ORT-FI-8184-DevOps 2024S2-G6-AMARTINEZ

Agustín Martinez - 274479

Tutor: Federico Barceló

Índice

Presentación del problema

La transición digital de una empresa líder en retail reveló una brecha crítica en la colaboración entre los equipos de desarrollo y operaciones, afectando el lanzamiento de una nueva aplicación clave para mejorar la experiencia del cliente.

La desconexión cultural y organizativa entre ambos equipos generó despliegues problemáticos que impactaron la estabilidad del sistema y la experiencia del usuario. Este desafío destacó la necesidad de fomentar una cultura de colaboración, comunicación efectiva y objetivos compartidos para superar las barreras existentes. La dirección ejecutiva solicitó un plan integral que no solo resuelva los problemas técnicos, sino que también impulse un cambio cultural, asegurando agilidad y resiliencia operativa a largo plazo

---

Solución propuesta

La solución se basa en implementar una cultura DevOps que fomente la colaboración y el trabajo conjunto entre los equipos de desarrollo y operaciones. En lugar de trabajar de forma separada, ambos equipos comparten objetivos comunes, promoviendo una comunicación fluida y una responsabilidad conjunta durante todo el ciclo de desarrollo.

Este enfoque prioriza la integración continua y la entrega frecuente de valor, permitiendo identificar y resolver problemas rápidamente. Además, se busca crear un ambiente de aprendizaje constante, donde la retroalimentación y la mejora continua sean clave.

Con esta metodología, la empresa no solo enfrentará los desafíos actuales, sino que también desarrollará una operación más ágil y adaptable, fortaleciendo su competitividad a largo plazo.

Herramientas utilizadas

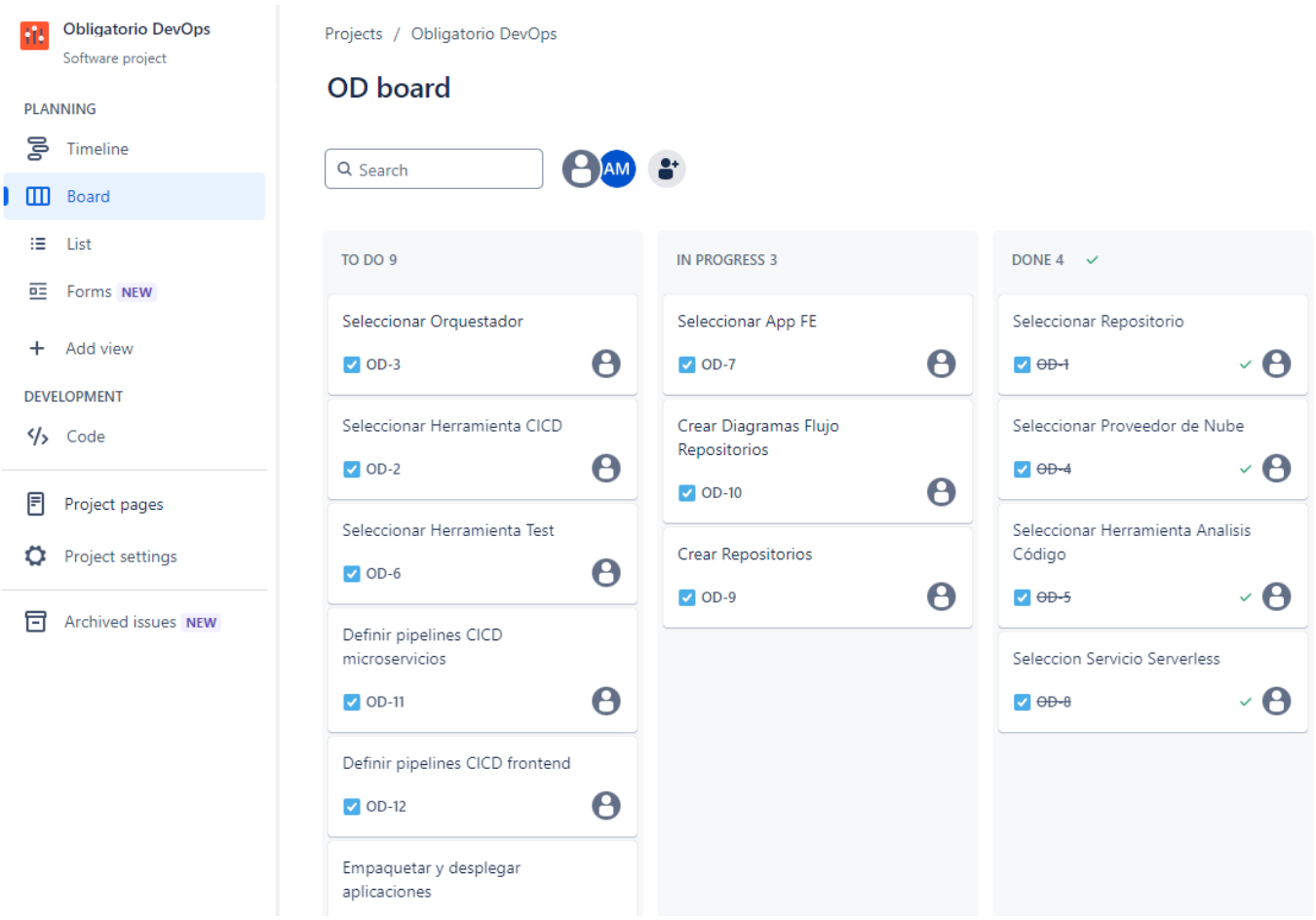
Listado de herramientas y tecnologías utilizadas en el proyecto:

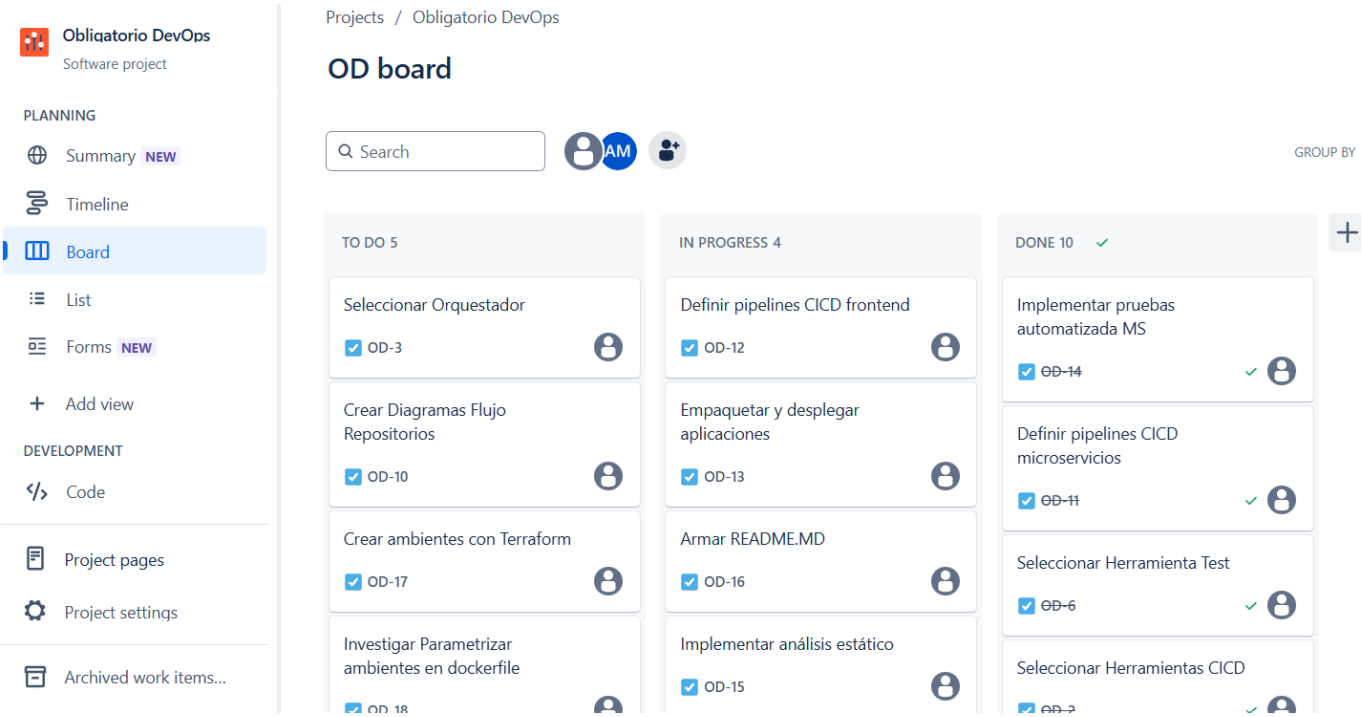
- **Herramienta de Versionado**: GitHub
- **Herramienta de CI/CD**: GitHub Actions
- **Aplicativo de FE a buildear y desplegar**: React
- **Herramienta para análisis de código estático**: SonarCloud
- **Herramienta para análisis de prueba extra**: JUnit & Mockito
- **Cloud provider**: AWS
- **Orquestador**: AWS ECS
- **Servicio serverless a usar**: API Gateway
- **Herramienta para el IaC**: Terraform

