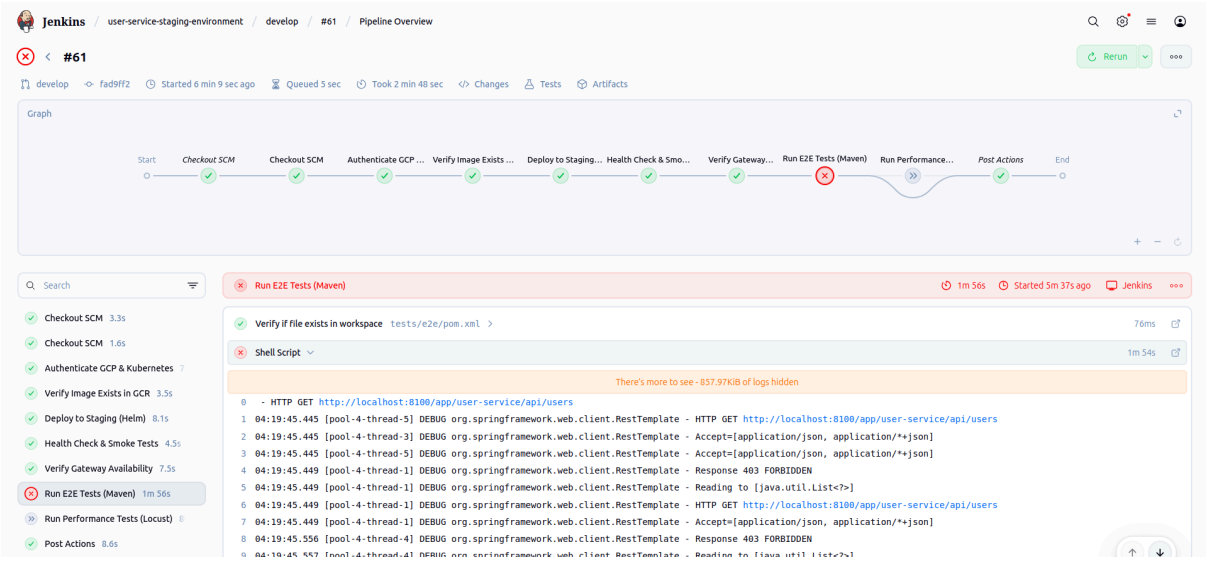
```
geoffrey0pv@Jarvis:~/Projects/2025-2/Ingeosft5/ecommerce-microservice-backend-app$ kubectl get all
  -n staging | grep -E "(proxy-client|NAME)"
NAME                                    READY   STATUS    RESTARTS   AGE
pod/proxy-client-69f9757587-xqssv       1/1     Running   0          16m
NAME                       TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
service/proxy-client       NodePort   10.22.10.27    <none>        80:30900/TCP     24m
NAME                               READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/proxy-client       1/1     1            1           24m
NAME                                          DESIRED   CURRENT   READY   AGE
replicaset.apps/proxy-client-69f9757587       1         1         1       16m
replicaset.apps/proxy-client-95cd4cd59        0         0         0       24m
```

Despliegue de proxy-client para testing de pruebas e2e en el entorno de staging de nuestro cluster de k8s en Google Cloud Platform

