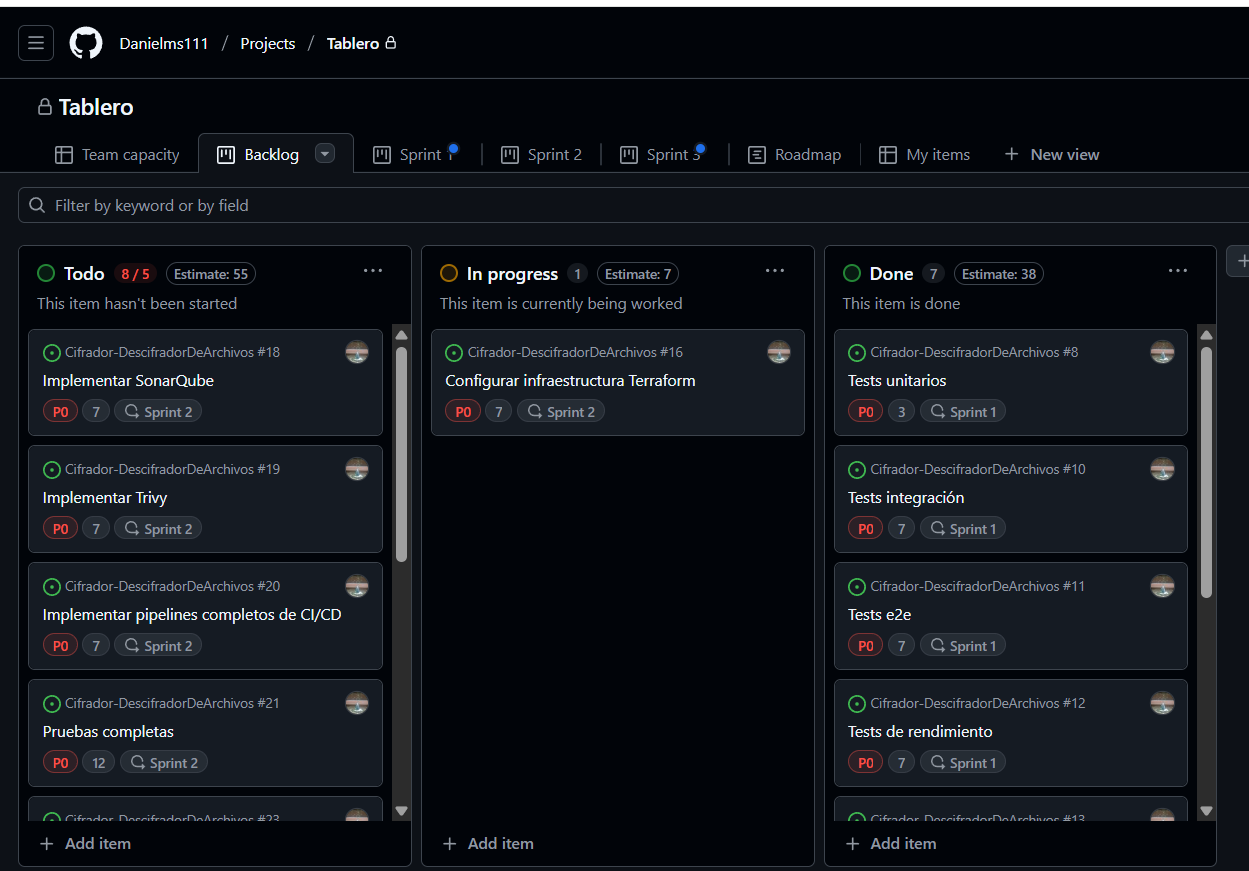
**Proyecto final**

**Diego Mueses - Daniel Montezuma**

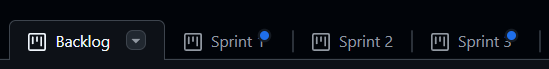
**Metodología Ágil y Estrategia de Branching**

Para el desarrollo del proyecto, se ha decidido utilizar la metodología ágil Scrum, ya que permite una planificación iterativa y una mejora continua a través de sprints cortos y objetivos claros. Esta metodología facilita el seguimiento del progreso y la adaptación a cambios en los requerimientos.

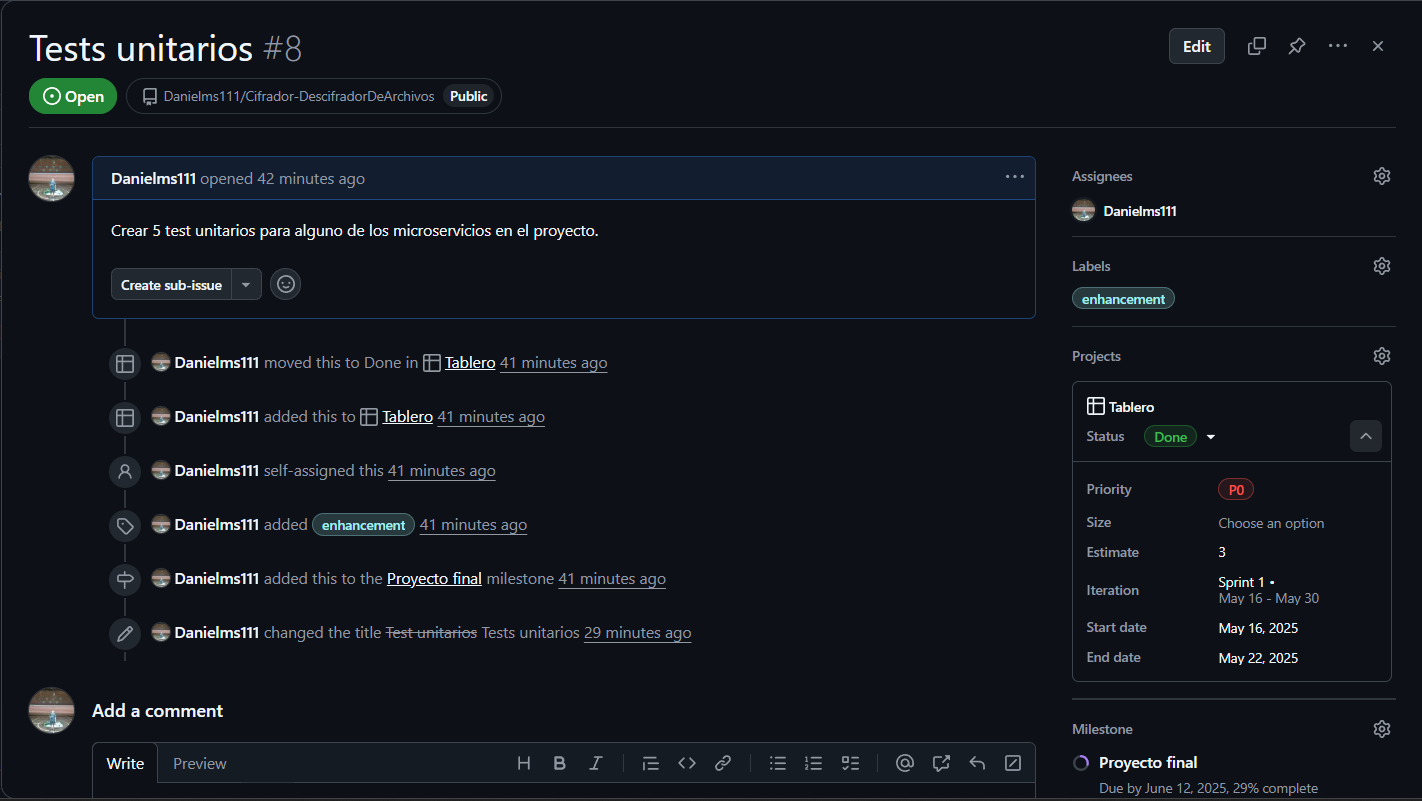
Se ha creado un tablero de proyecto en GitHub Projects, el cual está estructurado en columnas que representan el flujo de trabajo: *Backlog*, *To Do*, *In Progress*, *In Review* y *Done*. En este tablero se han registrado todas las historias de usuario, cada una con sus respectivos criterios de aceptación y etiquetas para facilitar su seguimiento.



El desarrollo se organizará en 3 sprints completos, cada uno con una duración de dos semanas. Al inicio de cada sprint se realizará la planificación (Sprint Planning), se definirán las tareas prioritarias, y al final se llevará a cabo una retrospectiva para evaluar el desempeño del equipo y proponer mejoras.



La estimación de tareas se ha realizado utilizando la técnica de story points, asignando valores en función de la complejidad y el esfuerzo requerido. Esta estimación permite ajustar la carga de trabajo de cada sprint de forma equilibrada.



En cuanto al control de versiones, se utilizará la estrategia de GitFlow, que proporciona una estructura clara para el desarrollo colaborativo. Esta estrategia incluye ramas principales (main y develop), ramas de características (feature/\*), ramas de corrección (hotfix/\*) y de lanzamiento (release/\*), lo cual permite mantener un flujo de trabajo organizado y estable.

Toda la documentación de la metodología, el tablero, los sprints y la estrategia de branching estará disponible en el repositorio del proyecto para su consulta y seguimiento.

**Infraestructura como Código con Terraform**

**Estructura:**

La infraestructura fue organizada de forma modular en la carpeta k8s-cluster-terraform/, con la siguiente estructura:

**k8s-cluster-terraform/**

**├── main/ # Módulo principal reutilizable**

**│ ├── main.tf # Configuración principal del cluster**

**│ ├── variables.tf # Variables parameterizable**

**│ └── outputs.tf # Outputs del módulo**

**├── stage/ # Ambiente de staging**

**│ └── main.tf # Configuración específica para staging**

**└── prod/ # Ambiente de producción**

**├── main.tf # Configuración específica para producción**

**├── backend.tf # Configuración de backend remoto**

**└── kubeconfig-prod # Archivo de configuración de kubectl**

**Componentes de Infraestructura:**

**Cluster de Kubernetes (GKE):**

* **Proveedor:** Google Kubernetes Engine (GKE)
* **Tipo de clúster:** Clúster privado (sin acceso público al plano de control)
* **Configuración:** Clúster regional (distribuido en varias zonas para alta disponibilidad)
* **Módulo utilizado:** terraform-google-modules/kubernetes-engine v24.1.0
* **Autenticación:** Generación automática de archivo kubeconfig para conexión segura con kubectl.