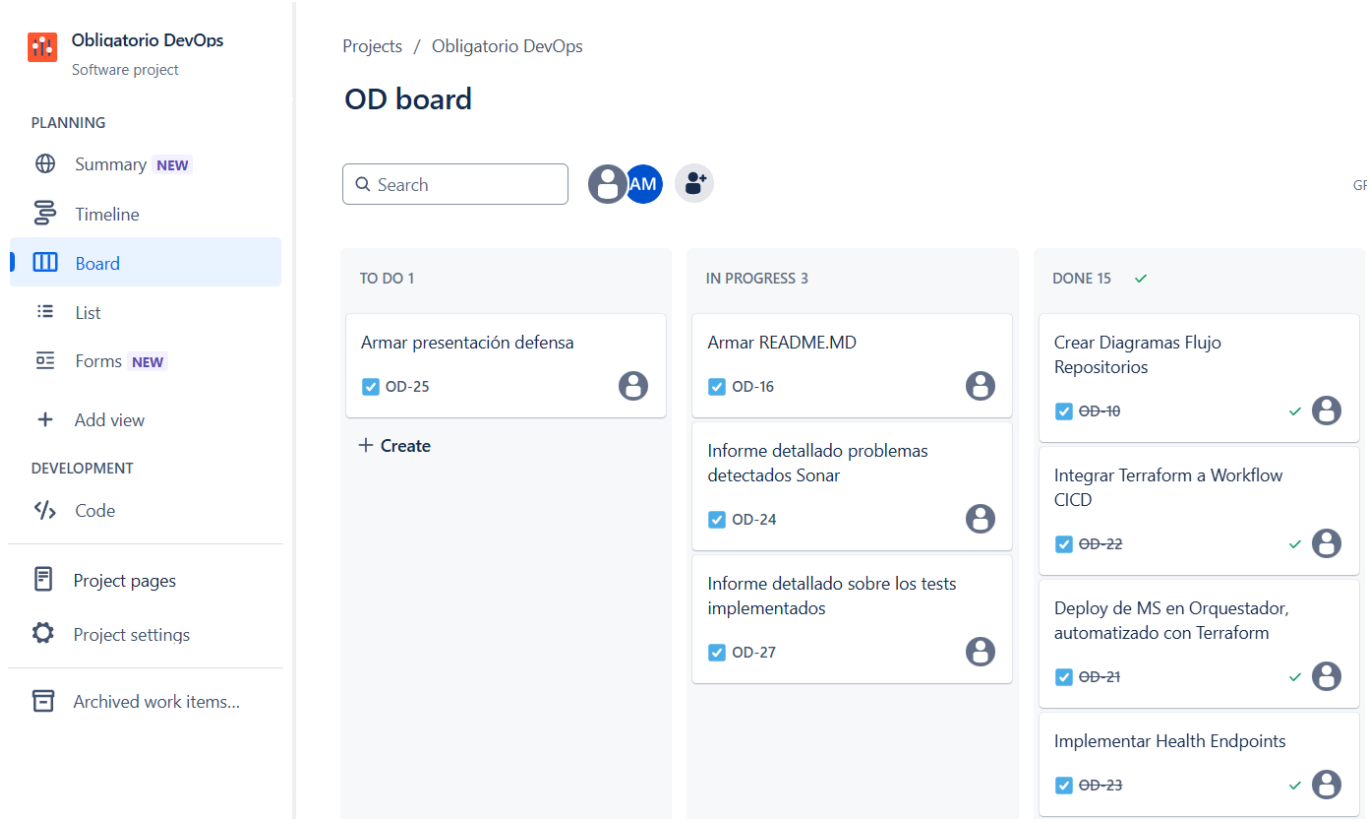
Planificación

Se decidió utilizar un tablero Kanban en Jira para la planificación y seguimiento de las tareas. Con esto logré tener una visualización clara y sencilla del flujo de trabajo. Además, contaba con la flexibilidad necesaria para poder agregar, modificar y eliminar tareas a medida que iba avanzando con la implementación.

A continuación, se muestran algunas imágenes a modo de evidencia de lo que fue la evolución del tablero.


Obligatorio DevOps
Software project
PLANNING
Timeline
Board
List
Forms NEW
Add view
DEVELOPMENT
Code
Project pages
Project settings
Archived issues NEW
Projects / Obligatorio DevOps
OD board
Search
AM
TO DO 9
Seleccionar Orquestador
OD-3
Seleccionar Herramienta CICD
OD-2
Seleccionar Herramienta Test
OD-6
Definir pipelines CICD microservicios
OD-11
Definir pipelines CICD frontend
OD-12
Empaquetar y desplegar aplicaciones
IN PROGRESS 3
Seleccionar App FE
OD-7
Crear Diagramas Flujo Repositorios
OD-10
Crear Repositorios
OD-9
DONE 4
Seleccionar Repositorio
OD-1
Seleccionar Proveedor de Nube
OD-4
Seleccionar Herramienta Analisis Código
OD-5
Seleccion Servicio Serverless
OD-8


Obligatorio DevOps
Software project
PLANNING
Summary NEW
Timeline
Board
List
Forms NEW
Add view
DEVELOPMENT
Code
Project pages
Project settings
Archived work items...
Projects / Obligatorio DevOps
OD board
Search
AM
GROUP BY
TO DO 5
Seleccionar Orquestador
OD-3
Crear Diagramas Flujo Repositorios
OD-10
Crear ambientes con Terraform
OD-17
Investigar Parametrizar ambientes en dockerfile
IN PROGRESS 4
Definir pipelines CICD frontend
OD-12
Empaquetar y desplegar aplicaciones
OD-13
Armar README.MD
OD-16
Implementar análisis estático
OD-15
DONE 10
Implementar pruebas automatizada MS
OD-14
Definir pipelines CICD microservicios
OD-11
Seleccionar Herramienta Test
OD-6
Seleccionar Herramientas CICD



Repositorio de código

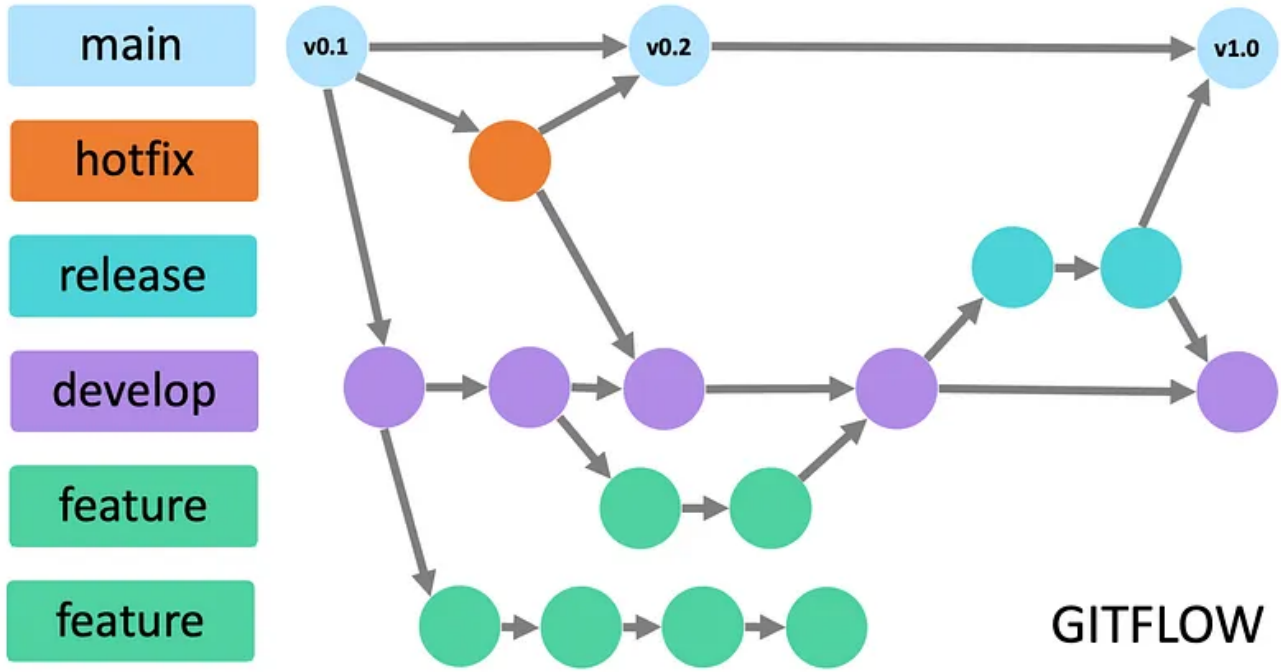
Como se mencionó anteriormente, GitHub fue la herramienta elegida como repositorio de código. Esta decisión fue tomada basándome en mi experiencia con la herramienta, su amplía adopción en la industria y la practicidad de utilizar GitHub Actions, herramienta la cual fue enseñada en el curso.

En GitHub se definió una organización con el nombre "obligatorio-devops-agustin" la cual contiene todos los repositorios de los que se hablará a continuación. Además se crearon los secrets necesarios para los flujos de CI/CD dentro de la misma, estos secrets contienen tokens para acceder a AWS, DockerHub y SonarCloud.

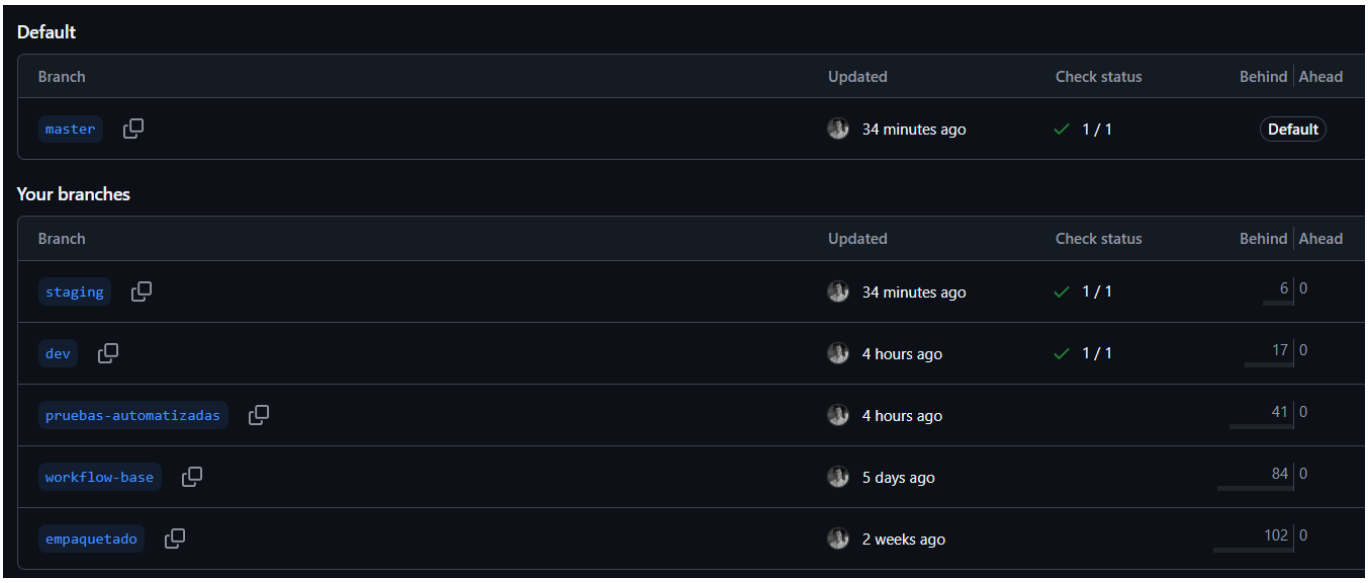
Secrets de la Organización

Estrategia de ramas

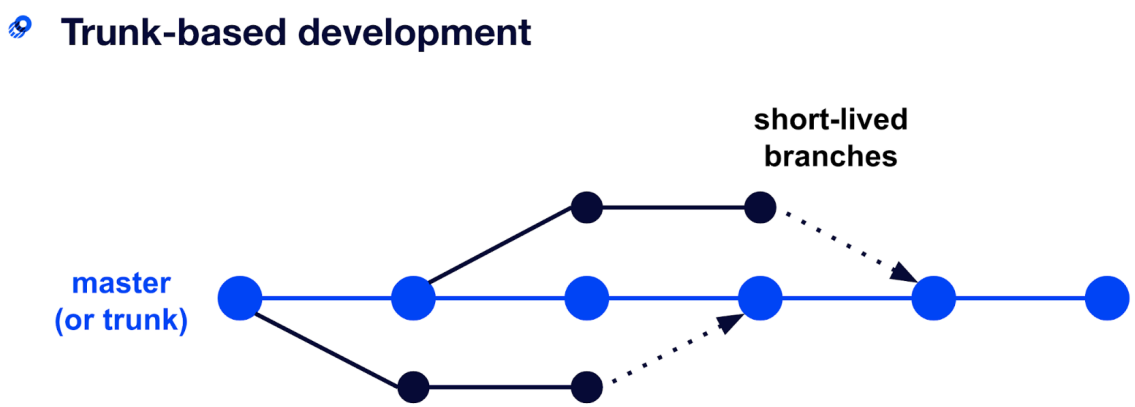
Tanto para los repositorios de microservicios como para el del frontend se decide utilizar la estrategia de ramas GitFlow. En estos repositorios se definen tres ramas principales las cuales son master, staging y dev. De esta forma tendremos la rama master por un lado, que apunta a ser una rama estable la cual está lista para producción en cualquier momento y la rama staging en donde ocurrirá la preparación de nuevas versiones. Además, existirán ramas específicas por cada característica a desarrollar (feature branch), esto aporta flexibilidad a los desarrolladores al momento de colaborar sin interferir entre sí.



