

# **Despliegue de Arquitectura Escalable en Kubernetes con enfoque GitOps en AWS con GitHub Actions y ArgoCD**

Luisa Fernanda Castaño Pino  
Juan Eduardo Yustes Nieto  
Santiago José Barraza Sining

Plataformas II  
Ingeniería de Software V

Ingeniería Telemática  
Ingeniería de Sistemas

Universidad ICESI

Christian David Flor Astudillo  
Juan Carlos Muñoz Fernández

<b>Introducción.....</b>	<b>5</b>
<b>Estructura del Proyecto.....</b>	<b>6</b>
<b>Flujo de trabajo entre ramas y entornos (GitFlow + Staging).....</b>	<b>7</b>
Uso del tablero Kanban durante el proyecto.....	10
Ramas implementadas.....	12
<b>Tests de los Microservicios.....</b>	<b>14</b>
user-service.....	15
Tests unitarios.....	15
Tests de integración.....	16
Tests e2e.....	18
Ejecución.....	19
product-service.....	20
Tests unitarios.....	20
Tests de integración.....	21
Tests e2e.....	23
Ejecución.....	24
order-service.....	25
Tests unitarios.....	25
Tests de integración.....	26
Tests e2e.....	28
Ejecución.....	29
<b>Build del Proyecto (Backend).....</b>	<b>30</b>
<b>Arquitectura de Infraestructura en la Nube.....</b>	<b>31</b>
Infraestructura creada manualmente.....	32
Integración GitHub Actions + Terraform.....	33
DNS en GoDaddy.....	34
<b>Arquitectura de Kubernetes.....</b>	<b>34</b>
Despliegue y Gestión de Aplicaciones.....	34
Networking y Entrada de Tráfico.....	35
Certificados y Seguridad.....	35
Entornos de Despliegue.....	36
Contenedores e Imágenes.....	36
Flujo de Integración Continua y Entrega Continua (CI/CD).....	36
Argo CD.....	37
GitHub Actions.....	37
Monitoreo y Observabilidad.....	37
Monitoring (Métricas).....	38
Logging (Registros).....	38
<b>Patrones de Cloud.....</b>	<b>38</b>
Patrón Retry.....	38
Patrón External Configuration Store (Configmaps).....	39

Patrón Health Endpoint Monitoring.....	39
Patrón Sidecar.....	39
<b>Addons del Cluster de EKS.....</b>	<b>40</b>
Amazon VPC CNI.....	40
kube-proxy.....	40
CoreDNS.....	40
Amazon EBS CSI Driver.....	41
<b>Configuración de Red y Seguridad.....</b>	<b>41</b>
Servicios Kubernetes configurados correctamente.....	41
Implementación de Ingress Controller para el acceso a la API Gateway y al Frontend:.....	42
Instalación del NGINX Ingress Controller:.....	42
Configuración de Ingress para el Frontend.....	43
NetworkPolicies configuradas para restringir comunicación entre servicios:.....	44
2. Permisos específicos entre microservicios.....	45
3. Trazabilidad y control para observabilidad.....	45
4. Relaciones simétricas entre servicios.....	46
Uso de Calico para habilitar NetworkPolicies.....	46
Configuración de Red y Políticas de Red (Network Policies).....	47
Configuración de TLS/HTTPS para endpoints públicos:.....	47
1. Instalación de Cert-Manager.....	47
2. Creación del ClusterIssuer:.....	48
Implementación del Service Mesh con Linkerd para habilitar Sidecar Proxies.....	50
Implementación de ServiceAccounts con permisos mínimos necesarios RBAC:.....	53
Escaneo de imágenes en busca de vulnerabilidades:.....	54
<b>Gestión de configuración y secretos.....</b>	<b>62</b>
Migración de todas las configuraciones de Spring Boot a ConfigMaps:.....	62
Servicios que utilizan el ConfigMap.....	62
Implementación de Secrets para credenciales y datos sensibles.....	63
Implementación de rotación de secretos.....	64
Uso de variables de entorno y/o volúmenes para injectar configuraciones.....	65
Gestión centralizada de la configuración (aprovechando el servicio de Cloud Config)....	66
<b>Estrategias de Despliegue y CI/CD.....</b>	<b>66</b>
Pipeline de CI/CD completo con Jenkins, GitLab CI o GitHub Actions para todos los microservicios.....	66
Repositorios de ECR con las Imágenes de Docker.....	70
Uso de Helm Charts para empaquetar los microservicios.....	71
Implementación de dependencias correctas en el orden de despliegue.....	73
<b>Almacenamiento y Persistencia.....</b>	<b>74</b>
Implementación de Persistent Volumes y Persistent Volume Claims para las bases de datos.....	74
Bases de datos implementadas.....	75
Ventajas de la Implementación.....	75

Configuración de StorageClass adecuada para el entorno.....	75
<b>Observabilidad y Monitoreo.....</b>	<b>76</b>
Implementación completa de Prometheus + Grafana.....	76
Recursos a nivel de cluster.....	77
Recursos a nivel de namespace:.....	78
Vista a nivel de network:.....	78
Uso de PersistentVolumes por namespace:.....	80
Monitoreo de los Kubelets:.....	80
Recursos a nivel de nodo.....	81
Datasources.....	82
Aprovechamiento de los Actuator endpoints de Spring Boot para monitoreo.....	82
Configuración de alertas para situaciones críticas.....	83
Sistema de logging centralizado.....	85
Tracing con Zipkin.....	85
<b>Autoscaling y Pruebas de Rendimiento.....</b>	<b>86</b>
Implementación de Horizontal Pod Autoscaler.....	86
Pruebas de estrés con Locust.....	87
Pruebas de Caos con Chaos-Mesh.....	91
<b>Escaneo de Código con SonarQube.....</b>	<b>92</b>
Escaneo desde SonarCloud.....	92
Escaneo desde GitHub.....	96
Pruebas como Gate en PR.....	97
<b>Conclusiones.....</b>	<b>98</b>
<b>Referencias:.....</b>	<b>99</b>

## **Introducción**

La evolución de los sistemas distribuidos y el auge de la computación en la nube han llevado al desarrollo de arquitecturas modernas basadas en microservicios, altamente escalables, resilientes y mantenibles, en este contexto, el presente proyecto se enmarca en el curso Plataformas II y tiene como objetivo desplegar una solución e-commerce compuesta por múltiples microservicios, sobre un clúster de Kubernetes en Amazon Web Services (AWS), aplicando principios avanzados de DevOps y GitOps para garantizar una automatización completa del ciclo de vida de las aplicaciones.

El punto de partida fue un repositorio base con microservicios desarrollados en Spring Boot, que incluyen servicios como usuario, producto, orden, pago, envío, favoritos, discovery, configuración centralizada, API Gateway y cliente proxy, así que a partir de este backend, se implementaron pruebas unitarias, de integración y end-to-end para garantizar su funcionamiento correcto y robusto, para que posteriormente, se automatizó la construcción de imágenes Docker y su despliegue en AWS ECR mediante GitHub Actions, asegurando así la integración continua (CI).

En cuanto a la infraestructura, se diseñó e implementó mediante Terraform, utilizando un enfoque Infraestructura como Código (IaC), desplegando componentes como VPC, subredes públicas, grupos de seguridad, clúster de EKS (Elastic Kubernetes Service), balanceadores de carga y almacenamiento persistente. Para la gestión del estado remoto y la concurrencia en Terraform se utilizaron servicios como S3 y DynamoDB, respectivamente, adicionalmente, se configuró el dominio personalizado lsj-app.xyz en GoDaddy, apuntando a los endpoints públicos de la infraestructura mediante registros DNS y certificados TLS automáticos gestionados con Cert-Manager y Let's Encrypt.

La arquitectura de Kubernetes se estructuró cuidadosamente en múltiples namespaces (dev, stage y master) para separar los entornos de desarrollo, preproducción y producción, pues esta organización permitió aplicar una estrategia de despliegue basada en GitOps, utilizando ArgoCD como herramienta principal para sincronizar automáticamente el estado deseado (definido en Helm Charts versionados en Git) con el estado actual del clúster, evitando así intervenciones manuales en el proceso de entrega continua (CD).

En cuanto al despliegue de las aplicaciones, se utilizó Helm para empaquetar cada microservicio con sus respectivos manifiestos Kubernetes, lo que facilitó la parametrización de variables por entorno (values-dev.yaml, values-stage.yaml, etc.) y el versionamiento de cada release. Las pipelines CI/CD fueron diseñadas para construir, testear, subir las imágenes y finalmente actualizar las referencias de versión en los charts, lo que permitía que ArgoCD detectará automáticamente los cambios y ejecutará los despliegues.

La estrategia de control de versiones y flujos de trabajo entre ramas evolucionó a lo largo del desarrollo del proyecto. Inicialmente, se utilizó GitHub Flow por su agilidad para iterar rápidamente en fases tempranas. Sin embargo, para garantizar una gestión más estructurada y segura en ambientes múltiples, se migró a Git Flow, donde las ramas dev, stage y master se

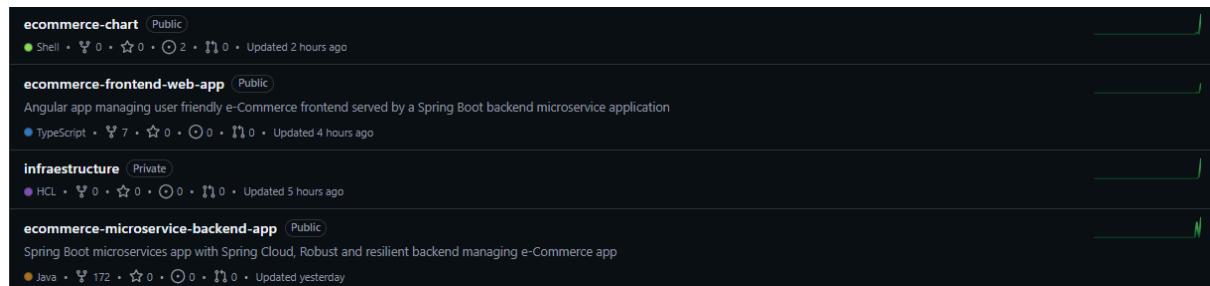
alinean directamente con los entornos correspondientes en Kubernetes, facilitando así la trazabilidad, control y estabilidad del sistema.

En términos de seguridad y control, se implementaron NetworkPolicies usando Calico, aislando el tráfico entre pods según necesidades explícitas, junto con ServiceAccounts configurados bajo el principio de mínimo privilegio. Para monitoreo y observabilidad se integró Prometheus, Grafana, ELK Stack y Zipkin, permitiendo trazabilidad de eventos, visualización de métricas y diagnóstico centralizado de errores. Se configuraron alertas en Prometheus y pruebas de estrés con Locust para validar el comportamiento del sistema bajo carga.

Asimismo, se abordaron aspectos críticos de configuración y gestión de secretos, migrando variables sensibles a Kubernetes Secrets, utilizando ConfigMaps centralizados, e implementando un sistema de rotación automática de secretos como demostración académica, todo esto permitió construir un sistema seguro, flexible y preparado para producción.

En resumen, este proyecto representa la integración práctica de múltiples herramientas, estándares y patrones de diseño modernos (como Retry, External Configuration Store, Sidecar y Health Check), aplicados a una arquitectura real de microservicios en la nube. La solución resultante es altamente automatizada, observable, segura y escalable, demostrando un dominio integral en la implementación de plataformas distribuidas modernas bajo un enfoque GitOps.

## Estructura del Proyecto



Se tienen los repositorios, cada uno cumpliendo un papel específico:

- **infrastructure**: Dentro de sí están los archivos de terraform, en este caso, la infraestructura que se despliega está en la carpeta de “aws-terraform”, ya que en la carpeta
- **ecommerce-chart**: Contiene los manifiestos de helm de los microservicios, incluyendo los proyectos de los que ArgoCD va a gestionar su ciclo de vida. Dentro de la carpeta templates contiene todo lo necesario para desplegar los microservicios, es decir, los:
  - Deployments
  - Services
  - Configmaps

- Secrets
- NetworkPolicies
- HorizontalPodAutoscaler
- Ingress
- **ecommerce-microservice-backend-app:** Este repositorio contiene los microservicios que están hechos en spring boot, en los cuales en algunos de estos contienen un Dockerfile para construir su imagen usando las respectivas variables de entorno necesarias.
- **ecommerce-frontend-web-app:** Este repositorio contiene el código del frontend hecho en Angular, el cual, al igual que el de los microservicios, tiene un Dockerfile para construir la imagen. Este frontend lo único que hace es llamar al endpoint de categorías del product-service y mostrarlas en pantalla.

### **Flujo de trabajo entre ramas y entornos (GitFlow + Staging)**

Durante el desarrollo del proyecto se utilizó GitHub Flow por su simplicidad, ideal para iteraciones rápidas en un entorno de automatización con GitHub Actions, esta estrategia permitió crear ramas de funcionalidades (feature/\*) y realizar integraciones continuas hacia la rama principal (main) a través de Pull Requests, facilitando la colaboración y la validación automatizada del código.

Sin embargo, para el despliegue final y el mantenimiento del sistema en producción, se adoptó una estrategia basada en Git Flow, más robusta y adecuada para manejar múltiples entornos, esta transición responde a la necesidad de controlar y trazar claramente el paso del código por entornos como dev, stage y prod.

En Git Flow, se utilizan ramas permanentes como:

- dev: Integración de nuevas funcionalidades.
- stage: Validación previa a producción.
- master: Código estable y liberado en producción.

Cada rama está vinculada directamente a un namespace específico en Kubernetes sobre AWS, y cualquier modificación en ellas activa un flujo automatizado que incluye:

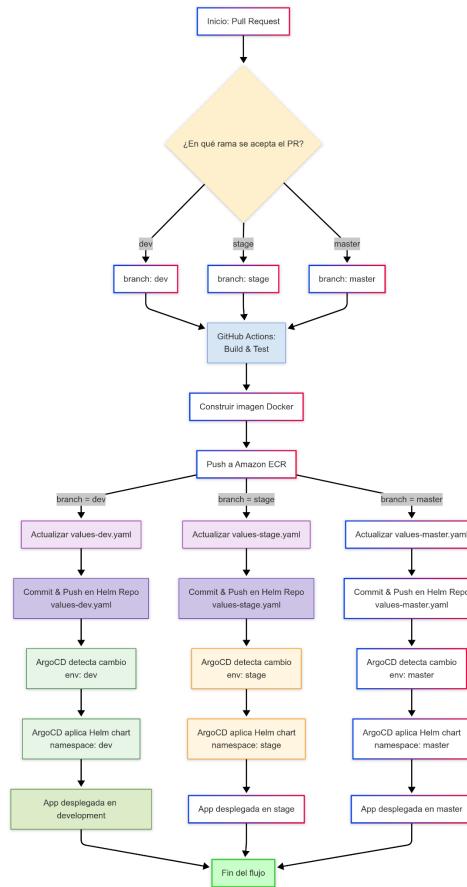
1. Ejecución de tests y construcción de imágenes Docker mediante GitHub Actions.
2. Publicación de la imagen en Amazon ECR.
3. Actualización del values.yaml correspondiente en el repositorio Helm.

#### 4. Detección automática de cambios por ArgoCD y despliegue en el namespace correspondiente.

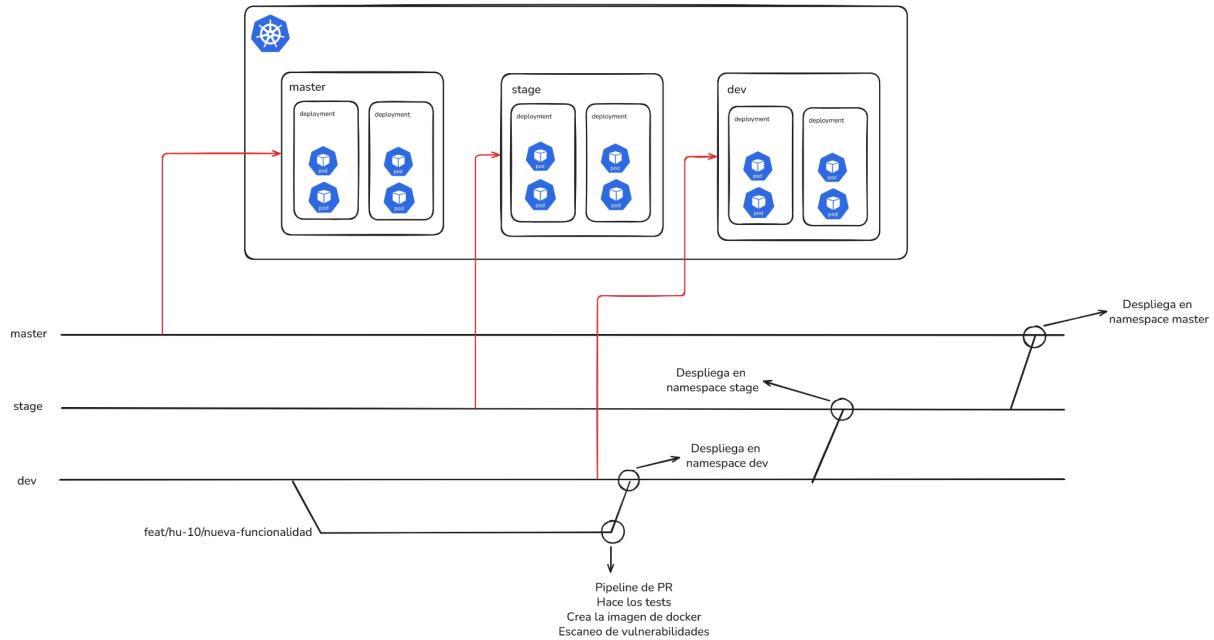
Esta estrategia garantiza un flujo de despliegue coherente, trazable y seguro, cumpliendo los principios de GitOps, además, permite realizar despliegues progresivos y controlar con precisión cada fase del ciclo de vida del software, asegurando estabilidad en producción sin sacrificar velocidad en desarrollo.

Las siguientes figuras ilustran:

- Figura 1: El flujo automatizado completo desde la aceptación de un Pull Request hasta el despliegue según el entorno.



- Figura 2: La relación entre ramas de Git Flow y namespaces en Kubernetes, mostrando cómo cada push activa un despliegue en el entorno adecuado.



## Metodología

Para el desarrollo de este proyecto se optó por la metodología Kanban, un enfoque ágil y visual que permitió gestionar eficazmente las tareas del equipo a lo largo de todo el ciclo de vida del proyecto, dado que el equipo estaba conformado por tres integrantes, se buscó una metodología simple pero funcional, que facilitara la organización del trabajo sin introducir una sobrecarga operativa innecesaria.

Kanban nos permitió:

- Visualizar el flujo de trabajo en tiempo real, identificando claramente qué tareas estaban en proceso, cuáles estaban pendientes y cuáles habían sido completadas.
- Limitar el trabajo en curso, lo cual fue fundamental para mantener un ritmo sostenible y evitar sobrecargar a los integrantes del equipo.
- Mejorar la colaboración y la transparencia, ya que todos los miembros tenían visibilidad del progreso del equipo y se podía redistribuir tareas cuando fuera necesario.

La herramienta principal utilizada para aplicar Kanban fue GitHub Projects, donde se configuró un tablero con columnas como Backlog, Planned, Doing, Blocked, Done. Cada tarea importante del proyecto (como configuración de infraestructura, desarrollo de pipelines, implementación de tests, despliegue, monitoreo, etc.) fue documentada en issues, permitiendo hacer seguimiento a los avances y facilitar las revisiones entre pares.

La elección de Kanban respondió también a la naturaleza continua del trabajo, dado que muchas de las actividades requerían ajustes frecuentes, despliegues iterativos y coordinación en paralelo entre infraestructura y desarrollo. Esto hacía que un enfoque como Scrum, con

sprints cerrados y roles formales, fuera innecesariamente rígido para nuestro equipo reducido y nuestros objetivos.

### Uso del tablero Kanban durante el proyecto

En esta primera etapa se visualiza el punto de partida del proyecto, de esta manera todo el trabajo identificado se encuentra acumulado en la columna de Backlog, y representa el conjunto completo de tareas, requerimientos e historias de usuario que deben ser abordadas.

The screenshot shows a Jira Kanban board with five columns: Backlog, Planned, Doing, Blocked, and Done. The 'Backlog' column contains 28 items, all of which are currently in the 'Planned' column. The 'Doing' column is empty. The 'Blocked' and 'Done' columns also contain no items. Each item in the backlog has a detailed description below it. Buttons for '+ Add item' are visible at the bottom of each column.

Column	Items
Backlog	28
Planned	28
Doing	0
Blocked	0
Done	0

Ya en este punto se observa el avance de tareas desde el Backlog hacia las columnas Planned y Doing. Las tareas en Planned han sido priorizadas y están listas para ser ejecutadas mientras que las tareas que están en Doing son las que el equipo está desarrollando activamente en este momento.

The screenshot shows the same Jira Kanban board after some tasks have been moved. Now, there are 23 items in the 'Planned' column and 5 items in the 'Doing' column. The 'Backlog' column now has 0 items. The 'Blocked' and 'Done' columns remain empty. The tasks in the 'Doing' column have descriptions indicating active work. Buttons for '+ Add item' are visible at the bottom of each column.

Column	Items
Backlog	0
Planned	23
Doing	5
Blocked	0
Done	0

En esta fase de ejecución, se contó con una gran cantidad de tareas en la columna Doing, con un flujo de trabajo constante con algunos desafíos, reflejados en la columna Blocked. Tareas

como el escaneo de imágenes de contenedores y el empaquetado con Helm Charts se han encontrado con impedimentos que se resolvieron antes para poder continuar.

This screenshot shows a Jira Kanban board with the following columns and their counts:

- Backlog: 0
- Planned: 15
- Doing: 10
- Blocked: 3
- Done: 0

The 'Doing' column contains 10 items, each with a brief description. The 'Blocked' column contains 3 items, with the last one highlighted by a blue box. The 'Done' column is currently empty.

Todas las tareas que estaban en las columnas anteriores han sido movidas a Done, indicando que no solo se completó el trabajo planificado, sino que también se resolvieron los impedimentos que habían surgido.

This screenshot shows the same Jira Kanban board after some tasks have been moved to the 'Done' column. The counts are now:

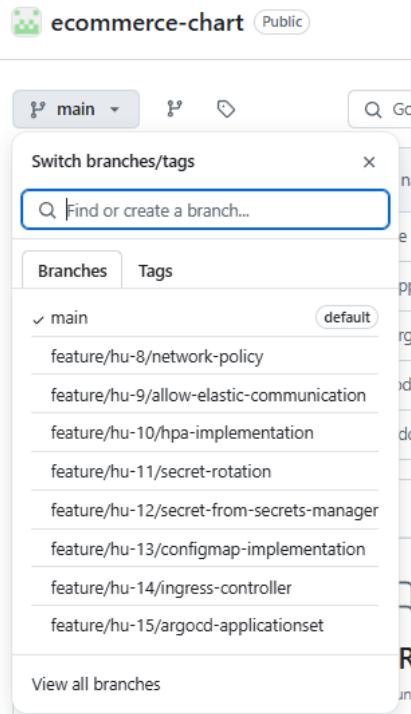
- Backlog: 0
- Planned: 0
- Doing: 0
- Blocked: 0
- Done: 28

The 'Done' column now contains 28 items, with the last one highlighted by a blue box. The other columns are empty.

## Ramas implementadas:

### *ecommerce-chart:*

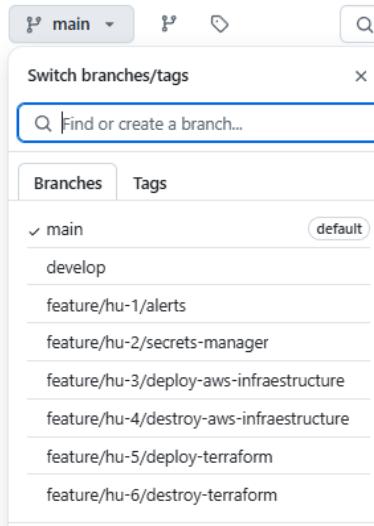
Este repositorio centraliza los Helm Charts, que son los paquetes para desplegar las aplicaciones en Kubernetes, con ramas de funcionalidades específicas, que a diferencia de un flujo con develop, aquí las ramas de feature se crean directamente para implementar capacidades específicas de Kubernetes.



### *Infraestructure:*

Este repositorio gestiona la Infraestructura, utilizando herramientas como Terraform. La estructura de ramas es la siguiente:

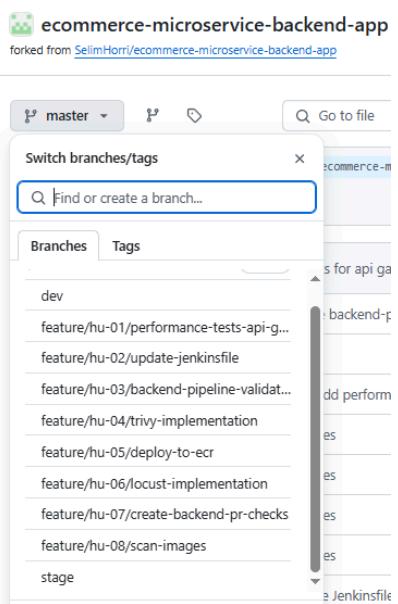
- Ramas principales: main y develop, se utiliza un modelo tipo GitFlow, donde main representa el estado de producción estable y develop es la rama de integración para nuevas funcionalidades.
- Ramas de funcionalidad: Cada nueva tarea o HU se desarrolla en su propia rama, con nombres descriptivos, indicando claramente el trabajo realizado, como deploy-aws-infrastructure, secrets-manager o alerts.



### *ecommerce-microservice-backend-app:*

Este es el repositorio del código fuente de la aplicación de backend, con una estructura de ramas siendo un claro indicador de un ciclo de vida de CI/CD bien definido:

- Ramas de entorno: Con un entorno principal dividido en: master, dev, stage, la existencia de ramas para los entornos de dev y stage demuestra que se realizan despliegues automáticos en diferentes etapas antes de llegar a producción.
- Ramas de funcionalidad: Las ramas como feature/hu-04/trivy-implementation y hu-08/scan-images muestran una fuerte preocupación por la seguridad, integrando escaneo de vulnerabilidades en el pipeline, además de otras funcionalidades como de subir las imágenes a un ecr, etc.

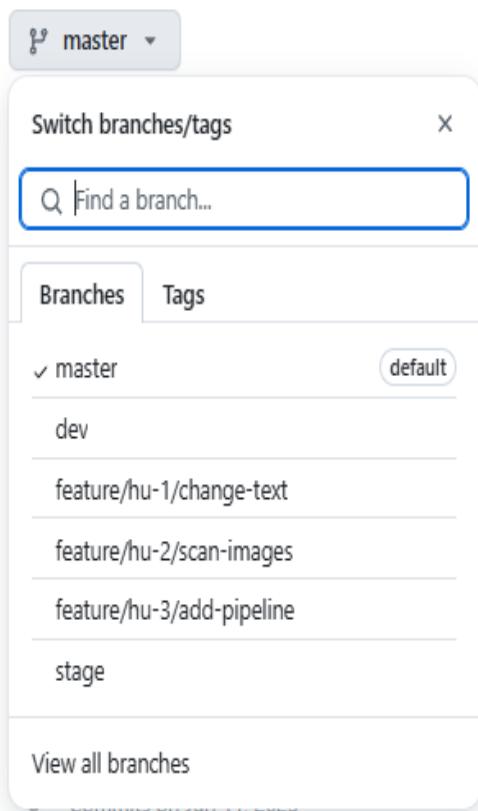


### *ecommerce-frontend-web-app:*

Este repositorio cuenta con similitudes al repositorio del backend, demostrando consistencia en las prácticas de desarrollo:

- Ramas de entorno: Con un entorno principal dividido en: master, dev, stage, la existencia de ramas para los entornos de dev y stage demuestra que se realizan despliegues automáticos en diferentes etapas antes de llegar a producción.
- Ramas de funcionalidad: La rama como feature/hu-2/scan-images muestra una fuerte preocupación por la seguridad, integrando escaneo de vulnerabilidades en las imágenes, además de otras funcionalidades como añadir las pipelines necesarias, etc.