**Networking:**

* Se creó una **red virtual (VPC)** para cada entorno (stage, prod) de manera aislada.
* Cada red incluye:
  + **Subred principal:** Rango 10.10.0.0/16 (asigna IPs internas a los nodos).
  + **Rango secundario para Pods:** 10.20.0.0/16.
  + **Rango secundario para Servicios:** 10.30.0.0/16.

Estas redes aseguran segmentación del tráfico y cumplimiento de las buenas prácticas de seguridad en Kubernetes.

**Node Pools:**

Configuración diferenciada por ambiente:

| **Ambientes** | **Tipo Máquina** | **Min nodos** | **Max nodos** | **Disco** |
| --- | --- | --- | --- | --- |
| Staging | e2-medium | 1 | 2 | 30GB |
| Producción | n2-highmem-2 | 3 | 5 | 30GB |

Estas configuraciones permiten escalar según la carga y necesidades de cada ambiente.

**Implementación Modular:**

* **Módulo Principal (main/):** Contiene toda la lógica reutilizable para definir redes, clústeres y autenticación. Actúa como una "plantilla" para crear entornos consistentes.
* **Ambientes (stage/, prod/):** Cada entorno utiliza el módulo principal, pero define sus propios parámetros: número de nodos, tipo de instancia, nombre del clúster, etc.

Esta separación permite aplicar cambios de forma segura y controlada en cada entorno.

**Backend remoto (GCS):**

Para garantizar seguridad y colaboración, se configuró el almacenamiento remoto del estado de Terraform en Google Cloud Storage (GCS). Esto permite:

* Compartir el estado entre diferentes usuarios/equipos.
* Tener versiones del estado para recuperar en caso de errores.
* Auditar los cambios realizados en la infraestructura.

El archivo backend.tf fue creado manualmente en cada ambiente para apuntar al bucket correspondiente.

**Beneficios:**

**Ventajas Técnicas:**

1. **Reutilización de Código:** El módulo principal es reutilizable entre ambientes
2. **Mantenibilidad:** Estructura clara y separación de responsabilidades
3. **Escalabilidad:** Configuración de auto-scaling integrada
4. **Seguridad:** Clusters privados con networking seguro
5. **Versionado:** Control de versiones del estado de infraestructura

Ventajas Operacionales:

1. **Despliegue Consistente:** Misma configuración base en todos los ambientes
2. **Automatización:** Aprovisionamiento automático de recursos
3. **Recuperación:** Estado remoto permite recuperación ante fallos
4. **Auditoría:** Historial completo de cambios en la infraestructura

**Costos aproximados:**

Ambiente prod:

* Nodos: n2-highmem-2
* 2 vCPU, 16 GB RAM
* Costo aprox: $67 USD/mes por nodo
* Cantidad: Mínimo 3, máximo 5 → estimaremos con 4 nodos promedio

PROMEDIO TOTAL = $67 x 4 = $268 USD

Ambiente stage:

* Nodos: e2-medium
* 2 vCPU, 4 GB RAM
* Costo aprox: $25 USD/mes por nodo
* Cantidad: Mínimo 1, máximo 2 → estimaremos con 1.5 nodos promedio

PROMEDIO TOTAL = $25 x 1.5 = $37.5 USD

**Ejecución de PODS:**

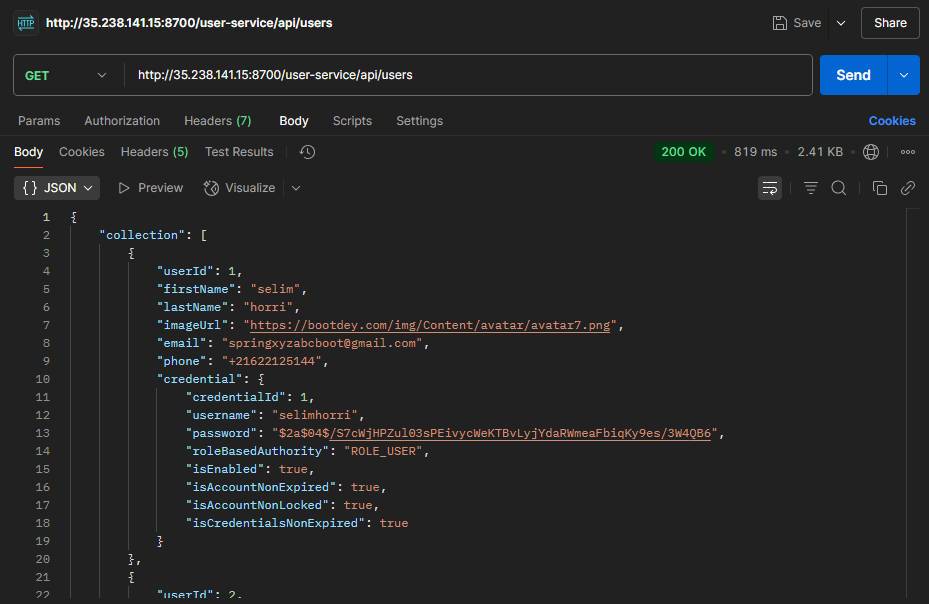
La siguiente imagen corresponde a la vista de cargas de trabajo (Workloads) dentro del clúster de producción (k8s-cluster-prod) en Google Kubernetes Engine (GKE):



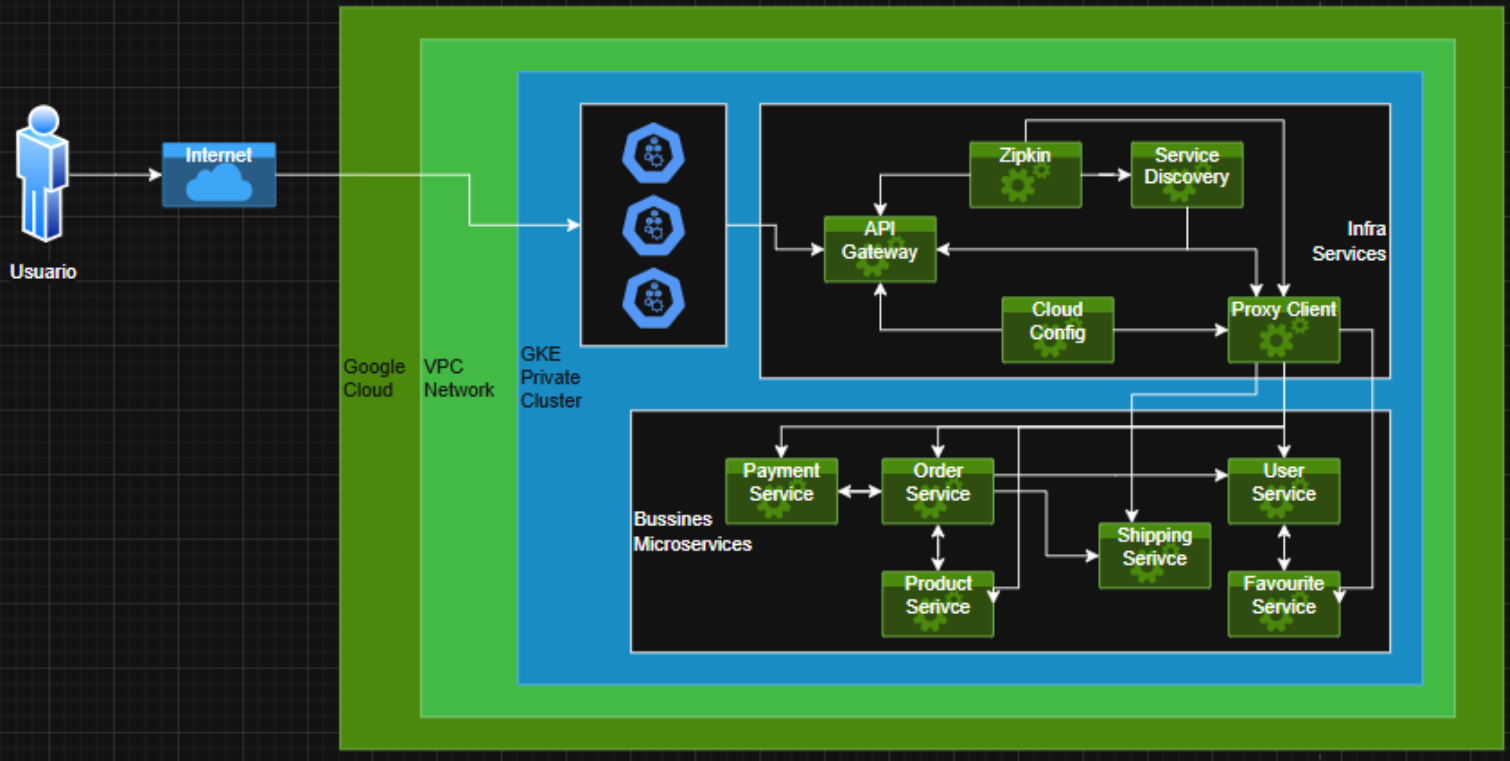
Esta captura muestra que todos los microservicios del sistema han sido desplegados correctamente y se encuentran en estado saludable (OK). Cada servicio aparece como un recurso de tipo Deployment, lo cual asegura alta disponibilidad y gestión automática de los Pods por parte del orquestador de Kubernetes.

**Ejemplo de funcionalidad (user-service):**





**Diagrama de arquitectectura:**

****

**Patrones de Diseño**

#### Patrones identificados en la arquitectura existente:

* **Gateway Pattern**Implementado con el api-gateway como punto central de entrada. Encapsula la lógica de enrutamiento y autentica solicitudes antes de redirigirlas a los microservicios correspondientes.
* **Service Discovery**A través de Eureka, los microservicios se registran dinámicamente y descubren otros servicios sin necesidad de configuración estática.
* **Config Server (Externalized Configuration)**cloud-config permite centralizar la configuración de los servicios y aplicar cambios sin recompilar ni reempaquetar.

**Patrones implementados durante el proyecto:**

**External Configuration**

* **Servicios:** Todos los microservicios importan su configuración desde cloud-config.
* **Beneficio:** Facilita el versionado de propiedades, cambio dinámico de configuraciones y portabilidad entre entornos.

**Retry + Timeout (Fault Tolerance)**

* **Aplicación:** Cada servicio tiene configuraciones que controlan el tiempo máximo de espera y el número de reintentos.
* **Beneficio:** Mejora la disponibilidad y control de errores transitorios.