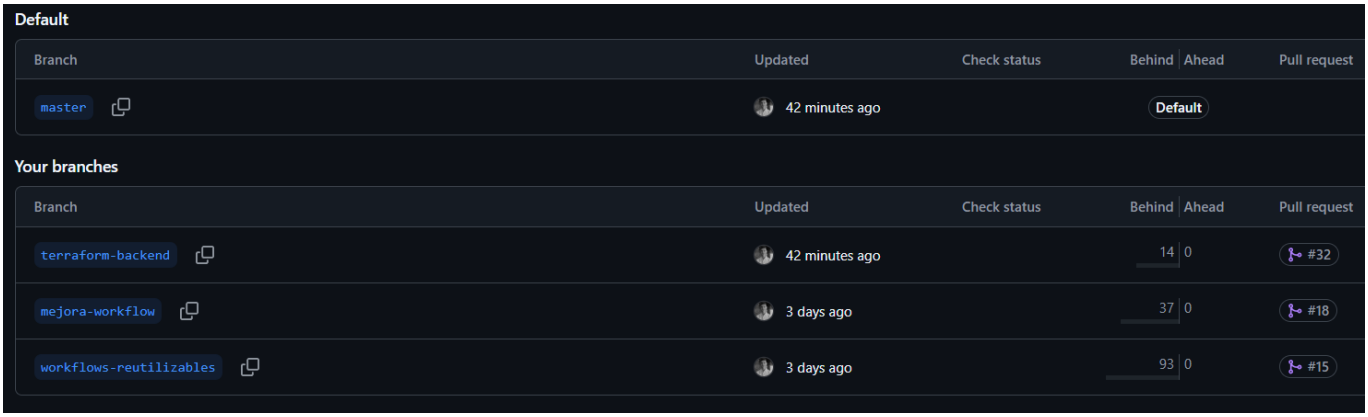
Evidencia Git Flow



Para el repositorio de DevOps se utiliza la estrategia de trunk-based. Debido a que este repositorio contendrá configuraciones de infraestructura y scripts de automatización, los cambios serán pequeños, incrementales y requerirán una rápida validación en los entornos de integración y producción. Con la estrategia de trunk-based fomentamos la agilidad, con commits frecuentes e integración continua, evitando largas ramas que pueden llegar a retrasar la implementación de nuevos cambios.



Evidencia Trunk-Based



Repositorio microservicios

El repositorio de microservicios es un mono-repo, esto quiere decir que contiene dentro del mismo, los 4 microservicios a desplegar. Además se incluye el archivo necesario para el flujo de CICD.

Repositorio frontend

Este repositorio contiene la aplicación del frontend seleccionada. Como se mencionó anteriormente, esta es la de React, específicamente la aplicación de "catalog". Además, incluye el archivo necesario para el flujo de CICD.

Repositorio devops

En este repositorio se pueden encontrar principalmente dos cosas. Por un lado, tenemos los archivos correspondientes a los flujos de CICD, tanto del repositorio de microservicios como el de frontend. Además de los archivos de terraform necesarios para desplegar toda la infraestructura como código.

Workflow CI/CD

Para los flujos de CI/CD decidí utilizar el siguiente enfoque:

Cada repositorio (microservicios y frontend) tiene un workflow el cual se ejecuta cada vez que se realiza un push a las ramas de master, staging o dev. Este workflow, se encarga de llamar a otro, el cual se encuentra en el repositorio de devops. Al llamarlo le pasa algunos parámetros necesarios para continuar con el flujo.

El workflow del repositorio frontend, pasa por parámetro los siguientes datos:

- Repositorio
- Branch
- Commit Hash

El workflow del repositorio de microservicios, pasa por parámetro los siguientes datos:

- Repositorio
- Branch
- Commit Hash
- Lista de microservicios

CI/CD Frontend

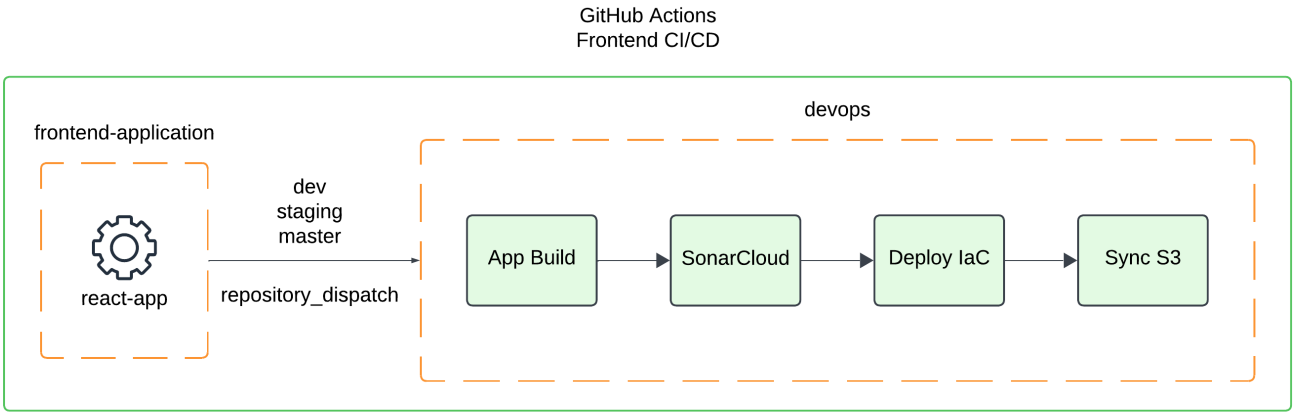
Como se mencionó anteriormente, el workflow implementado para el repositorio de la aplicación frontend se dispara cada vez que se realiza un push a las ramas de dev, staging o master. Al ocurrir esto, el flujo genera un trigger hacia el repositorio de devops, pasándole datos importantes como el repositorio y la rama sobre la que se hizo el push, así como también el hash del commit realizado.

El workflow en el repositorio devops se encarga de realizar las siguientes tareas:

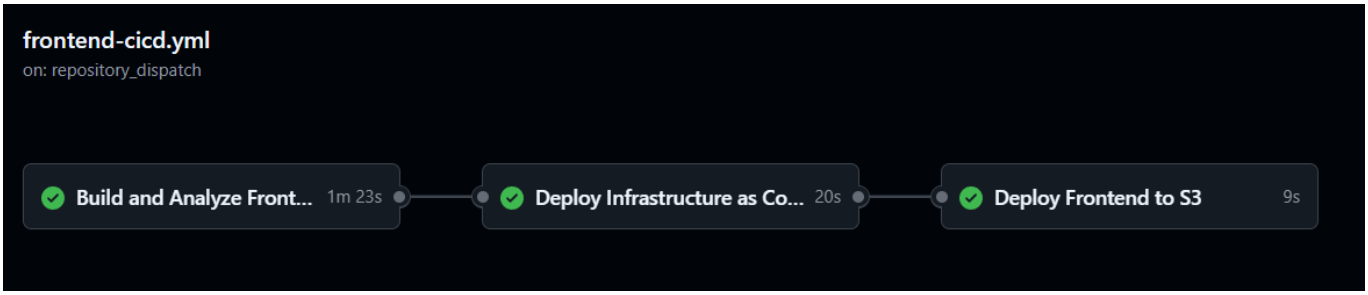
En primer lugar, está la tarea “Build and Analyze Frontend Application”, esta tarea realiza el checkout del código con los datos que le fueron proporcionados por el trigger, configura NodeJS para así poder realizar la instalación de dependencias de la aplicación frontend y luego construirla. Por último, se realiza un análisis sobre la calidad y seguridad del código utilizando la herramienta SonarCloud.

La segunda tarea es “Deploy Infrastructure as Code”, esta tarea se ejecuta únicamente si la tarea anterior finaliza correctamente. Se encarga de realizar un checkout de la rama master del repositorio de devops para así obtener los archivos de terraform necesarios, configura las credenciales para conectarse con AWS, las mismas las obtiene accediendo a los secrets configurados en la organización. Por último, se posiciona en el directorio donde se encuentra la configuración de terraform correspondiente al frontend y ejecuta los comandos de terraform init y terraform apply para aplicar el despliegue de la infraestructura necesaria.

La última tarea es la de “Deploy Frontend to S3” y al igual que la anterior, esta tarea se ejecuta únicamente si la anterior finaliza correctamente y sin ningún error. Nuevamente configura las credenciales de AWS basándose en los secrets de la organización y ejecuta el comando necesario para sincronizar la aplicación de frontend al bucket S3 desplegado en el paso anterior. Cabe aclarar que se utiliza un artifact de GitHub para acceder al build de la aplicación generado en la primera tarea.



Evidencia del flujo implementado



CI/CD Microservicios

Como se mencionó anteriormente, el workflow implementado para el repositorio de los microservicios se dispara cada vez que se realiza un push a las ramas de dev, staging o master. Al ocurrir esto, el flujo genera un trigger hacia el repositorio de devops, pasándole datos importantes como el repositorio y la rama sobre la que se hizo el push, así como también el hash del commit realizado y una lista de los microservicios a desplegar.