- **Pipeline de staging para el servicio User-Service**

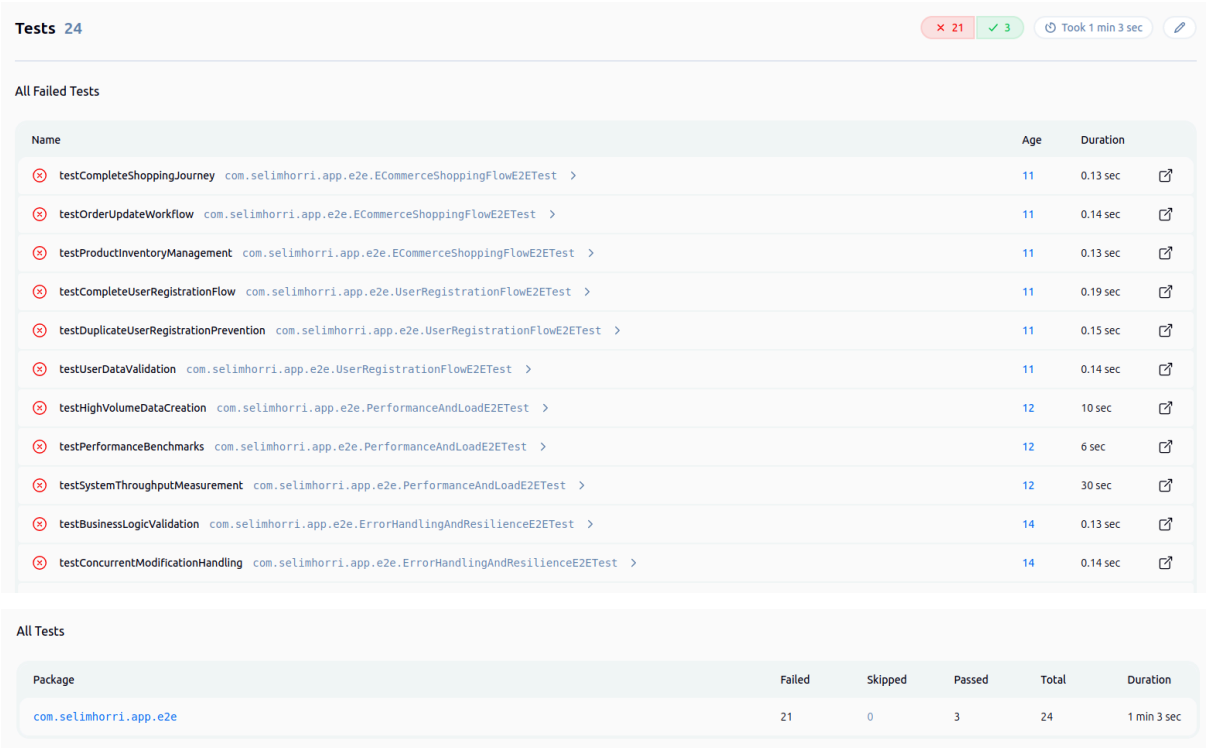


Resumen de la pipeline dónde se genero



Aquí podemos observar que nuestros test E2E ejecutados en el namespace de staging en el cluster de K8S en Google Cloud Platform a través de la pipeline de staging fallaron específicamente por un error de 403 Forbidden, ¿qué quiere decir esto? qué es un error de autenticación producto de un error en la configuración de seguridad de los tokens de JWT que los microservicios tenían previamente en construida, el cuál por temas de tiempo no sé pudo corregir correctamente.

Despues de 62 intento se puede ver cómo se avanzo en ejecución de cada etapa de forma exitosa en las definiciones de nuestra pipeline de staging



Aquí se ejecutaron 24 test e2e usando las IPs publicas de los ingress Controller que desplegamos en un servidor de nginx nos proveen para usarlas para el testeo

- **Pipeline de staging que se puede generalizar para todos los microservicios**

Build Once (Pipeline de DEV): Primero aclarar que el pipeline user-service-pipeline.groovy (DEV) es el único responsable de compilar, probar (unitariamente) y construir la imagen Docker. Al finalizar, etiqueta la imagen con latest-dev y la sube a nuestro Google Container Registry (GCR).

Deploy Many (Este Pipeline): Este pipeline de staging toma esa imagen latest-dev (un artefacto inmutable), la despliega en el clúster de Kubernetes de staging y ejecuta sobre ella un conjunto de pruebas más profundo (Health Checks, E2E y Rendimiento).

```
pipeline {
    agent any

    environment {
        IMAGE_NAME = "user-service"
        GCR_REGISTRY = "us-central1-docker.pkg.dev/ecommerce-backend-1760307199/ecommerce-microservice
        FULL_IMAGE_NAME = "${GCR_REGISTRY}/${IMAGE_NAME}"

        IMAGE_TAG = "latest-dev"

        GCP_CREDENTIALS = credentials('gke-credentials')
        GCP_PROJECT = "ecommerce-backend-1760307199"

        CLUSTER_NAME = "ecommerce-devops-cluster"
        CLUSTER_LOCATION_FLAG = "--region=us-central1"

        K8S_NAMESPACE = "staging"
        K8S_DEPLOYMENT_NAME = "user-service"
        K8S_CONTAINER_NAME = "user-service"
        K8S_SERVICE_NAME = "user-service"
        SERVICE_PORT = "8700"

        API_GATEWAY_SERVICE_NAME = "proxy-client"
    }
```