**CI/CD Avanzado**

Se implementó un pipeline completo y automatizado de CI/CD utilizando Jenkins, cubriendo todo el ciclo de vida de los microservicios del sistema. A continuación, se detallan cada uno de los subcomponentes implementados:

### **Pipelines completos de CI/CD (Jenkins)**

Se configuró un Jenkinsfile que define un pipeline multietapa ejecutado mediante un proyecto tipo Multibranch Pipeline. El pipeline considera:

* Compilación con Maven.
* Ejecución de pruebas unitarias, de integración y e2e.
* Análisis estático de código con SonarQube.
* Construcción de imágenes Docker para cada microservicio.
* Escaneo de vulnerabilidades de las imágenes Docker con Trivy.
* Publicación de imágenes a DockerHub.
* Despliegue en local o Kubernetes según el entorno.
* Análisis de resultados de pruebas de carga con Locust.
* Generación y publicación automática de Release Notes.

**Herramientas definidas:**

* JDK: JDK\_11
* Maven: MVN
* Docker
* kubectl

**Variables de Entorno:**

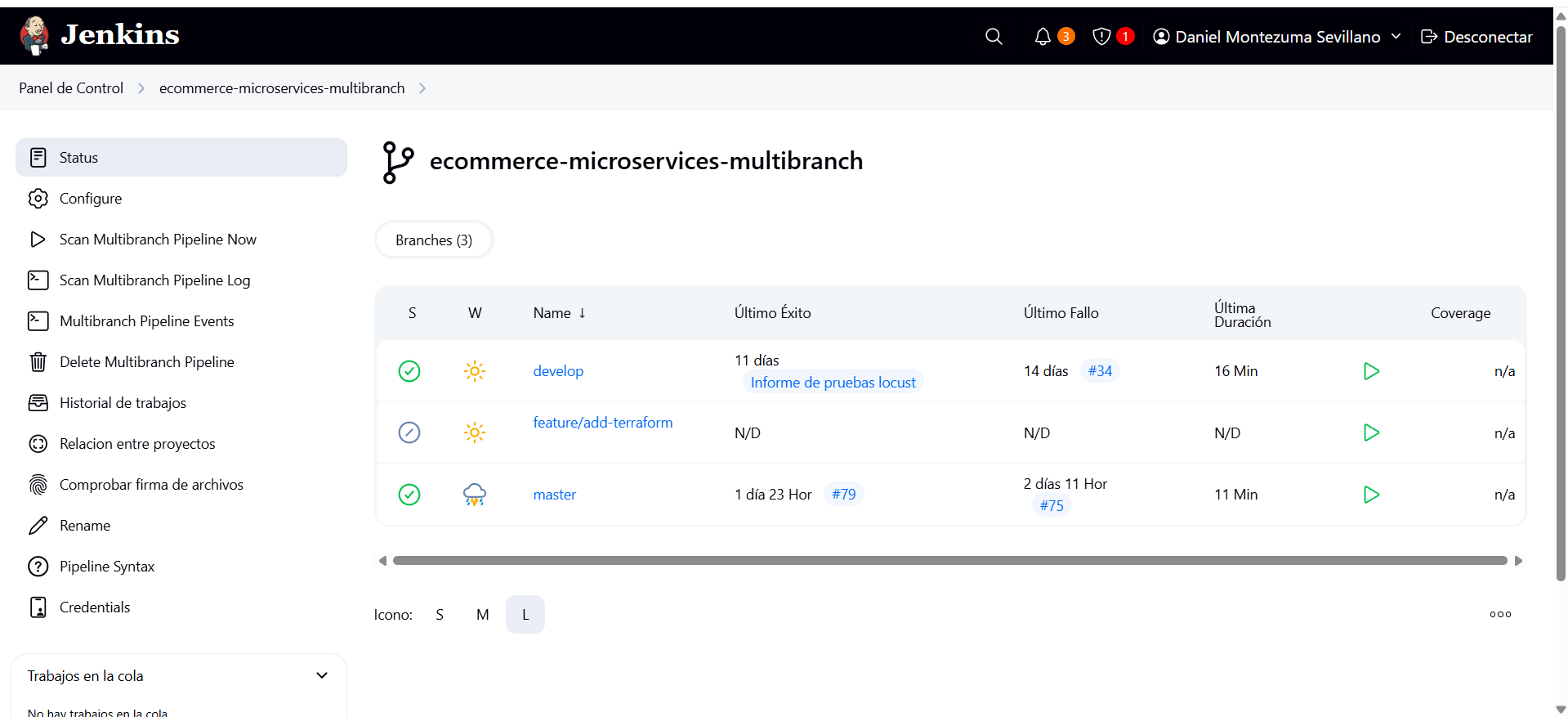
* DOCKERHUB\_CREDENTIALS, GITHUB\_TOKEN, ENVIRONMENT, IMAGE\_TAG, RELEASE\_VERSION

### **Ambientes separados**

Se implementaron tres entornos de ejecución, definidos automáticamente según la rama:

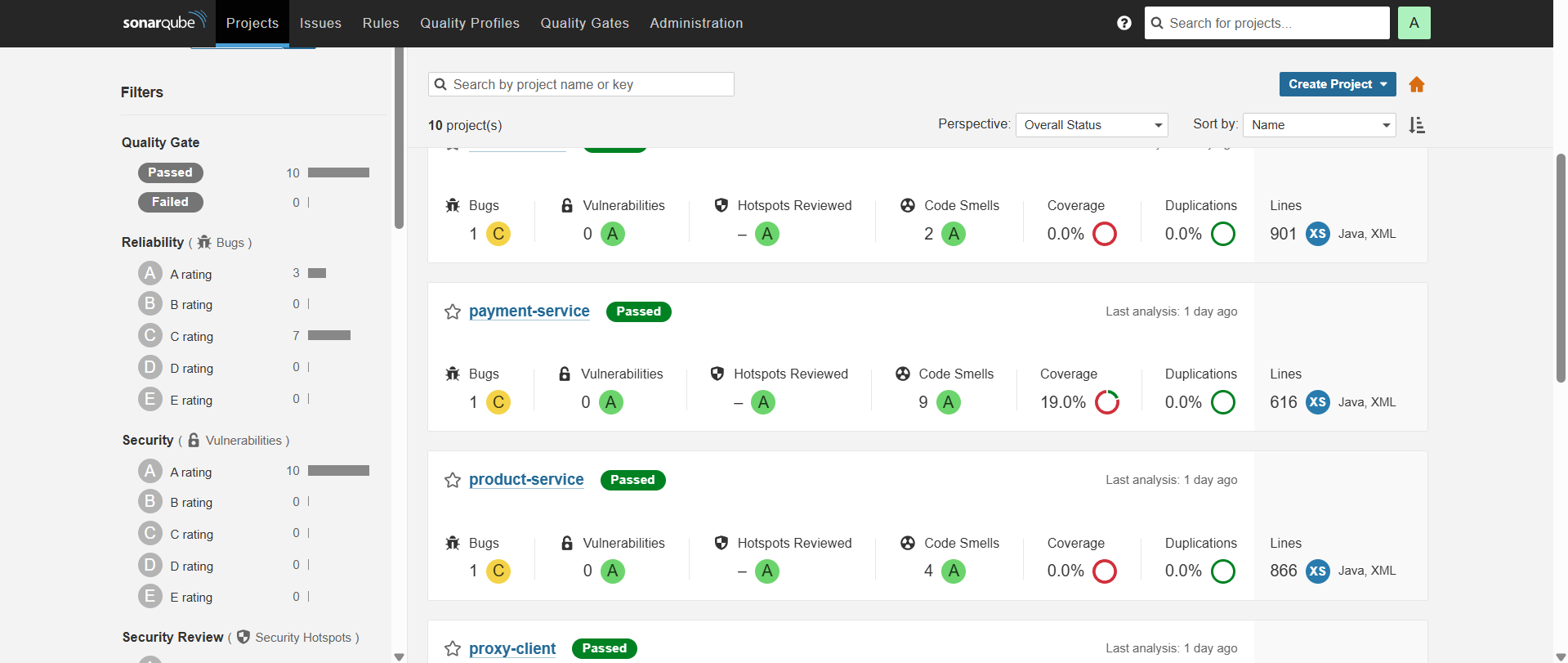
* develop ➔ ambiente de desarrollo con pruebas de carga.
* stage ➔ entorno preproductivo con pruebas de integración.
* main o master ➔ entorno de producción con despliegue controlado.

Cada entorno tiene su propio namespace en Kubernetes y configura variables de entorno distintas.



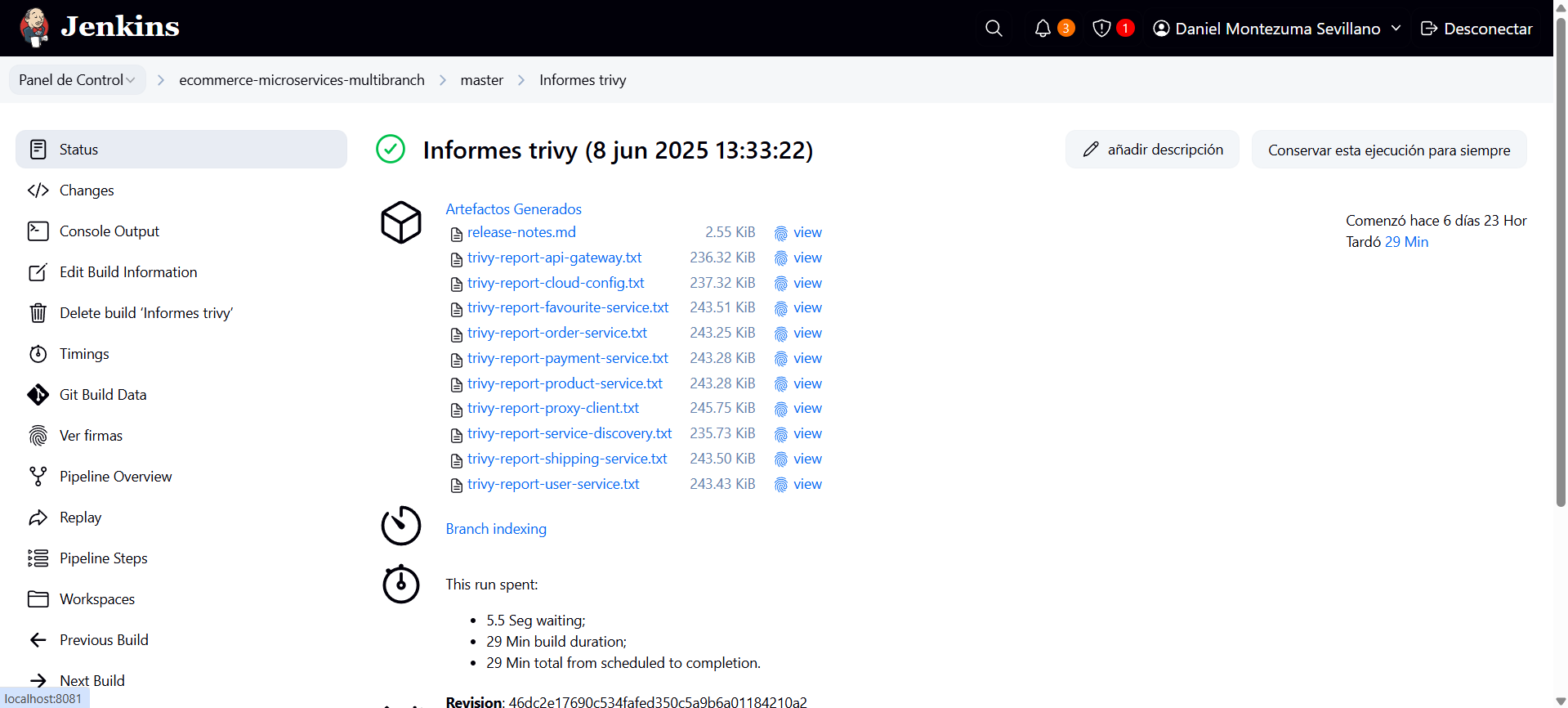
### **Análisis estático de código con SonarQube**