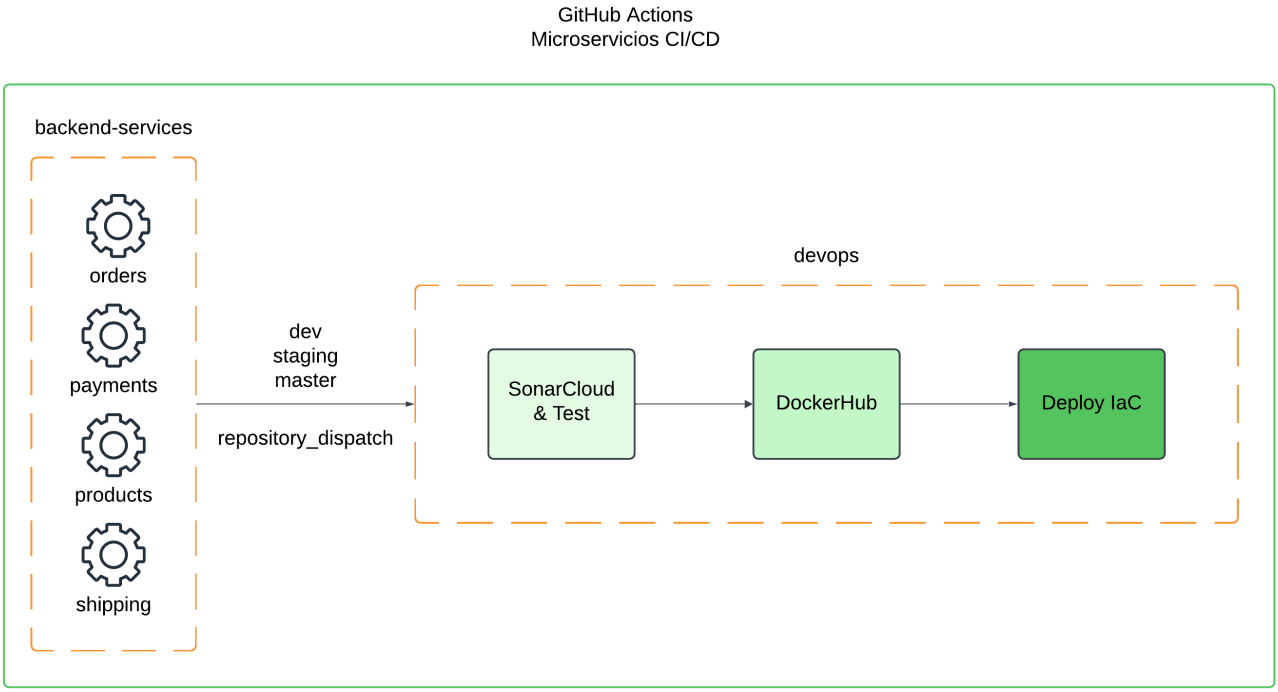
El workflow en el repositorio devops se encarga de realizar las siguientes tareas:

En primer lugar, está la tarea de “Build, Test, and Analyze Microservices”, esta tarea realiza el checkout del código con los datos que le fueron proporcionados por el trigger. Configura Java para poder ejecutar Maven y procede a realizar tanto la compilación como el análisis de los microservicios. Por cada microservicio

recibido desde el trigger en la lista, ejecuta el comando "mvn -B clean test", esto compila y ejecuta las pruebas implementadas, luego realiza el análisis del código con SonarCloud.

La segunda tarea es “Publish Images DockerHub”, esta tarea se ejecuta únicamente si la tarea anterior finaliza correctamente. Lo primero que se hace es recuperar el artifact empaquetado en el paso anterior, se autentica en DockerHub utilizando los secrets correspondientes en la organización y comienza a construir las imágenes. Por cada microservicio en la lista, genera un tag basándose en la rama y el commit, construye la imagen y sube la misma al repositorio de DockerHub.

Por último, está la tarea de “Deploy Infrastructure as Code”, al igual que la tarea anterior, esta solo se ejecuta si el paso anterior finaliza exitosamente. Comienza realizando un checkout del repositorio de devops para obtener los archivos necesarios de terraform, y configura las credenciales de AWS, utilizando los secrets de la organización. Luego se posiciona en el directorio donde se encuentra la configuración de terraform correspondiente al backend y ejecuta los comandos de terraform init y terraform apply, esto despliega la infraestructura necesaria con las imágenes de los microservicios actualizadas.



Evidencia del flujo implementado

Infraestructura como código

Para el manejo de la IaC se decidió utilizar terraform, debido a la robustez de la herramienta, su capacidad de integrarse con varios proveedores de nube y que fue la aprendida durante el curso. Esta tiene la capacidad de manejar distintos workspaces, en este caso decidí implementar un workspace por cada ambiente a desplegar,

los mismos son dev, staging y master. Por otra parte, para el despliegue de los microservicios, decidí utilizar módulos. Debido a que el despliegue de los microservicios implica desplegar varios recursos, el separar la solución en varios módulos facilita el entendimiento de la solución, además de que ayuda en el mantenimiento y escalabilidad de esta.

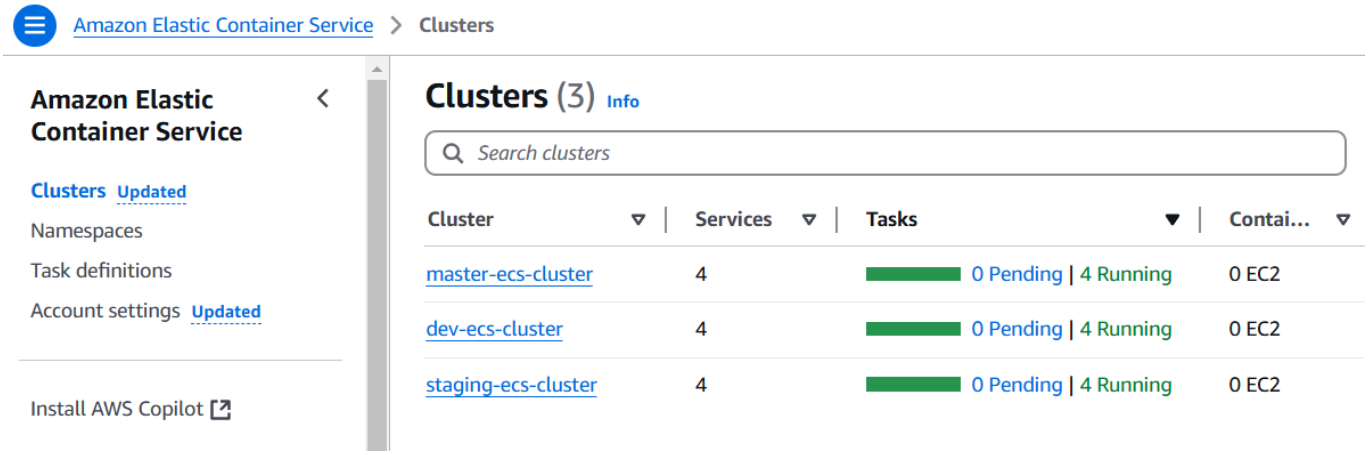
La estructura de archivos generada para terraform es la siguiente.

```
.
|-- backend
|   |-- backend.tf
|   |-- environments
|   |   |-- dev.tfvars
|   |   |-- master.tfvars
|   |   `-- staging.tfvars
|   |-- main.tf
|   |-- outputs.tf
|   `-- variables.tf
|-- frontend
|   |-- backend.tf
|   |-- environments
|   |   |-- dev.tfvars
|   |   |-- master.tfvars
|   |   `-- staging.tfvars
|   |-- main.tf
|   |-- outputs.tf
|   `-- variables.tf
`-- modules
    |-- alb
    |   |-- main.tf
    |   |-- outputs.tf
    |   `-- variables.tf
    |-- api_gateway
    |   |-- main.tf
    |   |-- outputs.tf
    |   `-- variables.tf
    |-- backend_state
    |   |-- main.tf
    |   |-- outputs.tf
    |   `-- variables.tf
    |-- ecs
    |   |-- main.tf
    |   |-- outputs.tf
    |   `-- variables.tf
    `-- vpc
        |-- main.tf
        |-- outputs.tf
        `-- variables.tf
```

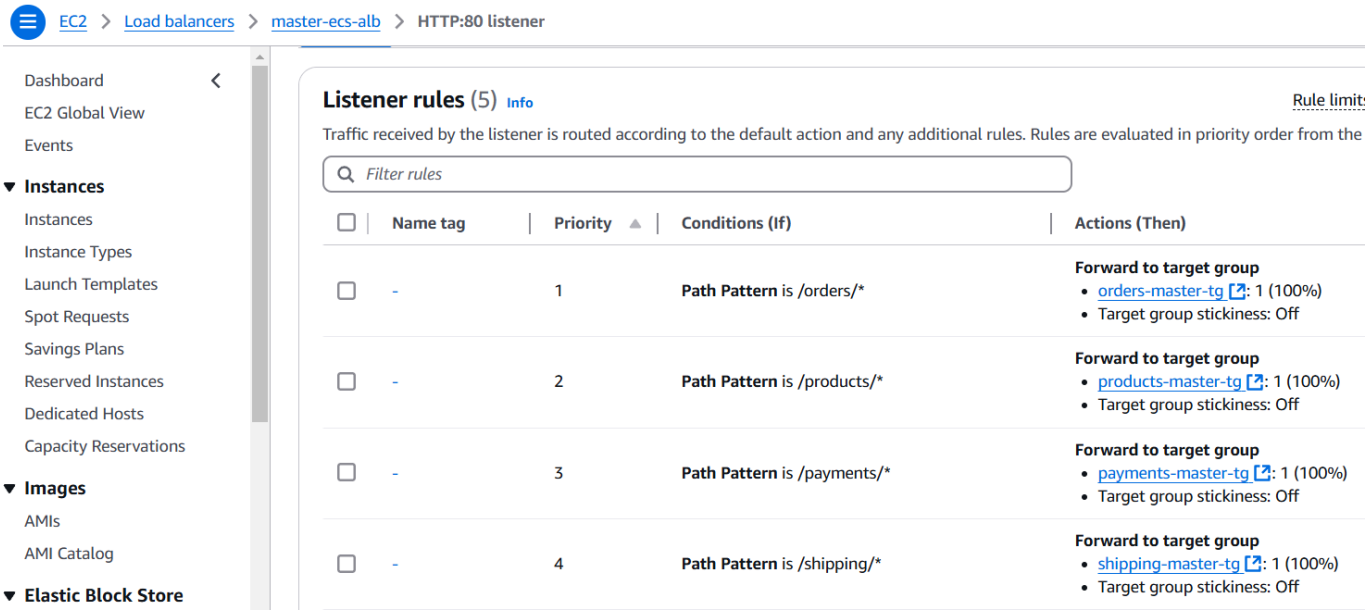
Terraform implementa un manejo del estado, mediante el cual se puede conocer el estado de la infraestructura desplegada en todo momento y facilita el mantenimiento de la misma. Este mantenimiento del estado se puede hacer de manera local aunque esto no es lo ideal para trabajar en conjunto con varias personas. Es por esto que se decidió utilizar una opción mas robusta, mediante la cual el estado es almacenado en AWS, utilizando un S3 Bucket y una tabla en DynamoDB.

A continuación, se deja evidencia de algunos de los recursos desplegados en AWS.