## Commits



## Tests de los Microservicios

Para realizar las pruebas requeridas, se escogieron los servicios: user-service, product-service y order-service, los cuales se encuentran en el repositorio de los microservicios. A continuación, se muestran los tests unitarios, de integración y e2e para cada uno de estos microservicios:

## user-service

### Tests unitarios

Se crea un mock para un usuario válido:

```
private UserDto buildValidUser(String email) {
    CredentialDto cred = CredentialDto.builder()
        .username("testuser")
        .password("1234")
        .isEnabled(true)
        .isAccountNonExpired(true)
        .isAccountNonLocked(true)
        .isCredentialsNonExpired(true)
        .build();

    return UserDto.builder()
        .firstName("Test")
        .lastName("User")
        .email(email)
        .credentialDto(cred)
        .build();
}
```

Los tests realizados son los siguientes:

- Guardado del usuario:

```
@Test
void testSaveUser() {
    UserDto userDto = buildValidUser("test@mail.com");
    when(userRepository.save(any())).thenReturn(UserMappingHelper.map(userDto));
    UserDto result = userService.save(userDto);
    assertNotNull(result);
    assertEquals("test@mail.com", result.getEmail());
}
```

- Encontrar usuario por id:

```
@Test
void testFindUserById() {
    UserDto userDto = buildValidUser("find@mail.com");
    when(userRepository.findById(1)).thenReturn(Optional.of(UserMappingHelper.map(userDto)));
    UserDto result = userService.findById(1);
    assertNotNull(result);
    assertEquals("find@mail.com", result.getEmail());
}
```

- Actualizar usuario:

```
@Test
void testUpdateUser() {
    UserDto userDto = buildValidUser("update@mail.com");
    when(userRepository.save(any())).thenReturn(UserMappingHelper.map(userDto));
    UserDto updated = userService.update(userDto);
    assertEquals("update@mail.com", updated.getEmail());
}
```

- Borrar usuario:

```
@Test
void testDeleteUser() {
    doNothing().when(userRepository).deleteById(1);
    assertDoesNotThrow(() -> userService.deleteById(1));
}
```

- Listar usuarios:

```
@Test
void testListUsers() {
    when(userRepository.findAll()).thenReturn(Collections.singletonList(UserMappingHelper.map(buildValidUser("list@mail.com"))));
    assertFalse(userService.findAll().isEmpty());
}
```

## Tests de integración

Se crea el mock:

```
private UserDto buildValidUser(String email) {
    CredentialDto cred = CredentialDto.builder()
        .username("testuser")
        .password("1234")
        .isEnabled(true)
        .isAccountNonExpired(true)
        .isAccountNonLocked(true)
        .isCredentialsNonExpired(true)
        .build();
    return UserDto.builder()
        .firstName("Test")
        .lastName("User")
        .email(email)
        .credentialDto(cred)
        .build();
}
```

Los tests son:

- Crear usuario:

```
@Test
void testCreateUser() throws Exception {
    UserDto userDto = buildValidUser("test@mail.com");
    mockMvc.perform(post("/api/users")
        .contentType(MediaType.APPLICATION_JSON)
        .content(objectMapper.writeValueAsString(userDto)))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.email").value("test@mail.com"));
}
```

- Obtener usuario por id:

```
@Test
void test GetUserById() throws Exception {
    User user = userRepository.save(new User());
    mockMvc.perform(get("/api/users/" + user.getUserId()))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.userId").value(user.getUserId()));
}
```

- Obtener usuario inexistente por id:

```
@Test
void testGetNonExistentUserById() throws Exception {
    mockMvc.perform(get("/api/users/999999"))
        .andExpect(status().isBadRequest())
        .andExpect(jsonPath("$.timestamp").exists());
}
```

- Listar usuarios:

```
@Test
void testListUsers() throws Exception {
    mockMvc.perform(get("/api/users"))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.collection").exists());
}
```

- Actualizar usuario:

```
@Test
void testUpdateUser() throws Exception {
    User user = userRepository.save(new User());
    UserDto userDto = buildValidUser("updated@mail.com");
    userDto.setUserId(user.getUserId());
    mockMvc.perform(put("/api/users/" + user.getUserId())
        .contentType(MediaType.APPLICATION_JSON)
        .content(objectMapper.writeValueAsString(userDto)))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.email").value("updated@mail.com"));
}
```

- Borrar usuario:

```
@Test
void testDeleteUser() throws Exception {
    User user = userRepository.save(new User());
    mockMvc.perform(delete("/api/users/" + user.getUserId()))
        .andExpect(status().isOk())
        .andExpect(content().string("true"));
}
```

## Tests e2e

Se crea el mock junto con una template REST:

```
@Autowired
private TestRestTemplate restTemplate;

private UserDto buildValidUser(String email) {
    CredentialDto cred = CredentialDto.builder()
        .username("testuser")
        .password("1234")
        .isEnabled(true)
        .isAccountNonExpired(true)
        .isAccountNonLocked(true)
        .isCredentialsNonExpired(true)
        .build();
    return UserDto.builder()
        .firstName("Test")
        .lastName("User")
        .email(email)
        .credentialDto(cred)
        .build();
}
```

Y los tests son:

- Usuario se registra:

```
@Test
void testUserRegisters() {
    UserDto user = buildValidUser("test@mail.com");
    ResponseEntity<UserDto> response = restTemplate.postForEntity("/api/users", user, UserDto.class);
    assertEquals(200, response.getStatusCodeValue());
    assertNotNull(response.getBody());
}
```

- Usuario borra la cuenta:

```
@Test
void testUserDeletesAccount() {
    UserDto user = restTemplate.postForEntity("/api/users", buildValidUser("test2@mail.com"), UserDto.class).getBody();
    restTemplate.delete("/api/users/" + user.getUserId());
    assertNotNull(user);
}
```

- Usuario lista otros usuarios:

```
@Test
void testUserViewsUserList() {
    ResponseEntity<DtoCollectionResponse> response = restTemplate.getForEntity("/api/users", DtoCollectionResponse.class);
    assertEquals(200, response.getStatusCodeValue());
    assertNotNull(response.getBody());
    assertTrue(response.getBody().getCollection() != null);
}
```

- Usuario actualiza email:

```
@Test
void testUserViewsUserList() {
    ResponseEntity<DtoCollectionResponse> response = restTemplate.getForEntity("/api/users", DtoCollectionResponse.class);
    assertEquals(200, response.getStatusCodeValue());
    assertNotNull(response.getBody());
    assertTrue(response.getBody().getCollection() != null);
}
```

- Usuario ve su perfil:

```
@Test
void testUserViewsProfile() {
    UserDto user = restTemplate.postForEntity("/api/users", buildValidUser("test4@mail.com"), UserDto.class).getBody();
    ResponseEntity<UserDto> response = restTemplate.getForEntity("/api/users/" + user.getUserId(), UserDto.class);
    if (response.getStatusCode().is2xxSuccessful()) {
        assertNotNull(response.getBody());
    } else {
        String errorJson = response.toString();
        assertTrue(errorJson.contains("timestamp"));
    }
}
```

- Usuario ve un perfil inexistente:

```
@Test
void testUserViewsNonExistentProfile() {
    ResponseEntity<String> response = restTemplate.getForEntity("/api/users/999999", String.class);
    assertEquals(HttpStatus.BAD_REQUEST, response.getStatusCode());
    assertTrue(response.getBody().contains("timestamp"));
}
```

## Ejecución

A continuación se puede ver la ejecución de los tests:

```
sjbarraza [~/ecommerce-microservice-backend-app/user-service] master ▾ Azure for Students
❯ ./mvnw test
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.selimhorri:user-service >-----
[INFO] Building user-service 0.1.0
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- maven-resources-plugin:3.2.0:resources (default-resources) @ user-service ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Using 'UTF-8' encoding to copy filtered properties files.
[INFO] Copying 4 resources
[INFO] Copying 11 resources
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ user-service ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:3.2.0:testResources (default-testResources) @ user-service ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Using 'UTF-8' encoding to copy filtered properties files.
[INFO] Copying 1 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:testCompile (default-testCompile) @ user-service ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.22.2:test (default-test) @ user-service ---
[INFO]
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running com.selimhorri.app.integration.UserServiceIntegrationTest
19:06:39.551 [main] DEBUG org.springframework.test.context.BootstrapUtils - Instantiating CacheAwareContextLoaderDelegate from class [org.springframework.test.context.cache.DefaultCacheAwareContextLoaderDelegate]
19:06:39.564 [main] DEBUG org.springframework.test.context.BootstrapUtils - Instantiating BootstrapContext using constructor [public org.springframework.test.context.sup
```

Y se ve un resultado satisfactorio:

```
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 32, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 24.441 s
[INFO] Finished at: 2025-05-26T19:06:59-05:00
[INFO] -----
```

### product-service

## Tests unitarios

Se crea el mock de un producto válido:

```
private ProductDto buildValidProduct(String title) {
    CategoryDto category = CategoryDto.builder().categoryId(1).categoryTitle("Cat1").build();
    return ProductDto.builder()
        .productId(1)
        .productTitle(title)
        .imageUrl("img.png")
        .sku("SKU1")
        .priceUnit(10.0)
        .quantity(5)
        .categoryDto(category)
        .build();
}
```

Tests:

- Guardar producto:

```
@Test
void testSaveProduct() {
    ProductDto productDto = buildValidProduct("test");
    when(productRepository.save(any())).thenReturn(ProductMappingHelper.map(productDto));
    ProductDto result = productService.save(productDto);
    assertNotNull(result);
    assertEquals("test", result.getProductTitle());
}
```

- Encontrar producto por id:

```
@Test
void testFindProductById() {
    ProductDto productDto = buildValidProduct("find");
    when(productRepository.findById(1)).thenReturn(Optional.of(ProductMappingHelper.map(productDto)));
    ProductDto result = productService.findById(1);
    assertNotNull(result);
    assertEquals("find", result.getProductTitle());
}
```

- Actualizar producto:

```
@Test
void testUpdateProduct() {
    ProductDto productDto = buildValidProduct("update");
    when(productRepository.save(any())).thenReturn(ProductMappingHelper.map(productDto));
    ProductDto updated = productService.update(productDto);
    assertEquals("update", updated.getProductTitle());
}
```

- Borrar producto:

```
@Test
void testUpdateProduct() {
    ProductDto productDto = buildValidProduct("update");
    when(productRepository.save(any())).thenReturn(ProductMappingHelper.map(productDto));
    ProductDto updated = productService.update(productDto);
    assertEquals("update", updated.getProductTitle());
}
```

- Listar productos:

```
@Test
void testListProducts() {
    when(productRepository.findAll()).thenReturn(Collections.singletonList(ProductMappingHelper.map(buildValidProduct("list"))));
    assertFalse(productService.findAll().isEmpty());
}
```

## Tests de integración

Se crea el mock de un producto válido:

```
private ProductDto buildValidProduct(String title, int categoryId) {
    CategoryDto category = CategoryDto.builder().categoryId(categoryId).categoryTitle("Cat1").build();
    return ProductDto.builder()
        .productId(1)
        .productTitle(title)
        .imageUrl("img.png")
        .sku("SKU1")
        .priceUnit(10.0)
        .quantity(5)
        .categoryDto(category)
        .build();
}
```

Tests:

- Crear producto:

```
@Test
void testCreateProduct() throws Exception {
    Category category = Category.builder().categoryId(1).categoryTitle("Cat1").build();
    categoryRepository.save(category);
    ProductDto productDto = buildValidProduct("TestProduct", 1);
    mockMvc.perform(post("/api/products")
        .contentType(MediaType.APPLICATION_JSON)
        .content(objectMapper.writeValueAsString(productDto)))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.productTitle").value("TestProduct"));
}
```

- Obtener producto por id:

```
@Test
void testGetProductById() throws Exception {
    Category category = Category.builder().categoryId(2).categoryTitle("Cat2").build();
    categoryRepository.save(category);
    Product product = productRepository.save(Product.builder().category(category).build());
    mockMvc.perform(get("/api/products/" + product.getProductId()))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.productId").value(product.getProductId()));
}
```

- Obtener producto inexistente por id:

```
@Test
void testGetNonExistentProductById() throws Exception {
    mockMvc.perform(get("/api/products/999999"))
        .andExpect(status().isBadRequest())
        .andExpect(jsonPath("$.timestamp").exists());
}
```

- Listar productos:

```
@Test
void testListProducts() throws Exception {
    Category category = Category.builder().categoryId(3).categoryTitle("Cat3").build();
    categoryRepository.save(category);
    productRepository.save(Product.builder().category(category).build());
    mockMvc.perform(get("/api/products"))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.collection").exists());
}
```

- Actualizar producto:

```
@Test
void testUpdateProduct() throws Exception {
    Category category = Category.builder().categoryId(4).categoryTitle("Cat4").build();
    categoryRepository.save(category);
    Product product = productRepository.save(Product.builder().category(category).build());
    ProductDto productDto = buildValidProduct("UpdatedProduct", 4);
    productDto.setProductId(product.getProductId());
    mockMvc.perform(put("/api/products/" + product.getProductId())
        .contentType(MediaType.APPLICATION_JSON)
        .content(objectMapper.writeValueAsString(productDto)))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.productTitle").value("UpdatedProduct"));
}
```

- Borrar producto:

```
@Test
void testDeleteProduct() throws Exception {
    Category category = Category.builder().categoryId(5).categoryTitle("Cat5").build();
    categoryRepository.save(category);
    Product product = productRepository.save(Product.builder().category(category).build());
    mockMvc.perform(delete("/api/products/" + product.getProductId()))
        .andExpect(status().isOk())
        .andExpect(content().string("true"));
}
```

## Tests e2e

Se crea el mock y una template REST:

```
@Autowired
private TestRestTemplate restTemplate;

private ProductDto buildValidProduct(String title) {
    CategoryDto category = CategoryDto.builder().categoryId(1).categoryTitle("Cat1").build();
    return ProductDto.builder()
        .productId(1)
        .productTitle(title)
        .imageUrl("img.png")
        .sku("SKU1")
        .priceUnit(10.0)
        .quantity(5)
        .categoryDto(category)
        .build();
}
```

Tests:

- Usuario crea producto:

```
@Test
void testUserCreatesProduct() {
    ProductDto product = buildValidProduct("TestProduct");
    ResponseEntity<ProductDto> response = restTemplate.postForEntity("/api/products", product, ProductDto.class);
    assertEquals(200, response.getStatusCodeValue());
    assertNotNull(response.getBody());
    assertEquals("TestProduct", response.getBody().getProductTitle());
}
```

- Usuario elimina producto:

```
@Test
void testUserDeletesProduct() {
    ProductDto product = restTemplate.postForEntity("/api/products", buildValidProduct("DeleteProduct"), ProductDto.class).getBody();
    restTemplate.delete("/api/products/" + product.getProductId());
    assertNotNull(product);
}
```

- Usuario obtiene lista de productos:

```
@Test
void testUserViewsProductList() {
    ResponseEntity<DtoCollectionResponse> response = restTemplate.getForEntity("/api/products", DtoCollectionResponse.class);
    if (response.getStatusCode().is2xxSuccessful()) {
        assertNotNull(response.getBody());
        assertTrue(response.getBody().getCollection() != null);
    } else {
        String errorJson = response.toString();
        assertTrue(errorJson.contains("timestamp"));
    }
}
```

- Usuario actualiza producto:

```
@Test
void testUserUpdatesProduct() {
    ProductDto product = restTemplate.postForEntity("/api/products", buildValidProduct("UpdateProduct"), ProductDto.class).getBody();
    product.setProductTitle("UpdatedProduct");
    restTemplate.put("/api/products/" + product.getProductId(), product);
    assertNotNull(product);
}
```

- Usuario obtiene detalles de un producto:

```
@Test
void testUserViewsProductDetails() {
    ProductDto product = restTemplate.postForEntity("/api/products", buildValidProduct("DetailProduct"), ProductDto.class).getBody();
    ResponseEntity<ProductDto> response = restTemplate.getForEntity("/api/products/" + product.getId(), ProductDto.class);
    if (response.getStatusCode().is2xxSuccessful()) {
        assertNotNull(response.getBody());
        assertEquals("DetailProduct", response.getBody().getTitle());
    } else {
        String errorJson = response.toString();
        assertTrue(errorJson.contains("timestamp"));
    }
}
```

- Usuario obtiene producto inexistente:

```
@Test
void testUserViewsNonExistentProduct() {
    ResponseEntity<String> response = restTemplate.getForEntity("/api/products/999999", String.class);
    assertEquals(HttpStatus.BAD_REQUEST, response.getStatusCode());
    assertTrue(response.getBody().contains("timestamp"));
}
```

## Ejecución

Se realiza la ejecución de todos los tests para el microservicio:

```
sjbarraza [~/ecommerce-microservice-backend-app/product-service] $ mvnw test
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.selimhorri:product-service >-----
[INFO] Building product-service 0.1.0
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- maven-resources-plugin:3.2.0:resources (default-resources) @ product-service ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Using 'UTF-8' encoding to copy filtered properties files.
[INFO] Copying 4 resources
[INFO] Copying 0 resources
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ product-service ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:3.2.0:testResources (default-testResources) @ product-service ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Using 'UTF-8' encoding to copy filtered properties files.
[INFO] skip non existing resourceDirectory /home/sjbarraza/projects/university/platforms2/microservices-project-k8s-jenkins/ecommerce-microservice-backend-app/product-service/src/test/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:testCompile (default-testCompile) @ product-service ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.22.2:test (default-test) @ product-service ---
[INFO]
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running com.selimhorri.app.integration.ProductServiceIntegrationTest
```

Se puede ver que todos pasan correctamente:

```
2025-05-26 19:09:46.746  INFO [PRODUCT-SERVICE,,] 59806 --- [ionShutdownHook] com.netflix.discovery.DiscoveryClient      : Completed shut down of DiscoveryClient
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 32, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 32.300 s
[INFO] Finished at: 2025-05-26T19:09:46-05:00
[INFO] -----
```

## order-service

### Tests unitarios

Se crea el mock de una orden válida:

```
private OrderDto buildValidOrder() {
    CartDto cart = CartDto.builder().cartId(1).userId(1).build();
    return OrderDto.builder()
        .orderId(1)
        .orderDate(LocalDateTime.now())
        .orderDesc("Test order")
        .orderFee(100.0)
        .cartDto(cart)
        .build();
}
```

Tests:

- Guardar orden:

```
@Test
void testSaveOrder() {
    OrderDto orderDto = buildValidOrder();
    when(orderRepository.save(any())).thenReturn(OrderMappingHelper.map(orderDto));
    OrderDto result = orderService.save(orderDto);
    assertNotNull(result);
    assertEquals("Test order", result.getOrderDesc());
}
```

- Encontrar orden por id:

```
@Test
void testFindOrderById() {
    OrderDto orderDto = buildValidOrder();
    when(orderRepository.findById(1)).thenReturn(Optional.of(OrderMappingHelper.map(orderDto)));
    OrderDto result = orderService.findById(1);
    assertNotNull(result);
    assertEquals("Test order", result.getOrderDesc());
}
```

- Actualizar orden:

```
@Test
void testUpdateOrder() {
    OrderDto orderDto = buildValidOrder();
    when(orderRepository.save(any())).thenReturn(OrderMappingHelper.map(orderDto));
    OrderDto updated = orderService.update(orderDto);
    assertEquals("Test order", updated.getOrderDesc());
}
```

- Borrar orden:

```
@Test
void testDeleteOrder() {
    doNothing().when(orderRepository).delete(any());
    when(orderRepository.findById(1)).thenReturn(Optional.of(OrderMappingHelper.map(buildValidOrder())));
    assertDoesNotThrow(() -> orderService.deleteById(1));
}
```

- Listar órdenes:

```
@Test
void testListOrders() {
    when(orderRepository.findAll()).thenReturn(Collections.singletonList(OrderMappingHelper.map(buildValidOrder())));
    assertFalse(orderService.findAll().isEmpty());
}
```

## Tests de integración

Se crea el mock de una orden válida:

```
private OrderDto buildValidOrder(int cartId) {
    CartDto cart = CartDto.builder().cartId(cartId).userId(1).build();
    return OrderDto.builder()
        .orderId(1)
        .orderDate(LocalDateTime.now())
        .orderDesc("Test order")
        .orderFee(100.0)
        .cartDto(cart)
        .build();
}
```

Tests:

- Crear una orden:

```
@Test
void testCreateOrder() throws Exception {
    Cart cart = Cart.builder().cartId(1).userId(1).build();
    cartRepository.save(cart);
    OrderDto orderDto = buildValidOrder(1);
    mockMvc.perform(post("/api/orders")
        .contentType(MediaType.APPLICATION_JSON)
        .content(objectMapper.writeValueAsString(orderDto)))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.orderDesc").value("Test order"));
}
```

- Obtener orden por id:

```
@Test
void testGetOrderById() throws Exception {
    Cart cart = Cart.builder().cartId(2).userId(1).build();
    cartRepository.save(cart);
    Order order = orderRepository.save(Order.builder().cart(cart).build());
    mockMvc.perform(get("/api/orders/" + order.getOrderId()))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.orderId").value(order.getOrderId()));
}
```

- Obtener orden inexistente por id:

```
@Test
void testGetNonExistentOrderById() throws Exception {
    mockMvc.perform(get("/api/orders/999999"))
        .andExpect(status().isBadRequest())
        .andExpect(jsonPath("$.timestamp").exists());
}
```

- Listar ordenes:

```
@Test
void testListOrders() throws Exception {
    Cart cart = Cart.builder().cartId(3).userId(1).build();
    cartRepository.save(cart);
    orderRepository.save(Order.builder().cart(cart).build());
    mockMvc.perform(get("/api/orders"))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.collection").exists());
}
```

- Actualizar orden:

```
@Test
void testUpdateOrder() throws Exception {
    Cart cart = Cart.builder().cartId(4).userId(1).build();
    cartRepository.save(cart);
    Order order = orderRepository.save(Order.builder().cart(cart).build());
    OrderDto orderDto = buildValidOrder(4);
    orderDto.setOrderId(order.getOrderId());
    mockMvc.perform(put("/api/orders/" + order.getOrderId())
        .contentType(MediaType.APPLICATION_JSON)
        .content(objectMapper.writeValueAsString(orderDto)))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.orderDesc").value("Test order"));
}
```

- Borrar orden:

```
@Test
void testDeleteOrder() throws Exception {
    Cart cart = Cart.builder().cartId(5).userId(1).build();
    cartRepository.save(cart);
    Order order = orderRepository.save(Order.builder().cart(cart).build());
    mockMvc.perform(delete("/api/orders/" + order.getOrderId()))
        .andExpect(status().isOk())
        .andExpect(content().string("true"));
}
```

## Tests e2e

Se crea el mock y una template REST:

```
@Autowired  
private TestRestTemplate restTemplate;  
  
private OrderDto buildValidOrder() {  
    CartDto cart = CartDto.builder().cartId(1).userId(1).build();  
    return OrderDto.builder()  
        .orderId(1)  
        .orderDate(LocalDateTime.now())  
        .orderDesc("Test order")  
        .orderFee(100.0)  
        .cartDto(cart)  
        .build();  
}
```

Tests:

- Usuario hace una orden:

```
@Test  
void testUserPlacesOrder() {  
    OrderDto order = buildValidOrder();  
    ResponseEntity<OrderDto> response = restTemplate.postForEntity("/api/orders", order, OrderDto.class);  
    assertEquals(200, response.getStatusCodeValue());  
    assertNotNull(response.getBody());  
    assertEquals("Test order", response.getBody().getOrderDesc());  
}
```

- Usuario borra una orden:

```
@Test  
void testUserDeletesOrder() {  
    OrderDto order = restTemplate.postForEntity("/api/orders", buildValidOrder(), OrderDto.class).getBody();  
    restTemplate.delete("/api/orders/" + order.getOrderId());  
    assertNotNull(order);  
}
```

- Usuario obtiene lista de órdenes:

```
@Test  
void testUserViewsOrderList() {  
    ResponseEntity<DtoCollectionResponse> response = restTemplate.getForEntity("/api/orders", DtoCollectionResponse.class);  
    if (response.getStatusCode().is2xxSuccessful()) {  
        assertNotNull(response.getBody());  
        assertTrue(response.getBody().getCollection() != null);  
    } else {  
        String errorJson = response.toString();  
        assertTrue(errorJson.contains("timestamp"));  
    }  
}
```

- Usuario actualiza una orden:

```
@Test  
void testUserUpdatesOrder() {  
    OrderDto order = restTemplate.postForEntity("/api/orders", buildValidOrder(), OrderDto.class).getBody();  
    order.setOrderDesc("Updated order");  
    restTemplate.put("/api/orders/" + order.getOrderId(), order);  
    assertNotNull(order);  
}
```

- Usuario obtiene detalles de una orden:

```

    @Test
    void testUserViewsOrderDetails() {
        OrderDto order = restTemplate.postForEntity("/api/orders", buildValidOrder(), OrderDto.class).getBody();
        ResponseEntity<OrderDto> response = restTemplate.getForEntity("/api/orders/" + order.getOrderId(), OrderDto.class);
        if (response.getStatusCode().is2xxSuccessful()) {
            assertNotNull(response.getBody());
            assertEquals("Test order", response.getBody().getOrderDesc());
        } else {
            String errorJson = response.toString();
            assertTrue(errorJson.contains("timestamp"));
        }
    }
}

```

- Usuario obtiene orden inexistente:

```

    @Test
    void testUserViewsNonExistentOrder() {
        ResponseEntity<String> response = restTemplate.getForEntity("/api/orders/999999", String.class);
        assertEquals(HttpStatus.BAD_REQUEST, response.getStatusCode());
        assertTrue(response.getBody().contains("timestamp"));
    }
}

```

## Ejecución

Se ejecutan todos los tests:

```

sjbarraza [■ .../ecommerce-microservice-backend-app/order-service] P master ▾ Azure for Students
❯ ./mvnw test
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.selimhorri:order-service >-----
[INFO] Building order-service 0.1.0
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- maven-resources-plugin:3.2.0:resources (default-resources) @ order-service ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Using 'UTF-8' encoding to copy filtered properties files.
[INFO] Copying 4 resources
[INFO] Copying 5 resources
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ order-service ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:3.2.0:testResources (default-testResources) @ order-service ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Using 'UTF-8' encoding to copy filtered properties files.
[INFO] skip non existing resourceDirectory /home/sjbarraza/projects/university/platforms2/microservices-project-k8s-jenkins/ecommerce-microservice-backend-app/order-service/src/test/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:testCompile (default-testCompile) @ order-service ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.22.2:test (default-test) @ order-service ---
[INFO]
[INFO] -----
[INFO] T E S T S
[INFO] -----

```

Se puede ver que todos pasan:

```

[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 32, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 32.575 s
[INFO] Finished at: 2025-05-26T19:12:21-05:00
[INFO] -----