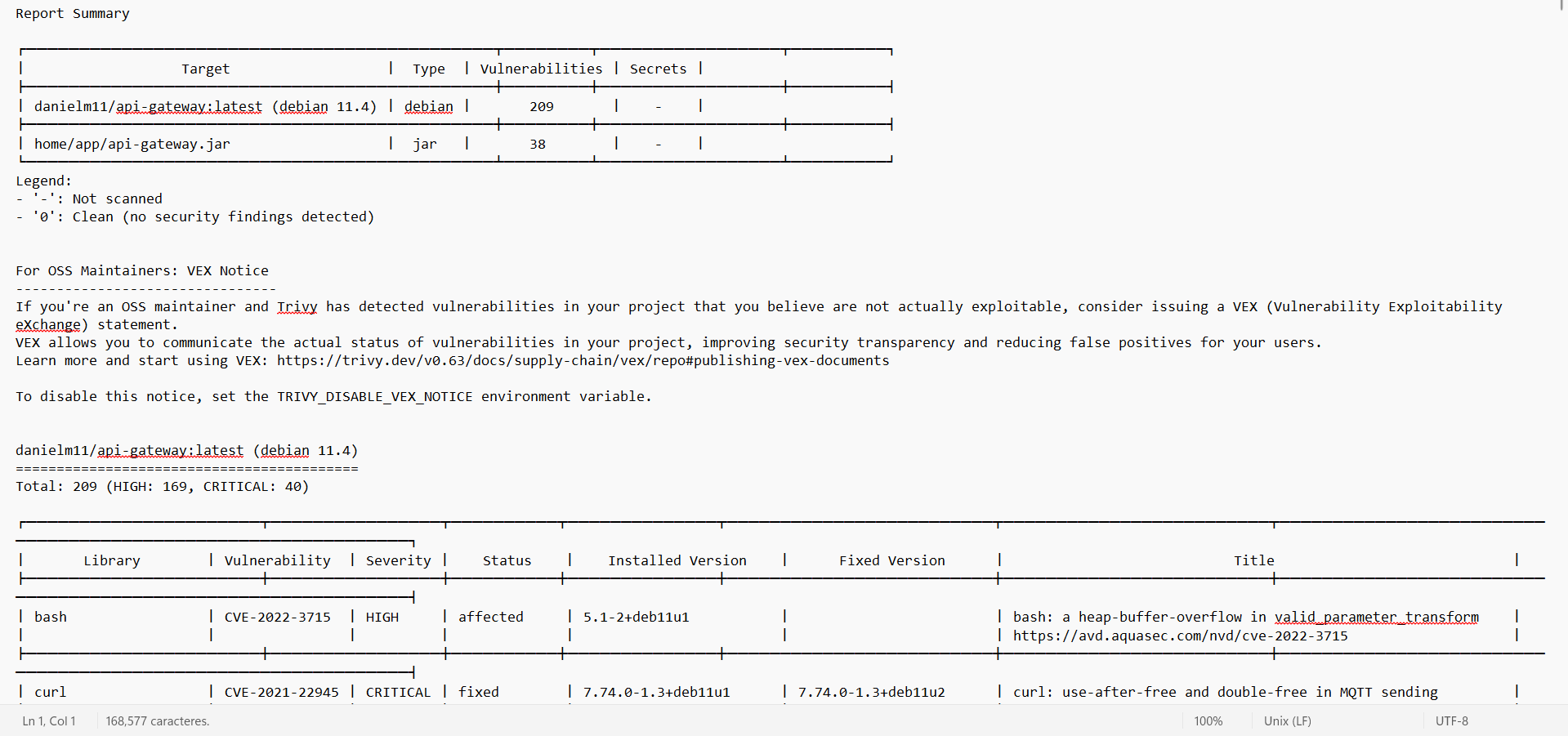
Mediante el plugin de Maven, cada servicio es analizado por SonarQube. En el user-service, se integra también con Jacoco para medir la cobertura de pruebas.



### **Trivy para escaneo de vulnerabilidades**

Trivy escanea cada imagen Docker generada, buscando vulnerabilidades CRITICAL y HIGH. Los reportes se almacenan como artefactos por Jenkins.



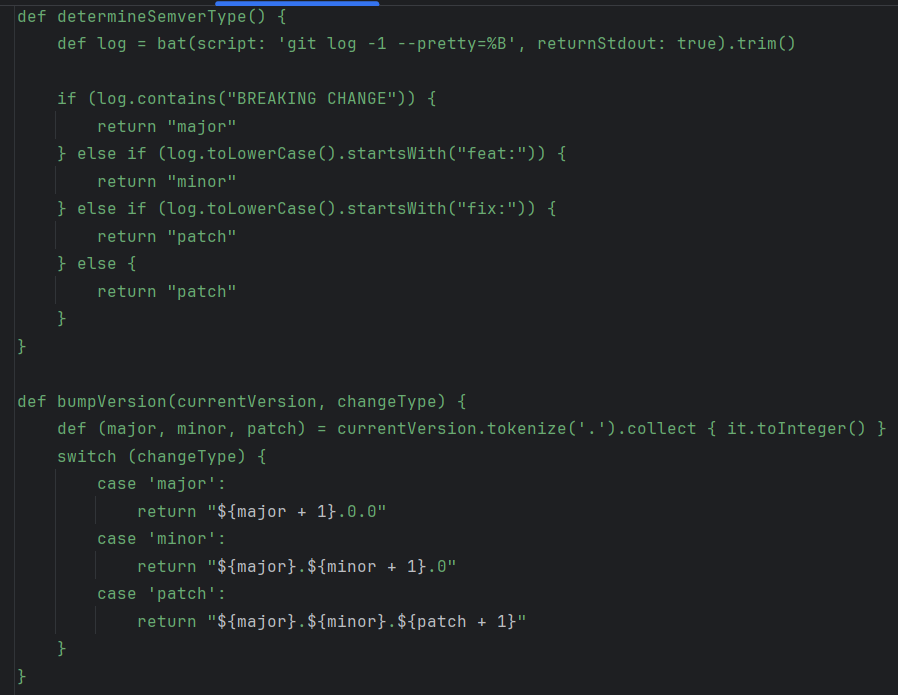


### **Versionado semántico automático**

Jenkins analiza el mensaje del último commit para determinar el tipo de versión:

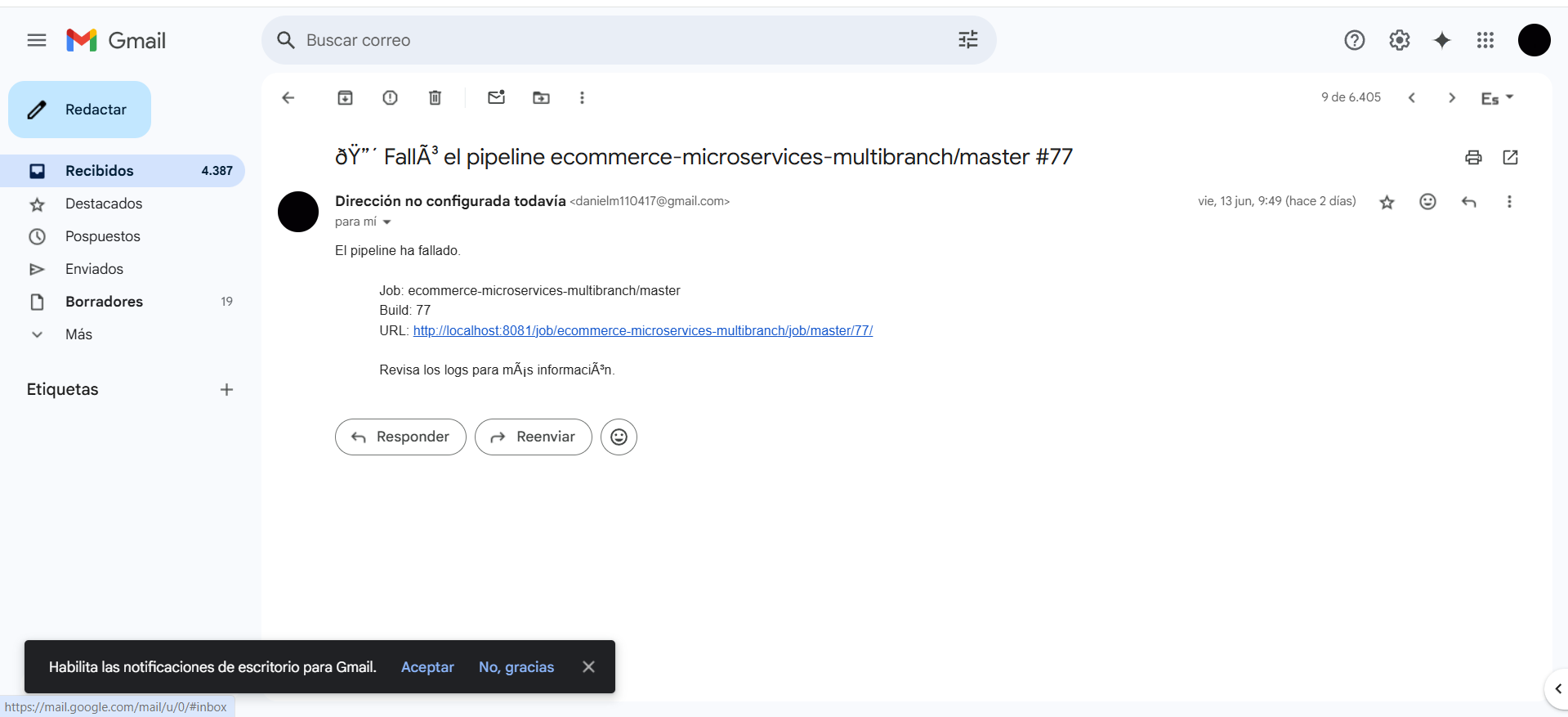
* feat: ➔ minor
* fix: ➔ patch
* BREAKING CHANGE ➔ major

Genera la versión resultante, crea el git tag y lo sube a GitHub.