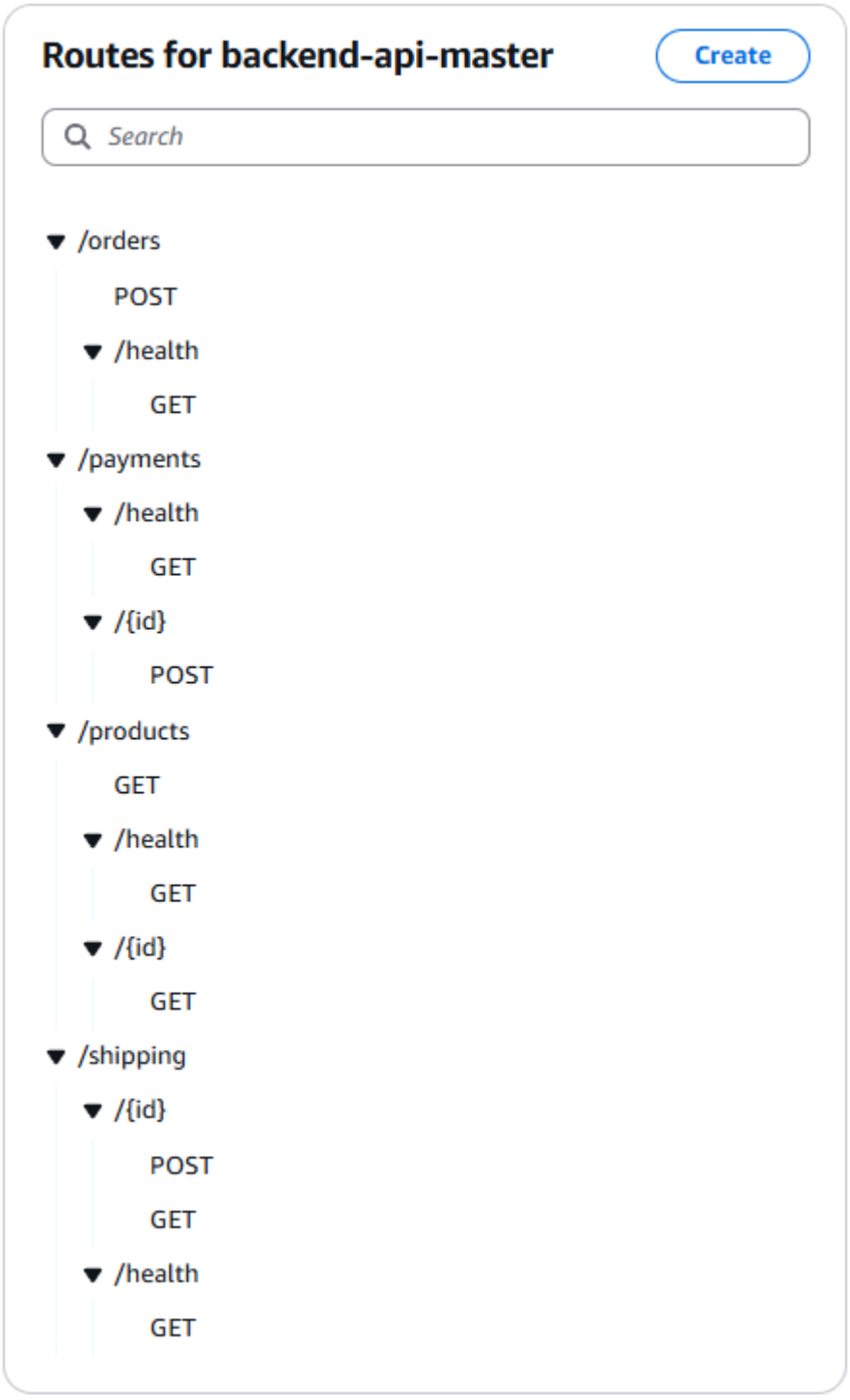
Cluster ECS



Application Load Balancer (ALB)



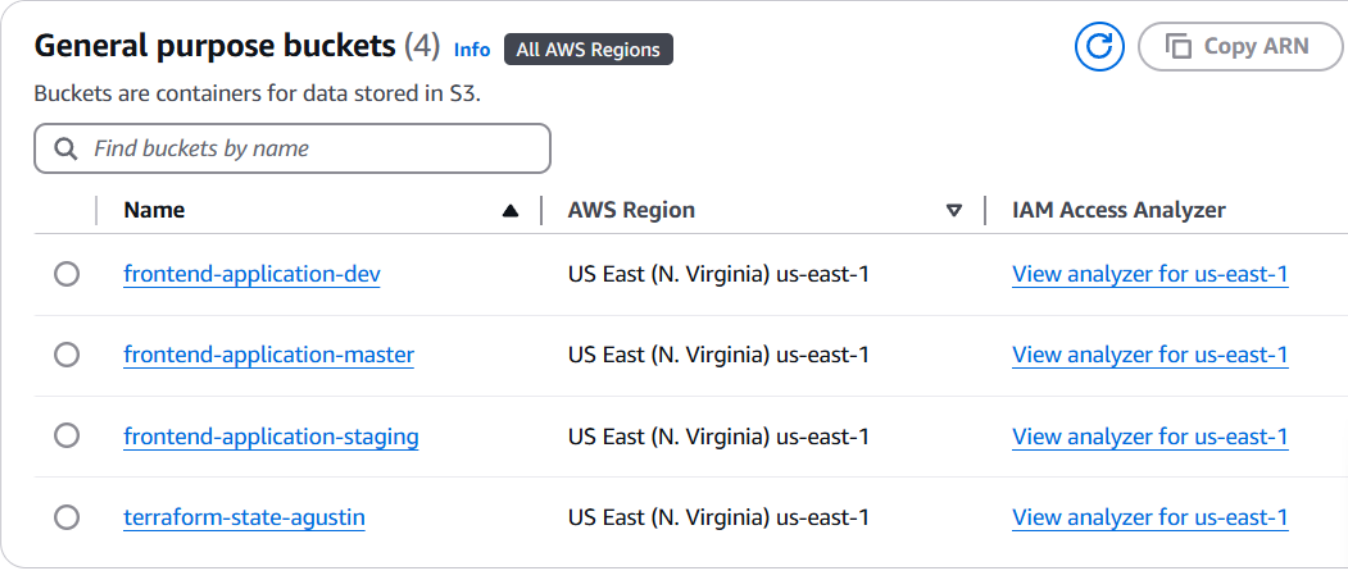
API GW



Como se puede ver a continuación, tenemos tres S3 Buckets desplegados con el nombre frontend-application-{branch}, estos buckets son los que contienen la aplicación de frontend, compilada con el contenido del último commit realizado en las ramas de dev, staging y master respectivamente.

Además podemos ver el bucket terraform-state-agustin, en este último se almacena el estado de terraform mencionado anteriormente.

S3 Buckets



General purpose buckets (4) Info All AWS Regions

Copy ARN

Buckets are containers for data stored in S3.

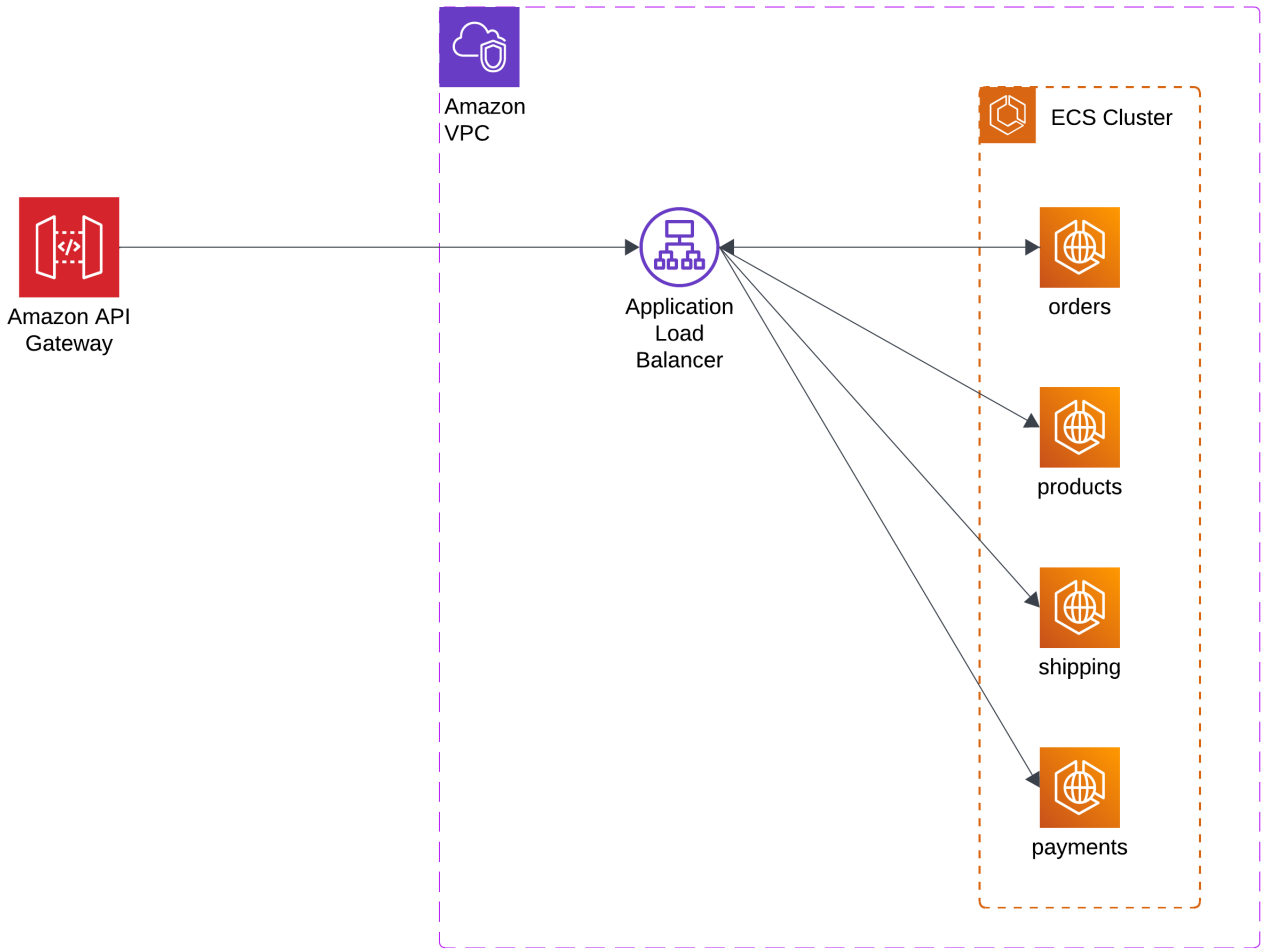
Find buckets by name

Name	AWS Region	IAM Access Analyzer
frontend-application-dev	US East (N. Virginia) us-east-1	View analyzer for us-east-1
frontend-application-master	US East (N. Virginia) us-east-1	View analyzer for us-east-1
frontend-application-staging	US East (N. Virginia) us-east-1	View analyzer for us-east-1
terraform-state-agustin	US East (N. Virginia) us-east-1	View analyzer for us-east-1

A continuación, se dejan diagramas realizados a grandes rasgos, sobre la infraestructura desplegada y como esta interactúa entre si. Por un lado, tenemos la solución para el backend, y por otro lado, para el frontend respectivamente.

Aclaración: El siguiente diagrama está simplificado para comprender la estructura y funcionamiento básico de la solución. No se incluyen en el mismo, algunos componentes fundamentales para el correcto funcionamiento de esta como lo son los target group, subredes, internet gateway y NAT gateway, aunque forman parte de la solución final.