```

## **Build del Proyecto (Backend)**

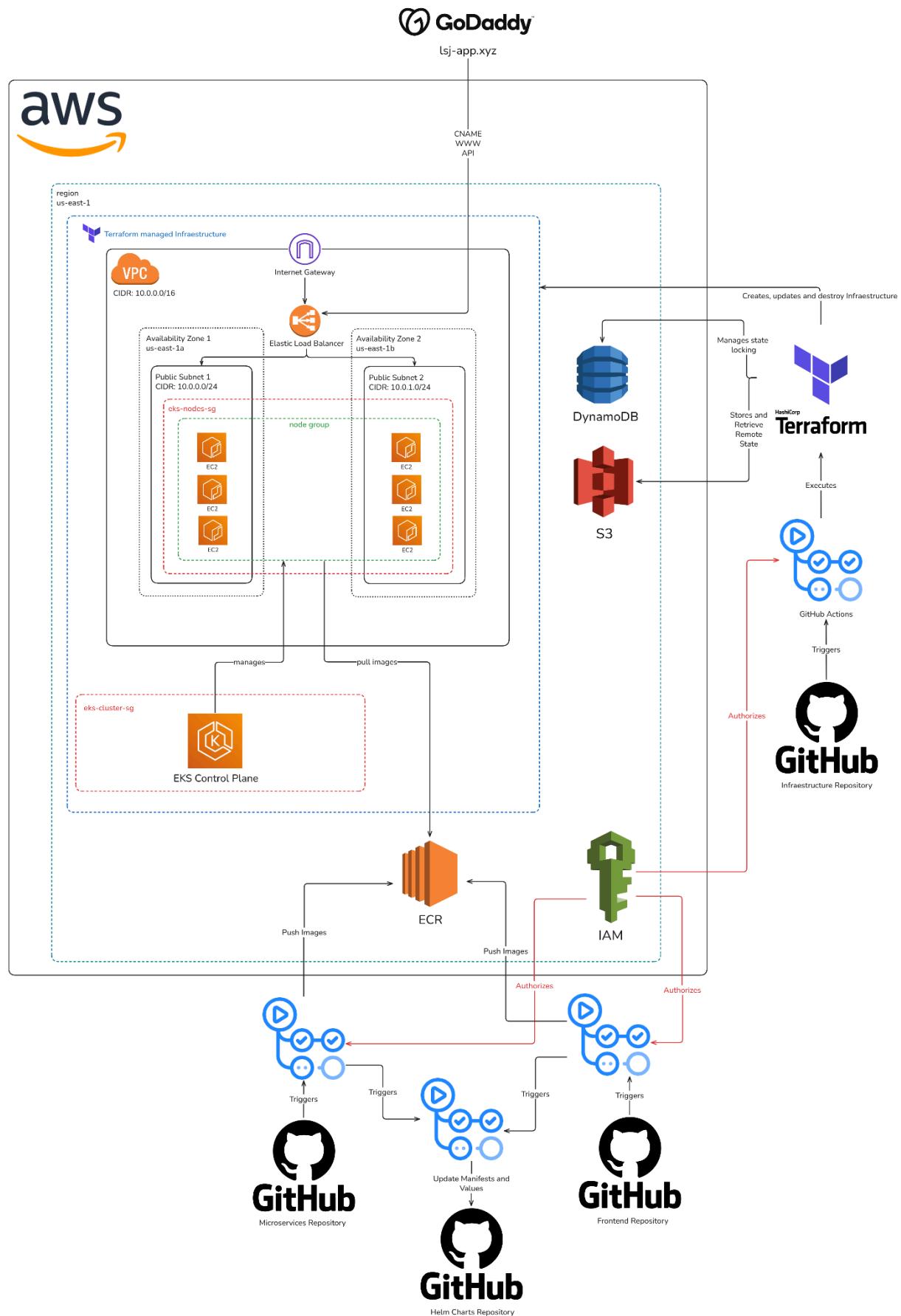
Se usa el comando “./mvnw clean package” para realizar el build completo de todos los microservicios del backend, esto ejecuta los tests de los microservicios anteriormente definidos:

```
sjbarraza [■ ..\ecommerce-microservice-backend-app] P master | 🍀 V0.1.0 ▾ Azure for Students
❯ ./mvnw clean package
[INFO] Scanning for projects...
[INFO] ...
[INFO] Reactor Build Order:
[INFO]
[INFO] ecommerce-microservice-backend           [pom]
[INFO] service-discovery                      [jar]
[INFO] cloud-config                            [jar]
[INFO] api-gateway                            [jar]
[INFO] proxy-client                           [jar]
[INFO] user-service                           [jar]
[INFO] product-service                        [jar]
[INFO] favourite-service                      [jar]
[INFO] order-service                          [jar]
[INFO] shipping-service                       [jar]
[INFO] payment-service                        [jar]
[INFO]
[INFO] -----< com.selimhorri:ecommerce-microservice-backend >-----
[INFO] Building ecommerce-microservice-backend 0.1.0          [1/11]
[INFO] [ pom ] -----
```

Se puede ver que el build finaliza de forma satisfactoria. Nótese que para los microservicios que se implementaron los tests el tiempo de build fue mayor, esto se debe a que para realizar los tests tiene que levantar el microservicio temporalmente, proceso que es tardado:

```
[INFO] ecommerce-microservice-backend           : SUCCESS [ 1.831 s]
[INFO] service-discovery                     : SUCCESS [ 4.650 s]
[INFO] cloud-config                           : SUCCESS [ 3.208 s]
[INFO] api-gateway                           : SUCCESS [ 3.698 s]
[INFO] proxy-client                          : SUCCESS [ 6.839 s]
[INFO] user-service                          : SUCCESS [ 44.281 s]
[INFO] product-service                       : SUCCESS [ 51.843 s]
[INFO] favourite-service                     : SUCCESS [ 4.252 s]
[INFO] order-service                         : SUCCESS [ 45.141 s]
[INFO] shipping-service                      : SUCCESS [ 3.445 s]
[INFO] payment-service                       : SUCCESS [ 7.290 s]
[INFO]
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 02:56 min
[INFO] Finished at: 2025-05-26T19:45:36-05:00
[INFO] -----
```

## Arquitectura de Infraestructura en la Nube



A continuación, para realizar un despliegue de los microservicios se explicará el diagrama de arquitectura de la infraestructura en la nube y todos sus elementos, los cuales serán divididos en dos grupos, la infraestructura creada manualmente y la infraestructura gestionada por terraform.

### **Infraestructura creada manualmente**

- **IAM:** Se usa credenciales de IAM para autorizar a las pipelines de los distintos repositorios para realizar cambios en AWS, sin embargo, al tratarse de un laboratorio, los roles de IAM no son un recurso que haya podido ser gestionado, sino que todo se ejecuta bajo el rol LabRole. Para autorizar a las pipelines se tienen tres parámetros que deben ser creados en los secrets de GitHub: AWS\_ACCESS\_KEY\_ID, AWS\_SECRET\_ACCESS\_KEY y AWS\_SESSION\_TOKEN, los cuales son generados cada vez que se inicia el laboratorio.
- **Bucket de S3:** Este Bucket se crea con el fin de almacenar el estado remoto de terraform, el cual se guarda en un archivo llamado tfstate. El objetivo con el cual esto se realiza es debido a que la creación de la infraestructura se realiza por medio de una pipeline de GitHub Actions en el repositorio de infraestructura, por lo que en cada ejecución de esta, el runner no tendrá contexto, y de no haber estado remoto, cada runner creará su estado local, por lo que no tendrá conocimiento del estado actual de la infraestructura.
- **DynamoDB:** Se crea una tabla llamada terraform-locks en DynamoDB, el motivo de esto es que terraform necesita almacenar algo llamado “lock”, el cual es un bloqueo que impide que haya dos o más ejecuciones simultáneas que alteren el estado de la infraestructura a través de terraform. Por lo que si una pipeline está ejecutando un “terraform destroy” y cuando esta no ha finalizado, se ejecuta una pipeline que hace “terraform apply”, entonces esta segunda fallará, debido a que este bloqueo existe, impidiendo que se corrompa la infraestructura.
- **ECR:** Los repositorios de ECR que almacenan las imágenes de Docker de los microservicios no son un recurso que se cree manualmente, sin embargo, este no es gestionado por terraform, ya que las pipelines de los repositorios de microservicios y de frontend son las que se encargan de subir sus imágenes de Docker a un repositorio de ECR.

### Infraestructura Gestionada Por Terraform

- **EKS:** El clúster de EKS es el núcleo de la infraestructura orquestada con Terraform. Se encarga de proporcionar un control plane, el cual es administrado para Kubernetes en AWS. Terraform crea este plano de control, el cual se encarga de gestionar el estado del clúster, la planificación de pods, y otras funciones críticas del sistema Kubernetes, sin necesidad de que el usuario lo administre directamente.

- **Node group:** Los node groups son conjuntos de instancias EC2 que actúan como nodos de trabajo dentro del clúster de EKS. Estas instancias se configuran automáticamente por Terraform y están asociadas a subredes públicas distribuidas en múltiples zonas de disponibilidad (us-east-1a y us-east-1b) para garantizar alta disponibilidad y tolerancia a fallos. Los nodos están protegidos por un security group específico (eks-nodes-sg) y son los responsables de ejecutar los contenedores y workloads desplegados en Kubernetes.
- **VPC y Subnets:** Se define una red privada virtual (VPC) con un rango CIDR de 10.0.0.0/16. Dentro de esta VPC se crean dos subredes públicas (10.0.0.0/24 y 10.0.1.0/24) distribuidas entre las zonas de disponibilidad mencionadas. Estas subredes alojan los nodos del clúster.
- **Internet Gateway y Load Balancer:** La VPC está conectada a un Internet Gateway que permite el tráfico entrante y saliente desde y hacia Internet. Un Elastic Load Balancer distribuye el tráfico entrante a los servicios desplegados en el clúster de manera eficiente y balanceada. Este Load Balancer se crea automáticamente cuando hay un servicio de tipo LoadBalancer en el cluster de Kubernetes, y su tipo es un Network Load Balancer.
- **Security Groups:** Terraform define distintas reglas de seguridad mediante dos security groups:
  - eks-cluster-sg: controla el tráfico que puede comunicarse con el plano de control de EKS.
  - eks-nodes-sg: regula el tráfico que entra y sale de los nodos del clúster. Solo permite tráfico entrante desde el security group del EKS.

### **Integración GitHub Actions + Terraform**

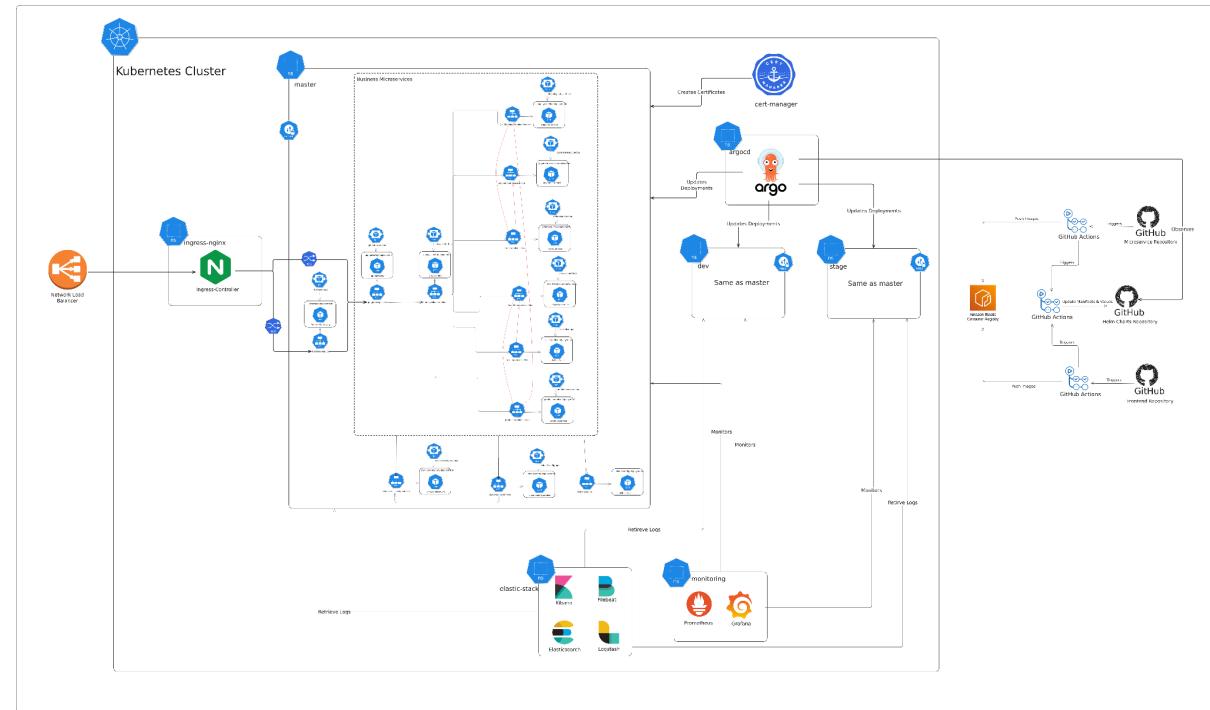
- El repositorio de infraestructura en GitHub contiene los archivos de configuración de Terraform y una pipeline que automatiza la creación, modificación o destrucción de recursos.
- Utiliza secrets con credenciales temporales de AWS para autenticarse.
- El estado remoto se guarda en un bucket S3 para mantener el contexto entre ejecuciones.
- Un lock de DynamoDB asegura que no se realicen ejecuciones concurrentes de Terraform que puedan causar conflictos en el estado.

## DNS en GoDaddy

El dominio lsj-app.xyz se gestiona en GoDaddy. Las entradas DNS (como www o api) se configuran con CNAME para apuntar al Load Balancer creado en AWS, permitiendo el acceso público a las aplicaciones desplegadas. Como se ve a continuación:

<input type="checkbox"/>	CNAME	api	ab298314bc4974ddbb88a4fdacc2ee60-1628897084.us-east-1.elb.amazonaws.com.	1 Hora		
<input type="checkbox"/>	CNAME	www	ab298314bc4974ddbb88a4fdacc2ee60-1628897084.us-east-1.elb.amazonaws.com.	1 Hora		
<input type="checkbox"/>	CNAME	_domainconnect	_domainconnect.gd.domaincontrol.com.	1 Hora		

## Arquitectura de Kubernetes



Esta arquitectura describe un sistema de microservicios moderno, nativo de la nube, desplegado en un clúster de Kubernetes. Utiliza un enfoque de GitOps para la automatización del despliegue (CI/CD), con herramientas robustas para el monitoreo, la observabilidad, el enrutamiento de tráfico y la seguridad. El sistema está diseñado para ser escalable, resiliente y mantenable.

## Despliegue y Gestión de Aplicaciones

La base del despliegue de aplicaciones es el Clúster de Kubernetes. Dentro de este clúster, las aplicaciones se organizan lógicamente en Namespaces. Cada microservicio (como payment-service, order-service, etc.) se gestiona mediante un conjunto de objetos de Kubernetes:

- **Deployment:** Define el estado deseado de la aplicación, como la imagen del contenedor a usar y el número de réplicas. Kubernetes se asegura de que este estado se mantenga, reiniciando pods si fallan.
- **Pod:** Es la unidad de despliegue más pequeña. Cada pod ejecuta una o más instancias de un contenedor de la aplicación.
- **Service (SVC):** Proporciona un punto de acceso de red estable (una IP y un DNS internos) para un conjunto de pods. Esto permite que los microservicios se comuniquen entre sí sin necesidad de conocer las IPs de los pods individuales, que son efímeras.
- **Horizontal Pod Autoscaler (HPA):** Escala automáticamente el número de pods en un Deployment basándose en métricas de uso, como el consumo de CPU o memoria. Esto asegura que la aplicación tenga los recursos necesarios para manejar la carga de trabajo, pero sin desperdiciarlos.

### Networking y Entrada de Tráfico

El flujo de tráfico desde el usuario final hasta los microservicios está claramente definido:

- **Tráfico Externo (DNS - GoDaddy):** El punto de partida es un registro DNS que está en GoDaddy que apunta un dominio (lsj-app.xyz) a la dirección IP pública del Network Load Balancer (NLB), haciendo uso de subdominios (www.lsj-app.xyz y api.lsj-app.xyz).
- **Network Load Balancer:** Este es el primer punto de entrada a la infraestructura en AWS. Recibe el tráfico externo y lo reenvía al clúster de Kubernetes, específicamente al servicio del Ingress Controller.
- **Ingress-Nginx Controller:** Este componente, que se ejecuta dentro del clúster en su propio namespace (ingress-nginx), actúa como un proxy inverso inteligente. Su trabajo es leer las reglas definidas en los recursos Ingress.
- **Ingress Resources (ING):** Son objetos de Kubernetes que definen cómo se debe enrutar el tráfico HTTP/S desde el exterior hacia los Services dentro del clúster. Por ejemplo, una regla podría decir: "El tráfico que llega a api.tu-empresa.com/orders debe ser dirigido al order-service". El diagrama muestra Ingresses para frontend-gateway y api-gateway.
- **Service y Pods:** Finalmente, el Ingress Controller enruta la solicitud al Service correspondiente, que a su vez balancea la carga entre los pods saludables del microservicio de destino.

### Certificados y Seguridad

**cert-manager:** Esta herramienta automatiza la gestión de certificados TLS/SSL en el clúster. Se integra con el Ingress Controller para:

- Detectar recursos Ingress que requieren HTTPS (Frontend Ingre).
- Solicitar y obtener automáticamente certificados de autoridades de certificación como Let's Encrypt.

- Renovar los certificados antes de que expiren.
- Configurar el Ingress para que utilice estos certificados, asegurando así que la comunicación esté cifrada (HTTPS).

**Network Policies (netpol)**: El ícono netpol dentro de los namespaces indica el uso de Políticas de Red. Estas actúan como un firewall a nivel de pod, controlando qué pods pueden comunicarse entre sí. Por ejemplo, se puede configurar una política para que solo el api-gateway pueda comunicarse con el payment-service, bloqueando cualquier otro tipo de acceso no autorizado.

### Entornos de Despliegue

La arquitectura soporta múltiples entornos para un ciclo de vida de desarrollo seguro y controlado.

- **Namespaces por Entorno**: Se utilizan namespaces de Kubernetes para aislar los entornos: master (producción), stage (pre-producción) y dev (desarrollo).
- **Paridad de Entornos**: Los entornos dev y stage son réplicas de la arquitectura de producción (master). Esto es una buena práctica, ya que permite probar los cambios en un entorno idéntico al de producción, reduciendo el riesgo de errores inesperados.
- **Gestión con Argo CD**: Argo CD gestiona los despliegues en cada uno de estos entornos, apuntando a cada proyecto en el repositorio de Helm Charts para aplicar configuraciones específicas de cada entorno (values-dev.yaml, values-stage.yaml o values-master.yaml).

### Contenedores e Imágenes

Los contenedores son la unidad de empaquetado y distribución de las aplicaciones.

**Creación de Imágenes**: El pipeline de GitHub Actions es responsable de construir las imágenes de contenedor (Docker) a partir del código fuente de los repositorios (Microservice Repository, Frontend Repository).

**Registro de Contenedores**: Una vez construidas, las imágenes se etiquetan (generalmente con el hash del commit o una versión semántica) y se suben a Amazon Elastic Container Registry (ECR), que actúa como el repositorio central y privado de imágenes.

**Despliegue de Imágenes**: Cuando Argo CD actualiza un Deployment en Kubernetes, el clúster se encarga de extraer (pull) la nueva versión de la imagen desde ECR para crear los nuevos pods.

### Flujo de Integración Continua y Entrega Continua (CI/CD)

Este es el corazón de la automatización del sistema y sigue los principios de GitOps.

CI (Integración Continua) - A cargo de GitHub Actions:

1. **Trigger**: Un desarrollador realiza push o un pull request es aceptado en una de las repositorios en GitHub.

2. **Build & Push:** GitHub Actions se activa, ejecuta pruebas, construye la imagen del contenedor y la empuja a Amazon ECR.
3. **Update Manifests:** Como último paso, el mismo workflow de GitHub Actions modifica el repositorio de configuración (Helm Charts Repository). Concretamente, actualiza un archivo (como values.yaml) para que apunte a la nueva etiqueta de la imagen que acaba de subir a ECR. Este paso es el enlace crucial entre CI y CD.

CD (Entrega Continua) - A cargo de Argo CD:

1. **Observe:** Argo CD está configurado para "observar" el GitHub Helm Charts Repository. Monitorea constantemente este repositorio en busca de cambios.
2. **Detect & Sync:** Cuando detecta el commit realizado por GitHub Actions (el cambio en la etiqueta de la imagen), Argo CD se da cuenta de que el estado actual del clúster (Live State) ya no coincide con el estado deseado en Git (Target State).
3. **Deploy:** Argo CD procede a sincronizar el estado. Aplica los manifiestos actualizados de Helm al clúster de Kubernetes, lo que hace que el Deployment correspondiente se actualice de forma controlada (rolling update), reemplazando los pods antiguos por los nuevos que usan la nueva imagen.

### Argo CD

Argo CD es la herramienta central para el Despliegue Continuo (CD) basada en GitOps. Su función es asegurar que el estado del clúster de Kubernetes sea un reflejo fiel de lo que está definido en un repositorio Git (en este caso, el Helm Charts Repository).

- **Fuente de la Verdad:** El repositorio de Git es la única fuente de la verdad para la configuración de las aplicaciones.
- **Automatización:** Elimina la necesidad de despliegues manuales (con kubectl apply). Los cambios se aplican automáticamente al fusionar código en la rama principal del repositorio de configuración.
- **Visibilidad:** Proporciona una interfaz de usuario para ver el estado de los despliegues, el historial de sincronización y detectar cualquier deriva de configuración.

### GitHub Actions

GitHub Actions es el motor de la Integración Continua (CI). Es una herramienta de automatización de flujos de trabajo integrada en GitHub. En esta arquitectura, es responsable de:

- Reaccionar a eventos de Git (como push o pull\_request).
- Construir, probar y empaquetar el código fuente en imágenes de contenedor.
- Publicar artefactos (imágenes de contenedor en ECR).
- Orquestar el "hand-off" a la CD, actualizando el repositorio de Helm que Argo CD está observando.

### Monitoreo y Observabilidad

La arquitectura cuenta con dos sistemas complementarios para entender el comportamiento del sistema en tiempo real.

## Monitoring (Métricas)

- Prometheus: Se encarga de recopilar métricas numéricas en series de tiempo (CPU, memoria, latencia de peticiones, tasas de error) de los microservicios y del propio clúster.
- Grafana: Se conecta a Prometheus como fuente de datos para visualizar estas métricas en paneles y dashboards interactivos, permitiendo a los equipos monitorear la salud del sistema y configurar alertas.

## Logging (Registros)

Elastic Stack (ELK): Se utiliza para la gestión centralizada de logs.

- **Filebeat**: Se ejecuta en cada nodo del clúster y recoge los logs de los contenedores.
- **Logstash**: Procesa y enriquece estos logs.
- **Elasticsearch**: Indexa y almacena los logs de manera eficiente.
- **Kibana**: Proporciona una potente interfaz de usuario para buscar, analizar y visualizar los logs, lo cual es crucial para la depuración de errores.

## Patrones de Cloud

### Patrón Retry

En el código del frontend se implementó el patrón de retry, cuyo objetivo es que si el api gateway no responde a alguna solicitud, entonces el frontend la intentará de nuevo un número determinado de veces. Si no responde después de MAX RETRIES, entonces para con el intento. A continuación se puede ver el código:

```
console.log('Error recibido:', error.status, error.statusText);
if (retries < this.MAX_RETRIES &&
    (error.status === 0 || error.status === 404 || error.status >= 500)) {
  retries++;
  console.log(`Reintentando petición ${retries}/${this.MAX_RETRIES} debido a error ${error.status}...`);
  return timer(this.RETRY_DELAY * retries);
}
console.log('No se reintentará más la petición');
return throwError(() => error);
```

Y aquí, se puede ver un ejemplo de lo que ocurre, para esto se ejecutó el frontend sin ejecutar el api-gateway o los demás microservicios:

```
✖ ▶ GET http://localhost:4200/app/api/categories 404 (Not Found)
Error recibido: 404 Not Found
Reintentando petición (1/3) debido a error 404...
root
Content Script re-injected or page loaded
[webpack-dev-server] Live Reloading enabled.
✖ ▶ GET http://localhost:4200/app/api/categories 404 (Not Found)
Error recibido: 404 Not Found
Reintentando petición (2/3) debido a error 404...
✖ ▶ GET http://localhost:4200/app/api/categories 404 (Not Found)
Error recibido: 404 Not Found
Reintentando petición (3/3) debido a error 404...
>
```

### **Patrón External Configuration Store (Configmaps)**

Para implementar el patrón de external configuration store se hizo uso de configmaps, los cuales se encargan de almacenar de forma externa a los microservicios las configuraciones de estos, por lo que los deployments deben hacer uso del configmap para poder funcionar y tener las variables de entorno correctas, esto sirve para centralizar la configuración y poder realizar cambios de manera rápida en configuraciones que son compartidas por múltiples microservicios. Para mayor información, diríjase a la sección “*Migración de todas las configuraciones de Spring Boot a ConfigMaps*”.

### **Patrón Health Endpoint Monitoring**

El patrón Health Endpoint Monitoring fue implementado mediante liveness y readiness probes haciendo uso de los actuator endpoints de spring boot. La información acerca de la implementación se puede encontrar en la sección “*Aprovechamiento de los Actuator endpoints de Spring Boot para monitoreo*”.

### **Patrón Sidecar**

El patrón Sidecar fue implementado mediante la integración de Linkerd como proxy de red para cada uno de los microservicios desplegados en el ambiente de producción. Este proxy actúa como un contenedor adicional dentro del mismo pod, encargándose de gestionar tareas transversales como la comunicación segura, balanceo de carga, observabilidad y control de tráfico, sin necesidad de modificar la lógica de negocio de los microservicios. De esta forma, se desacoplan responsabilidades relacionadas con la red, facilitando la implementación de prácticas de service mesh y mejorando la resiliencia de toda la arquitectura. Para más detalles

sobre la configuración y despliegue de Linkerd, consulte la sección “Implementación del Service Mesh con Linkerd para habilitar Sidecar Proxies”.

## Addons del Cluster de EKS

En los archivos de terraform se definieron los siguientes addons para que el cluster de EKS pudiera funcionar correctamente:

The screenshot shows the AWS EKS Add-ons management interface. It lists four active add-ons:

- Amazon EBS CSI Driver**: Category storage, Status Active, Version v1.44.0-eksbuild.1, EKS Pod Identity -, IAM role for service account (IRSA) Not set.
- kube-proxy**: Category networking, Status Active, Version v1.28.8-eksbuild.5, EKS Pod Identity -, IAM role for service account (IRSA) Not set. Includes a blue "Update version" button.
- CoreDNS**: Category networking, Status Active, Version v1.10.1-eksbuild.7, EKS Pod Identity -, IAM role for service account (IRSA) Not set. Includes a blue "Update version" button.
- Amazon VPC CNI**: Category networking, Status Active, Version v1.19.0-eksbuild.1, EKS Pod Identity -, IAM role for service account (IRSA) Not set. Includes a blue "Update version" button.

## Amazon VPC CNI

Este complemento permite la conectividad de red de los pods dentro del clúster de Kubernetes usando la red de Amazon VPC. Gestiona interfaces de red elásticas (ENI) para asignar direcciones IP a los pods. Su configuración y detalles se describen en la sección “Configuración de Red y Seguridad”.

### **kube-proxy**

**kube-proxy** es un componente de red de Kubernetes que implementa reglas de red en cada nodo del clúster para permitir la comunicación entre pods y servicios, y para dirigir el tráfico de red hacia los endpoints correctos. Administra reglas de iptables o IPVS para manejar el enrutamiento del tráfico y el balanceo de carga interno.

### **CoreDNS**

CoreDNS es el servidor de nombres de dominio (DNS) para Kubernetes. Proporciona resolución de nombres de servicios dentro del clúster, permitiendo que los pods localicen otros servicios y recursos mediante nombres DNS. Es un componente clave para el descubrimiento de servicios en Kubernetes.

## **Amazon EBS CSI Driver**

Este complemento permite a Kubernetes gestionar volúmenes de almacenamiento de Amazon Elastic Block Store (EBS) de forma dinámica. Facilita la provisión, el montaje y la eliminación automática de volúmenes persistentes, lo que es esencial para aplicaciones que necesitan almacenamiento de datos duradero.

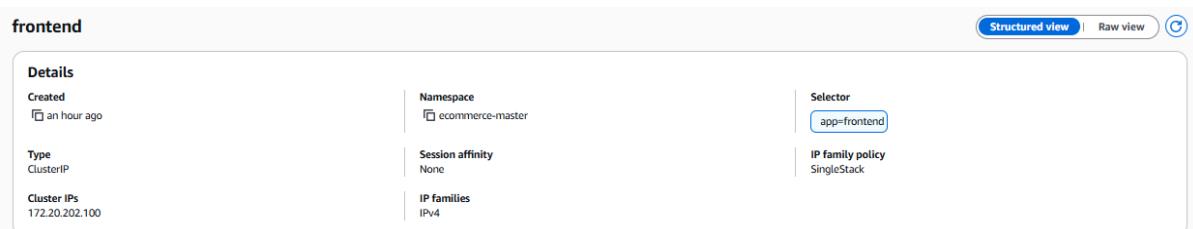
## **Configuración de Red y Seguridad**

### **Servicios Kubernetes configurados correctamente**

En el clúster de Kubernetes se han configurado correctamente distintos tipos de servicios según el propósito de cada componente, en este caso se habla solo del frontend, sin embargo en los demás servicios se trata también de clusterIP, ya que el tráfico externo se hace a través del ingress, por ello es cluster IP sin necesidad de recurrir a otros tipos como NodePort o LoadBalancer:

#### **1. Servicio de tipo ClusterIP para el frontend:**

El servicio que expone la aplicación frontend utiliza el tipo ClusterIP, lo cual permite la comunicación interna entre los pods y otros componentes del clúster sin exponerlo directamente al exterior. Este servicio se ubica en todos los namespaces, como el namespace ecommerce-master, ecommerce-dev y ecommerce-stage y selecciona los pods etiquetados con app=frontend.