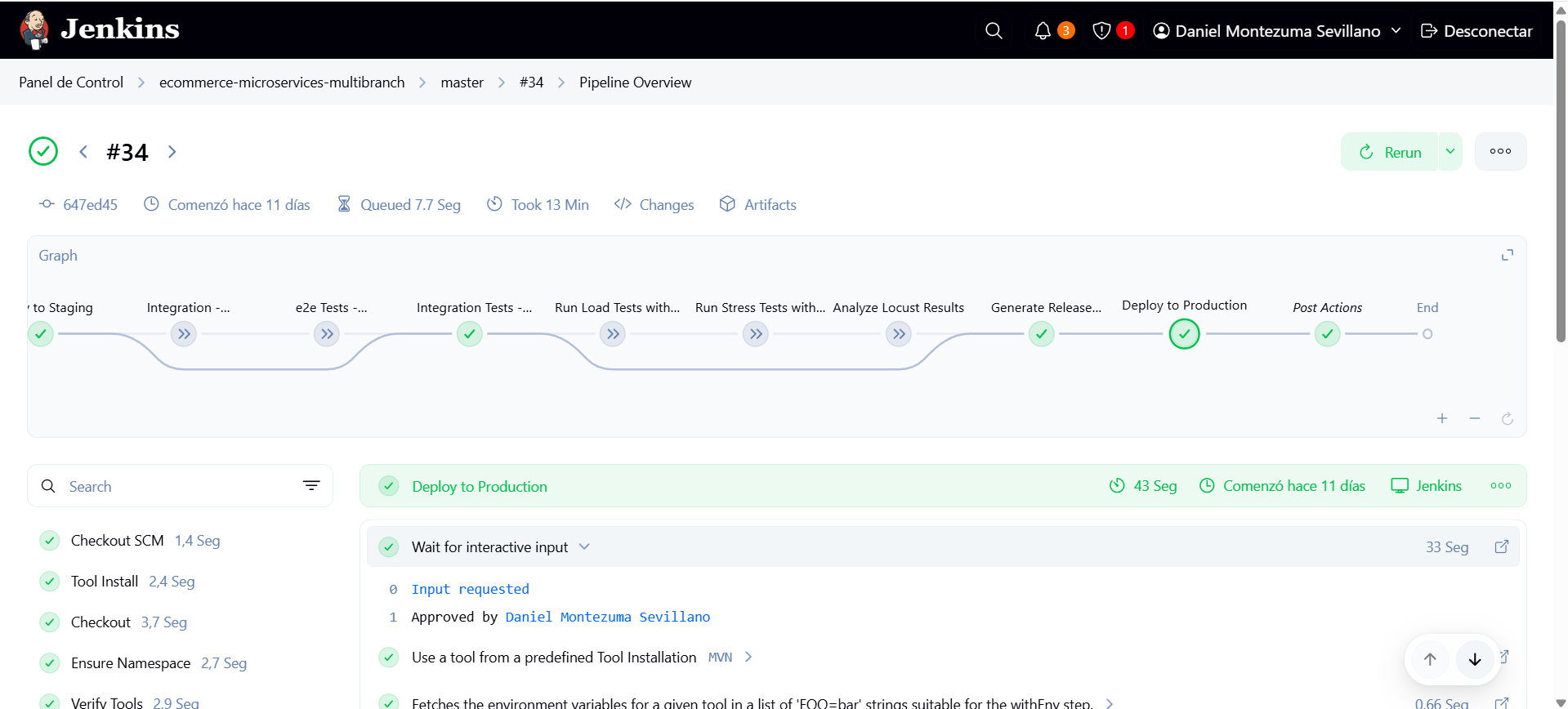
### **Notificaciones automáticas por fallos**

Si el pipeline falla, se envía un correo automático al desarrollador, se configuró en el POST del pipeline de Jenkins.



### **Aprobación manual para producción**

El despliegue a prod requiere la aprobación manual mediante input en el pipeline. Además, se valida la versión y se generan las notas de la release.

****

**Pruebas Completas**

El pipeline automatiza la ejecución de distintas pruebas según el entorno y rama:

### **Pruebas unitarias**

Se implementaron pruebas unitarias en user-service, validadas mediante JUnit y ejecutadas en Jenkins. Se asegura la cobertura mínima y se integran con Jacoco.

### **Pruebas de integración**

Implementadas en user-service y payment-service, se encargan de validar lógica que interactúa con otras capas o bases de datos.

### **Pruebas End-to-End (E2E)**

Se utilizaron pruebas e2e desde un módulo independiente e2e-tests, que simula flujos completos de usuario para user-service y product-service.

### **Pruebas de rendimiento y estrés (Locust)**

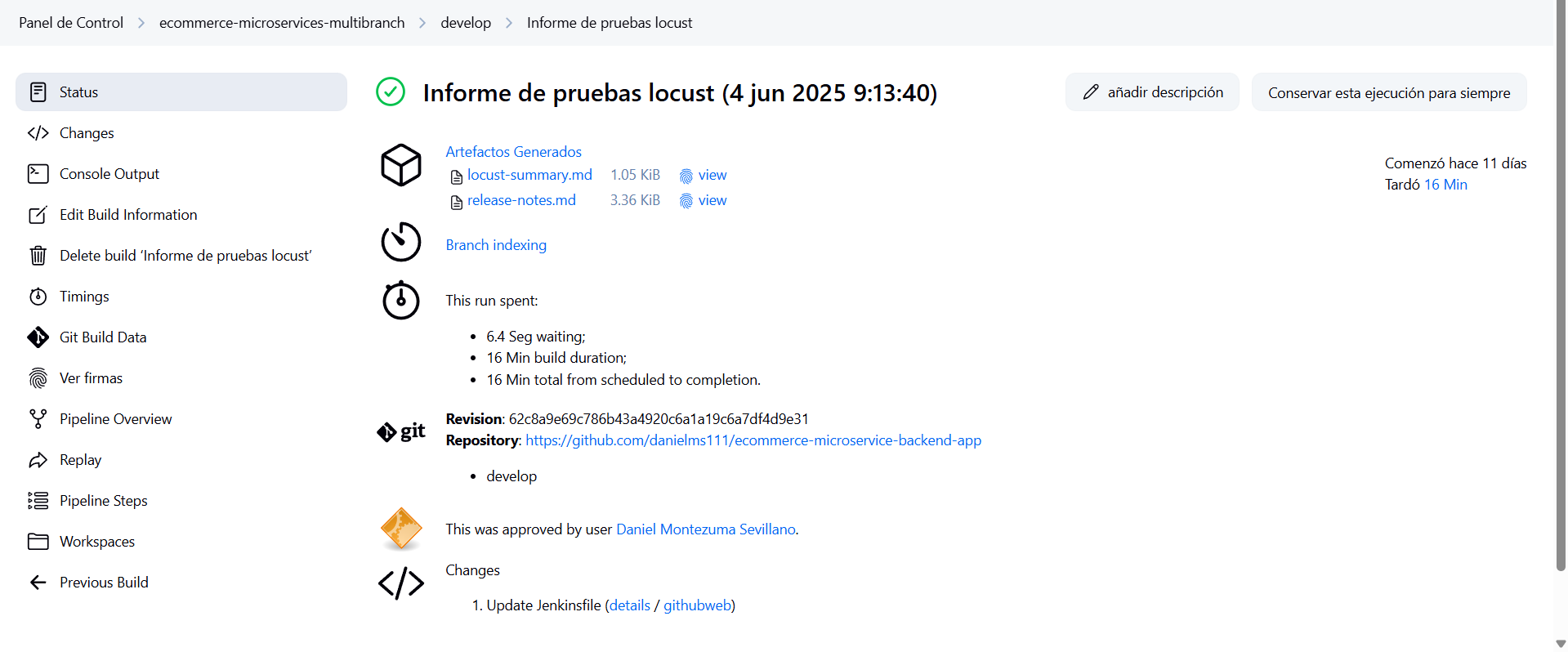
Se automatizó la ejecución de pruebas de carga y estrés para:

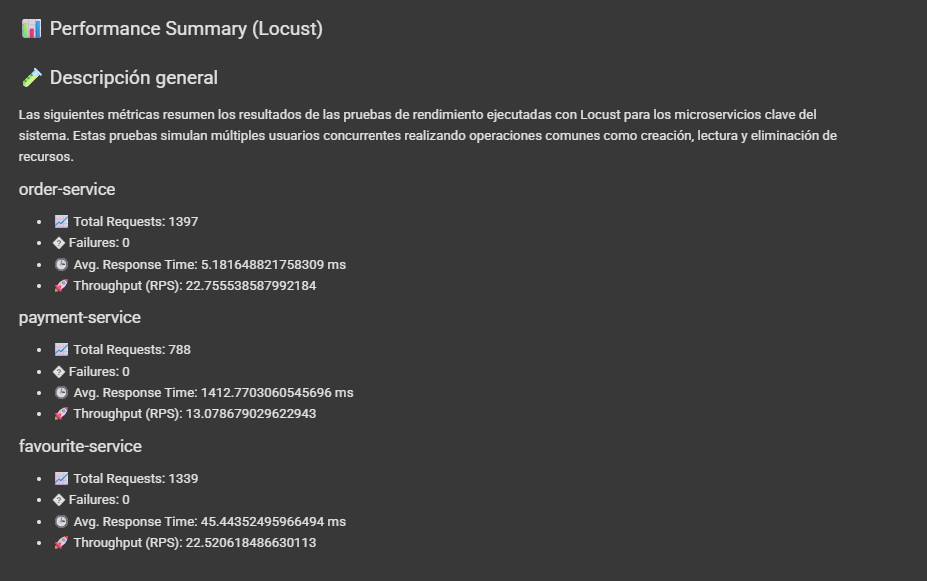
* order-service
* payment-service
* favourite-service

Se utilizaron contenedores Docker que ejecutan locust en modo headless, generando archivos .csv con resultados (RPS, tiempo de respuesta, fallos, etc.)