Microservicios



Frontend

Test automatizados

Los tests implementados en los microservicios son una buena herramienta que nos ayudan a garantizar la calidad y estabilidad del sistema. Combinando pruebas unitarias, de integración y de endpoints, logramos validar que cada componente cumpla su función y que las interacciones entre ellos se comporten según lo esperado.

Las pruebas unitarias, fueron diseñadas para verificar que los métodos internos funcionen correctamente, incluso cuando hablamos de casos bordes o datos inesperados. Por otra parte, las pruebas de integración aseguran que los componentes interactúen adecuadamente entre sí y que el sistema responda de forma consistente al interactuar con los otros microservicios.

Algo a destacar en las pruebas implementadas, es el uso de mocks. Utilizando la herramienta Mockito, tenemos la posibilidad de simular la respuesta de servicios externos. Esto permite recrear situaciones como pagos fallidos, falta de stock de productos o errores en los envíos, asegurando que el sistema maneje cada caso adecuadamente.

Por último, con las pruebas sobre los endpoints, podemos evaluar que los endpoint respondan correctamente, utilizando MockMvc por ejemplo, para simular solicitudes reales. Estas pruebas no solo validan los códigos de respuesta, sino también el contenido de las respuestas.

Combinando todo esto, los tests nos ayudan a identificar problemas no solo durante el desarrollo, sino que también durante el flujo CI/CD. La ejecución automática de los mismos, permite validar cada cambio en el código antes de ser desplegado, bloqueando el proceso en caso de encontrar cualquier problema. Esto nos ayuda a garantizar la estabilidad y funcionalidad de los servicios, incluso trabajando en un ambiente dinámico donde los despliegues son frecuentes.

Ejemplo de tests implementados para el microservicio de products:

```java
    @Test  Agustin
    void testList() {
        Collection<Product> products = productsLogic.list();
        assertNotNull(products);
        assertEquals( expected: 3, products.size());
    }

    @Test  Agustin
    void testGetProduct() {
        Product product = productsLogic.getProduct( id: "123");
        assertNotNull(product);
        assertEquals( expected: "123", product.getId());
        assertEquals( expected: "Producto 123", product.getName());
    }

    @Test  Agustin
    void testGetProductNotFound() {
        Product product = productsLogic.getProduct( id: "999");
        assertNull(product);
    }

    @Test  Agustin
    void testHasProduct() {
        assertTrue(productsLogic.hasProduct( id: "123"));
        assertFalse(productsLogic.hasProduct( id: "999"));
    }
```

```java
    @Test  Agustin
    void testIndex() throws Exception {
        mockMvc.perform(get( urlTemplate: "/products")
                        .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(jsonPath( expression: "$").isArray())
                .andExpect(jsonPath( expression: "$[*].id").value(org.hamcrest.Matchers.containsInAnyOrder( ...items: "123
    }


    @Test  Agustin
    void testGetProductExists() throws Exception {
        mockMvc.perform(get( urlTemplate: "/products/{id}",  ...uriVars: "123")
                        .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(jsonPath( expression: "$.id").value( expectedValue: "123"))
                .andExpect(jsonPath( expression: "$.name").value( expectedValue: "Producto 123"));
    }

    @Test  Agustin
    void testGetProductNotExists() throws Exception {
        mockMvc.perform(get( urlTemplate: "/products/{id}",  ...uriVars: "999")
                        .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isNotFound());
    }
}
```

A continuación, se muestra evidencia sobre la ejecución de los test implementados para los microservicios:

Payments

```
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.423 s - in uy.edu.ort.devops.paymentsserviceexample.endpoints.PaymentsEndpointTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
[INFO]
```

```
[INFO] Running uy.edu.ort.devops.paymentsserviceexample.logic.PaymentsLogicTest
2024-12-08 16:00:28.264  INFO 1830 --- [           main] u.e.o.d.p.logic.PaymentsLogic            : Paying result: PaymentStatus{orderId='order123', success=true, description='Done.'}
2024-12-08 16:00:28.268  INFO 1830 --- [           main] u.e.o.d.p.logic.PaymentsLogic            : Paying result: PaymentStatus{orderId='order456', success=true, description='Done.'}
2024-12-08 16:00:28.269  INFO 1830 --- [           main] u.e.o.d.p.logic.PaymentsLogic            : Paying result: PaymentStatus{orderId='order456', success=false, description='No money.'}
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.003 s - in uy.edu.ort.devops.paymentsserviceexample.logic.PaymentsLogicTest
```

Products

```
[INFO] Running uy.edu.ort.devops.productsserviceexample.logic.ProductsLogicTest
16:01:03.563 [main] INFO uy.edu.ort.devops.productsserviceexample.logic.ProductsLogic - Listing products
16:01:03.576 [main] INFO uy.edu.ort.devops.productsserviceexample.logic.ProductsLogic - Checking if has product: 123
16:01:03.577 [main] INFO uy.edu.ort.devops.productsserviceexample.logic.ProductsLogic - Checking if has product: 999
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.097 s - in uy.edu.ort.devops.productsserviceexample.logic.ProductsLogicTest
```

```
2024-12-08 16:01:05.102  INFO 2182 --- [           main] u.e.o.d.p.e.ProductsEndpointTest         : Started ProductsEndpointTest in 1.257 seconds (JVM running for 2.022)
2024-12-08 16:01:05.431  INFO 2182 --- [           main] u.e.o.d.p.logic.ProductsLogic            : Checking if has product: 999
2024-12-08 16:01:05.460  INFO 2182 --- [           main] u.e.o.d.p.logic.ProductsLogic            : Checking if has product: 123
2024-12-08 16:01:05.532  INFO 2182 --- [           main] u.e.o.d.p.logic.ProductsLogic            : Listing products
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.96 s - in uy.edu.ort.devops.productsserviceexample.endpoints.ProductsEndpointTest
```

Shipping

```
[INFO] Running uy.edu.ort.devops.shippingserviceexample.logic.ShippingLogicTest
2024-12-08 16:01:41.531  INFO 2532 --- [           main] u.e.o.d.s.logic.ShippingLogic            : Getting shipping: a
2024-12-08 16:01:41.538  INFO 2532 --- [           main] u.e.o.d.s.logic.ShippingLogic            : Adding shipping: d
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.002 s - in uy.edu.ort.devops.shippingserviceexample.logic.ShippingLogicTest
```

```
2024-12-08 16:01:41.994  INFO 2532 --- [           main] u.e.o.d.s.e.ShippingEndpointTest         : Started ShippingEndpointTest in 0.425 seconds (JVM running for 2.746)
2024-12-08 16:01:42.027  INFO 2532 --- [           main] u.e.o.d.s.logic.ShippingLogic            : Adding shipping: d
2024-12-08 16:01:42.063  INFO 2532 --- [           main] u.e.o.d.s.logic.ShippingLogic            : Getting shipping: a
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.509 s - in uy.edu.ort.devops.shippingserviceexample.endpoints.ShippingEndpointTest
```

Orders

```
[INFO] Running uy.edu.ort.devops.ordersserviceexample.logic.OrdersLogicTest
16:02:26.283 [main] INFO uy.edu.ort.devops.ordersserviceexample.logic.OrdersLogic - Creating order.
16:02:26.284 [main] INFO uy.edu.ort.devops.ordersserviceexample.logic.OrdersLogic - Checking products.
16:02:26.284 [main] INFO uy.edu.ort.devops.ordersserviceexample.logic.OrdersLogic - Invoking products service.
16:02:26.299 [main] INFO uy.edu.ort.devops.ordersserviceexample.logic.OrdersLogic - Error in products: Missing: product1.
16:02:26.310 [main] INFO uy.edu.ort.devops.ordersserviceexample.logic.OrdersLogic - Creating order.
16:02:26.311 [main] INFO uy.edu.ort.devops.ordersserviceexample.logic.OrdersLogic - Checking products.
16:02:26.311 [main] INFO uy.edu.ort.devops.ordersserviceexample.logic.OrdersLogic - Invoking products service.
16:02:26.312 [main] INFO uy.edu.ort.devops.ordersserviceexample.logic.OrdersLogic - Error in products: No stock: product1.
16:02:26.317 [main] INFO uy.edu.ort.devops.ordersserviceexample.logic.OrdersLogic - Creating order.
16:02:26.318 [main] INFO uy.edu.ort.devops.ordersserviceexample.logic.OrdersLogic - Checking products.
16:02:26.318 [main] INFO uy.edu.ort.devops.ordersserviceexample.logic.OrdersLogic - Invoking products service.
16:02:26.319 [main] INFO uy.edu.ort.devops.ordersserviceexample.logic.OrdersLogic - Products ok.
16:02:26.320 [main] INFO uy.edu.ort.devops.ordersserviceexample.logic.OrdersLogic - Invoking payments service.
16:02:26.320 [main] INFO uy.edu.ort.devops.ordersserviceexample.logic.OrdersLogic - Payment ok.
16:02:26.320 [main] INFO uy.edu.ort.devops.ordersserviceexample.logic.OrdersLogic - Invoking shipping service.
16:02:26.324 [main] INFO uy.edu.ort.devops.ordersserviceexample.logic.OrdersLogic - Creating order.
16:02:26.324 [main] INFO uy.edu.ort.devops.ordersserviceexample.logic.OrdersLogic - Checking products.
16:02:26.324 [main] INFO uy.edu.ort.devops.ordersserviceexample.logic.OrdersLogic - Invoking products service.
16:02:26.325 [main] INFO uy.edu.ort.devops.ordersserviceexample.logic.OrdersLogic - Products ok.
16:02:26.325 [main] INFO uy.edu.ort.devops.ordersserviceexample.logic.OrdersLogic - Invoking payments service.
16:02:26.325 [main] INFO uy.edu.ort.devops.ordersserviceexample.logic.OrdersLogic - Error in payment: Insufficient funds
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.563 s - in uy.edu.ort.devops.ordersserviceexample.logic.OrdersLogicTest
```

```
2024-12-08 16:02:32.679  INFO 2974 --- [           main] u.e.o.d.o.endpoints.OrdersEndpointTest   : Started OrdersEndpointTest in 1.219 seconds (JVM running for 2.482)
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.564 s - in uy.edu.ort.devops.ordersserviceexample.endpoints.OrdersEndpointTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
```