The screenshot shows the 'frontend' service configuration in the Kubernetes dashboard. The service is of type ClusterIP, created an hour ago, and is associated with the ecommerce-master namespace. It has no session affinity and uses IPv4. The selector is defined by the label app=frontend. The service has one cluster IP address: 172.20.202.100.

Details	Namespace	Selector
Created: an hour ago	ecommerce-master	app=frontend
Type: ClusterIP	Session affinity: None	IP family policy: SingleStack
Cluster IPs: 172.20.202.100	IP families: IPv4	

Este tipo de servicio es útil para mantener una arquitectura segura, ya que evita exponer directamente el frontend, permitiendo que el acceso externo se controle únicamente a través del Ingress Controller y TLS.

#### **2. Servicio de tipo LoadBalancer para el Ingress Controller:**

El Ingress Controller implementado con NGINX se expone mediante un servicio de tipo LoadBalancer, que en su configuración crea internamente puertos NodePort para las conexiones HTTP y HTTPS.

**Details**

**Created**: 2 hours ago

**Namespace**: ingress-nginx

**Finalizers**: service.kubernetes.io/load-balancer<cleanup>

**Type**: LoadBalancer

**IP Family policy**: SingleStack

**Cluster IPs**: 172.20.223.191

**Selector**:

- app.kubernetes.io/component=controller
- app.kubernetes.io/instance=ingress-nginx
- app.kubernetes.io/name=ingress-nginx

**Session affinity**: None

**IP families**: IPv4

**Ports (2)**

Name	Protocol	Application protocol	Port	Target port	Node port
http	TCP	http	80	http	30887
https	TCP	https	443	https	32667

**Endpoints (2)**

IP	Hostname	Node name	Target reference
10.0.1.114:443	-	ip-10-0-1-211.ec2.internal	ingress-nginx-controller-7f9b84b7df-z8xw6
10.0.1.114:80	-	ip-10-0-1-211.ec2.internal	ingress-nginx-controller-7f9b84b7df-z8xw6

Se puede ver que el nombre del servicio es `ingress-nginx-controller`, en un namespace llamado `ingress-nginx` de tipo `LoadBalancer`, de aquí viene la url que se configura en el DNS de GoDaddy, los protocolos habilitados son `HTTP` para el puerto `80` y `HTTPS` para el puerto `443`. Gracias a esta configuración, tanto el frontend como la API Gateway pueden recibir tráfico externo de manera segura mediante `HTTPS` (`TLS`), utilizando dominios personalizados configurados en GoDaddy y certificados obtenidos por Cert-Manager con Let's Encrypt.

### **Implementación de Ingress Controller para el acceso a la API Gateway y al Frontend:**

Con el objetivo de gestionar el acceso externo a los servicios internos del clúster de Kubernetes, se implementó un Ingress Controller basado en NGINX, que permite definir reglas de entrada `HTTP/HTTPS` y enrutar solicitudes a los servicios adecuados, manteniendo control y seguridad sobre los puntos de entrada.

#### **Instalación del NGINX Ingress Controller:**

La instalación se realizó utilizando Helm, una herramienta de gestión de paquetes para Kubernetes. El despliegue incluyó las siguientes configuraciones clave:

```

- name: Install NGINX Ingress Controller
  run: |
    helm upgrade --install ingress-nginx ingress-nginx/ingress-nginx \
      --namespace ingress-nginx \
      --create-namespace \
      --set controller.service.type=LoadBalancer \
      --set controller.service.annotations."service\\.beta\\.kubernetes\\.io/aws-load-balancer-type\"="nlb" \
      --wait
  
```

- Se desplegó el Ingress Controller en un espacio de nombres dedicado llamado ingress-nginx.
- El servicio fue expuesto mediante un LoadBalancer externo, lo cual permite recibir tráfico desde fuera del clúster.
- Se utilizó un Network Load Balancer (NLB) para mejorar la escalabilidad y disponibilidad del tráfico entrante.

**Service and networking: Services (2)**

Service is an abstract way to expose an application running on a set of Pods as a network service. [Learn more](#)

Name	Created
ingress-nginx-controller	24 minutes ago
ingress-nginx-controller-admission	24 minutes ago

**ad8f70754ee044b74b88bb44fc1a538d**

**Details**

Load balancer type	Status	VPC	Load balancer IP address type
Network	Active	vpc-00921e859841bc85c	IPv4
Scheme	Hosted zone	Availability Zones	Date created
Internet-facing	Z26RNL4JYFTOTI	subnet-08d5091d98f53f51f us-east-1a (use1-az6) subnet-063118b9736cceddf us-east-1b (use1-az1)	June 12, 2025, 23:29 (UTC-05:00)
Load balancer ARN	DNS name info		
arn:aws:elasticloadbalancing:us-east-1:211125706306:loadbalancer/net/ad8f70754ee044b74b88bb44fc1a538d/ed222ee8b4730292	ad8f70754ee044b74b88bb44fc1a538d-ed222ee8b4730292.elb.us-east-1.amazonaws.com (A Record)		

**Listeners** | Network mapping | Resource map | Security | Monitoring | Integrations | Attributes | Capacity | Tags

**Listeners (2)**

A listener checks for connection requests using the protocol and port that you configure. Traffic received by a Network Load Balancer listener is forwarded to the selected target group.

Protocol:Port	Default action	ARN	Security policy	Default SSL/TLS certificate	ALPN policy	Tags
TCP:443	Forward to target group k8s-ingress-ingress-3bb5838c6	ARN	Not applicable	Not applicable	None	2 tags
TCP:80	Forward to target group k8s-ingress-ingress-ea144bbcea	ARN	Not applicable	Not applicable	None	2 tags

## Configuración de Ingress para el Frontend

Se definió un recurso Ingress llamado frontend-ingress, encargado de enrutar el tráfico HTTP/HTTPS dirigido al dominio correspondiente del frontend hacia el servicio frontend interno en el puerto 80. Este recurso también incluyó configuraciones TLS para habilitar el uso de certificados y cifrado seguro.

## Configuración de Ingress para el API Gateway

Del mismo modo, se configuró un recurso Ingress denominado api-gateway-ingress, este se verá más adelante en la sección de gestión de configuración y secretos, este dirige el tráfico entrante hacia el servicio api-gateway a través del puerto 8080. También incluye TLS y anotaciones específicas para integrarse con cert-manager, que gestiona automáticamente los certificados SSL.

Service and networking: Ingresses (4)	
Ingress is an API object that manages external access to the services in a cluster, typically HTTP. <a href="#">Learn more</a>	
ecommerce-master	View details
<input type="button"/> <a href="#">Filter Ingresses by name</a>	< 1 >
Name	Created
api-gateway-ingress	2 minutes ago
cm-acme-http-solver-ddd19	2 minutes ago
cm-acme-http-solver-dx7gv	2 minutes ago
frontend-ingress	2 minutes ago

### NetworkPolicies configuradas para restringir comunicación entre servicios:

Con el objetivo de garantizar la seguridad en la comunicación entre los distintos microservicios del clúster, se implementaron políticas de red (NetworkPolicies) en Kubernetes. Estas políticas permiten establecer reglas específicas sobre qué pods pueden comunicarse entre sí y en qué puertos, reduciendo significativamente la superficie de ataque y permitiendo un control más granular del tráfico interno del clúster.

Policy: NetworkPolicies (23)	
Network Policy is a specification of how groups of Pods are allowed to communicate with each other and with other network endpoints. <a href="#">Learn more</a>	
ecommerce-master	View details
<input type="button"/> <a href="#">Filter NetworkPolicies by name</a>	< 1 2 3 >
Name	Created
allow-all-egress	11 minutes ago
allow-api-from-ingress	11 minutes ago
allow-cloud-ingress	11 minutes ago
allow-favourite-and-product	11 minutes ago
allow-favourite-and-user	11 minutes ago
allow-favourite-from-proxy	11 minutes ago
allow-order-and-payment	11 minutes ago
allow-order-and-shipping	11 minutes ago
allow-order-from-proxy	11 minutes ago
allow-payment-and-order	11 minutes ago

Las NetworkPolicies aplicadas responden a los siguientes puntos:

#### 1. Aislamiento controlado entre pods

Por defecto, los pods permiten tráfico de red entrante y saliente sin restricciones. Para controlar esto, se implementó la política allow-all-egress, la cual permite que todos los pods puedan comunicarse hacia afuera, pero deja restringido el tráfico de entrada (Ingress), que

debe habilitarse explícitamente por cada servicio. Esta política establece un aislamiento base y fuerza a definir de manera intencionada las conexiones válidas entre pods.

## 2. Permisos específicos entre microservicios

Se definieron múltiples políticas de Ingress para cada pod que necesita recibir solicitudes, estas políticas indican claramente qué pods pueden conectarse con qué otros, y sobre qué puertos. A continuación, se detallan las relaciones habilitadas entre servicios:

- api-gateway permite el ingreso desde api-gateway-ingress por el puerto 8080.
- proxy-client puede ser accedido por api-gateway en el puerto 8900.
- cloud-config solo permite comunicación desde servicios internos específicos en el puerto 9296.
- favourite-service permite conexiones desde product-service (puerto 8800) y desde proxy-client.
- product-service permite conexiones desde favourite-service (puerto 8500) y desde proxy-client.
- payment-service permite conexiones desde order-service (puerto 8400) y desde proxy-client.
- order-service permite conexiones desde payment-service (puerto 8300), shipping-service (puerto 8300) y proxy-client.
- shipping-service permite conexiones desde order-service, product-service y proxy-client (puerto 8600).
- user-service y favourite-service pueden comunicarse mutuamente, así como recibir solicitudes desde proxy-client.
- service-discovery (Eureka) permite el ingreso de solicitudes de todos los servicios que necesitan registrarse (puerto 8761).
- zipkin permite tráfico desde todos los servicios para enviar trazas distribuidas (puerto 9411).

Cada regla Ingress define no sólo el origen permitido sino también el puerto exacto, evitando accesos innecesarios o generales.

## 3. Trazabilidad y control para observabilidad

Servicios como zipkin, cloud config y service-discovery reciben tráfico de múltiples servicios, pero cada acceso está controlado y limitado a los puertos requeridos (9411, 9296 y 8761 respectivamente), permitiendo así una correcta trazabilidad de llamadas y registro sin comprometer la seguridad general del clúster.

#### 4. Relaciones simétricas entre servicios

Se aplicaron reglas de doble vía (mutuas) para servicios que requieren comunicación bidireccional, como order-service y payment-service, shipping-service y product-service, o user-service y favourite-service. Cada servicio tiene una NetworkPolicy que define explícitamente los permisos de ingreso desde su contraparte.

#### 5. Principio de mínimo privilegio

Todas las políticas aplicadas siguen el principio de mínimo privilegio: solo los servicios estrictamente necesarios pueden comunicarse entre sí, en los puertos que se requieren para su funcionamiento, y no se permiten conexiones abiertas o sin control.

#### Uso de Calico para habilitar NetworkPolicies

Para que las NetworkPolicies funcionen correctamente en el clúster de Kubernetes, es necesario contar con un plugin de red (CNI) que soporte su implementación. En este caso, se utilizó Calico en su modalidad policy-only, que permite aplicar políticas de red sin modificar el plano de datos de red ya existente del clúster (vpc cnf).

La instalación de Calico se realizó mediante la aplicación del manifiesto oficial correspondiente a la versión v3.28.2, que habilita exclusivamente el motor de políticas:

```
- name: Install Calico Network Policies
  run: |
    kubectl apply -f https://raw.githubusercontent.com/projectcalico/calico/v3.28.2/manifests/calico-policy-only.yaml
    kubectl wait --for condition=ready pods -l k8s-app=calico-kube-controllers -n kube-system --timeout=300s
```

Con esta configuración, Calico actúa como el componente responsable de aplicar y hacer cumplir las NetworkPolicies definidas en el clúster, sin alterar la infraestructura de red subyacente. Esta elección permite mantener flexibilidad en la configuración del entorno mientras se aprovechan las capacidades avanzadas de segmentación y control de tráfico que proporciona Calico, tales como:

- Aislamiento entre pods según etiquetas.
- Control específico de puertos y protocolos.
- Escalabilidad para clústeres de gran tamaño.
- Compatibilidad con herramientas de observabilidad.

El uso de Calico resultó fundamental para lograr una implementación efectiva del principio de mínimo privilegio en las comunicaciones internas del clúster, asegurando así una mayor protección ante accesos no autorizados y posibles vulnerabilidades.

### **Configuración de Red y Políticas de Red (Network Policies)**

El clúster de Kubernetes desplegado en AWS EKS utiliza el add-on oficial de Amazon VPC CNI para gestionar la red de pods y servicios. Este componente es fundamental para habilitar la comunicación dentro del clúster, permitiendo que cada pod reciba una dirección IP directamente desde la VPC. Se encuentra instalado y activo el add-on Amazon VPC CNI, como:

- Versión: v1.19.0-eksbuild.1
- Categoría: Networking
- Estado: Activo

Este CNI proporciona integración nativa entre Kubernetes y la red de AWS, lo que significa que los pods pueden comunicarse con otros servicios dentro y fuera del clúster utilizando direcciones IP privadas. Esto mejora el rendimiento de red y simplifica la configuración de seguridad mediante Security Groups y políticas de red basadas en VPC.



Esta combinación permite mantener el rendimiento de red proporcionado por AWS, mientras se gana el control granular sobre la comunicación entre recursos del clúster.

### **Configuración de TLS/HTTPS para endpoints públicos:**

Con el objetivo de asegurar el tráfico hacia los servicios expuestos públicamente (como el frontend y el API Gateway), se implementó la configuración de TLS/HTTPS utilizando Cert-Manager junto con Let's Encrypt como proveedor de certificados SSL gratuitos.

#### **1. Instalación de Cert-Manager**

Se utilizó Helm para instalar Cert-Manager, encargado de emitir y renovar certificados TLS automáticamente:

```
- name: Install Cert Manager
run: |
  helm upgrade --install cert-manager jetstack/cert-manager \
    --namespace cert-manager \
    --create-namespace \
    --version v1.16.1 \
    --set crds.enabled=true \
    --wait
```

## 2. Creación del ClusterIssuer:

Se definió un recurso ClusterIssuer para obtener certificados firmados por Let's Encrypt en su entorno de producción, usando el método de validación HTTP-01. Este recurso permite que Cert-Manager genere certificados para los dominios personalizados de forma automática.

```
! cluster-issuer.yaml ! values-master.yaml ! values.yaml ! allow...
charts > ecommerce > templates > ! cluster-issuer.yaml
1  {{< if .Values.certManager.enabled >}}
2    apiVersion: cert-manager.io/v1
3    kind: ClusterIssuer
4    metadata:
5      name: letsencrypt-prod
6    spec:
7      acme:
8        email: {{ .Values.certManager.email }}
9        server: https://acme-v02.api.letsencrypt.org/directory
10       privateKeyRef:
11         name: letsencrypt-prod-key
12       solvers:
13         - http01:
14           ingress:
15             class: {{ .Values.ingress.class }}
16 {{< end >}}
```

## 3. Configuración de los Ingress:

Los certificados generados se aplican automáticamente a los recursos Ingress definidos para los servicios frontend y api-gateway. Estos recursos hacen referencia a los dominios públicos personalizados y a los secretos TLS generados:

```
domains:
  frontend:
    host: www.lsj-app.xyz
    tlsSecret: frontend-tls
  api:
    host: api.lsj-app.xyz
    tlsSecret: api-gateway-tls
certManager:
```

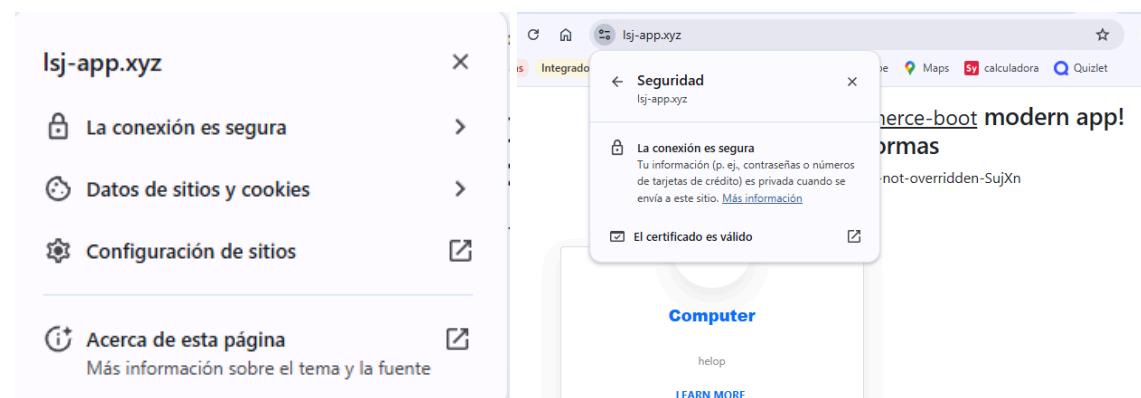
Esto permite que los Ingress utilicen estos valores desde el archivo values-master.yaml para establecer las rutas y aplicar el cifrado TLS.

## 4. Uso de Dominio Personal (GoDaddy):

Para utilizar un dominio personalizado registrado en GoDaddy (www.lsj-app.xyz), fue necesario configurar los registros DNS tipo CNAME en GoDaddy apuntando a la IP pública asignada por el Load Balancer:

	CNAME	api	ad8f70754ee0 44b74b88bb4 4fc1a538d-ed 222ee8b4730 292.elb.us-eas t-1.amazonaw s.com.	1 Hora		
	CNAME	www	ad8f70754ee0 44b74b88bb4 4fc1a538d-ed 222ee8b4730 292.elb.us-eas t-1.amazonaw s.com.	1 Hora		

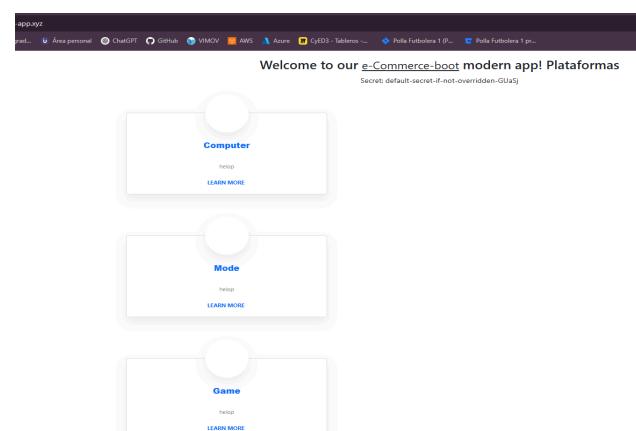
Esto permite que Cert-Manager valide los dominios y emita certificados válidos mediante Let's Encrypt, habilitando el acceso seguro vía HTTPS a los servicios públicos desplegados en el clúster.



## 5. Acceso Seguro a la API Gateway desde Herramientas Externas (Postman):

El dominio personalizado del API Gateway (<https://api.lsj-app.xyz>) es utilizado para acceder a los servicios del backend desde herramientas externas como Postman. Gracias a la configuración de TLS con Cert-Manager y Let's Encrypt, este endpoint expone una conexión segura (HTTPS), permitiendo realizar pruebas de consumo de APIs, autenticación y envío de peticiones de forma cifrada.

A continuación, una pequeña demostración sobre el funcionamiento de la url `api.lsj-api.xyz`, primero se puede observar el estado inicial del frontend:



Se usa postman y la url que hace referencia a la api, tal cual como se mencionó anteriormente, útil para demostrar el envío de peticiones:

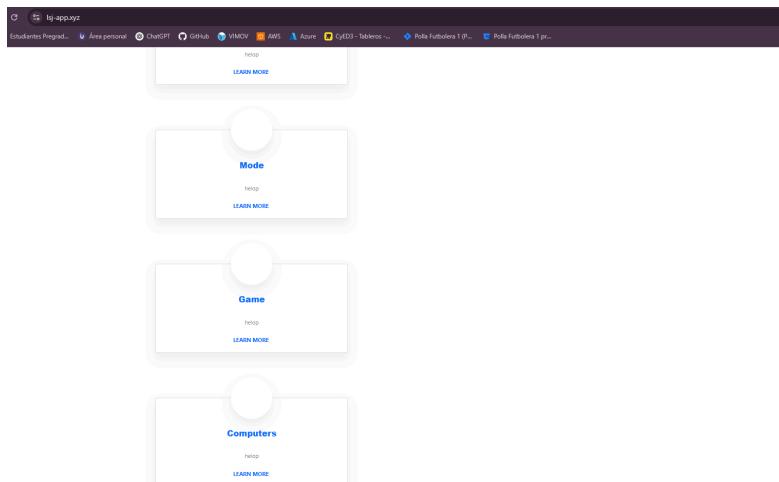
The screenshot shows a Postman interface with a POST request to `https://api.lsj-app.xyz/api/categories`. The request body is a JSON object with the following structure:

```
1: { "categoryTitle": "Computers", "parentCategory": { "categoryId": "2" } }
```

The response status is 200 OK, with a response time of 176 ms and a response size of 657 B. The response body is also a JSON object:

```
1: { "categoryId": 6, "categoryTitle": "Computers", "imageUrl": null, "parentCategory": { "categoryId": 2, "categoryTitle": null, "imageUrl": null } }
```

Y de acuerdo a eso, se puede observar como se le agrega una sección, y de esta manera se demuestra el funcionamiento de la api:



### Implementación del Service Mesh con Linkerd para habilitar Sidecar Proxies

Para garantizar la seguridad, observabilidad y control del tráfico entre microservicios, se implementó Linkerd como Service Mesh dentro del clúster Kubernetes. Esta herramienta permite injectar proxies sidecar de forma automática en cada pod, lo que habilita características avanzadas como mTLS (Mutual TLS), balanceo de carga, métricas y trazabilidad, sin necesidad de modificar el código de los servicios.

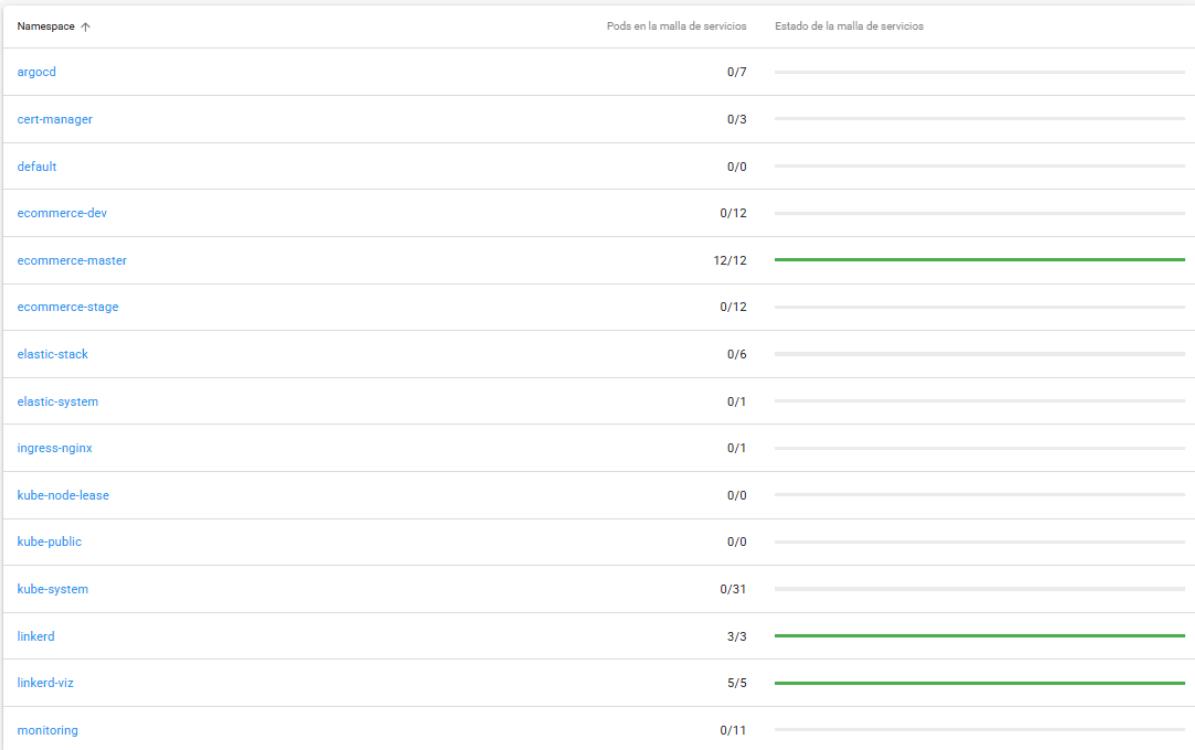
La imagen siguiente presenta el estado del plano de control de Linkerd, donde se confirma que los componentes esenciales (Destination, Identity y Proxy Injector) están desplegados y funcionando correctamente. Además, se evidencia que la extensión viz, encargada de la visualización, está instalada y activa en su namespace correspondiente:

Plano de control		Componentes
		3
Deployment	Pods	Estado del Pod
Destination	1	●
Identity	1	●
Proxy Injector	1	●

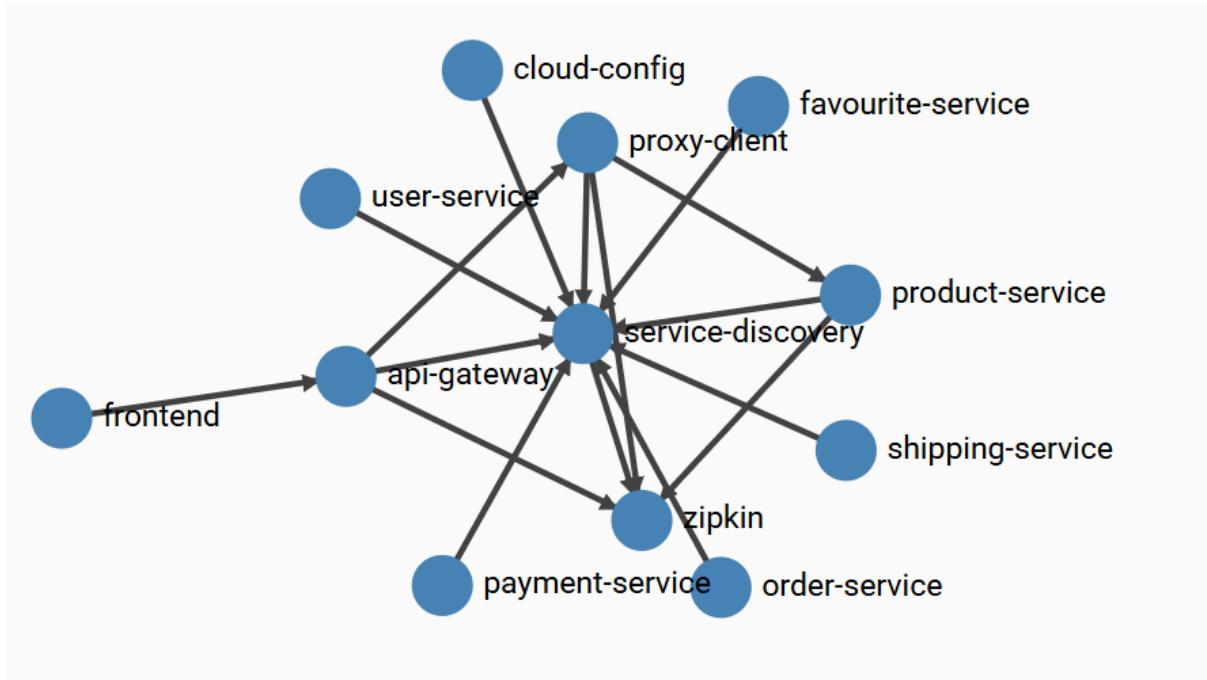
  

Installed Extensions	
Nombre	Namespace
viz	linkerd-viz

A continuación, se muestra un resumen del estado de la malla de servicios por namespace. Aquí se puede observar que todos los pods en los namespaces ecommerce-master (el escogido) y Linkerd están integrados en la malla y reportan un estado saludable, indicando que la inyección de los proxies sidecar es exitosa en cada microservicio:



En la siguiente imagen se puede ver gráficamente la topología de comunicación entre los servicios del namespace ecommerce-master. Cada nodo representa un microservicio y las flechas indican el flujo de tráfico entre ellos, lo que permite monitorear la interacción y desempeño dentro de la malla de servicios gestionada por Linkerd:



En la sección de métricas HTTP, se pueden consultar indicadores clave como la tasa de éxito de las solicitudes, la cantidad de peticiones por segundo (PPS) y las métricas de latencia para cada pod inscrito en la malla. Todos los pods muestran una tasa de éxito del 100% y latencias bajas, lo cual refleja un buen rendimiento de la malla y de la aplicación en general:

Métricas HTTP	Pod ↑	↑ En la malla de servicios	↑ Tasa de éxito	↑ PPS	↑ Latencia P50	↑ Latencia P95	↑ Latencia P99
	api-gateway-56bbc59d7c-869w9	1/1	100.00% ●	0.3	1 ms	5 ms	5 ms
	cloud-config-69d6cc796-kd2cm	1/1	100.00% ●	0.3	1 ms	2 ms	2 ms
	favourite-service-6dbbc75f85-98dk7	1/1	100.00% ●	0.3	1 ms	3 ms	3 ms
	frontend-657bd9bd76-znq4m	1/1	100.00% ●	0.3	1 ms	1 ms	1 ms
	order-service-68bf4dd9f6-g5b2q	1/1	100.00% ●	0.3	1 ms	2 ms	2 ms
	payment-service-cd9b457f-4thw4	1/1	100.00% ●	0.3	1 ms	2 ms	2 ms
	product-service-f5777566d-j9srg	1/1	100.00% ●	0.3	1 ms	2 ms	2 ms
	proxy-client-6fc74d8596-fs5gl	1/1	100.00% ●	0.3	1 ms	2 ms	2 ms
	service-discovery-77f7d484d7-t9mc6	1/1	100.00% ●	0.9	1 ms	1 ms	2 ms
	shipping-service-cd9698bbc-r966g	1/1	100.00% ●	0.6	2 ms	3 ms	3 ms
	user-service-c66646d8d-ff4sr	1/1	100.00% ●	0.6	1 ms	3 ms	3 ms
	zipkin-546bf9547b-fgr6n	1/1	100.00% ●	0.7	1 ms	3 ms	3 ms

Por último, se muestra la salida del comando “linkerd viz edges”, la cual detalla las conexiones entre los deployments dentro del namespace ecommerce-master. La columna SECURED muestra una marca de verificación para cada conexión, confirmando que todas las comunicaciones están protegidas mediante TLS mutuo gestionado por Linkerd:

SRC	DST	SRC_NS	DST_NS	SECURED
api-gateway	service-discovery	ecommerce-master	ecommerce-master	✓
cloud-config	service-discovery	ecommerce-master	ecommerce-master	✓
favourite-service	service-discovery	ecommerce-master	ecommerce-master	✓
order-service	service-discovery	ecommerce-master	ecommerce-master	✓
payment-service	service-discovery	ecommerce-master	ecommerce-master	✓
product-service	service-discovery	ecommerce-master	ecommerce-master	✓
proxy-client	service-discovery	ecommerce-master	ecommerce-master	✓
service-discovery	zipkin	ecommerce-master	ecommerce-master	✓
shipping-service	service-discovery	ecommerce-master	ecommerce-master	✓
user-service	service-discovery	ecommerce-master	ecommerce-master	✓
prometheus	api-gateway	linkerd-viz	ecommerce-master	✓
prometheus	cloud-config	linkerd-viz	ecommerce-master	✓
prometheus	favourite-service	linkerd-viz	ecommerce-master	✓
prometheus	frontend	linkerd-viz	ecommerce-master	✓
prometheus	order-service	linkerd-viz	ecommerce-master	✓
prometheus	payment-service	linkerd-viz	ecommerce-master	✓
prometheus	product-service	linkerd-viz	ecommerce-master	✓
prometheus	proxy-client	linkerd-viz	ecommerce-master	✓
prometheus	service-discovery	linkerd-viz	ecommerce-master	✓
prometheus	shipping-service	linkerd-viz	ecommerce-master	✓
prometheus	user-service	linkerd-viz	ecommerce-master	✓
prometheus	zipkin	linkerd-viz	ecommerce-master	✓

### Implementación de ServiceAccounts con permisos mínimos necesarios RBAC:

Los ServiceAccounts de ArgoCD son identidades de Kubernetes esenciales para su operación segura y funcional, se crean para seguir el principio de menor privilegio, asignando a cada componente de ArgoCD (como el controlador de aplicaciones, el servidor API y el servidor de repositorios) solo los permisos estrictamente necesarios para cumplir su función específica dentro del clúster, esto mejora la seguridad al limitar el impacto potencial de cualquier componente individual y permite una gestión de permisos más granular y clara.

Estos ServiceAccounts, vinculados a Roles y RoleBindings de Kubernetes, permiten que el controlador de aplicaciones gestione los recursos de las aplicaciones (deployments, services, etc.) en los clústeres de destino, que el servidor API interactúe con los usuarios y gestione las entidades de ArgoCD, y que el servidor de repositorios acceda y procese los manifiestos de Git. En conjunto, esta configuración RBAC posibilita que ArgoCD automatice los despliegues y mantenga las aplicaciones sincronizadas con su estado deseado definido en Git, operando de manera autónoma y segura.

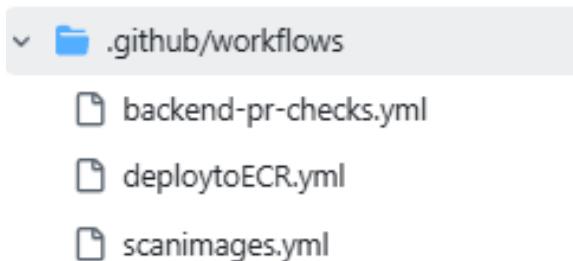
Authentication: ServiceAccounts (8)		<a href="#">View details</a>
Service account provides an identity for processes that run in a Pod. <a href="#">Learn more</a>		
Name	Created	
argocd-application-controller	22 minutes ago	
argocd-applicationset-controller	22 minutes ago	
argocd-dex-server	22 minutes ago	
argocd-notifications-controller	22 minutes ago	
argocd-redis-secret-init	22 minutes ago	
argocd-repo-server	22 minutes ago	
argocd-server	22 minutes ago	
default	22 minutes ago	

Por otro lado se tiene el ServiceAccount para un "secret rotator" es crucial para automatizar la gestión y actualización periódica de credenciales sensibles (como contraseñas o API keys) dentro del clúster Kubernetes, reforzando la seguridad. Al asignarle una identidad dedicada con permisos específicos mediante RBAC —principalmente para leer, actualizar y crear Secrets, y potencialmente para reiniciar pods que los consumen— se asegura que solo este componente pueda realizar estas operaciones críticas, y esto minimiza la exposición de secretos y reduce el riesgo asociado a credenciales obsoletas o comprometidas, todo bajo el principio de menor privilegio.

Authentication: ServiceAccounts (2)		<a href="#">View details</a>
Service account provides an identity for processes that run in a Pod. <a href="#">Learn more</a>		
Name	Created	
default	3 minutes ago	
frontend-secret-rotator-sa	3 minutes ago	

### Escaneo de imágenes en busca de vulnerabilidades:

Para realizar el escaneo de imágenes en busca de posibles vulnerabilidades, se hizo la implementación de Trivy, que es una herramienta que permite escanear vulnerabilidades y malas configuraciones en imágenes, contenedores, repositorios, entre otros. Se realizó una pipeline respectiva en cada repositorio entre ecommerce-microservice-backend-app y entre ecommerce-frontend-web-app, esta pipeline se llamó: scanimages.yml:



### *ecommerce-microservice-backend-app:*

Este workflow de GitHub Actions permite construir y escanear imágenes Docker automáticamente con Trivy cada vez que se abre Pull Request a la rama dev, pero también se puede ejecutar de forma manual, y si se detectan vulnerabilidades críticas o altas, la pipeline falla.

luisapino feat: add changes

Code Blame 93 lines (73 loc) · 3.01 KB

```
1 name: CI - Build and Scan Docker Images on PR to dev
2
3 on:
4   pull_request:
5     types: [opened, synchronize]
6     branches:
7       - dev
8   workflow_dispatch:
9
10 permissions:
11   contents: read
12   security-events: write
13
14 env:
15   TRIVY_VERSION: "0.63.0"
16
17 jobs:
18   build-and-scan:
19     name: Build & Scan Images with Trivy
20     runs-on: ubuntu-latest
21
22     steps:
23       - name: Checkout code
24         uses: actions/checkout@v4
25
26       - name: Set up JDK 17
27         uses: actions/setup-java@v4
28       with:
29         distribution: 'temurin'
30         java-version: '17'
31         cache: 'maven'
32
33       - name: Build application JARs with Maven
34         run: ./mvnw clean package -DskipTests
35
36       - name: Install Trivy
37         run: |
38           VERSION=${TRIVY_VERSION:-"0.63.0"}
39           curl -sfL https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install.sh | sh -s -- -b
40           trivy --version
41
42       - name: Build and Scan Images in a Loop
43         run: |
44           OVERALL_EXIT_CODE=0
45
46           services=( cloud-config service-discovery api-gateway proxy-client order-service payment-service pro
47           mkdir -p trivy-results
48
49           for service_name in "${services[@]}"; do
50             CONTEXT_PATH="./${service_name}"
```

En este caso al ejecutar la pipeline en el repositorio de ecommerce-microservice-backend-app, la pipeline falló, queriendo decir que fueron encontradas vulnerabilidades:

The screenshot shows a GitHub repository interface with the path 'ecommerce-frontend-web-app'. The 'Actions' tab is selected, displaying a list of workflow runs. One run is highlighted with a red error icon, indicating a failure. The run details show it was triggered by a pull request, event type 'CI - Build and Scan Frontend Image on PR', branch 'master', and actor 'Manually run by luisapino'. The run completed 39 minutes ago with a duration of 1m 27s.

En la pipeline, al ver los fallos dice entonces que fueron encontradas vulnerabilidades críticas, por ejemplo en proxy-client, order-service, payment-service, product-service, shipping-service, user-service y entre otras vulnerabilidades que fueron encontradas:

```
267 2025-06-11T20:50:04Z  WARN  Using severities from other vendors for some vulnerabilities. Read
https://trivy.dev/v0.63/docs/scanner/vulnerability#severity-selection for details.
268 Error: Trivy found critical vulnerabilities in api-gateway. The build will be marked as failed.
269 -----
270 Processing service: proxy-client
271 -----  

280 Processing service: proxy-client
281 -----  

282 2025-06-11T20:50:12Z  WARN  Using severities from other vendors for some vulnerabilities. Read
https://trivy.dev/v0.63/docs/scanner/vulnerability#severity-selection for details.
283 Error: Trivy found critical vulnerabilities in proxy-client. The build will be marked as failed.
284 -----
285 Processing service: order-service  

295 2025-06-11T20:50:20Z  WARN  Using severities from other vendors for some vulnerabilities. Read
https://trivy.dev/v0.63/docs/scanner/vulnerability#severity-selection for details.
296 Error: Trivy found critical vulnerabilities in order-service. The build will be marked as failed.
297 -----
298 Processing service: payment-service  

308 2025-06-11T20:50:29Z  WARN  Using severities from other vendors for some vulnerabilities. Read
https://trivy.dev/v0.63/docs/scanner/vulnerability#severity-selection for details.
309 Error: Trivy found critical vulnerabilities in payment-service. The build will be marked as
failed.
310 -----
311 Processing service: product-service  

321 2025-06-11T20:50:37Z  WARN  Using severities from other vendors for some vulnerabilities. Read
https://trivy.dev/v0.63/docs/scanner/vulnerability#severity-selection for details.
322 Error: Trivy found critical vulnerabilities in product-service. The build will be marked as
failed.
323 -----
324 Processing service: shipping-service
```

```

https://trivy.dev/v0.63/docs/scanner/vulnerability#severity-selection for details.

Error: Trivy found critical vulnerabilities in shipping-service. The build will be marked as failed.

-----
Processing service: user-service
-----
```

En github en la sección de Security salen entonces las vulnerabilidades encontradas, en este caso fueron encontradas 2,786:

The screenshot shows the GitHub repository interface for the 'ecommerce-microservice-backend-app'. The 'Security' tab is selected, displaying 2,786 vulnerabilities. A prominent message states: 'Trivy is reporting errors. Check the [status page](#) for help.' Below this, a search bar shows the query 'is:open branch:master'. The main list of vulnerabilities includes the following entries:

- Template injection in thymeleaf-spring5** (Critical) Library master
  - #2782 opened 1 hour ago • Detected by Trivy in home/.../lib/thymeleaf-spring5-3.0.12... :1
- Template injection in thymeleaf-spring5** (Critical) Library master
  - #2781 opened 1 hour ago • Detected by Trivy in home/.../lib/thymeleaf-spring5-3.0.12... :1
- spring-framework: RCE via Data Binding on JDK 9+** (Critical) Library master
  - #2776 opened 1 hour ago • Detected by Trivy in home/.../lib/spring-webmvc-5.3.13.jar :1
- spring-framework: RCE via Data Binding on JDK 9+** (Critical) Library master
  - #2775 opened 1 hour ago • Detected by Trivy in home/.../lib/spring-webmvc-5.3.13.jar :1
- spring: HttpInvokerServiceExporter readRemoteInvocation method untrusted java deserialization** (Critical) Library master
  - #2768 opened 1 hour ago • Detected by Trivy in home/.../lib/spring-web-5.3.13.jar :1
- spring: HttpInvokerServiceExporter readRemoteInvocation method untrusted java deserialization** (Critical) Library master
  - #2767 opened 1 hour ago • Detected by Trivy in home/.../lib/spring-web-5.3.13.jar :1
- spring-framework: RCE via Data Binding on JDK 9+** (Critical) Library master
  - #2760 opened 1 hour ago • Detected by Trivy in home/.../lib/spring-beans-5.3.13.jar :1

Si se ingresa a la primera opción sale entonces la siguiente información:

## Template injection in thymeleaf-spring5

[Open](#) in `master` yesterday

home/app/user-service.jar/BOOT-INF/lib/thymeleaf-spring5-3.0.12.RELEASE.jar:1

**Preview unavailable**  
Sorry, we couldn't find this file in the repository.

Package: org.thymeleaf:thymeleaf-spring5  
Installed Version: 3.0.12.RELEASE  
Vulnerability CVE-2021-43466  
Severity: CRITICAL  
Fixed Version: 3.0.13.RELEASE  
Link: [CVE-2021-43466](#)

Trivy

Tool	Rule ID
Trivy	CVE-2021-43466

Vulnerability CVE-2021-43466

Severity	Package	Fixed Version	Link
CRITICAL	org.thymeleaf:thymeleaf-spring5	3.0.13.RELEASE	<a href="#">CVE-2021-43466</a>

In the thymeleaf-spring5:3.0.12 component, thymeleaf combined with specific scenarios in template injection may lead to remote code execution.

Show less ^

Es una vulnerabilidad de severidad crítica en el componente thymeleaf-spring5 en la versión 3.0.12.RELEASE, utilizada en el módulo user-service.

Hay un link en la sección de Trivy CVE-2021-43466:

 **aqua vulnerability database** ≡

**Vulnerabilities** Misconfiguration Runtime Security Compliance

 **CVE Vulnerabilities**

## CVE-2021-43466

Improper Control of Generation of Code ('Code Injection')

Published: Nov 09, 2021 | Modified: Nov 21, 2024

In the thymeleaf-spring5:3.0.12 component, thymeleaf combined with specific scenarios in template injection may lead to remote code execution.

**Weakness** 

The product constructs all or part of a code segment using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the syntax or behavior of the intended code segment.

**Affected Software** 

Name	Vendor	Start Version	End Version
Thymeleaf	Thymeleaf	3.0.12	3.0.12

**CVSS 3.x**  **9.8** **CRITICAL** Source: NVD

CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

**CVSS 2.x**  6.8 MEDIUM

RedHat/V2 

RedHat/V3 

Ubuntu 

**Additional information**

NVD <https://nvd.nist.gov/vuln/detail/CVE-2021-43466>

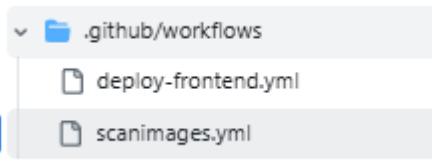
Esto nos indica que el problema se presenta cuando Thymeleaf, en combinación con ciertos escenarios de template injection, permite que un atacante pueda ejecutar código remoto (RCE - Remote Code Execution). Además nos indica que en nivel de criticidad se encuentra en un 9.8, es por ello que es relevante y grave.

Algunas recomendaciones en este caso serían:

- Actualizar inmediatamente a la versión segura.
- Validación estricta de entradas.
- Usar mecanismos de aislamiento.
- Evitar la construcción dinámica del código basada en entrada del usuario.
- Evitar las validaciones solo basadas en listas negras.

#### *ecommerce-frontend-web-app:*

En el repositorio de ecommerce frontend también se implementó una pipeline con Trivy para poder identificar vulnerabilidades:



Al igual que la primera mencionada, también se puede ejecutar en el momento que se realiza un pull request para poder analizar las imágenes, pero también se puede ejecutar de manera manual, solo construye las imágenes y las analiza, no ejecuta nada más, pero si falla la pipeline si se encuentran vulnerabilidades:

A screenshot of a GitHub code editor displaying the content of a YAML configuration file named 'ci-frontend-scan.yml'. The code defines a GitHub Actions workflow for scanning Docker images with Trivy. It includes sections for 'name', 'on', 'permissions', 'env', and 'jobs'. The 'jobs' section contains a single job named 'build-and-scan-frontend' which runs on 'ubuntu-latest' and performs steps to checkout code, install Trivy, build the image, and run Trivy against it. The code also includes comments and logic to verify the Dockerfile exists before proceeding.

```
1 # Nombre del archivo: .github/workflows/ci-frontend-scan.yml
2
3 name: CI - Build and Scan Frontend Image on PR to dev
4
5 on:
6   pull_request:
7     types: [opened, synchronize]
8     branches:
9       - dev
10    workflow_dispatch:
11
12 permissions:
13   contents: read
14   security-events: write
15
16 env:
17   TRIVY_VERSION: "0.63.0"
18
19 APP_NAME: "frontend-app"
20
21 jobs:
22   build-and-scan-frontend:
23     name: Build & Scan Frontend Image
24     runs-on: ubuntu-latest
25
26     steps:
27       - name: Checkout code
28         uses: actions/checkout@v4
29
30       # --- No necesitamos Java/Maven para el frontend, así que esos pasos se eliminan ---
31
32       - name: Install Trivy
33         run: |
34           VERSION=${TRIVY_VERSION}
35           curl -sL https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install.sh | sh -s -- -i
36           trivy --version
37
38       - name: Build and Scan Image
39         run: |
40           mkdir -p trivy-results
41
42           CONTEXT_PATH=.
43           DOCKERFILE_PATH=./Dockerfile
44
45           # Verificamos si el Dockerfile existe antes de continuar.
46           if [ ! -f "$DOCKERFILE_PATH" ]; then
47             echo "::error::Dockerfile not found at $DOCKERFILE_PATH. Cannot build image."
48             exit 1
```

En la sección de security fueron detectadas 102:

The screenshot shows the GitHub Code scanning interface for the repository 'microservices-project-k8s-jenkins / ecommerce-fr...'. The sidebar on the left lists various sections: Overview, Reporting, Policy, Advisories, Vulnerability alerts, Dependabot, Code scanning (which is selected and has 102 items), and Secret scanning. The main area is titled 'Code scanning' and displays a list of 102 open vulnerabilities. A message at the top says 'Trivy is reporting errors. Check the [status page](#) for help.' There is a search bar with the query 'is:open branch:master'. The results are filtered by tool (Trivy) and language (all). The first few vulnerabilities listed are:

- golang: net/netip: Unexpected behavior from Is methods for IPv4-mapped IPv6 addresses (Critical, Library, master)
- golang: html/template: improper handling of JavaScript whitespace (Critical, Library, master)
- golang: html/template: backticks not treated as string delimiters (Critical, Library, master)
- golang: crypto/elliptic: IsOnCurve returns true for invalid field elements (Critical, Library, master)
- webpack: avoid cross-realm objects (Critical, Library, master)
- Insufficient validation when decoding a Socket.IO packet (Critical, Library, master)
- minimist: prototype pollution (Critical, Library, master)
- loader-utils: prototype pollution in function (Critical, master)

Si se selecciona la primera opción nos indica lo siguiente, es una vulnerabilidad CVE-2024-24790 en la librería estándar de Go (stdlib):

golang: net/netip: Unexpected behavior from Is methods for IPv4-mapped IPv6 addresses

Dismiss alert ▾

Open in master yesterday

home/app/node\_modules/esbuild-linux-64/bin/esbuild:1 Library

Preview unavailable  
Sorry, we couldn't find this file in the repository.

Package: stdlib  
Installed Version: v1.17.2  
Vulnerability CVE-2024-24790  
Severity: CRITICAL  
Fixed Version: 1.21.11, 1.22.4  
Link: CVE-2024-24790

Trivy

Tool Rule ID  
Trivy CVE-2024-24790

Vulnerability CVE-2024-24790

Severity	Package	Fixed Version	Link
CRITICAL	stdlib	1.21.11, 1.22.4	CVE-2024-24790

The various Is methods (IsPrivate, IsLoopback, etc) did not work as expected for IPv4-mapped IPv6 addresses, returning false for addresses which would return true in their traditional IPv4 forms.

Show less ^

Severity Critical

Affected branches master default First detected 1 day ago

Development

Link a branch, pull request, or create a new branch to start working on this alert.

Security campaigns No campaign is currently tracking this alert.

Tags critical security vulnerability

En este caso, CVE-2024-24790 afecta al paquete net/netip de Go en las versiones 1.21.11 y 1.22.4. El problema específico es con los métodos Is (como IsPrivate, IsLoopback, etc.) que no funcionan correctamente con direcciones IPv6 mapeadas a IPv4. Cuenta con una severidad crítica.

aqua vulnerability database

Vulnerabilities Misconfiguration Runtime Security Compliance

 CVE Vulnerabilities

# CVE-2024-24790

Published: Jun 05, 2024 | Modified: Nov 21, 2024

The various Is methods (IsPrivate, IsLoopback, etc) did not work as expected for IPv4-mapped IPv6 addresses, returning false for addresses which would return true in their traditional IPv4 forms.

**Affected Software**

Name	Vendor	Start Version	CVSS 3.x	CVSS 2.x	RedHat/V2	RedHat/V3	Ubuntu	Additional information	
Go	Golang	*	9.8 CRITICAL	CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H	CVSS 2.x	RedHat/V2	RedHat/V3	NVD	Source: NVD
Go	Golang	1.22.0 (including)	MEDIUM	CVSS 2.x	RedHat/V2	RedHat/V3	Ubuntu	(excluding)	NVD
Cost Management for RHEL 8	RedHat	costmanagement/costmanagement-operator-bundle:3.3.1-1	MEDIUM	CVSS 2.x	RedHat/V2	RedHat/V3	Ubuntu	(excluding)	NVD
Cost	RedHat	costmanagement/costmanagement-operator-bundle:3.3.1-1	MEDIUM	CVSS 2.x	RedHat/V2	RedHat/V3	Ubuntu	(excluding)	NVD

CVSS 3.0: 9.8 CRITICAL (Source: NVD)  
CVSS 2.0: 6.7 MODERATE (Source: NVD)  
CWE: https://cwe.mitre.org/data/definitions/.html

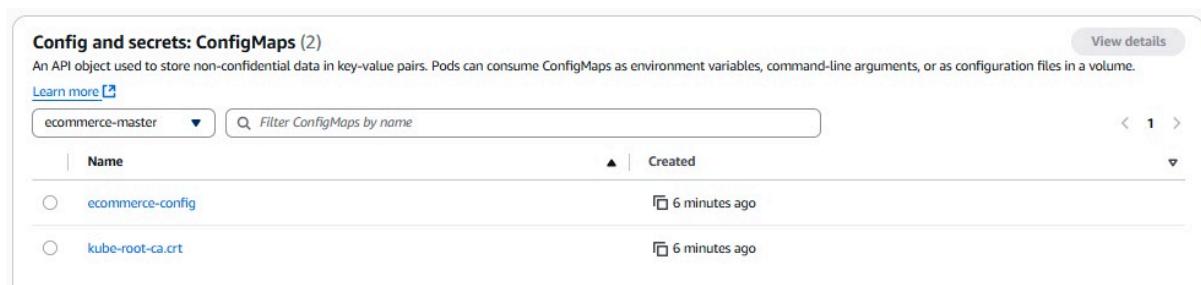
Algunas recomendaciones serían:

- Actualizar Go a versión parcheada más reciente.
- Revisar código que use métodos IsPrivate, IsLoopback en net/netip.
- Reconstruir imagen Docker con versión actualizada.

## **Gestión de configuración y secretos**

### **Migración de todas las configuraciones de Spring Boot a ConfigMaps:**

Con el objetivo de centralizar, simplificar y facilitar el mantenimiento de la configuración de los microservicios desarrollados con Spring Boot, se realizó la migración de todas las variables de entorno necesarias desde los values.yaml individuales hacia un único ConfigMap reutilizable: ecommerce-config.



Name	Created
ecommerce-config	6 minutes ago
kube-root-ca.crt	6 minutes ago

Esta migración permite desacoplar la lógica de configuración de cada microservicio y reutilizar valores comunes en múltiples componentes, evitando duplicación innecesaria de código y mejorando la consistencia en entornos Kubernetes.

#### **• Cambios realizados:**

Anteriormente, cada microservicio Spring Boot (por ejemplo: user-service, payment-service, api-gateway, etc.) definía sus propias variables de entorno directamente en la sección env del Helm chart. Estas incluían configuraciones repetitivas como la URL de Zipkin, la zona de disponibilidad de Eureka, el perfil activo de Spring, etc.

Para reducir esta redundancia, se identificaron variables comunes y se unificaron dentro del archivo ConfigMap ecommerce-config. Las plantillas Helm (Helm templates) fueron actualizadas para que estas variables sean leídas desde el ConfigMap en tiempo de despliegue.

## **Servicios que utilizan el ConfigMap**

El ConfigMap es consumido por los siguientes microservicios:

- cloud-config
- service-discovery
- api-gateway
- proxy-client
- order-service

- payment-service
- product-service
- shipping-service
- user-service
- favourite-service

Cada uno de ellos ahora lee variables como SPRING\_PROFILES\_ACTIVE, SPRING\_CONFIG\_IMPORT, y configuraciones de EUREKA\_\* desde el ConfigMap unificado, evitando duplicación.

- **Beneficios esperados:**

La migración de las configuraciones a un ConfigMap centralizado ofrece múltiples beneficios, entre ellos una reutilización eficiente, ya que varios servicios pueden acceder a las mismas variables desde una única fuente, lo que reduce la duplicación. Además, mejora la mantenibilidad, permitiendo modificar un valor una sola vez en el ConfigMap para que el cambio se refleje en todos los servicios relacionados. También aporta claridad, al separar claramente la lógica de despliegue de la configuración de los servicios. Finalmente, incrementa la portabilidad, ya que esta estructura facilita la integración con herramientas como ArgoCD para una gestión avanzada de entornos.

#### **Implementación de Secrets para credenciales y datos sensibles**

En la nube fueron subidos tres secretos, cada uno con un propósito específico. Uno de ellos es frontend-secret, utilizado como ejercicio académico para ilustrar la rotación de secretos. Este secreto se muestra constantemente en la pantalla del frontend y sirve como ejemplo práctico para demostrar cómo puede cambiarse dinámicamente gracias a otros componentes del sistema.