### **Reportes de pruebas**

Jenkins genera un archivo locust-summary.md que resume los resultados por microservicio.





### **Automatización por entorno**

* En ramas feature/ y develop se ejecutan pruebas e2e y Locust.
* En ramas release/ y main, solo pruebas unitarias y de integración.

**Change Management y Release Notes**

* **Proceso de Change Management:**

**1. Solicitud del Cambio**

- Toda solicitud de cambio debe registrarse en Pull Requests.

- La descripción debe incluir:

- Tipo de cambio (bugfix, feature, improvement)

- Afectación (microservicio)

- Impacto estimado

**2. Evaluación del Cambio**

- Revisión técnica por mínimo un revisor.

- Validación de cobertura de pruebas.

**3. Aprobación del Cambio**

- Pull Request aprobado por mínimo 1 miembro del equipo.

- Solo ramas `release/\*`, `master` o `hotfix/\*` son desplegadas automáticamente.

**4. Implementación**

- Jenkins ejecuta pipeline de CI/CD.

- El cambio se despliega en entorno `stage`.

**5. Verificación**

- Se validan pruebas unitarias, e2e, integración y de rendimiento.

- Se aprueba despliegue vía Pull Request.

**6. Documentación**

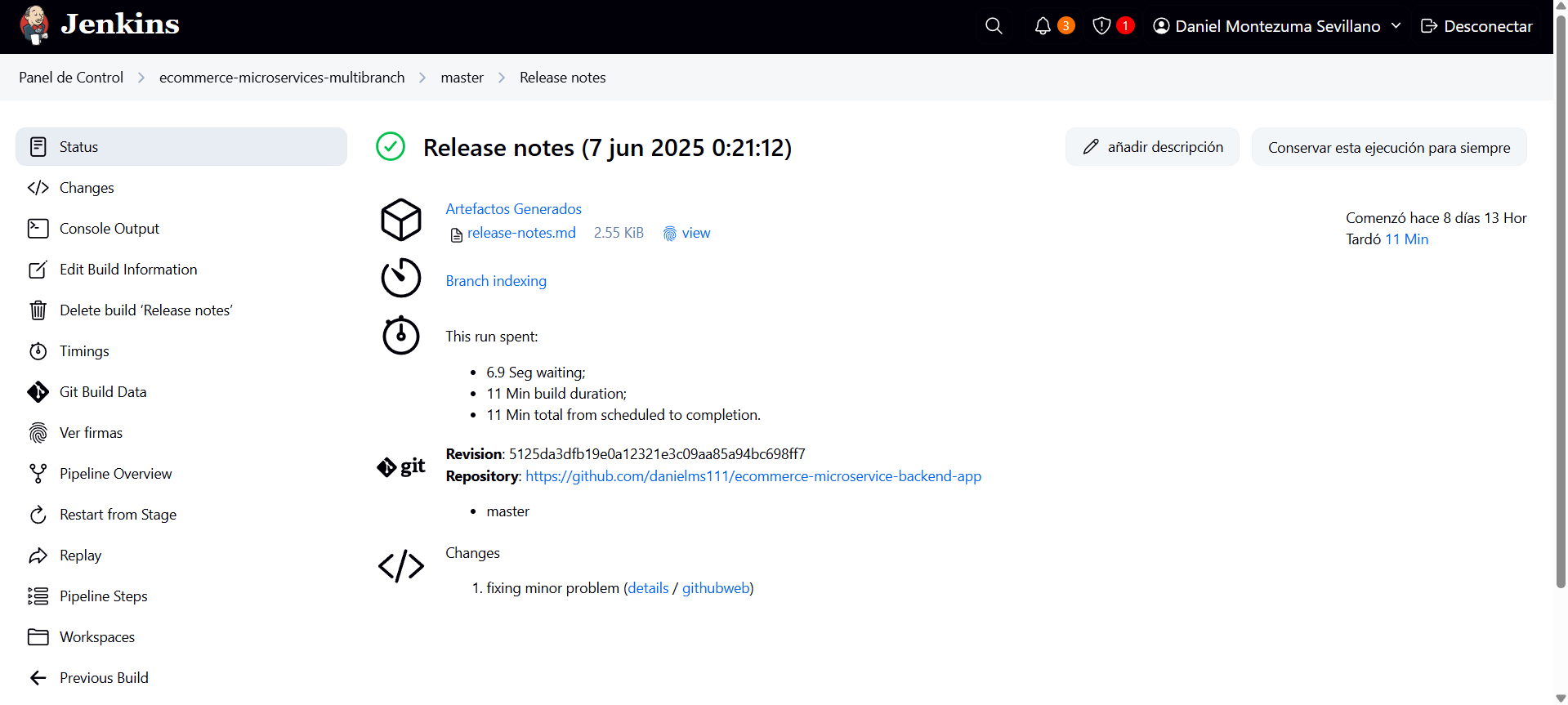
- Jenkins genera release notes.

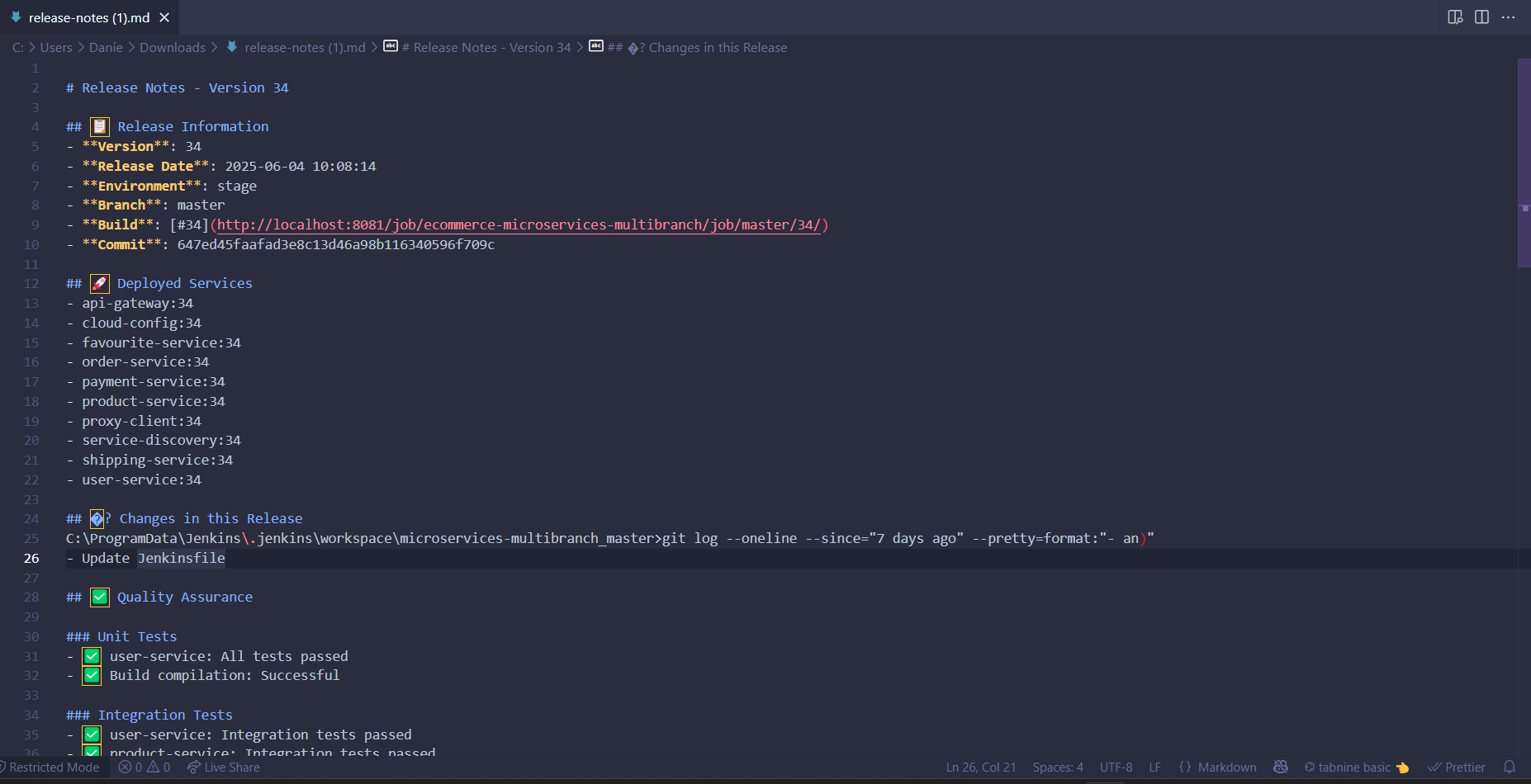
- Se crea etiqueta semántica (v1.2.0, v2.0.1, etc).