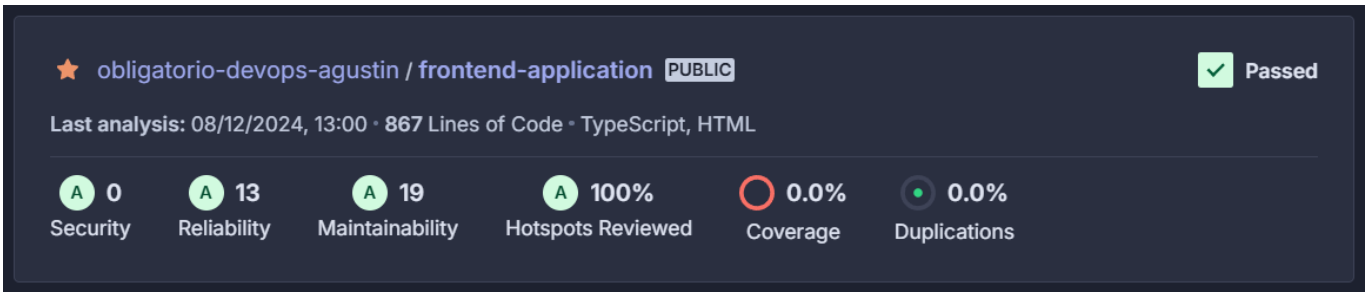
Análisis de código

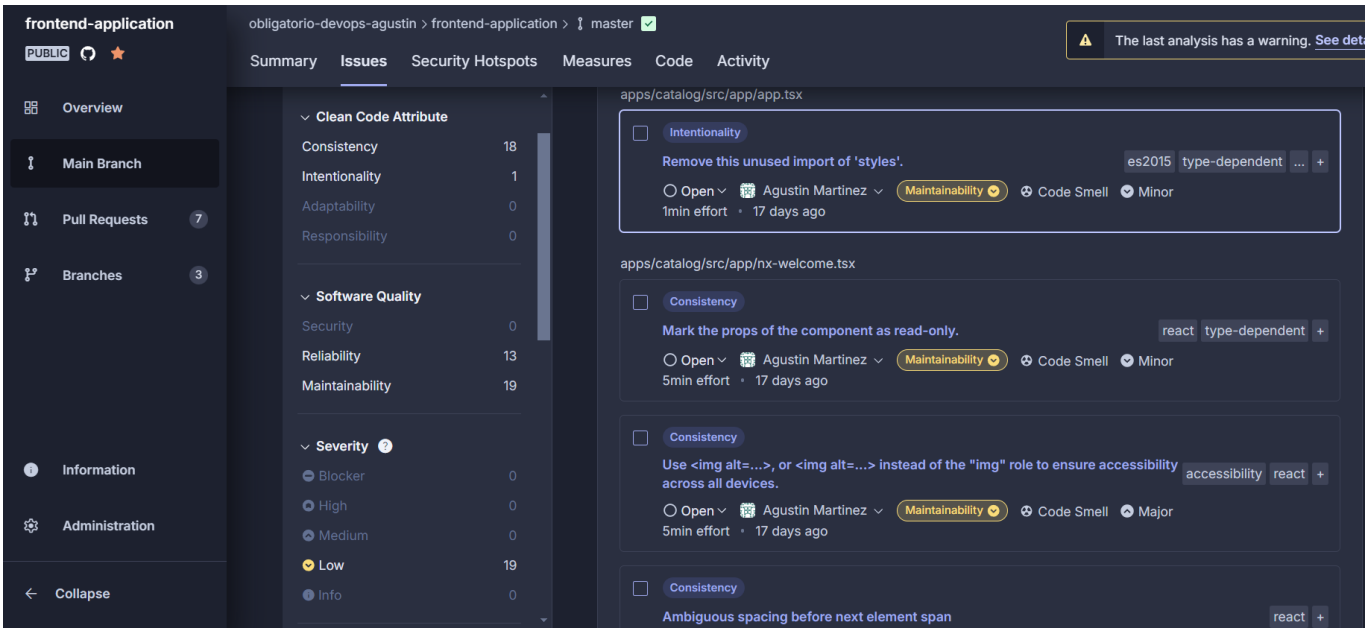
Como herramienta para el análisis del código se decide utilizar SonarCloud, esta es crucial para garantizar la calidad del código, ya que permite identificar de manera automatizada errores, vulnerabilidades de seguridad y problemas de mantenimiento en las aplicaciones. Al integrarse en los flujos de CI/CD, asegura que el código cumpla con estándares de calidad antes de ser desplegado.

Para esta solución, se crearon cinco repositorios de código en SonarCloud, uno por cada microservicio además de uno para la aplicación frontend. También se realizaron configuraciones sobre estos para asegurar que todas las ramas que se van a estar analizando (master, staging y dev) sean consideradas como "long-lived branches". SonarCloud, al entender que estas ramas son long-lived analiza todo el código de la rama y no solo el código nuevo, además ofrece métricas completas sobre el estado general de la cálidad del código en la rama, por ejemplo, cobertura, vulnerabilidades, duplicación y mantenimiento.

Resultado análisis frontend

Los resultados del análisis de la aplicación frontend fueron los siguientes:





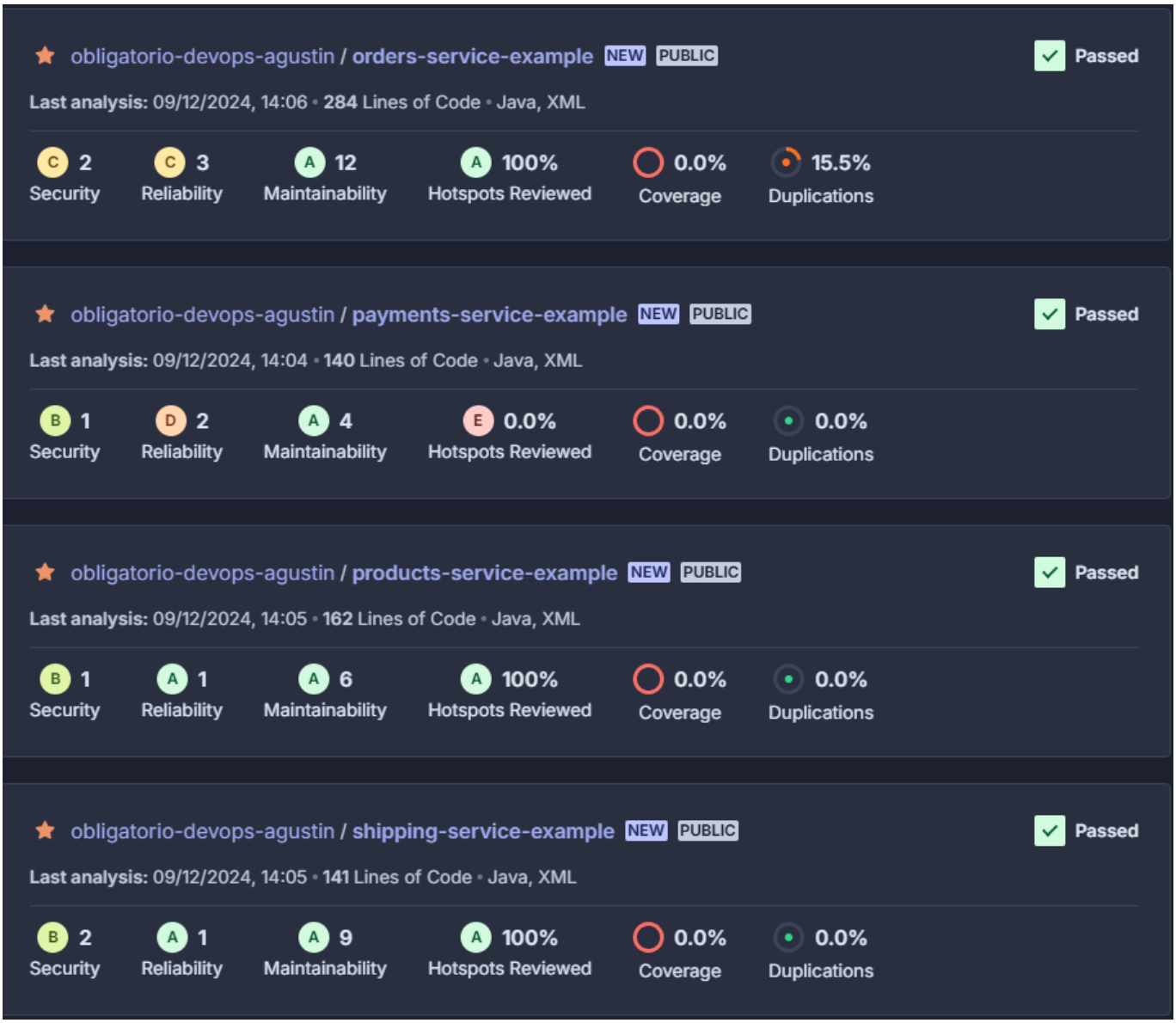
Como se puede observar en las imágenes, respecto a incidentes de seguridad, la aplicación frontend no tiene ninguno lo cual es un aspecto muy positivo.

Por otra parte, si miramos en detalle los problemas que se encuentran en esta, la severidad de los mismos es baja y tienen una calificación de "A" por lo que la calidad del código es muy buena. Los problemas encontrados están relacionados con el espaciado del código y propiedades que no se usan. Este tipo de problemas puede afectar tanto la confiabilidad como el mantenimiento del código, aunque al tener una severidad baja, el impacto es mínimo.

Podemos observar que el proyecto tiene una calificación en "Quality Gate" de "Passed", esto singifica que, según ciertas métricas, el código está listo para desplegarse en producción.

Resultado análisis backend

Los resultados del análisis de los microservicios fueron los siguientes:

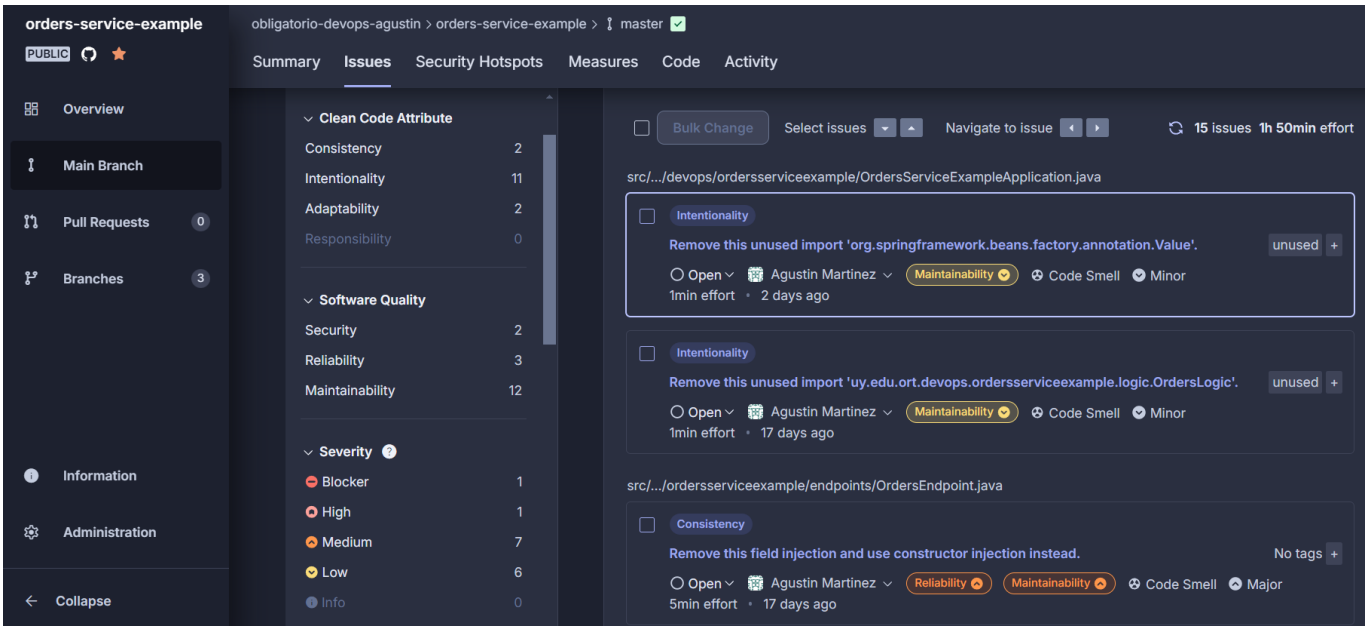


Como primera observación, se puede destacar que todos los microservicios tienen una calificación en "Quality Gate" de "Passed", por lo que todos están listos para ser desplegados en producción en lo que respecta a calidad del código.