Name	Created
api-gateway-tls-9b27j	6 minutes ago
frontend-secret	6 minutes ago
frontend-tls-hsr7q	6 minutes ago

Los otros dos secretos están relacionados con los certificados TLS. Estos certificados son fundamentales ya que la aplicación ha sido expuesta al público mediante GoDaddy usando HTTPS, lo que proporciona una conexión segura para los usuarios. Al proteger las

comunicaciones con TLS, se garantiza que los datos transmitidos estén cifrados y autenticados, cumpliendo con buenas prácticas de seguridad en entornos productivos.

### **Implementación de rotación de secretos**

Para probar de forma académica la rotación de secretos, se utilizó el secreto frontend-secret, que se muestra en el frontend. A través de un CronJob, un Job y un ServiceAccount, este secreto se rota automáticamente cada cierto tiempo.

El objetivo es automatizar la ejecución periódica de la rotación de secretos según un horario definido, asegurando que las credenciales se actualicen consistentemente sin intervención manual. Esto mejora la higiene de seguridad y reduce el riesgo de exposición prolongada de información sensible.

El CronJob no realiza directamente la rotación, sino que en cada intervalo programado instancia un Job que ejecuta la lógica de rotación. Este Job, por ser una tarea finita y orientada a lotes, ejecuta un Pod que:

- Obtiene un nuevo valor para el secreto.
- Actualiza el objeto Secret en Kubernetes.
- Reinicia los Pods afectados (en este caso, el frontend), ya que estos cargan el valor del secreto en el momento de su creación y no lo actualizan dinámicamente.

The screenshot shows a list of CronJobs in the Kubernetes interface. The title is "Workloads: CronJobs (3)". It includes a "View details" button, a dropdown for "All Namespaces", a search bar, and a page number indicator "1". The table has columns for "Name" and "Created". Three entries are listed, all created 4 minutes ago:

Name	Created
frontend-secret-rotator-cronjob	4 minutes ago
frontend-secret-rotator-cronjob	4 minutes ago
frontend-secret-rotator-cronjob	4 minutes ago

Workloads: Jobs (3)							<a href="#">View details</a>
All Namespaces		Filter Jobs by name			< 1 >		
	Name	Namespace	Type	Created	Pod count	Status	
○	frontend-secret-rotator-cronjob-29163175	ecommerce-dev	jobs	a minute ago	1	<div style="width: 100%;">1 Ready   0 Failed   1 Completed</div>	
○	frontend-secret-rotator-cronjob-29163175	ecommerce-master	jobs	a minute ago	1	<div style="width: 100%;">1 Ready   0 Failed   1 Completed</div>	
○	frontend-secret-rotator-cronjob-29163175	ecommerce-stage	jobs	a minute ago	1	<div style="width: 100%;">1 Ready   0 Failed   1 Completed</div>	

Gracias a la resiliencia de Kubernetes, este reinicio es imperceptible para los usuarios y se gestiona de forma confiable. Además, Kubernetes garantiza que el Job se ejecute hasta completarse, marcando su estado como exitoso o fallido según corresponda, lo cual permite una supervisión y auditoría clara del proceso de rotación.

### Uso de variables de entorno y/o volúmenes para inyectar configuraciones

```
env:
- name: SPRING_PROFILES_ACTIVE
  valueFrom:
    configMapKeyRef:
      name: ecommerce-config
      key: SPRING_PROFILES_ACTIVE
- name: SPRING_ZIPKIN_BASE_URL
  valueFrom:
    configMapKeyRef:
      name: ecommerce-config
      key: SPRING_ZIPKIN_BASE_URL
- name: EUREKA_CLIENT_REGION
  valueFrom:
    configMapKeyRef:
      name: ecommerce-config
      key: EUREKA_CLIENT_REGION
```

En la imagen se muestra cómo, mediante variables de entorno definidas en un manifiesto de Kubernetes, se inyectan configuraciones dinámicas desde un ConfigMap hacia los contenedores de la aplicación. Esta práctica permite que parámetros como rutas de servicios o configuraciones específicas se gestionen externamente, garantizando que la aplicación use valores actualizados sin necesidad de modificar la imagen del contenedor.

### **Gestión centralizada de la configuración (aprovechando el servicio de Cloud Config)**

En este esquema, toda la configuración de Spring (perfiles activos, rutas de Zipkin, propiedades de Eureka, importación de Cloud Config, nombres de host, etc.) queda centralizada en un único ConfigMap (ecommerce-config), que es referenciado por el despliegue del propio servicio de Cloud Config y por cada microservicio (por ejemplo payment-service). Mediante env.valueFrom.configMapKeyRef en los manifiestos de Kubernetes, cada contenedor inyecta dinámicamente sus variables de entorno desde ese ConfigMap común, incluyendo SPRING\_CONFIG\_IMPORT para conectar con el servidor de configuración y descargar sus propiedades específicas en tiempo de ejecución. De este modo se garantiza coherencia entre entornos, se simplifican actualizaciones de parámetros sin necesidad de reconstruir imágenes y se mantiene un único punto de verdad para todos los servicios del ecosistema.

En el configmap se puede ver así:

```
# SPRING_CONFIG_IMPORT de service-discovery, api-gateway, proxy-client, order-service, payment-service
SPRING_CONFIG_IMPORT: {{ index .Values "service-discovery" "env" "SPRING_CONFIG_IMPORT" | quote }}
```

Y en el values.yaml:

```
SPRING_CONFIG_IMPORT: "optional:configserver:http://cloud-config:9296/"
```

### **Estrategias de Despliegue y CI/CD**

#### **Pipeline de CI/CD completo con Jenkins, GitLab CI o GitHub Actions para todos los microservicios**

Para hacer el despliegue automatizado de la aplicación se usaron pipelines desarrolladas en GitHub Actions, teniendo en cuenta la estructura de la organización de GitHub donde hizo el desarrollo, hay que tener en cuenta que hay 4 repositorios a destacar: infrastructure, microservice-backend, ecommerce-backend y ecommerce-chart. En ese orden de ideas estos mismos repositorios tienen cada uno al menos una pipeline para el funcionamiento, y vamos a ver paso a paso cómo funcionan:

- Infrastructure:  
Este repositorio de infraestructura contiene dos workflows principales de GitHub Actions:
  - Workflow de Despliegue de Infraestructura (deploy-infra.yml):
    - Propósito: Crear la infraestructura en AWS según las definiciones de Terraform, también implementando los repos de Helm para que se suban los manifiestos relacionados, asimismo implementando ECK, ArgoCD, etc.
    - Recursos Gestionados: Esto incluye, pero no se limita a:
      - El clúster de Amazon EKS (TF\_EKS\_CLUSTER\_NAME).
      - Namespaces dentro del clúster de Kubernetes.
      - Roles y políticas IAM necesarios para EKS y otras integraciones.

- Configuración de VPC, subredes, grupos de seguridad.
- El repositorio ECR (TF\_ECR\_NAME), si su creación no se delega al pipeline de aplicación.
- ArgoCD.
- ECK.
- Add Helm Repos.
- Etc.

[← Deploy Infrastructure to AWS](#)

## Deploy Infrastructure to AWS #70

[Summary](#)

Manually triggered 4 hours ago by santiagobarrazas · e39c055 · main Status: Success Total duration: 17m 42s Artifacts: 1

Jobs

- ✓ deploy\_infra
- ✓ bootstrap\_cluster

Run details

- ⌚ Usage
- ⌚ Workflow file

deploy-aws.yml  
on: workflow\_dispatch

```

graph LR
    A[deploy_infra] -- "12m 40s" --> B[bootstrap_cluster]
    
```

**deploy\_infra**  
succeeded 5 hours ago in 12m 40s

> ⌚ Set up job  
> ⌚ Checkout repository  
> ⌚ Set up Terraform  
> ⌚ Terraform init  
> ⌚ Terraform validate  
> ⌚ Terraform plan  
> ⌚ Terraform apply  
> ⌚ Terraform destroy  
> ⌚ Export Terraform outputs  
> ⌚ Upload Terraform Outputs Artifact  
> ⌚ Post Checkout repository  
> ⌚ Complete job

```

bootstrap_cluster
succeeded 5 hours ago in 4m 56s

> ✓ Set up job
> ✓ Checkout repository
> ✓ Download Terraform Outputs
> ✓ Load Terraform Outputs into Env Vars
> ✓ Configure AWS credentials
> ✓ Configure kubectl for EKS
> ✓ Install Helm
> ✓ Add Helm Repos
> ✓ Wait for cluster to be ready
> ✓ Install Metrics Server
> ✓ Install NGINX Ingress Controller
> ✓ Install Cert Manager
> ✓ Install Calico Network Policies
> ✓ Install ECK Operator
> ✓ Wait for ECK CRDs
> ✓ Install ECK Stack with Logstash
> ✓ Install Prometheus and Grafana
> ✓ Create Alert Rules as PrometheusRule
> ✓ Setup ArgoCD Bootstrap
> ✓ Verify Installations
> ✓ Post Configure AWS credentials
> ✓ Post Checkout repository
> ✓ Complete job

```

- Workflow de Destrucción de Infraestructura (destroy-infra.yml):
  - Propósito: Eliminar de forma controlada todos los recursos creados por Terraform.

← Destroy AWS Infrastructure

**✓ Destroy AWS Infrastructure #33**

Summary	Manually triggered 1 hour ago	Status	Total duration	Artifacts
santiagobarrazas → e39c055 main	Success	11m 56s	-	

**destroy-aws.yml**  
on: workflow\_dispatch

**destroy** 11m 53s

```

> ✓ Set up job
> ✓ Checkout repository
> ✓ Set up Terraform
> ✓ Terraform Init
> ✓ Export Terraform Outputs
> ✓ Check if EKS Cluster Exists
> ✓ Graceful Kubernetes Cleanup
> ✓ Terraform Destroy
> ✓ Post Checkout repository
> ✓ Complete job

```

Una vez ya está la infraestructura arriba, la parte principal de nuestro proceso de despliegue de la aplicación reside en dos repositorios Git principales, el primero es ecommerce-microservice-backend-app, que alberga el código fuente de todos los microservicios Java, junto con sus respectivos Dockerfiles, este repositorio también contiene el workflow principal de GitHub Actions que orquesta la construcción y el registro de las imágenes. El segundo repositorio, ecommerce-chart, el cual está dedicado a nuestro Helm chart, que define cómo se despliega la aplicación en Kubernetes, este repositorio del chart también incluye un workflow de GitHub Actions, pero su función es diferente: se encarga de actualizar las referencias de las imágenes dentro del chart cuando nuevas versiones están disponibles.

Deploy on PR Merge and Trigger Chart Update #40

Summary

Jobs

build-push-and-trigger-on-merge

Run details

Usage

Workflow file

build-push-and-trigger-on-merge

succeeded 2 hours ago in 3m 45s

- > Set up job
- > Checkout Microservices Repository
- > Extract Version and Determine Environment
- > Set up AWS credentials
- > Login to Amazon ECR
- > Determine Base ECR Registry URI
- > DEBUG - ECR URIs
- > Set up JDK 17
- > Build with Maven
- > Build, Tag, and Push Docker Images (Per Service ECR Repo)
- > Trigger chart repository update
- > Generate Release Notes from PR
- > Upload Release Notes
- > Deployment Summary
- > Post Set up JDK 17
- > Post Login to Amazon ECR
- > Post Set up AWS credentials
- > Post Checkout Microservices Repository
- > Complete job

Cuando se realizan cambios en el código de los microservicios y se integran en la rama principal, el workflow de GitHub Actions en el repositorio de microservicios se pone en marcha, este proceso compila el código Java, construye las imágenes Docker para cada servicio y las sube a cada una a un repositorio en el ECR de AWS, cada imagen se etiqueta de

forma distintiva, combinando el nombre del servicio y un número de versión único derivado de la ejecución del workflow, además de una etiqueta {servicio}-latest para referencia.

### Repositorios de ECR con las Imágenes de Docker

Una vez que las imágenes están seguras en ECR, el workflow de microservicios no despliega directamente, en su lugar, envía una notificación (un evento repository\_dispatch) al repositorio ecommerce-chart. Este evento contiene toda la información necesaria sobre las nuevas imágenes, incluyendo la URI del registro ECR, el nombre del repositorio ECR y las versiones de los servicios actualizados.

Al recibir esta notificación, el workflow dedicado en el repositorio ecommerce-chart se activa. Su tarea es modificar el archivo charts/e-commerce/values-master.yaml, actualizando las secciones image.repository e image.tag para cada microservicio afectado, de modo que apunten a las nuevas versiones de las imágenes en ECR, estos cambios se commitean y se suben (push) automáticamente a la rama principal del repositorio del chart.

The screenshot shows a dark-themed user interface for a CI/CD pipeline. At the top, there's a header with a back arrow labeled "Update Helm Chart Image Tags" and the text "update-image-tags #28" next to a green checkmark icon. Below the header, there are several navigation links: "Summary", "Jobs", "Run details", "Usage", and "Workflow file". The "Jobs" link is currently selected, as indicated by a green background and a white checkmark icon. On the right side, a detailed view of a single job is displayed. The job is titled "update-chart-values" and has a status of "succeeded 2 hours ago in 25s". A list of steps is shown, each with a green checkmark icon and a brief description: "Set up job", "Checkout Chart Repository", "Install yq and jq", "Update Image Details in Target Values File", "Commit and Push Changes", "Post Checkout Chart Repository", and "Complete job".

## Commit a50ca92

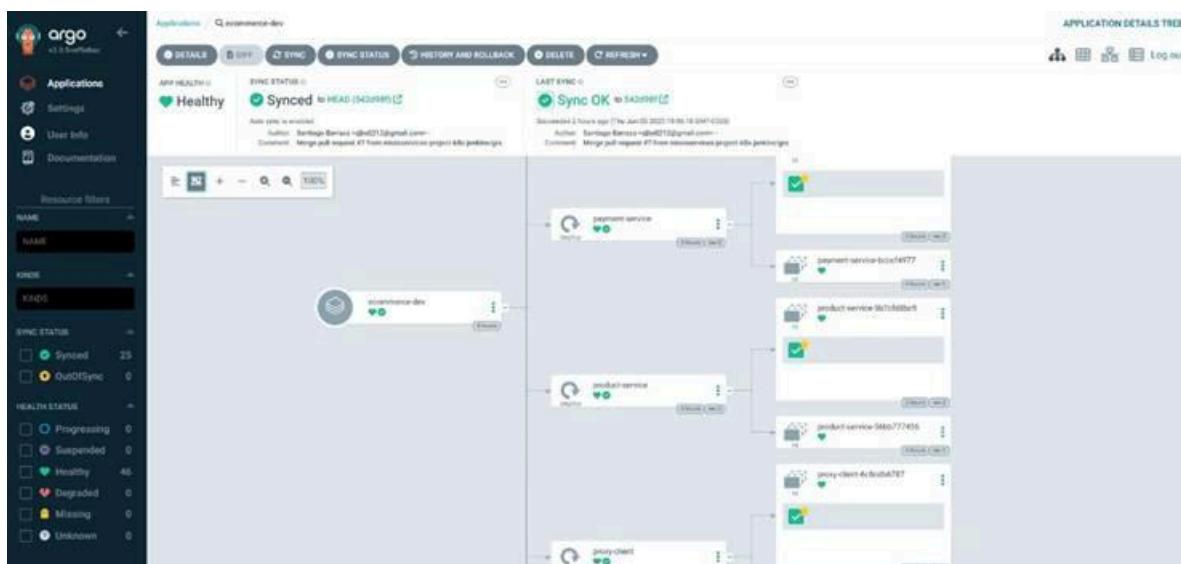
 **github-actions[bot]** committed 2 hours ago

**Update Helm chart (charts/ecommerce/values-stage.yaml) for update-image-tags**

Images to version v1.3.0

Triggered by PR merge to 'stage' in source repo

Aquí es donde ArgoCD entra en juego. Configurado para monitorizar el repositorio ecommerce-chart, ArgoCD detecta los nuevos commits en values-master.yaml. Siguiendo los principios de GitOps, ArgoCD compara el estado deseado (definido en el chart) con el estado actual del clúster de EKS y aplica los cambios necesarios para sincronizarlos. Esto se traduce en que los deployments de Kubernetes se actualizan para utilizar las nuevas imágenes, completando así el ciclo de despliegue de la aplicación.



### Uso de Helm Charts para empaquetar los microservicios

La gestión y el despliegue de la arquitectura de microservicios sobre Kubernetes EKS se realiza mediante el uso de Helm, una potente herramienta de empaquetado que nos ofrece un método estandarizado y versionado para definir, instalar y actualizar nuestras aplicaciones, lo cual es fundamental para mantener la coherencia y reproducibilidad en un entorno complejo como el nuestro.

Helm funciona como un gestor de paquetes específico para Kubernetes, y la unidad fundamental en Helm es el Chart, su adopción nos brinda múltiples beneficios, incluyendo la consistencia en el empaquetado, la capacidad de parametrizar configuraciones a través de plantillas, el versionado de releases para facilitar seguimientos y rollbacks, y una simplificación general del ciclo de vida de las aplicaciones en Kubernetes, haciendo que esto sea especialmente valioso para nosotros, ya que la parametrización es clave para el flujo de CI/CD del proyecto, permitiendo que las imágenes Docker y sus tags se actualicen dinámicamente con cada nueva versión.

Este proyecto utiliza un Chart de Helm principal denominado ecommerce, que reside en el repositorio ecommerce-chart, dentro de la ruta charts/ecommerce/. Este Chart centraliza la definición de despliegue para todos nuestros microservicios, pues su estructura interna contiene varios componentes clave, así que el archivo Chart.yaml provee metadatos esenciales como el nombre y la versión del Chart, mientras que el directorio templates/ alberga los manifiestos de Kubernetes en forma de plantillas Go, donde definimos los Deployments, Services y otros recursos para cada microservicio, haciendo que estos manifiestos sean dinámicos, utilizando marcadores de posición que se rellenan con valores específicos durante el despliegue.

Un componente crucial es el archivo values-master.yaml (hay uno para cada ambiente, e.g: values-dev.yaml). Este archivo externaliza todas las configuraciones específicas de cada microservicio, como la URI completa de la imagen en Amazon ECR, el tag de la imagen (ej. v1.5.6), el número de réplicas, variables de entorno, puertos, y recursos de CPU/memoria. Es este archivo el que se modifica automáticamente por nuestra pipeline de CI/CD para reflejar las nuevas versiones de los microservicios.

```
user-service:
  image:
    repository: 211125706306.dkr.ecr.us-east-1.amazonaws.com/user-service-prod
    tag: v1.3.0
  enabled: true
  replicas: 1
favourite-service:
  image:
    repository: 211125706306.dkr.ecr.us-east-1.amazonaws.com/favourite-service-prod
    tag: v1.3.0
  enabled: true
  replicas: 1
ingress:
  enabled: true
  class: nginx
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-prod
    nginx.ingress.kubernetes.io/rewrite-target: /
domains:
  frontend:
    host: www.lsj-app.xyz
    tlsSecret: frontend-tls
  api:
    host: api.lsj-app.xyz
    tlsSecret: api-gateway-tls
certManager:
  email: luisapino.28@gmail.com
  enabled: true
```

El flujo de despliegue que hemos implementado integra Helm con nuestras herramientas de CI/CD y GitOps es el mismo mencionado en el punto anterior de despliegue, donde se explica paso a paso como hacen sinergia las pipelines con el fin de actualizar de forma automática estos manifiestos de Helm y que el ArgoCD se encargue de sincronizar la nueva información.

Finalmente el mantenimiento y las actualizaciones futuras del sistema siguen este mismo paradigma; cambios de configuración simples, como ajustar el número de réplicas o modificar una variable de entorno, se realizan directamente en los values, también modificaciones estructurales más profundas, como añadir un nuevo microservicio o cambiar la forma en que se realizan las health probes, implican editar las plantillas en el directorio templates/ y, potencialmente, añadir nuevas secciones al archivo values destino. Este enfoque estructurado y automatizado, centrado en Helm, nos permite gestionar eficazmente la complejidad inherente a una arquitectura de microservicios desplegada en Kubernetes.

### Implementación de dependencias correctas en el orden de despliegue

En nuestra arquitectura de microservicios, la correcta inicialización del sistema depende de que ciertos servicios estén disponibles antes que otros, por ejemplo, los servicios de negocio como order-service requieren que el service-discovery (Eureka) y el cloud-config estén operativos, pues para gestionar estas interdependencias, combinamos la resiliencia de nuestros microservicios Spring Boot, que incluyen mecanismos de reintento para conectarse a servicios de configuración y descubrimiento, con las capacidades de sondeo de salud de Kubernetes, también las Readiness Probes son cruciales, ya que impiden que un microservicio reciba tráfico hasta que sus dependencias fundamentales estén listas y él mismo sea completamente funcional, asegurando así que los servicios no intenten operar prematuramente.

Al desplegar nuestra aplicación mediante el Chart de Helm ecommerce, que agrupa todos los microservicios, ArgoCD aplica los manifiestos al clúster EKS. Aunque no hay un orden estricto impuesto por defecto entre Deployments, los servicios de infraestructura más ligeros (cloud-config, service-discovery) tienden a iniciarse más rápido, lo que a menudo satisface las necesidades de los servicios de negocio que arrancan posteriormente. Para un control más explícito, se implementó initContainers, que ejecutan comprobaciones de disponibilidad de servicios críticos antes de permitir el inicio del contenedor principal de la aplicación.

```
initContainers:
- name: wait-for-dependencies
  image: busybox
  command: ['sh', '-c', 'until nc -z zipkin 9411 && nc -z cloud-config 9296 && nc -z service-discovery 8761; do echo waiting for dependencies; sleep 2; done;']
```

Para un control avanzado de la secuencia de despliegue, ArgoCD introduce el concepto de Olas de Sincronización (Sync Waves), estas permiten asignar prioridades a los recursos en nuestras plantillas Helm, instruyendo a ArgoCD para que despliegue y verifique la salud de los componentes en un orden específico, por ejemplo, podríamos asignar a los servicios de

infraestructura una ola con un peso numérico menor para asegurar su disponibilidad antes de que ArgoCD proceda con los microservicios de negocio, la capacidad de ArgoCD para pausar la sincronización si los recursos en olas anteriores no alcanzan un estado saludable complementa esta estrategia, creando un flujo de despliegue más ordenado y robusto.

En la práctica actual, nuestra estrategia se basa en la resiliencia de los servicios y en ReadinessProbes bien definidas, lo cual ha resultado funcional., no obstante, la implementación de Sync Waves de ArgoCD representa una mejora futura valiosa para formalizar y granularizar aún más el orden de despliegue, especialmente si surgen escenarios de dependencia más complejos. Este enfoque combinado nos permite gestionar eficazmente las dependencias inherentes a una arquitectura de microservicios, asegurando un arranque estable y predecible de nuestra plataforma.

## Almacenamiento y Persistencia

### Implementación de Persistent Volumes y Persistent Volume Claims para las bases de datos

Los Persistent Volumes son recursos de almacenamiento a nivel de clúster que proporcionan almacenamiento persistente independiente del ciclo de vida de los pods, actualmente están implementados los siguientes:

Storage: PersistentVolumes (6)	
Persistent Volume is an API object that represents a piece of storage in the cluster. Available as a general, pluggable resource that persists beyond the lifecycle of any individual Pod. <a href="#">Learn more</a>	
Q Filter PersistentVolumes by name	View details
Name	Created
pvc-32bb5a03-0651-4b20-bec1-ff9a8b261da4	29 minutes ago
pvc-6f84c4f2-cd35-499b-9b35-17ba4b835277	30 minutes ago
pvc-860f7ffc-9cfe-4940-80f6-3f4276b8eb77	30 minutes ago
pvc-a87ccff5-232e-4286-b26b-6cd55f65bfed	29 minutes ago
pvc-dc5f8d45-126f-4eb4-8590-0a3b5112f8dd	30 minutes ago
pvc-fe696dab-b1c0-4685-a926-0effe514487b	30 minutes ago

Por su lado los Persistent Volume Claims son solicitudes de almacenamiento realizadas por aplicaciones que se vinculan automáticamente con los Persistent Volumes disponibles, en este proyecto se encuentran implementados para el alert manager, elk y prometheus:

Storage: PersistentVolumeClaims (6)		<a href="#">View details</a>
A request for storage by a user, similar to a Pod. <a href="#">Learn more</a>		
All Namespaces	Filter PersistentVolumeClaims by name	< 1 >
Name		Created
<a href="#">alertmanager-kube-prometheus-stack-alertmanager-db-alertmanager-kube-prometheus-stack-alertmanager-0</a>		30 minutes ago
<a href="#">elasticsearch-data-elasticsearch-es-default-0</a>		31 minutes ago
<a href="#">elasticsearch-data-elasticsearch-es-default-1</a>		31 minutes ago
<a href="#">elasticsearch-data-elasticsearch-es-default-2</a>		31 minutes ago
<a href="#">logstash-data-logstash-ls-beats-ls-0</a>		31 minutes ago
<a href="#">prometheus-kube-prometheus-stack-prometheus-db-prometheus-kube-prometheus-stack-prometheus-0</a>		30 minutes ago

### Bases de datos implementadas

- Elasticsearch Cluster:** Cuenta con una configuración de 3 nodos (elasticsearch-es-default-0/1/2), cada nodo tiene su propio PVC para garantizar la persistencia de datos y almacenamiento distribuido para alta disponibilidad
- Prometheus:** PVC dedicado para la base de datos de métricas, cuenta con almacenamiento persistente para retención de datos históricos.
- AlertManager:** PVC para almacenamiento de configuraciones y estado de alertas, cuenta con persistencia de reglas y notificaciones.
- Logstash:** PVC para almacenamiento temporal y procesamiento de logs, cuenta con un Buffer persistente para garantizar no pérdida de datos

### Ventajas de la Implementación

La implementación de Persistent Volumes y PVCs proporciona múltiples beneficios críticos para el entorno de producción: garantiza la persistencia de datos permitiendo que la información sobreviva a reinicios y recreaciones de pods, ofrece escalabilidad mediante el almacenamiento independiente de cada componente, asegura alta disponibilidad a través de la configuración de Elasticsearch con múltiples nodos y almacenamiento distribuido, y automatiza completamente el proceso mediante el provisioning automático de volúmenes en AWS EBS, eliminando la necesidad de intervención manual en la gestión del almacenamiento.

### Configuración de StorageClass adecuada para el entorno

El clúster utiliza una StorageClass llamada gp2 que está configurada específicamente para el entorno AWS EKS.

The screenshot shows the AWS Storage Classes console. At the top, it says "Storage: StorageClasses (1)". Below that, a note states "Storage Class provides a way for administrators to describe different available storage types." with a "Learn more" link. A search bar labeled "Filter StorageClasses by name" contains the placeholder "gp2". To the right of the search bar are navigation icons: a left arrow, a page number "1", and a right arrow. Below the search bar, there's a header row with columns "Name" and "Created". A single row is listed below, showing "gp2" in the Name column and "37 minutes ago" in the Created column. There are also up and down arrows for sorting.

La StorageClass personalizada llamada gp2 está diseñada para gestionar el aprovisionamiento dinámico de volúmenes persistentes (PersistentVolumes) en el entorno de Kubernetes que se ejecuta sobre Amazon EKS (Elastic Kubernetes Service). Esta clase de almacenamiento está basada en los volúmenes EBS (Elastic Block Store) de tipo General Purpose SSD (gp2) de AWS, los cuales ofrecen un buen equilibrio entre costo y rendimiento para una amplia variedad de cargas de trabajo.

Cuando una aplicación desplegada en el clúster solicita almacenamiento persistente mediante un PersistentVolumeClaim (PVC) que hace referencia a la StorageClass gp2, Kubernetes se encarga automáticamente de aprovisionar un volumen EBS de tipo gp2 y de conectarlo al pod correspondiente. Esto permite que los datos de la aplicación se mantengan disponibles incluso si el pod se reinicia o se mueve a otro nodo.