Aquí se definen todas las variables de entorno importantes que necesita conocer la pipeline de staging, cómo las credenciales de acceso al cluster de K8S en GCP, el nombre del proyecto, el nombre del namespace, el nombre de la imagén, los puertos, etc.

```
stages {

    stage('Checkout SCM') {
        steps {
            checkout scm
            echo "📦 Iniciando despliegue a STAGING"
            echo "📦 Imagen a desplegar: ${FULL_IMAGE_NAME}:${IMAGE_TAG}"
        }
    }

    stage('Authenticate GCP & Kubernetes') {
        steps {
            script {
                sh """
                    echo "🔐 Autenticando con GCP..."
                    gcloud auth activate-service-account --key-file=\${GCP_CREDENTIALS}
                    gcloud config set project \${GCP_PROJECT}
                    gcloud auth configure-docker us-central1-docker.pkg.dev --quiet
                    echo "☸️ Obteniendo credenciales de GKE..."
                    gcloud container clusters get-credentials \${CLUSTER_NAME} \${CLUSTER_LOCATION_FLAG} --project
                """
            }
        }
    }
```

Aquí de primeras descarga el código fuente del repositorio (la rama develop o el PR). No lo usamos para compilar, sino para tener acceso a los manifiestos de Kubernetes (manifests-gcp/) y a los scripts de prueba (tests/e2e/, tests/performance/) que se necesitan en etapas posteriores.

Luego utiliza la credencial gke-credentials para autenticar al agente de Jenkins contra Google Cloud. Este es el paso de "inicio de sesión". Le da permiso al pipeline para ejecutar comandos gcloud y kubectl en nuestro proyecto (ecommerce-backend-1760307199) y conectarse a nuestro clúster de GKE (ecommerce-devops-cluster).

```
    stage('Verify Image Exists in GCR') {
        steps {
            script {
                sh """
                    echo "🔍 Verificando \${FULL_IMAGE_NAME}:\${IMAGE_TAG}..."
                    gcloud artifacts docker images describe \${FULL_IMAGE_NAME}:\${IMAGE_TAG} || {
                        echo "❌ ERROR: Imagen no encontrada"
                        echo "Asegúrate de que el pipeline de DEV ('user-service-pipeline.groovy') haya corrido ex
                        exit 1
                    }
                    echo "✅ Imagen verificada."
                """
            }
        }
    }
}
```

Bueno, aquí se comprueba que la imagen que vamos a desplegar (user-service:latest-dev) realmente exista en nuestro Google Container Registry.
Esto es un "paso de seguridad" crucial. Si esta etapa falla, significa que el pipeline de DEV (Build) falló o nunca se ejecutó. Esto evita intentar desplegar una imagen rota o inexistente, ahorrándonos tiempo de diagnóstico.

```
stage('Deploy to Staging (Helm)') {
    steps {
        script {
            sh """
                echo "🚀 Desplegando a \${K8S_NAMESPACE} usando Helm..."
                kubectl create namespace \${K8S_NAMESPACE} --dry-run=client -o yaml | kubectl apply -f -

                echo "📋 Aplicando/Actualizando Chart de Helm: \${K8S_DEPLOYMENT_NAME}"

                # Deshabilitamos Eureka para que el pod arranque solo
                helm upgrade --install \${K8S_DEPLOYMENT_NAME} manifests-gcp/user-service/ \
                    --namespace \${K8S_NAMESPACE} \
                    --set image.tag=\${IMAGE_TAG} \
                    --set env[4].value="false" \
                    --set env[5].value="false" \
                    --wait --timeout=5m

                echo "✅ Despliegue completado."
            """
        }
    }
}
```

Esta Etapa es el corazón del despliegue. Usa helm upgrade --install para desplegar el user-service en el clúster.