También vemos que todos los microservicios tienen algún problema relacionado con la seguridad. Si bien la calificación de estos errores está entre "B" y "C", siendo "B" buena y "C" aceptable, se recomienda prestar atención a estos problemas y dedicar tiempo a resolver los mismos para tener un código libre de problemas relacionados a la seguridad del mismo.

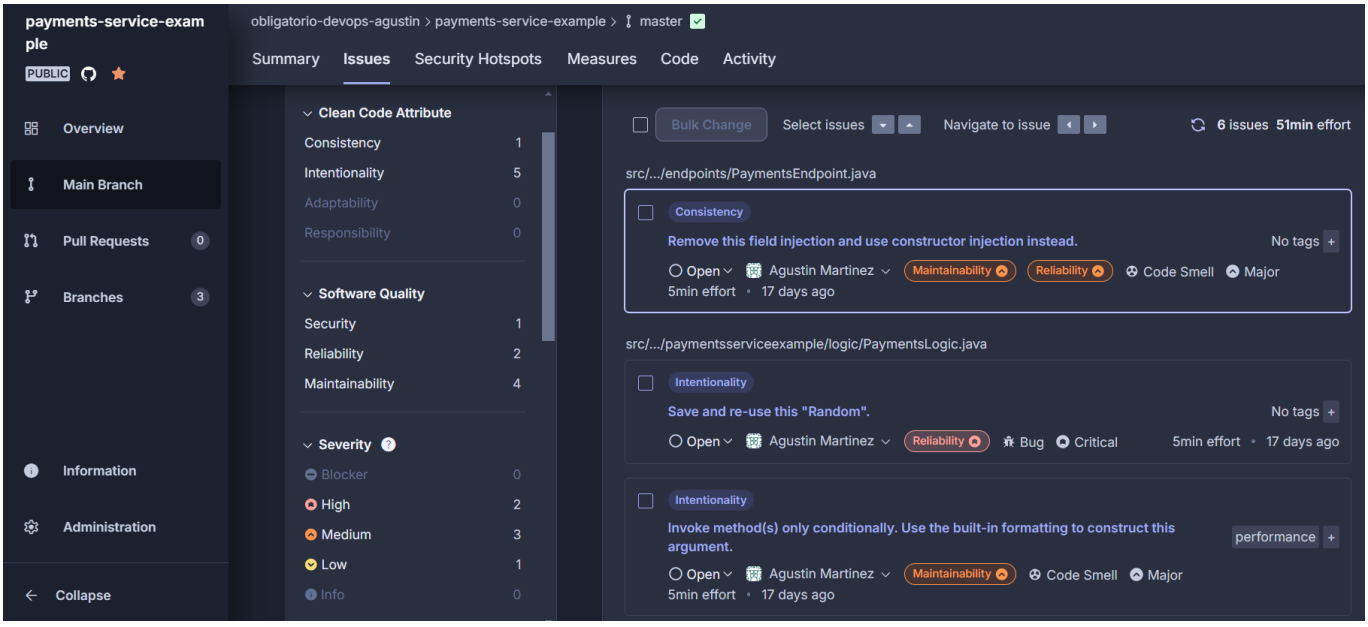
Si vamos al detalle sobre el análisis de cada servicio podemos observar lo siguiente:

Orders



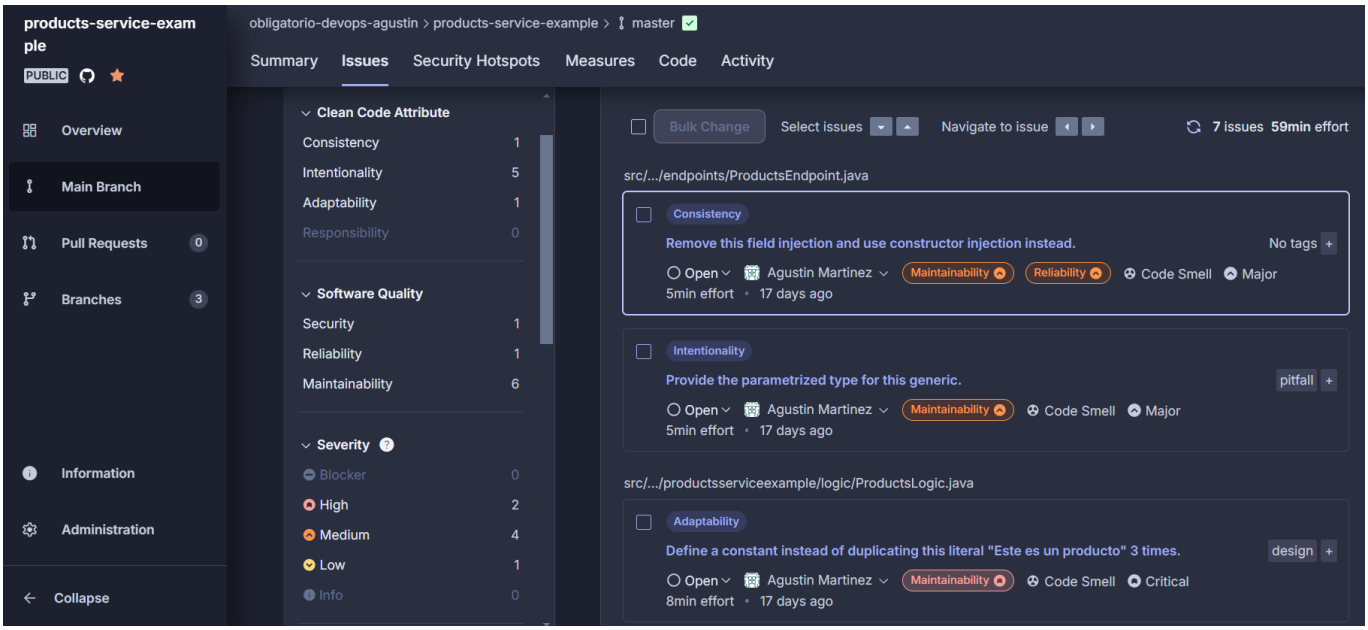
En el caso del microservicio Orders, se encuentran problemas que afectan varias categorías. Aunque la mayoría tienen un bajo impacto en la calidad del código, se encuentra un error con severidad bloqueante y otro con severidad alta. La recomendación es atacar estos dos problemas en particular, de manera inmediata, para así poder remediarlos y lograr una solución de mayor calidad para desplegar en producción.

Payments



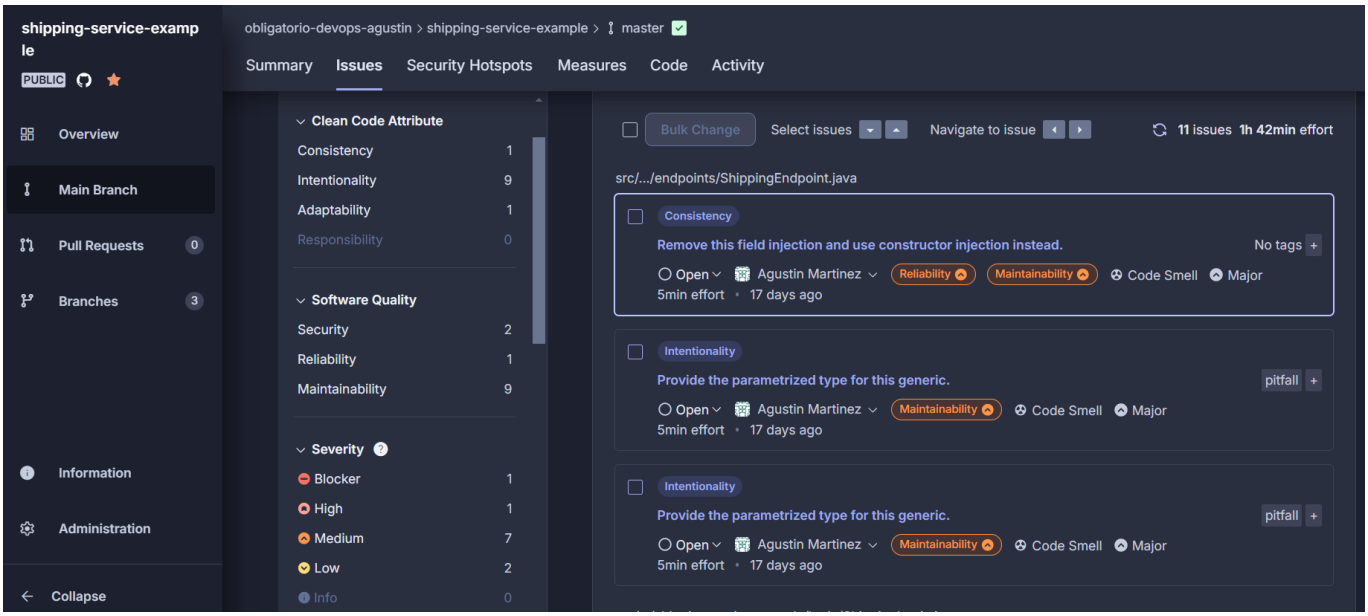
En el caso del microservicio Payments, nuevamente se encuentran distintos problemas que afectan varias categorías. La mayoría tienen bajo impacto en lo que respecta a cálidad de código, pero en particular hay dos problemas con una severidad alta que afectan tanto la confiabilidad como el mantenimiento de la solución. Resolver estos errores debería ser prioridad para el equipo.

Products



Al igual que en el anterior, para el microservicio de products se encuentran dos problemas con severidad alta. Estos problemas tienen un alto impacto en la mantenibilidad de la solución. Nuevamente se recomienda enfocarse en estos problemas encontrados para lograr encontrar una solución para los mismos.

Shipping



En el microservicio de shipping se encuentra un problema bloqueante y uno alto, los cuales afectan directamente la mantenibilidad del código. Al igual que en el caso del microservicio orders, se recomienda atacar estos dos problemas de inmediato para lograr remediarlos y desplegar el microservicio en producción sin estos.