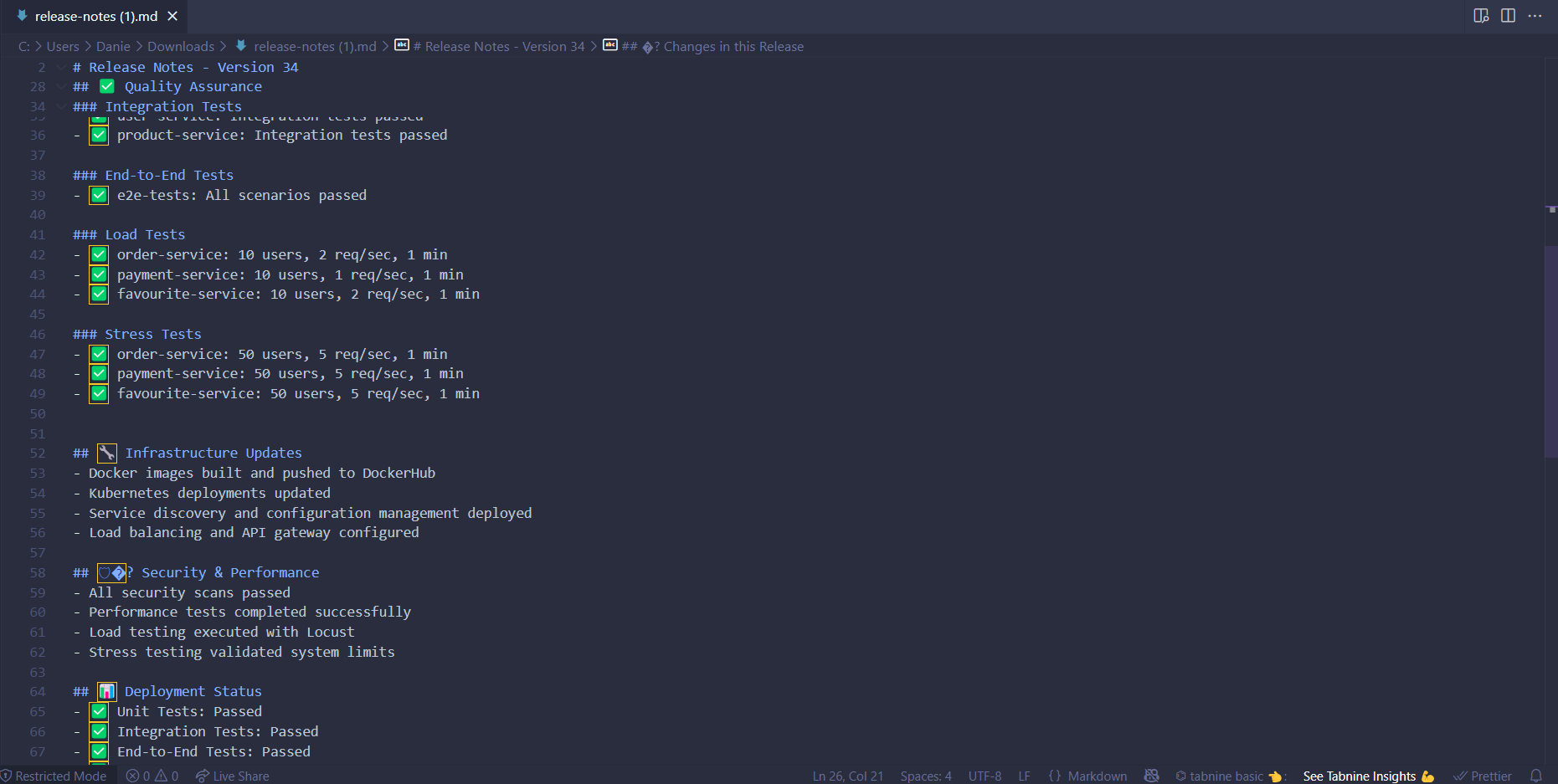
* **Implementar generación automática de Release Notes**

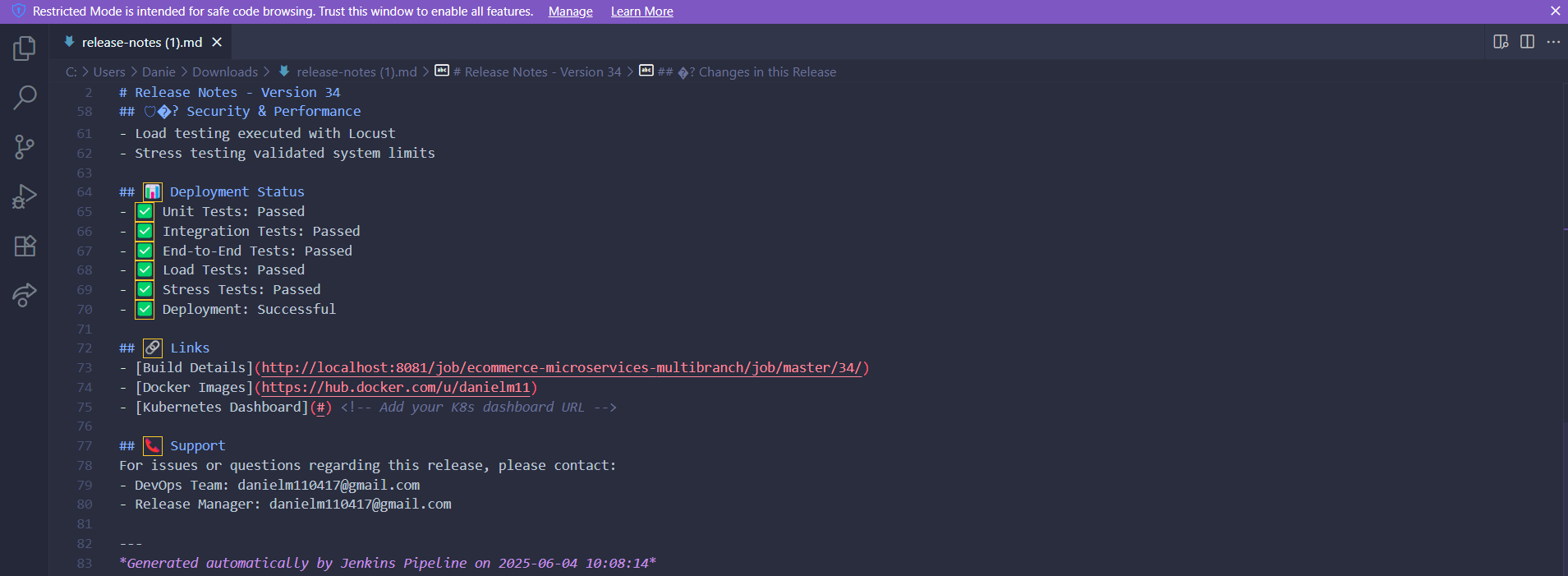
El pipeline genera un archivo Markdown con:

* Versión
* Fecha y entorno
* Servicios desplegados y sus imágenes
* Commits relevantes
* Resultados de pruebas
* Links a artefactos y recursos









* **Documentar planes de rollback**

**Objetivo**

Restaurar una versión previa del sistema en caso de fallos tras un despliegue, garantizando continuidad del servicio.

**Arquitectura Base**

- Microservicios Dockerizados

- Orquestación con Kubernetes

- CI/CD con Jenkins

- Control de versiones con Git

---

**Métodos de Rollback Disponibles**

**1. Rollback de imágenes en Kubernetes**

Cada microservicio es desplegado con un `Deployment.yaml` que especifica una imagen por versión (ej: `my-service:’version’`).

**Estrategia:**

- Usar `kubectl rollout undo` para revertir al último deployment exitoso:

kubectl rollout undo deployment/<deployment-name> -n <namespace>

### **2. Rollback desde Jenkins**

Si el pipeline falló tras deploy en prod, ejecutar manualmente el stage de rollback:

**stage('Rollback') {**

**steps {**

**sh 'kubectl rollout undo deployment/user-deployment -n prod'**

**}**

**}**

Se puede dejar este stage comentado o controlado por un input manual.

## **Condiciones para activar rollback**

* Fallo crítico en pruebas.
* Métricas de rendimiento fuera de rango (latencia, errores 5xx).
* Notificación de error por parte del equipo QA o DevOps.

## **Validación post-rollback**

* Verificar que el sistema se restableció correctamente.
* Consultar logs con kubectl logs.
* Correr tests o endpoints básicos.
* **Implementar sistema de etiquetado de releases**

En nuestro Jenkinsfile, dentro del bloque stage('Deploy to Production'), tenemos el siguiente fragmento de código:

**createGitTag(RELEASE\_VERSION)**

Este método createGitTag(version) lo tenemos definido al final del pipeline, y hace lo siguiente:

1. Configura el usuario de Git dentro de Jenkins.
2. Crea un tag con el prefijo v y el número de versión.
3. Hace push de ese tag remoto al repositorio de GitHub.

**def createGitTag(version) {**

**try {**

**withCredentials([string(credentialsId: 'github-token', variable: 'GITHUB\_TOKEN')]) {**

**bat """**

**git config user.email "jenkins@company.com"**

**git config user.name "Jenkins CI"**

**git tag -a v${version} -m "Release version ${version}"**

**git push https://%GITHUB\_TOKEN%@github.com/danielm11/your-repo.git v${version}**

**"""**

**}**

**echo "Git tag v${version} created successfully"**

**} catch (Exception e) {**

**echo "Warning: Could not create Git tag: ${e.getMessage()}"**

**}**

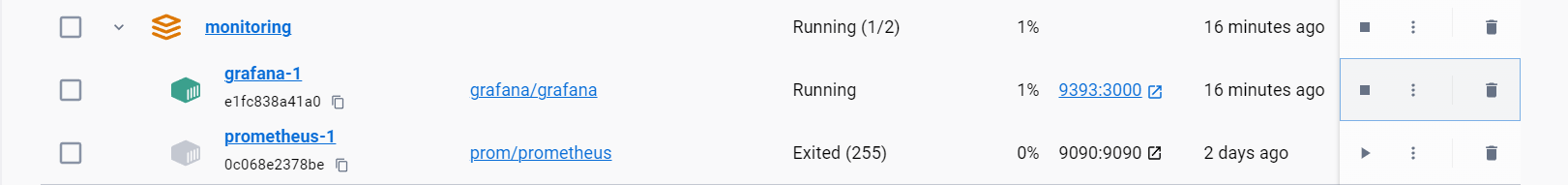
**}**

**Observabilidad y Monitoreo**

### **Stack Prometheus y Grafana**

Se implementó con Docker Compose un stack de observabilidad:

* Prometheus recolecta métricas desde /actuator/prometheus de cada microservicio.
* Grafana visualiza métricas en dashboards personalizados.