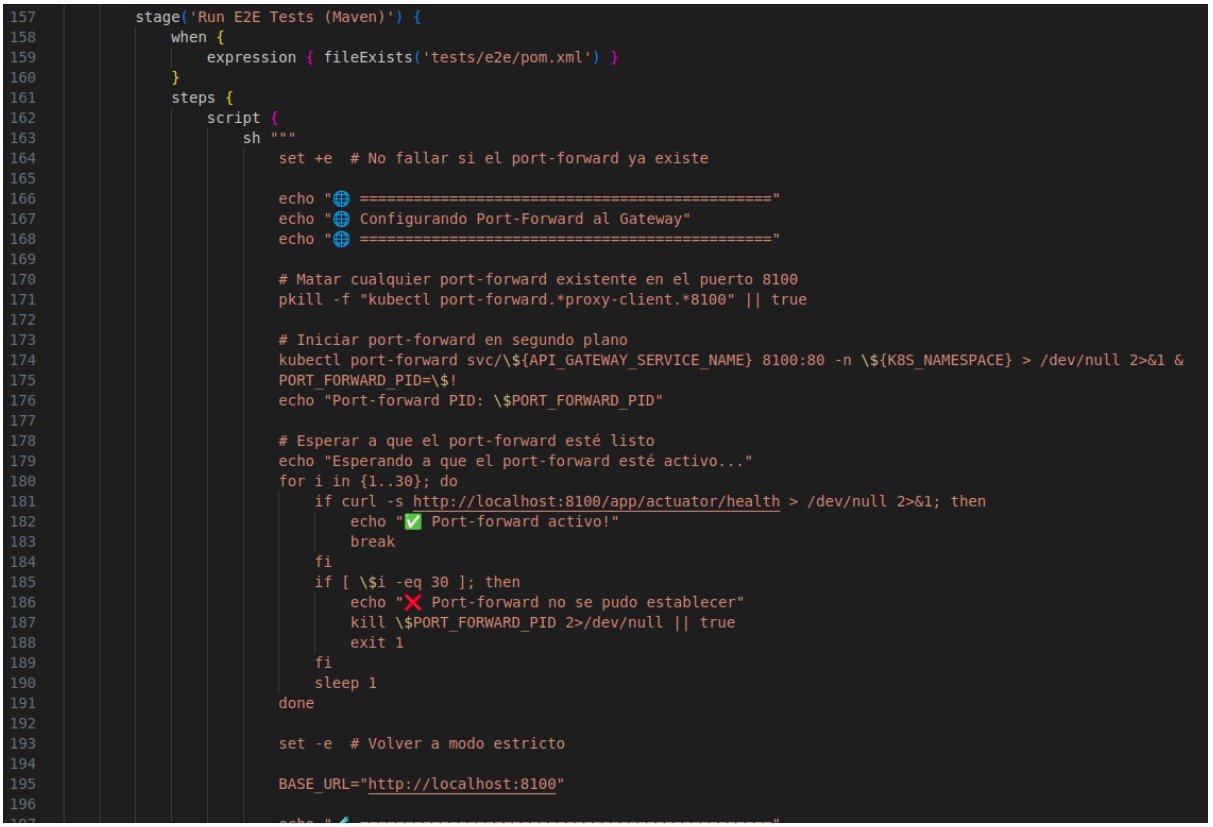
- helm upgrade --install: Actualiza el servicio si ya existe o lo instala si es la primera vez.
- --set image.tag=latest-dev: Le dice a Helm que ignore el tag por defecto del values.yaml y use la imagen latest-dev que acabamos de verificar.
- --set env[4].value="false": (¡Importante!) Deshabilita el registro en Eureka. Esto es un "atajo" de staging para que el servicio arranque sin depender de que Eureka esté desplegado, permitiéndonos probarlo de forma aislada.
- --wait: Le ordena a Helm que no termine hasta que Kubernetes reporte que el despliegue fue exitoso y los pods están listos (Ready).

```
stage('Health Check & Smoke Tests') {
    steps {
        script {
            sh """
                echo "🏥 Ejecutando health checks..."

                kubectl wait --for=condition=ready pod \
                    -l app=\${K8S_DEPLOYMENT_NAME} \
                    -n \${K8S_NAMESPACE} \
                    --timeout=300s

                POD_NAME=\$(kubectl get pods -n \${K8S_NAMESPACE} \
                    -l app=\${K8S_DEPLOYMENT_NAME} \
                    -o jsonpath='{.items[0].metadata.name}')

                echo "🎯 Testing pod: \$POD_NAME en puerto \${SERVICE_PORT}"

                kubectl exec \$POD_NAME -n \${K8S_NAMESPACE} -- \
                    curl -f http://localhost:\${SERVICE_PORT}/user-service/actuator/health || {
                        echo "⚠️ Health check falló"
                        kubectl logs \$POD_NAME -n \${K8S_NAMESPACE} --tail=50
                        exit 1
                    }

                echo "✅ Health check passed!"
            """
        }
    }
}
```