## Observabilidad y Monitoreo

### Implementación completa de Prometheus + Grafana

Se implementó Grafana + Prometheus + AlertManager mediante helm, esta instalación se realiza en la pipeline que monta la infraestructura, ya que después de terminar de montarla, instala utilidades en el cluster de EKS, como en este caso, el kube-prometheus-stack. A continuación, se pueden ver los Deployments y los StatefulSets correspondientes a cada componente del stack de monitoreo:

kubectl get deployments -n monitoring				
NAME	READY	UP-TO-DATE	AVAILABLE	AGE
kube-prometheus-stack-grafana	1/1	1	1	167m
kube-prometheus-stack-kube-state-metrics	1/1	1	1	167m
kube-prometheus-stack-operator	1/1	1	1	167m

kubectl get statefulsets -n monitoring				
NAME	READY	AGE		
alertmanager-kube-prometheus-stack-alertmanager	1/1	167m		
prometheus-kube-prometheus-stack-prometheus	1/1	167m		

Se puede ver Grafana en funcionamiento con diferentes dashboards:

## Recursos a nivel de cluster

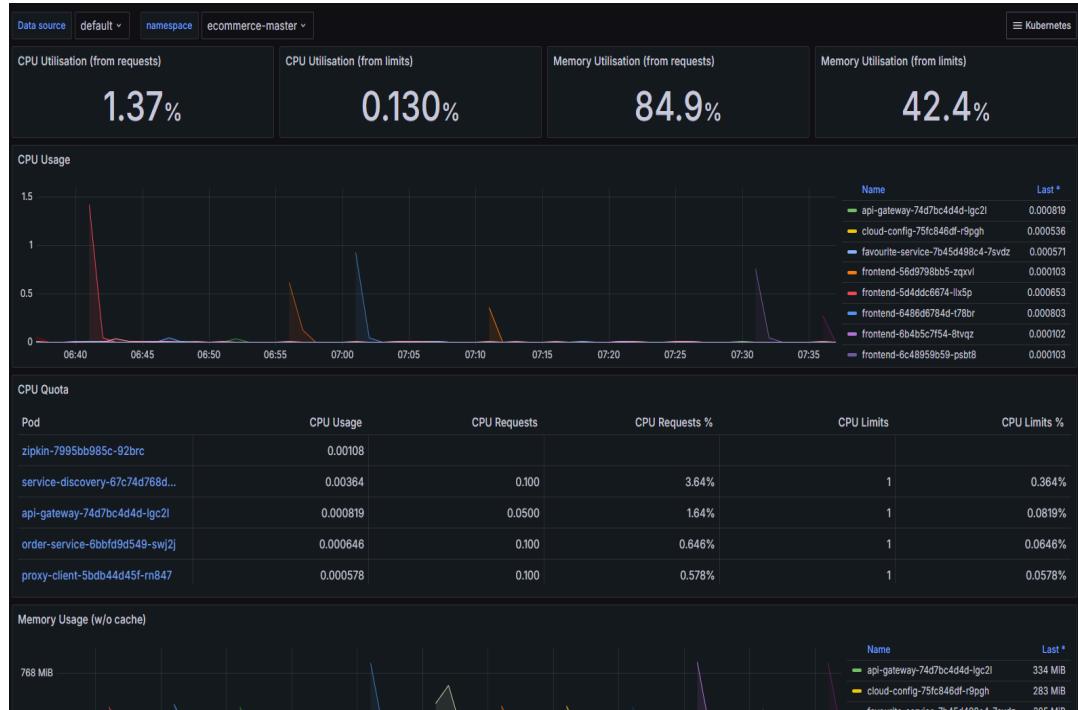


Este panel principal ofrece una vista general del estado de salud y consumo de recursos de todo el clúster de Kubernetes. Los indicadores clave (KPIs) en la parte superior muestran el uso actual de CPU y memoria, junto con el compromiso de recursos (Requests y Limits).

Se observa un uso de CPU bajo (7.06%) pero un uso de memoria considerable (56.7%). Es importante notar que el compromiso de límites de CPU está en 286% y el de memoria en 115%. Esto indica una estrategia de overcommit, donde la suma de los límites de todos los pods excede la capacidad real del clúster, confiando en que no todos los servicios demandarán sus recursos máximos simultáneamente. La tabla inferior desglosa el consumo por namespace, permitiendo identificar rápidamente qué áreas del clúster son las más demandantes.

## Recursos a nivel de namespace:

Este dashboard filtra los recursos para enfocarse exclusivamente en el namespace ecommerce-master, que contiene los microservicios principales de la aplicación.



Se puede apreciar que la utilización de CPU es muy baja en relación a lo solicitado (1.37% from requests), mientras que la utilización de memoria es significativamente más alta (84.9% from requests). Esto sugiere que las solicitudes (requests) de memoria están bien ajustadas a la realidad del consumo, pero las de CPU podrían estar sobredimensionadas para la carga actual. Los gráficos y la tabla de CPU Quota permiten analizar el consumo a nivel de pod individual, como api-gateway y frontend.

## Vista a nivel de network:

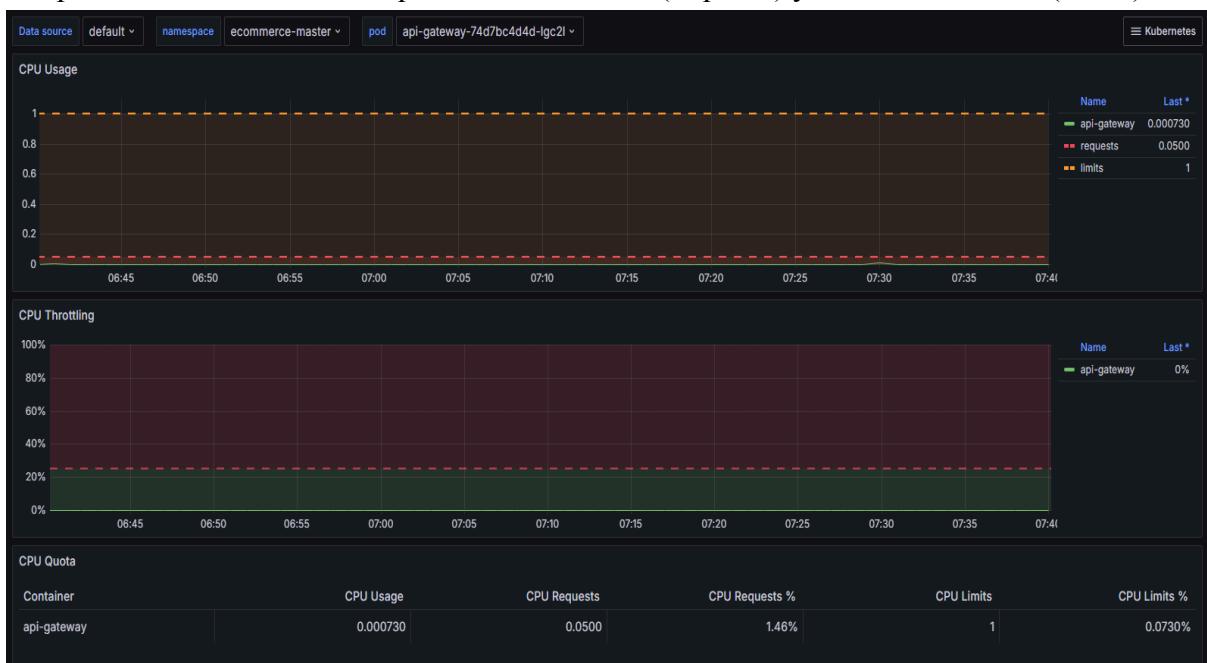
Este panel monitoriza el tráfico de red en todo el clúster, desglosado por namespace. Los gráficos superiores muestran el volumen de bytes recibidos (Current Rate of Bytes Received) y transmitidos (Current Rate of Bytes Transmitted) a lo largo del tiempo.



Los namespaces ecommerce-master, ecommerce-stage y elastic-stack son los que presentan mayor actividad de red. El patrón cíclico y constante en la recepción de bytes sugiere comunicaciones regulares, posiblemente debidas al scraping de métricas de Prometheus o a chequeos de salud (health checks). La tabla Current Status ofrece un resumen cuantitativo del tráfico y los paquetes por segundo para cada namespace.

### *Recursos a nivel de pod:*

Esta vista detalla el comportamiento de un pod específico: api-gateway-74d7bc4d4d-lgc21 dentro del namespace ecommerce-master. El gráfico muestra el uso de CPU del pod en comparación con los recursos que tiene solicitados (requests) y su límite máximo (limits).



Se observa que el uso de CPU es mínimo y estable, muy por debajo de la solicitud de 0.05 cores. El panel CPU Throttling es crucial: al mantenerse en 0%, confirma que el pod nunca ha alcanzado su límite de CPU, por lo que su rendimiento no está siendo estrangulado por falta de recursos. Este es un indicador clave de un pod saludable y con recursos correctamente asignados para su carga de trabajo.

#### Uso de PersistentVolumes por namespace:

Este dashboard monitoriza el almacenamiento persistente, que es vital para las aplicaciones con estado (stateful), en esta imagen se analiza un PersistentVolumeClaim utilizado por elasticsearch.



Los paneles muestran tanto el uso del espacio en disco (Volume Space Usage) como el uso de inodos (Volume Inodes Usage), con un 2.23% de espacio y un 0.941% de inodos utilizados, es evidente que el volumen tiene una capacidad más que suficiente para las necesidades actuales, eliminando cualquier preocupación inmediata sobre el almacenamiento.

#### Monitoreo de los Kubelets:

El Kubelet es el agente primario que se ejecuta en cada nodo del clúster, y en este panel es fundamental para asegurar que los nodos están operando correctamente y comunicándose con el plano de control:



Hay 6 Kubelets en ejecución (es decir, 6 nodos en el clúster), gestionando un total de 96 pods y 213 contenedores, los gráficos de Operation Rate y Operation Error Rate son indicadores clave de salud; una tasa de error de cero, como la que se muestra, indica que los Kubelets están ejecutando sus operaciones sin problemas.

### Recursos a nivel de nodo

A diferencia de la vista de clúster, este dashboard se enfoca en los recursos de un único nodo trabajador, por esto esta vista permite diagnosticar problemas de rendimiento a nivel de máquina individual:



Se observa que el uso de CPU del nodo es bajo y su Load Average se mantiene muy por debajo del número de núcleos lógicos, lo cual es óptimo, además el uso de memoria es estable en torno al 54.3%. El uso de disco en la partición raíz (/) es del 44.3%, indicando que hay espacio disponible, en general, este nodo se encuentra saludable y sin saturación de recursos.

## Datasources

Esta imagen muestra la pantalla de configuración de las fuentes de datos en Grafana:

The screenshot shows the Grafana 'Data sources' configuration page. At the top right is a blue button labeled '+ Add new data source'. Below it is a search bar and a dropdown menu set to 'Sort by A-Z'. There are two data source entries: 'Alertmanager' (Alertmanager | http://kube-prometheus-stack-alertmanager.monitoring:9093) and 'Prometheus' (Prometheus | http://kube-prometheus-stack-prometheus.monitoring:9090 | default). Each entry has a 'Build a dashboard' and an 'Explore' button.

## Aprovechamiento de los Actuator endpoints de Spring Boot para monitoreo

Durante el proceso de despliegue del microservicio user-service en la nube utilizando GitOps, se aprovechó el endpoint /actuator/health proporcionado por Spring Boot Actuator para implementar mecanismos de monitoreo y verificación de estado mediante las sondas liveness y readiness de Kubernetes, esta sonda liveness permite detectar si la aplicación ha dejado de funcionar correctamente, reiniciando el contenedor automáticamente en caso de fallos críticos, y por el otro lado, la sonda readiness evalúa si el servicio está listo para recibir tráfico, evitando que reciba peticiones antes de estar completamente inicializado.

Esto asegura que el sistema sea más robusto, minimiza el tiempo fuera de servicio ante fallos, y permite a la plataforma de orquestación mantener la disponibilidad del microservicio de forma autónoma.

```
livenessProbe:
  httpGet:
    path: /user-service/actuator/health
    port: {{ index .Values "user-service" "port" }}
  initialDelaySeconds: 30
  periodSeconds: 10
  timeoutSeconds: 5
  failureThreshold: 3

readinessProbe:
  httpGet:
    path: /user-service/actuator/health
    port: {{ index .Values "user-service" "port" }}
  initialDelaySeconds: 10
  periodSeconds: 5
  timeoutSeconds: 5
  failureThreshold: 3
```

Y como resultado, el despliegue del servicio fue exitoso, como se refleja en el estado “1 Ready | 0 Failed” mostrado en el dashboard de Kubernetes, indicando que el pod está corriendo correctamente y en condiciones óptimas.



## Configuración de alertas para situaciones críticas

Para la implementación de alertas, esta se hizo por medio de Prometheus, con ayuda de la pipeline de infraestructura, específicamente en la parte en la cual se instala Prometheus y Grafana, se determina las alertas que se quieren montar y con cuanto se deben activar, etc. Una vez arriba y conectado el Prometheus podemos ver las alarmas que tiene asociadas, en este caso se puede evidenciar una llamada HighCPUUsage, la cual está inactiva.

The screenshot shows the Prometheus Alerts interface. At the top, there are filter checkboxes for 'active (1)', 'Pending (1)', and 'Firing (0)'. A search bar says 'Filter by name or labels'. Below that, a file path is shown: '/etc/prometheus/rules/prometheus-kube-prometheus-stack-prometheus-rules-0/monitoring-custom-alert-rules-3c30f80a-8768-4576-9cb8-a64034f44b8d.yaml > example.rules'. A button on the right says 'pending (1)'. A section titled 'HighCPUUsage (1 active)' is expanded, showing the alert configuration. The configuration details include:

```
name: HighCPUUsage
expr: sum by (pod) (rate(container_cpu_usage_seconds_total[1m])) > 0.8
for: 1m
labels:
  severity: warning
annotations:
  description: Pod {{ $labels.pod }} is using more than 80% CPU.
  summary: High CPU usage detected on pod
```

Below this, a table lists the alert's state and creation details:

Labels	State	Active Since	Value
alarmname=HighCPUUsage, pod=stress-cpu, severity=warning	PENDING	2025-06-12T21:00:36.392631288Z	0.9936753872296604

Para probar su funcionamiento se aplican y ejecutan pruebas de estrés sobre la aplicación para llevar los nodos al límite y así la CPU de los pods también pase la barrera de la alerta, la cual se activa al 80% de la CPU.

```
sjbarraza [~] ➜ Azure for Students
) vim stress-cpu.yaml

sjbarraza [~] ➜ Azure for Students
) kubectl apply -f stress-cpu.yaml
pod/stress-cpu created

sjbarraza [~] ➜ Azure for Students
) kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
stress-cpu  1/1     Running   0          6s

sjbarraza [~] ➜ Azure for Students
) kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
stress-cpu  1/1     Running   0          19s
```

```

apiVersion: v1
kind: Pod
metadata:
  name: stress-cpu
spec:
  containers:
  - name: stress
    image: program/stress
    args: [--cpu, "1"]
    resources:
      limits:
        cpu: "1"
      requests:
        cpu: "0.5"

```

Y luego podemos ver como al pasar el tiempo de ejecución la alerta se inicia automáticamente para que se pueda entrar en acción de ser necesario o seguir monitoreando su comportamiento.

The screenshot shows the Prometheus Alertmanager interface with the following details:

- Alert Status:** Active (141), Pending (0), Firing (5).
- Filter:** Filter by name or labels.
- Rule:** /etc/prometheus/rules/prometheus-kube-prometheus-stack-prometheus-rulefiles-0/monitoring-custom-alert-rules-3c30f80a-8768-4576-9cb8-a64034f44b8d.yaml > example.rules
- Alert Description:** HighCPUUsage (1 active)
- Alert Configuration:**

```

name: HighCPUUsage
expr: sum by (pod) (rate(container_cpu_usage_seconds_total{image!=""}[1m])) > 0.8
for: 1m
labels:
  severity: warning
annotations:
  description: Pod {{ $labels.pod }} is using more than 80% CPU.
  summary: High CPU usage detected on pod

```
- Table:**

Labels	State	Active Since	Value
alarmname=HighCPUUsage, pod=stress-cpu, severity=warning	FIRING	2025-06-12T21:00:36.392631288Z	0.995965179317808

Luego de la prueba pasamos a borrar la prueba de estrés.

```

sjbarraza 🖥 ~ ⚙️ Azure for Students
❯ kubectl delete pod stress-cpu
pod "stress-cpu" deleted

```

Y finalmente al volver los pods a la normalidad se vuelve a apagar la alarma.

The screenshot shows the Prometheus Alertmanager interface with the following details:

- Alert Status:** Inactive (0).
- Rule:** /etc/prometheus/rules/prometheus-kube-prometheus-stack-prometheus-rulefiles-0/monitoring-custom-alert-rules-3c30f80a-8768-4576-9cb8-a64034f44b8d.yaml > example.rules
- Alert Description:** HighCPUUsage (0 active)
- Alert Configuration:**

```

name: HighCPUUsage
expr: sum by (pod) (rate(container_cpu_usage_seconds_total{image!=""}[1m])) > 0.8
for: 1m
labels:
  severity: warning
annotations:
  description: Pod {{ $labels.pod }} is using more than 80% CPU.
  summary: High CPU usage detected on pod

```

## Sistema de logging centralizado

Se implementó el ELK stack, la instalación de éste, al igual que el stack de Prometheus se realizó en la pipeline de la infraestructura usando helm. A continuación, se puede ver los deployments y los statefulsets que componen al stack:

```
› kubectl get deployments -n elastic-stack
```

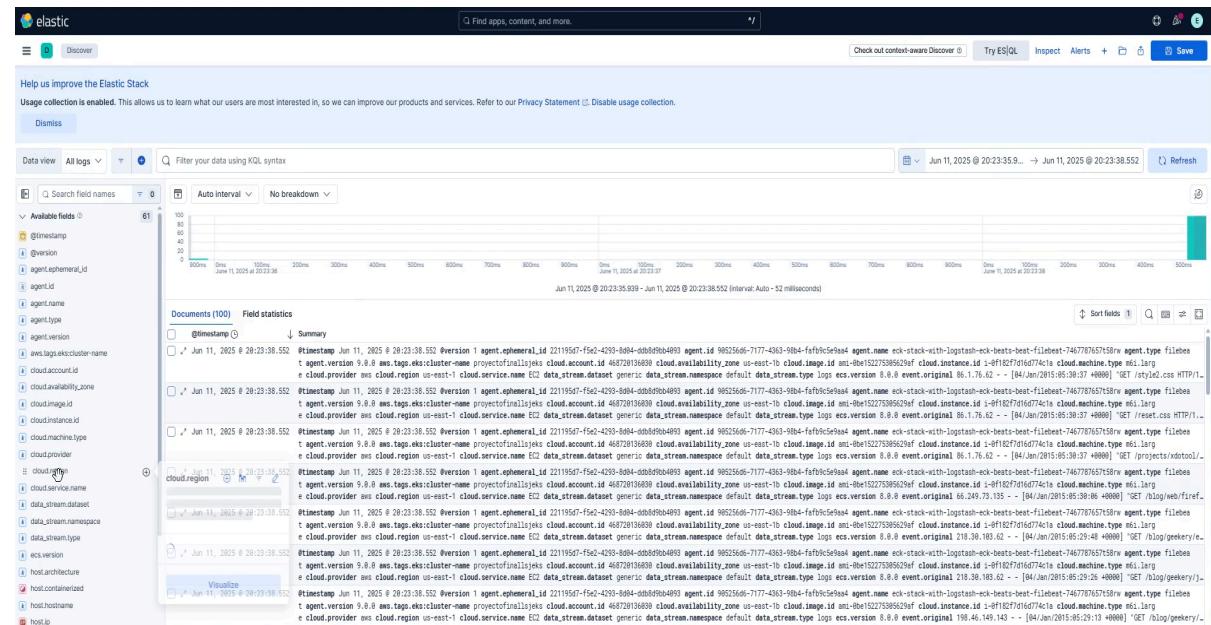
NAME	READY	UP-TO-DATE	AVAILABLE	AGE
eck-stack-with-logstash-eck-beats-beat-filebeat	1/1	1	1	177m
eck-stack-with-logstash-eck-kibana-kb	1/1	1	1	176m

```
> kubectl get statefulsets -n elastic-stack
```

NAME	READY	AGE
elasticsearch-es-default	3/3	177m
logstash-ls-beats-ls	1/1	177m

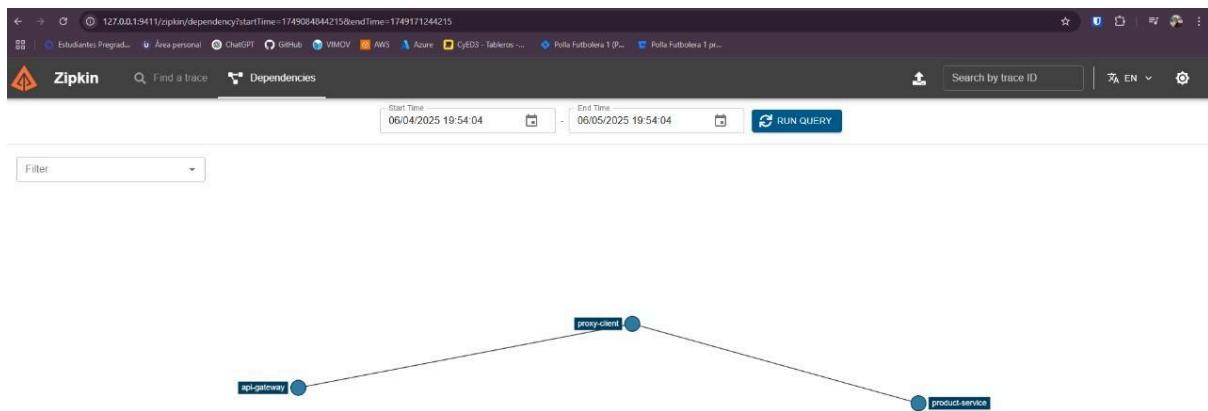
Se puede ver, que el stack está conformado por Filebeat, Logstash, Elasticsearch y Kibana.

A continuación, se puede ver que los logs se ven correctamente en la interfaz gráfica de Kibana:

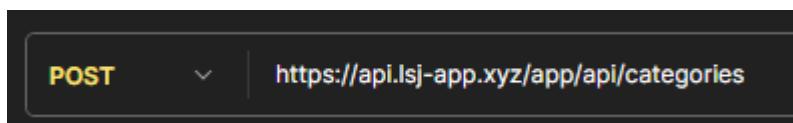


## Tracing con Zipkin

Al hacer una solicitud al api-gateway mediante postman, se creó la traza en Zipkin que se puede ver a continuación:



Esta traza representa que la solicitud llegó al api-gateway desde el frontend (el frontend no envía trazas a zipkin), luego pasó por el proxy-client y finalmente llegó al product-service. La llamada a la API fue:



Por lo que llegó correctamente a la API de productos, ya que el product-service es el que se encarga de gestionar las categorías.

### Autoscaling y Pruebas de Rendimiento

#### Implementación de Horizontal Pod Autoscaler

Se implementaron Horizontal Pod Autoscaler (HPA) para cada microservicio, su definición es la siguiente:

```

apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: user-service
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: user-service
  minReplicas: 1
  maxReplicas: 2
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50

```

Está basado en métricas de CPU, se puede extender a métricas de memoria para controlar también este factor.

### Pruebas de estrés con Locust

Con el objetivo de evaluar el comportamiento del sistema desplegado bajo condiciones de carga, se realizaron pruebas de rendimiento utilizando la herramienta Locust. Estas pruebas se ejecutaron sobre el entorno en la nube accediendo al dominio <https://api.lsj-app.xyz> (el cual fue configurado previamente como parte de la estrategia GitOps), el escenario simulado incluyó múltiples usuarios concurrentes accediendo a diferentes endpoints del sistema, representando interacciones típicas y condiciones de estrés.

The screenshot shows the Locust web interface for starting a new load test. It has a dark background with white text and input fields. The main title is "Start new load test". There are three input fields: "Number of users (peak concurrency)" with value "200", "Spawn rate (users started/second)" with value "5", and "Host (e.g. http://www.example.com)" with value "https://api.lsj-app.xyz". Below these fields are two buttons: "Advanced options" and a large green "Start swarming" button. A cursor arrow is visible at the bottom center, pointing towards the "Start swarming" button.

Se definieron dos tipos principales de usuarios virtuales: ApiGatewayUser, encargado de simular peticiones de uso habitual, y ApiGatewayStressUser, cuyo propósito era generar tráfico de alta frecuencia para evaluar el comportamiento bajo presión, las tareas simuladas incluyeron consultas a los servicios de usuarios, productos, órdenes y favoritos, además de pruebas de manejo de errores y solicitudes rápidas consecutivas.

## Locust Test Report

During: 11/6/2025, 5:22:50 p.m. - 11/6/2025, 5:26:24 p.m.

Target Host: https://api.lsj-app.xyz

Script: locustfile.py

### Request Statistics

Method	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	/favourite-service/api/favourites	6706	0	467	95	7892	1589	31.4	0.0
GET	/order-service/api/orders	7014	0	469	88	9773	524	32.8	0.0
GET	/order-service/api/orders/999999	391	0	378	93	4021	121	1.8	0.0
GET	/product-service/api/products	6954	0	460	89	9771	708	32.6	0.0
GET	/user-service/api/users	7082	0	479	90	7917	3999	33.2	0.0
GET	/user-service/api/users/999999	391	0	635	94	7379	120	1.8	0.0
<b>Aggregated</b>		<b>28538</b>	<b>0</b>	<b>470</b>	<b>88</b>	<b>9773</b>	<b>1670</b>	<b>133.6</b>	<b>0.0</b>

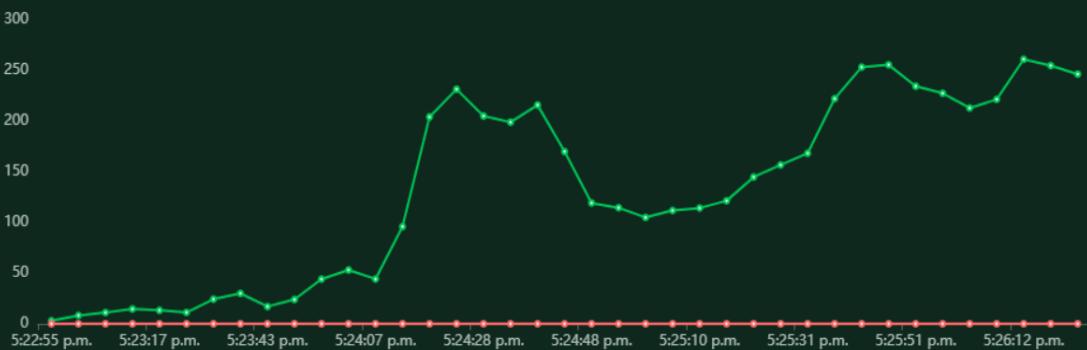
### Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
GET	/favourite-service/api/favourites	240	300	430	640	820	1100	4100	7900
GET	/order-service/api/orders	200	250	400	630	860	1200	4400	9800
GET	/order-service/api/orders/999999	190	230	340	550	860	1000	3000	4000
GET	/product-service/api/products	200	250	390	640	830	1200	4300	9800
GET	/user-service/api/users	220	280	410	640	830	1200	4500	7900
GET	/user-service/api/users/999999	250	360	570	750	1200	2700	5400	7400
<b>Aggregated</b>		<b>210</b>	<b>270</b>	<b>410</b>	<b>640</b>	<b>840</b>	<b>1200</b>	<b>4300</b>	<b>9800</b>

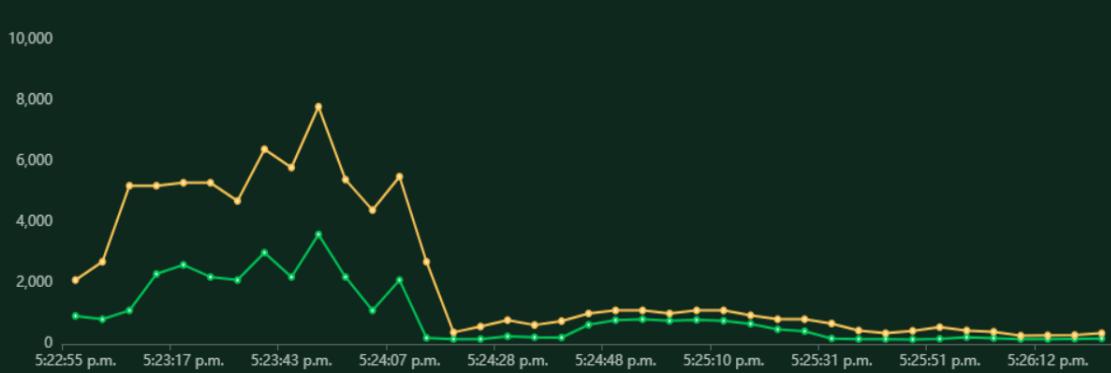
Durante el ensayo, se generaron más de 28,000 solicitudes sin registrar fallos, lo que demuestra una estabilidad notable del sistema. El tiempo promedio de respuesta fue de 470 milisegundos, con mínimos de 88 ms y máximos de 9,773 ms. La plataforma mantuvo un rendimiento sostenido cercano a los 134 requests por segundo, con picos de carga que se mantuvieron estables sin afectar la disponibilidad del sistema. Además, los percentiles de respuesta mostraron que el 95% de las solicitudes fueron respondidas en menos de 1,200 ms, lo que refleja una eficiencia considerable incluso en situaciones de alta demanda.