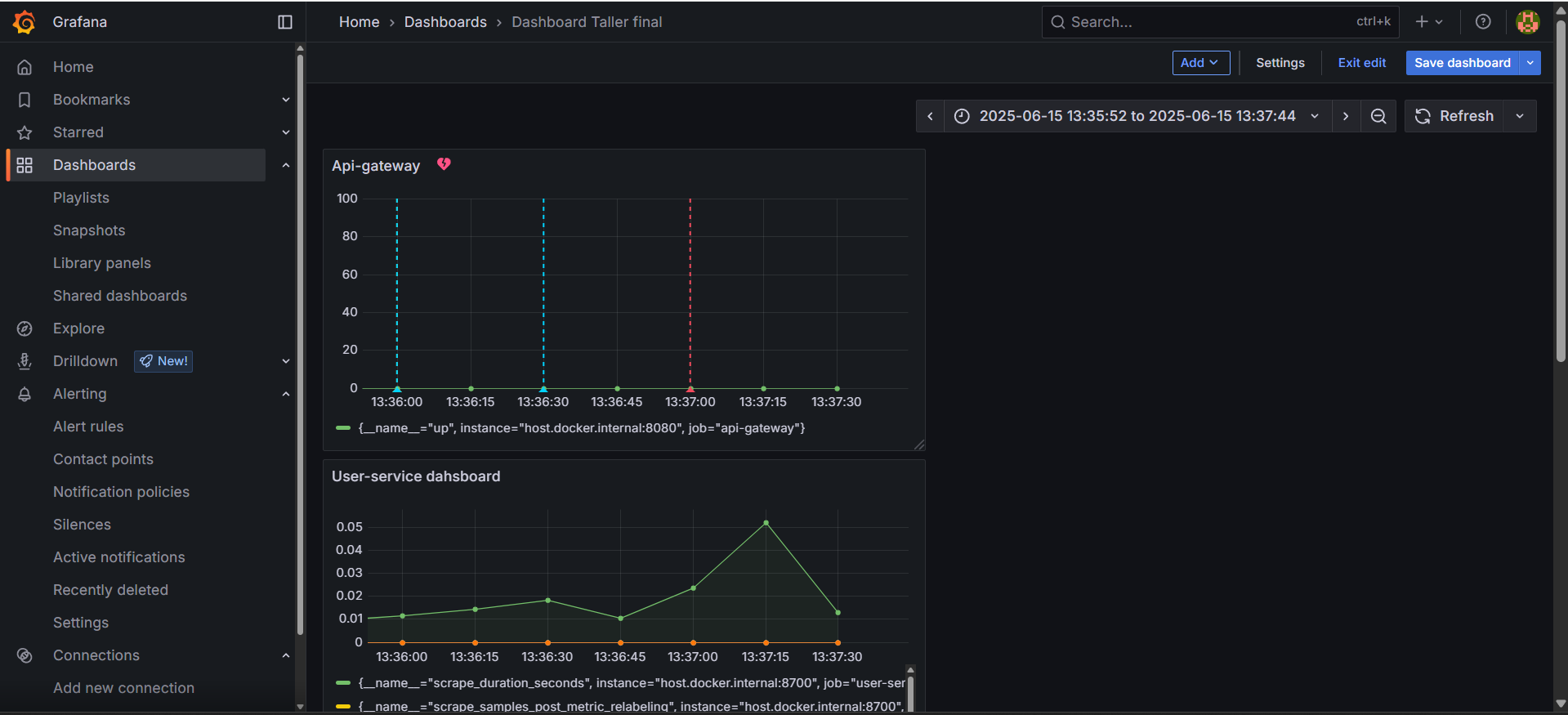
### **Dashboards**

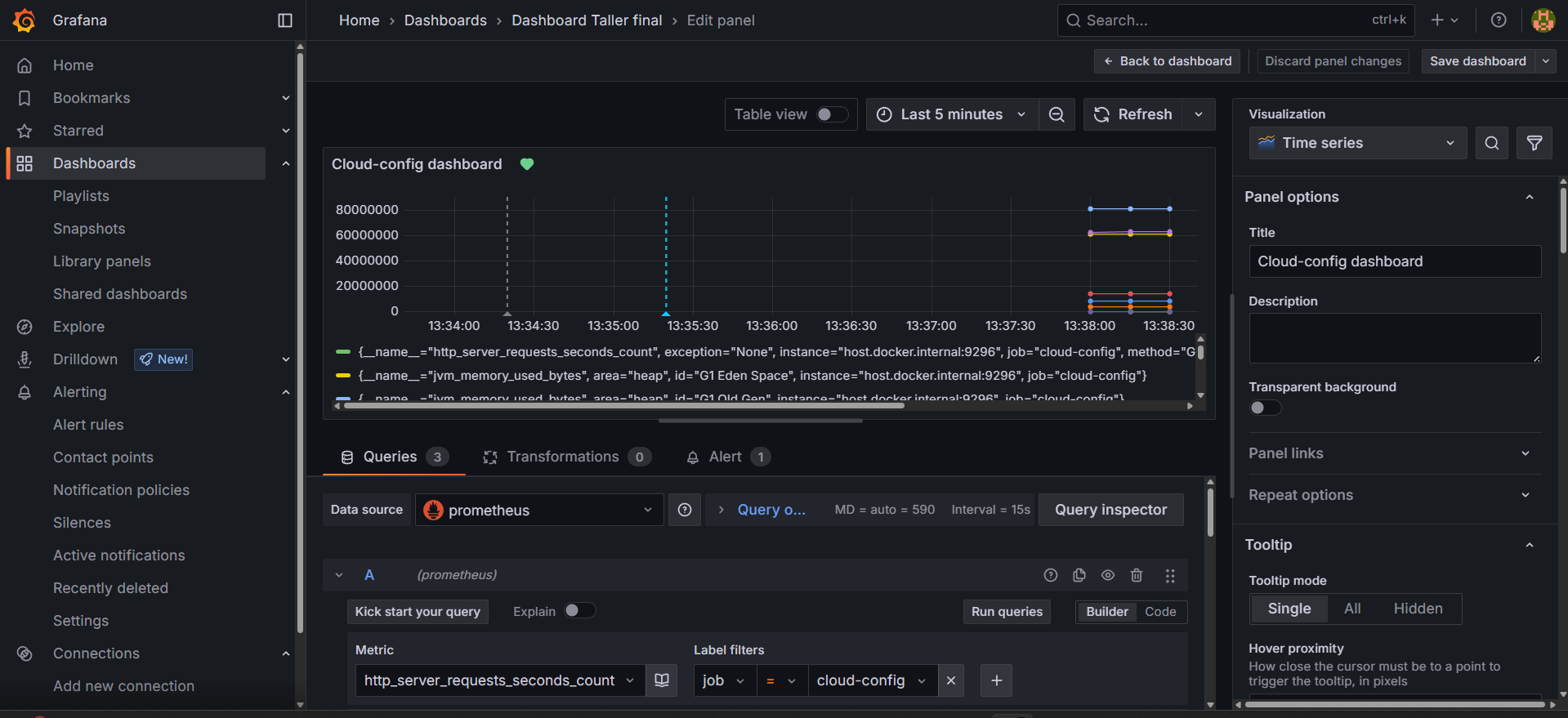
Se crearon dashboards para servicios como:

* user-service
* payment-service

Se muestran métricas como:

* Disponibilidad (up)
* Tiempo de respuesta promedio
* Errores HTTP





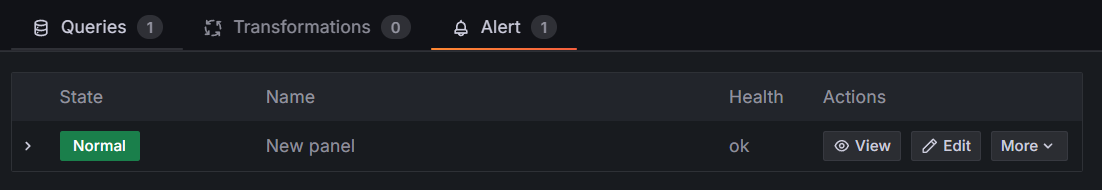
### **Alertas**

Se configuraron alertas en Grafana que se activan cuando:

* up = 0 (servicio caído)
* Tiempo de respuesta supera umbrales

Las alertas se enviaban por correo.

### 



### **Tracing distribuido (Zipkin)**

Se desplegó Zipkin en local y en Kubernetes. Todos los microservicios integraron Spring Cloud Sleuth:

* Cada petición genera una traza
* Las trazas se visualizan en la UI de Zipkin
* Se identifican cuellos de botella o servicios lentos

### **Health Checks y Probes**

Todos los deployments en Kubernetes tienen configuradas:

livenessProbe:

httpGet:

path: /actuator/health

port: <puerto>

readinessProbe:

httpGet:

path: /actuator/health

port: <puerto>

Esto permite a Kubernetes reiniciar o remover servicios según su estado.

### **Métricas de negocio**

Se implementaron endpoints de métricas específicas como:

* Cantidad de órdenes procesadas
* Total de pagos
* Productos favoritos por usuario Estas métricas se recolectan en Prometheus para análisis de negocio.

### **Seguridad**

#### **Escaneo continuo de vulnerabilidades**

* **Herramienta:** Trivy
* **Integración:** En la pipeline de Jenkins para cada imagen Docker de los microservicios.
* **Resultado:** Se generan reportes automáticos y se archivan para revisión.

#### **Gestión segura de secretos**

* **Herramienta:** Jenkins Credentials
* **Uso:** Tokens para SonarQube, GitHub, y DockerHub se almacenan cifrados y acceden usando withCredentials.

### **Manual básico de operaciones**

#### **Despliegue de microservicios**

1. Asegúrate de tener minikube y kubectl funcionando.
2. Ejecuta:

kubectl config current-context

kubectl create namespace ecommerce

1. Aplica los YAML desde k8s/:  
     
   kubectl apply -f k8s/cloud-config -n ecommerce

kubectl apply -f k8s/service-discovery -n ecommerce

kubectl apply -f k8s/api-gateway -n ecommerce

kubectl apply -f k8s/product-service -n ecommerce

...

#### **Rollback de despliegues**

1. Para revertir una versión fallida:  
     
   kubectl rollout undo deployment/<nombre-del-servicio> -n ecommerce
2. Desde Jenkins (opcional): ejecutar manualmente el stage Rollback.

#### **Ver logs**

kubectl logs deployment/<nombre-del-servicio> -n ecommerce

#### **Acceso a herramientas de monitoreo**

* Prometheus:<http://localhost:9090>
* Grafana:<http://localhost:3000>
* Kibana:<http://localhost:5602>

#### **Verificación de estado de microservicios**

kubectl get pods -n ecommerce

kubectl get svc -n ecommerce

#### **Reemplazo de secretos en Jenkins**

1. Ir a Jenkins > Manage Jenkins > Credentials
2. Buscar los IDs: github-token, sonarqube-token, password
3. Actualizar los valores si es necesario

**LINK AL REPOSITORIO:**https://github.com/Danielms111/ecommerce-microservice-backend-app/tree/master