Esto ejecuta un kubectl exec para "entrar" al pod recién creado y hacer un curl a su endpoint de salud (/actuator/health).
Esto confirma que la aplicación *dentro* del contenedor se inició correctamente y está respondiendo. El helm upgrade --wait solo nos dice que el *pod* está corriendo, pero esta prueba confirma que nuestra aplicación Spring Boot está viva.

```
stage('Verify Gateway Availability') {
    steps {
        script {
            sh """
                echo "🌐 Verificando disponibilidad del API Gateway (\${API_GATEWAY_SERVICE_NAME})..."

                kubectl wait --for=condition=ready pod \
                    -l app=\${API_GATEWAY_SERVICE_NAME} \
                    -n \${K8S_NAMESPACE} \
                    --timeout=300s

                GATEWAY_IP=\$(kubectl get svc \${API_GATEWAY_SERVICE_NAME} -n \${K8S_NAMESPACE} \
                    -o jsonpath='{.spec.clusterIP}')

                if [ -z "\$GATEWAY_IP" ]; then
                    echo "❌ No se pudo obtener la IP del servicio \${API_GATEWAY_SERVICE_NAME}"
                    exit 1
                fi

                echo "✅ Gateway ClusterIP: \$GATEWAY_IP"
                echo "\$GATEWAY_IP" > gateway-ip.txt

                echo "🔍 Verificando conectividad al Gateway en http://\$GATEWAY_IP:80/app/actuator/health"
                kubectl run test-gateway-\${BUILD_NUMBER} --image=curlimages/curl:latest \
                    -n \${K8S_NAMESPACE} --rm -i --restart=Never --timeout=60s -- \
                    curl -f --retry 5 --retry-delay 5 --retry-connrefused \
                    http://\$GATEWAY_IP:80/app/actuator/health || {
                        echo "⚠️ No se pudo conectar al Gateway internamente"
                        exit 1
                    }

                echo "✅ Gateway respondiendo correctamente"
            """
        }
    }
}
```

Aquí se despliega (si no existe) el proxy-client (nuestro API Gateway) y espera a que esté listo. Luego, obtiene su ClusterIP (la IP interna) y la guarda en un archivo gateway-ip.txt. Las pruebas E2E y Locust no deben apuntar al user-service directamente. Deben simular tráfico real, el cual siempre pasa por el Gateway. Esta etapa prepara el "punto de entrada" para las siguientes pruebas.



```
157         stage('Run E2E Tests (Maven)') {
158             when {
159                 expression { fileExists('tests/e2e/pom.xml') }
160             }
161             steps {
162                 script {
163                     sh """
164                         set +e  # No fallar si el port-forward ya existe
165 
166                         echo "🌐 ============================================="
167                         echo "🌐 Configurando Port-Forward al Gateway"
168                         echo "🌐 ============================================="
169 
170                         # Matar cualquier port-forward existente en el puerto 8100
171                         pkill -f "kubectl port-forward.*proxy-client.*8100" || true
172 
173                         # Iniciar port-forward en segundo plano
174                         kubectl port-forward svc/\${API_GATEWAY_SERVICE_NAME} 8100:80 -n \${K8S_NAMESPACE} > /dev/null 2>&1 &
175                         PORT_FORWARD_PID=\$!
176                         echo "Port-forward PID: \$PORT_FORWARD_PID"
177 
178                         # Esperar a que el port-forward esté listo
179                         echo "Esperando a que el port-forward esté activo..."
180                         for i in {1..30}; do
181                             if curl -s http://localhost:8100/app/actuator/health > /dev/null 2>&1; then
182                                 echo "✅ Port-forward activo!"
183                                 break
184                             fi
185                             if [ \$i -eq 30 ]; then
186                                 echo "❌ Port-forward no se pudo establecer"
187                                 kill \$PORT_FORWARD_PID 2>/dev/null || true
188                                 exit 1
189                             fi
190                             sleep 1
191                         done
192 
193                         set -e  # Volver a modo estricto
194 
195                         BASE_URL="http://localhost:8100"
196 
197                         echo "✅ ============================================="
```