## Charts

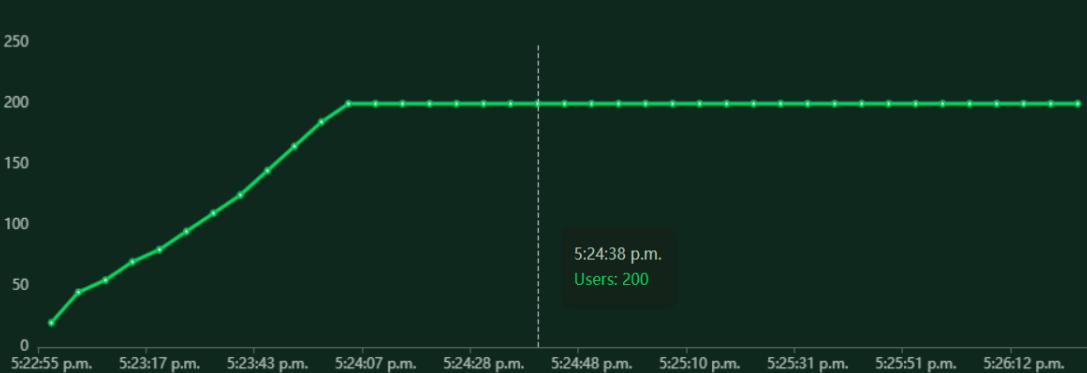
### Total Requests per Second



### Response Times (ms)



### Number of Users



Los gráficos generados durante la prueba respaldan estos resultados: el número de usuarios virtuales creció hasta estabilizarse en 200 sin generar interrupciones ni errores. Los tiempos de respuesta más exigentes ocurrieron en los primeros momentos de la carga, pero se establecieron rápidamente conforme el sistema se adaptó al tráfico simulado.

En conclusión, la prueba de rendimiento demostró que el sistema desplegado es robusto, responde adecuadamente bajo escenarios de uso intensivo y está bien preparado para escalar.

según sea necesario. Estos resultados validan tanto la arquitectura de microservicios implementada, garantizando que la solución es adecuada para entornos productivos exigentes.

A continuación, se muestra como se visualiza este aumento de solicitudes en las métricas en Linkerd, se puede ver como el api gateway y otros servicios como user service, favourite service, product service. Se nota que aumenta la latencia, sin embargo, la tasa de éxito sigue siendo del 100%:

#### Métricas HTTP

Pod ↑	↓ En la malla de servicios	↓ Tasa de éxito	↓ PPS	↓ Latencia P50	↓ Latencia P95	↓ Latencia P99
api-gateway-56bbc59d7c-869w9	1/1	100.00% ●	62.82	18 ms	181 ms	320 ms
cloud-config-69d6cc796-kd2cm	1/1	100.00% ●	0.3	1 ms	2 ms	2 ms
favourite-service-6dbc75f85-98dk7	1/1	100.00% ●	14.93	61 ms	286 ms	376 ms
frontend-847cd698f-gfckr	1/1	100.00% ●	0.3	1 ms	1 ms	1 ms
order-service-68bf4dd8f6-g5b2q	1/1	100.00% ●	19.72	1 ms	6 ms	9 ms
payment-service-cd9b457f-4thw4	1/1	100.00% ●	0.3	1 ms	2 ms	2 ms
product-service-f5777566d-j9srg	1/1	100.00% ●	54.83	6 ms	69 ms	135 ms
proxy-client-6fc74d8596-fs5gl	1/1	100.00% ●	0.3	1 ms	2 ms	2 ms
service-discovery-77f7d484d7-t9mc6	1/1	100.00% ●	0.9	1 ms	3 ms	4 ms
shipping-service-cd9698bbc-r966g	1/1	100.00% ●	0.6	3 ms	17 ms	19 ms
user-service-c66646d8d-ff4sr	1/1	100.00% ●	66.6	4 ms	30 ms	55 ms
zipkin-546bf9547b-fgr6n	1/1	100.00% ●	4.43	1 ms	1 ms	4 ms

También se puede ver directamente las métricas TCP, donde se puede ver el número de conexiones de cada servicio y los bytes de lectura y de escritura:

#### Métricas TCP

Pod ↑	↓ En la malla de servicios	↓ Conexiones	↓ Lectura Bytes / seg	↓ Escritura Bytes / seg
api-gateway-56bbc59d7c-869w9	1/1	70	51.498kB/s	187.73kB/s
cloud-config-69d6cc796-kd2cm	1/1	1	24.45B/s	1.619kB/s
favourite-service-6dbc75f85-98dk7	1/1	9	6.22kB/s	54.352kB/s
frontend-847cd698f-gfckr	1/1	1	24.43B/s	940.28B/s
order-service-68bf4dd8f6-g5b2q	1/1	5	6.602kB/s	21.118kB/s
payment-service-cd9b457f-4thw4	1/1	1	24.23B/s	1.679kB/s
product-service-f5777566d-j9srg	1/1	8	13.41kB/s	46.65kB/s
proxy-client-6fc74d8596-fs5gl	1/1	1	25.03B/s	1.945kB/s
service-discovery-77f7d484d7-t9mc6	1/1	11	91.42B/s	1.572kB/s
shipping-service-cd9698bbc-r966g	1/1	3	65.62B/s	2.64kB/s
user-service-c66646d8d-ff4sr	1/1	7	13.38kB/s	111.986kB/s
zipkin-546bf9547b-fgr6n	1/1	8	20.62kB/s	1.225kB/s

## Pruebas de Caos con Chaos-Mesh

Se instaló Chaos-Mesh para realizar pruebas bajo condiciones adversas en el clúster. Hay una gran variedad de tipos de caos que se pueden aplicar, en este caso, se puso un ejemplo para aumentar las latencias de product-service, como se ve a continuación:

```
sjbarraza [ ~ ] AWS Azure for Students
$ kubectl get schedule -n ecommerce-master
NAME                      AGE
delay-product-service-schedule   3m54s

sjbarraza [ ~ ] AWS Azure for Students
$ kubectl get networkchaos -n ecommerce-master
NAME                           ACTION      DURATION
delay-product-service-schedule-6zxwh   delay      1m
```

Como se puede ver, se creó un objeto en el cluster de tipo shcedule, que al momento de ejecutarse creó uno de tipo NetworkChaos. En la siguiente imagen se puede ver en Linkerd como estaba la red durante una prueba de estrés sin el chaos aplicado:

Métricas HTTP						
Deployment	En la malla de servicios	Tasa de éxito	PPS	Latencia P90	Latencia P95	Latencia P99
api-gateway	1/1	100.00% ✅	0.3	1 ms	9 ms	10 ms
cloud-config	1/1	100.00% ✅	0.3	1 ms	3 ms	3 ms
favourite-service	1/1	10.41% ✖	12.48	5 ms	11 ms	18 ms
frontend	1/1	100.00% ✅	0.3	1 ms	1 ms	1 ms
order-service	1/1	100.00% ✅	12.22	1 ms	4 ms	6 ms
payment-service	1/1	100.00% ✅	0.3	1 ms	2 ms	2 ms
product-service	1/1	100.00% ✅	5.43	1 ms	3 ms	16 ms
proxy-client	1/1	100.00% ✅	0.3	1 ms	3 ms	3 ms
service-discovery	1/1	100.00% ✅	0.93	1 ms	1 ms	2 ms
shipping-service	1/1	100.00% ✅	0.6	2 ms	8 ms	10 ms
user-service	1/1	100.00% ✅	29.03	1 ms	2 ms	5 ms
zipkin	1/1	100.00% ✅	4.58	1 ms	2 ms	78 ms

Y en la siguiente cuando el chaos está en ejecución:

Métricas HTTP						
Deployment	En la malla de servicios	Tasa de éxito	PPS	Latencia P50	Latencia P95	Latencia P99
api-gateway	1/1	100.00% ●	0.3	1 ms	5 ms	5 ms
cloud-config	1/1	100.00% ●	0.3	1 ms	2 ms	2 ms
favourite-service	1/1	100.00% ●	18.53	16 ms	34 ms	81 ms
frontend	1/1	100.00% ●	0.3	1 ms	1 ms	1 ms
order-service	1/1	100.00% ●	19.22	1 ms	6 ms	17 ms
payment-service	1/1	100.00% ●	0.3	1 ms	2 ms	2 ms
product-service	1/1	100.00% ●	73.42	1 ms	2 ms	9 ms
proxy-client	1/1	100.00% ●	0.3	1 ms	2 ms	2 ms
service-discovery	1/1	100.00% ●	0.98	1 ms	2 ms	3 ms
shipping-service	1/1	100.00% ●	0.6	2 ms	3 ms	3 ms
user-service	1/1	100.00% ●	74.5	1 ms	2 ms	10 ms
zipkin	1/1	100.00% ●	5.32	1 ms	6 ms	15 ms

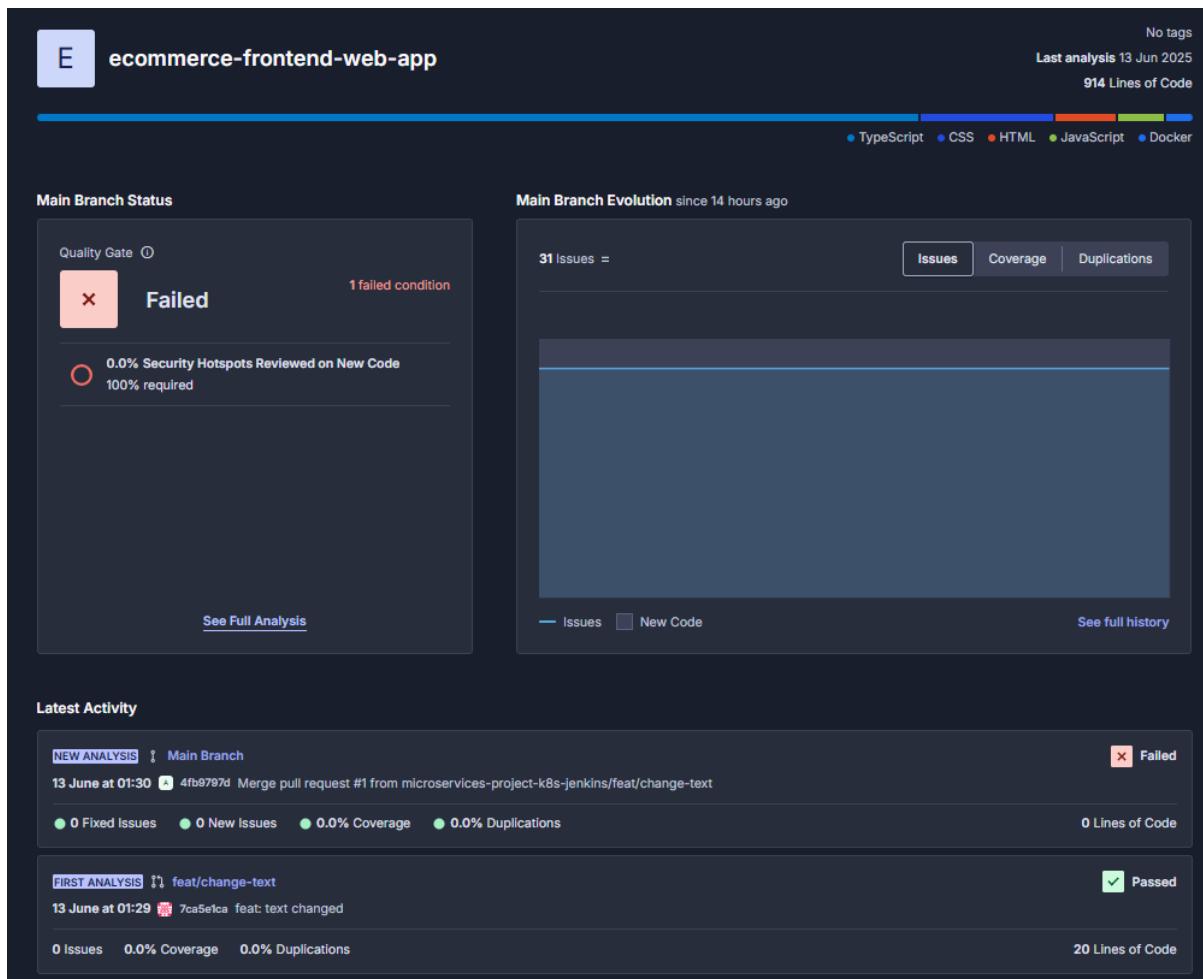
Se puede analizar que el favourite-service, un servicio que hace uso de product-service aumenta de forma significativa su latencia, pasando de 18ms a 81ms, es decir, un aumento de 4.5 veces.

### Escaneo de Código con SonarQube

#### Escaneo desde SonarCloud



El escaneo del código se hizo sobre el repositorio del frontend, ya que es más simple al este tener un solo código en lugar del de los microservicios, por lo que se puede escanear más fácilmente.



Se puede ver que hay problemas relacionados con el código, debido al status “failed” de este. Por otro lado, es importante notar que SonarQube escanea código por cada push a la rama principal, evaluando si este produce mayor cantidad de problemas, ya sea de mantenibilidad, confiabilidad o de seguridad.

The screenshot shows a code review interface with a sidebar for filtering issues. The sidebar includes sections for Software Quality (Security: 0, Reliability: 4, Maintainability: 27), Severity (Blocker: 0, High: 0, Medium: 21, Low: 10, Info: 0), Clean Code Attribute (Consistency: 7, Intentionality: 24, Adaptability: 0, Responsibility: 0), and various other filters like Type, Status, Security Category, and Creation Date.

The main area displays a list of issues:

- Dockerfile**
  - Merge this RUN instruction with the consecutive ones. (Maintainability)
  - Sort these package names alphabetically. (Maintainability)
  - Merge this RUN instruction with the consecutive ones. (Maintainability)
- src/app/app.component.html**
  - Add "lang" and/or "xml:lang" attributes to this "<html>" element (Reliability)
  - Add a <title> tag to this page. (Reliability)
- src/app/app.component.spec.ts**
  - Remove this commented out code. (Maintainability)

Se puede ver que el código del frontend tiene varios problemas, sin embargo, ninguno es de seguridad, ya que los que hay son de mantenibilidad y de confiabilidad.

The screenshot shows a detailed view of an issue in `src/app/app.component.html`. The issue is described as "Add <lang> and/or <xml:lang> attributes to this <html>" element. The code editor shows the following snippet:

```

<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.0/dist/css/bootstrap.min.css" integrity="sha384-KyZXEAg3QhQLjpGr+8fMAXLRk2voC2F3B092VXn8CA5QIVF02J3BCsw2P0p/H=" crossorigin="anonymous" />
    <link href="https://maxcdn.bootstrapcdn.com/font-awesome/4.7.0/css/font-awesome.min.css" rel="stylesheet" />
  </head>
  <body>
    <router-outlet></router-outlet>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
    <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js" integrity="sha384-J3Wyy0p3x81rRbZUAYoIly60rq5vrltaff/nJGzIxFdF4x0xIM+B07jRM" crossorigin="anonymous"></script>
  </body>
</html>

```

The right sidebar provides additional details about the issue:

- Tags:** accessibility
- Line affected:** L2
- Effort:** 2 min
- Introduced:** 3 years ago

Al entrar a uno de los problemas, se puede ver que uno de ellos es que no se indica el idioma de la página en la etiqueta <html>, lo que puede afectar el SEO de esta o impedir que la página interactúe correctamente con los traductores.

The screenshot shows a dark-themed user interface for a security analysis tool. At the top, there are tabs for 'Summary', 'Issues', 'Security Hotspots' (which is selected), 'Measures', 'Code', and 'Activity'. The main area is divided into two sections. On the left, a sidebar displays statistics: '0.0% Security Hotspots Reviewed', '5 Security Hotspots to review', 'Review priority: Medium' (with a 'Permission' category), and 'Review priority: Low' (with an 'Others' category). On the right, a detailed view of a specific hotspot is shown. The hotspot title is 'The "node" image runs with "root" as the default user. Make sure it is safe here.' It includes a note: 'Running containers as a privileged user is security-sensitive docker:S6471'. The status is 'To Review'. A 'Review' button is present. Below this, a code snippet from a Dockerfile is shown, highlighting the line 'FROM node:14-alpine' with a red box and the same error message: 'The "node" image runs with "root" as the default user. Make sure it is safe here.' The code snippet also shows other Dockerfile instructions like RUN apk add --no-cache nginx gettext bash, RUN npm i -g npm@8.1.4, and RUN npm i -g @angular/cli@13.0.3.

Se puede ver que encontró una vulnerabilidad en la imagen de docker, ya que la imagen base de node corre sobre el usuario root.

This screenshot shows a different part of the security analysis tool. On the left, a sidebar provides a 'Project Overview' with sections for 'Security' (Rating A, 0 vulnerabilities), 'Reliability', 'Maintainability', 'Security Review', 'Duplications', 'Size', and 'Complexity'. On the right, the main panel shows a 'Security Rating' for the project 'ecommerce-frontend-web-app'. It lists 82 files with their respective security ratings: src/app/model/address.spec.ts (A), src/app/model/address.ts (A), angular.json (A), src/app/app-routing.module.ts (A), src/app/app.component.css (A), src/app/app.component.html (A), src/app/app.component.spec.ts (A), src/app/app.component.ts (A), src/app/app.module.ts (A), src/app/model/request/authentication-request.spec.ts (A), src/app/model/request/authentication-request.ts (A), src/app/model/response/authentication-response.spec.ts (A), src/app/model/response/authentication-response.ts (A), src/app/service/authentication.service.spec.ts (A), and src/app/service/authentication.service.ts (A).

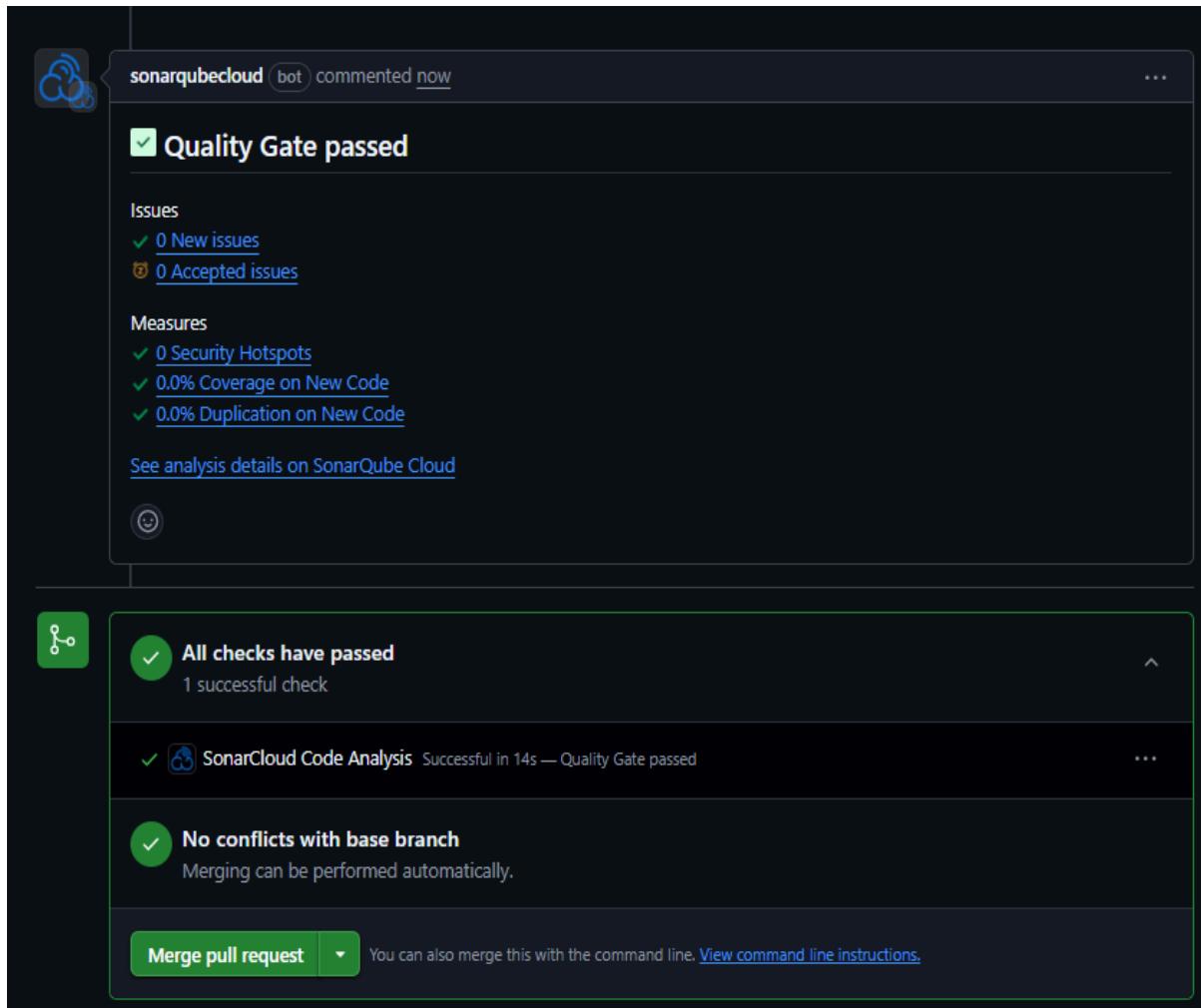
Se puede ver un security rating seccionado por cada archivo y carpeta del código, indicando cuales son los lugares en los que se concentra la mayor cantidad de problemas del código que está siendo analizado.

SonarQube también proporciona un resumen detallado de la calidad del código, organizado por archivo y carpeta. Este análisis identifica los tipos de vulnerabilidades o problemas presentes, e incluye métricas clave como el porcentaje de duplicación de código, lo que facilita la detección de áreas que requieren refactorización o mejoras.

### Escaneo desde GitHub

Se puede ver que al haber vinculado SonarQube al repositorio, que pueden ver sus escaneos desde GitHub.

## Pruebas como Gate en PR



Al realizar un Pull Request, SonarQube escanea el nuevo código, realizando una prueba tipo gate que valida automáticamente si este cumple con los estándares mínimos de calidad definidos. Como se muestra en la imagen, se verifica que no existan vulnerabilidades, code smells, duplicaciones ni security hotspots en el código introducido. En este caso, el análisis fue exitoso y el Quality Gate fue aprobado, lo que permite continuar con el proceso de integración y despliegue sin riesgos técnicos identificados. Esta integración temprana de análisis estático garantiza que solo se fusionen cambios saludables al repositorio principal.

## **Conclusiones**

La implementación del proyecto *Despliegue de arquitectura escalable en Kubernetes con enfoque GitOps* permitió comprobar la viabilidad y las ventajas de integrar tecnologías y metodologías modernas para gestionar aplicaciones nativas de la nube.

Una de las conclusiones más importantes es que automatizar todo el ciclo de vida con GitOps fue clave, además usar GitHub junto con GitHub Actions para integración continua y ArgoCD para entrega continua ayudó a construir un pipeline ágil, confiable y fácil de auditar. Este enfoque redujo al mínimo las intervenciones manuales en el clúster, lo que disminuyó errores humanos y mantuvo el estado de la infraestructura siempre coherente y reproducible.

Otro aspecto fundamental fue la seguridad, que no se trató como algo añadido, sino como una parte central desde el principio, ya que se aplicaron NetworkPolicies con Calico para restringir la comunicación entre microservicios, se integraron herramientas como Trivy para escanear vulnerabilidades en las imágenes, Cert-Manager para la gestión de certificados TLS, y se usaron ServiceAccounts junto con RBAC para una administración segura de accesos. También se incorporó SonarQube al pipeline para hacer análisis estático del código, lo que ayudó a detectar errores, vulnerabilidades y problemas de mantenibilidad, toda esta integración con GitHub permitió evaluar automáticamente cada cambio y fomentar buenas prácticas de desarrollo.

Además, se implementó Linkerd como service mesh, lo cual trajo beneficios adicionales en cuanto a seguridad, observabilidad y resiliencia, con Linkerd se permitió cifrar el tráfico entre servicios y detectar latencias anómalas, también facilitó el análisis de métricas a nivel de malla, complementando el monitoreo general del sistema y ayudando a identificar cuellos de botella o comportamientos inesperados entre servicios.

La infraestructura como código también tuvo un papel esencial, ya que gracias al uso de Terraform fue posible gestionar todos los recursos de AWS, incluyendo el clúster de EKS, la VPC y servicios de soporte como S3 y DynamoDB, facilitando la colaboración entre el equipo, y permitiendo versionar los cambios y proporcionó un entorno estable y fácilmente replicable.

En cuanto a la observabilidad, se implementó una solución completa, pues se usaron Prometheus y Grafana para recolectar métricas y generar alertas, la pila ELK para centralizar los logs, y Zipkin para trazar las peticiones distribuidas. Esta visibilidad permitió entender el comportamiento del sistema, resolver problemas más fácilmente y tomar decisiones informadas.

Por último, el diseño basado en microservicios, desplegado sobre Kubernetes, demostró ser resiliente y escalable, tanto que gracias a mecanismos como el HPA, Linkerd y patrones de tolerancia a fallos como el reinicio en el frontend, fue posible responder adecuadamente ante cargas altas. Además, se integró Chaos Mesh como herramienta de pruebas de caos, lo que permitió simular fallos controlados en el entorno de producción para validar la capacidad de

recuperación del sistema. Estas pruebas ayudaron a identificar puntos débiles y confirmar el comportamiento esperado ante condiciones adversas. Las pruebas de estrés con Locust también confirmaron la capacidad del sistema para escalar, y las pruebas de liveness y readiness ayudaron a garantizar disponibilidad y recuperación automática de los servicios.

En resumen, el proyecto fue una prueba práctica y efectiva de cómo desplegar y mantener sistemas modernos de manera segura, eficiente y escalable, la combinación de herramientas como Kubernetes, AWS, Terraform, GitOps, Linkerd y Chaos Mesh, junto con buenas prácticas de seguridad, automatización y observabilidad, constituye un modelo sólido para abordar los retos actuales del desarrollo en la nube.

### **Referencias:**

*Chaos Mesh Overview | Chaos Mesh.* (s. f.). <https://chaos-mesh.org/docs/>

*CVE-2021-43466 | Vulnerability Database | Aqua Security.* (s. f.). Aqua Vulnerability Database. <https://avd.aquasec.com/nvd/2021/cve-2021-43466/>

*CVE-2024-24790 | Vulnerability Database | Aqua Security.* (s. f.). Aqua Vulnerability Database. <https://avd.aquasec.com/nvd/2024/cve-2024-24790/>

*Elastic — the search AI company.* (s. f.). Elastic. <https://www.elastic.co/>

Grafana Labs. (s. f.). *Grafana: The open and composable observability platform | Grafana Labs.* <https://grafana.com/>

*Locust.io.* (s. f.). <https://locust.io/>

*Prometheus - Monitoring system & time series database.* (s. f.). <https://prometheus.io/>

*Sonar. (s. f.). Code Quality Tool & Secure Analysis with SonarQube.*

<https://www.sonarsource.com/es/products/sonarqube/>

*The only service mesh designed for human beings. (s. f.). Linkerd. <https://linkerd.io/>*

. (s. f.). <https://trivy.dev/latest/>