Aquí se ejecutan nuestras pruebas E2E de Java (tests/e2e/). Esta es la validación funcional clave. El pipeline lanza un contenedor de Maven, le da acceso a la IP del Gateway (del gateway-ip.txt) y ejecuta los flujos de usuario completos (como UserRegistrationFlowE2ETest.java) contra el entorno de staging *real*.
Reporte: Genera los reportes JUnit (.xml) que Jenkins puede leer para mostrar un historial de pruebas.

```
        stage('Run Performance Tests (Locust)') {
            when {
                expression { fileExists('tests/performance/ecommerce_load_test.py') }
            }
            steps {
                script {
                    sh """
                        set +e  # No fallar si el port-forward ya existe

                        echo "🌐 ============================================="
                        echo "🌐 Configurando Port-Forward al Gateway para Locust"
                        echo "🌐 ============================================="

                        # Matar cualquier port-forward existente en el puerto 8100
                        pkill -f "kubectl port-forward.*proxy-client.*8100" || true

                        # Iniciar port-forward en segundo plano
                        kubectl port-forward svc/\${API_GATEWAY_SERVICE_NAME} 8100:80 -n \${K8S_NAMESPACE} > /dev/null 2>&1 &
                        PORT_FORWARD_PID=\$!
                        echo "Port-forward PID: \$PORT_FORWARD_PID"

                        # Esperar a que el port-forward esté listo
                        echo "Esperando a que el port-forward esté activo..."
                        for i in {1..30}; do
                            if curl -s http://localhost:8100/app/actuator/health > /dev/null 2>&1; then
                                echo "✅ Port-forward activo!"
                                break
                            fi
                            if [ \$i -eq 30 ]; then
                                echo "❌ Port-forward no se pudo establecer"
                                kill \$PORT_FORWARD_PID 2>/dev/null || true
                                exit 1
                            fi
                            sleep 1
                        done

                        set -e  # Volver a modo estricto

                        BASE_URL="http://localhost:8100"
```

```
    post {
        success {
            script {
                sh """
                    echo "🎉 ✅ STAGING DEPLOY EXITOSO"
                    echo "📦 Imagen desplegada: \${FULL_IMAGE_NAME}:\${IMAGE_TAG}"
                    gcloud auth activate-service-account --key-file=\${GCP_CREDENTIALS}
                    gcloud auth revoke --all || true
                """
            }
        }
        failure {
            script {
                sh """
                    echo "🔐 Re-autenticando para operaciones de rollback..."
                    gcloud auth activate-service-account --key-file=\${GCP_CREDENTIALS}
                    gcloud config set project \${GCP_PROJECT}
                    gcloud container clusters get-credentials \${CLUSTER_NAME} \${CLUSTER_LOCATION_FLAG} --project \${GCP_PROJECT
                """

                def failedStage = env.STAGE_NAME ?: 'Unknown'

                sh """
                    echo "❌ 💥 STAGING DEPLOY FALLÓ"
                    echo "🔍 Fallo detectado en stage: ${failedStage}"

                    if [ "${failedStage}" = "Deploy to Staging (Helm)" ]; then
                        echo "🔄 Realizando rollback del despliegue fallido..."
                        helm rollback \${K8S_DEPLOYMENT_NAME} 0 -n \${K8S_NAMESPACE} || echo "⚠️ No hay revisión anterior para ro
                    else
                        echo "⚠️ Fallo en stage '${failedStage}'. El despliegue NO será revertido."
                    fi

                    echo "📋 Información de debug:"
                    kubectl get events -n \${K8S_NAMESPACE} --sort-by='.lastTimestamp' | tail -20
                    gcloud auth revoke --all || true
                """
            }
        }
        always {
            cleanWs()
```

Success: Si todo sale bien, limpia las credenciales de GCP y marca el build como exitoso.
Failure: Si cualquier etapa falla, se re-autentica en GKE y ejecuta helm rollback para revertir el despliegue fallido a la versión anterior que sí funcionaba. Luego, imprime los eventos de Kubernetes para un diagnóstico rápido.

Always: Sin importar si falla o tiene éxito, ejecuta cleanWs() para limpiar el workspace de Jenkins, dejándolo listo para la próxima ejecución.