Reporte técnico: Proyecto final de Sistemas Operativos y Laboratorio

1. Información del Proyecto

- **Título del Proyecto:** Arquitectura de Microservicios para Telconova Supports Suite
- **Curso/Materia:** Sistemas Operativos
- **Integrantes:**
  - Ricardo Contreras Garzón (ricardo.contreras1@udea.edu.co)
  - Juan Felipe Escobar Rendón (juan.escobar15@udea.edu.co)
  - Valentina Muñoz Rincón (valentina.munozr1@udea.edu.co)
- **Fecha de Entrega:** 17 julio 2025

2. Introducción

2.1. Objetivo del Proyecto

Desarrollar, implementar y desplegar una arquitectura completa de microservicios en Amazon Web Services aplicada a TelcoNova SupportSuite, que demuestre la implementación práctica y análisis cuantitativo de conceptos fundamentales de Sistemas Operativos, incluyendo gestión distribuida de procesos, comunicación interprocesos avanzada, virtualización mediante contenedores, scheduling inteligente, y monitorización integral de recursos del sistema.

2.2. Motivación y Justificación

Explique por qué eligió este proyecto o tema. ¿Cuál es su relevancia en el contexto de los sistemas operativos? ¿Por qué es interesante o importante?

La elección de este proyecto surge a raíz de un proyecto académico de la Universidad de Antioquia y a su vez de la necesidad actual de las empresas, de poder contar con sistemas de soporte técnico modernos, escalables y altamente disponibles. Las compañías como TelcoNova enfrentan diariamente retos significativos en la gestión de miles de órdenes de trabajo, la coordinación de técnicos en campo y la comunicación eficiente con los clientes. La adopción de una arquitectura de microservicios no solo responde a una demanda tecnológica, sino también a una exigencia del negocio para mejorar la eficiencia operativa, optimizar recursos y ofrecer una experiencia de usuario superior.

Desde la perspectiva académica y técnica, el proyecto tiene relevancia porque materializa conceptos fundamentales de los sistemas operativos en un entorno práctico y de aprendizaje. A través de la implementación en AWS y el uso de contenedores, se abordan principios clave como

la gestión de procesos, la comunicación interprocesos avanzada (IPC), el scheduling inteligente, y la virtualización a nivel de sistema operativo.

La arquitectura de microservicios, utilizando contenedores y basada en eventos (Event-Driven Architecture), constituye actualmente el estándar en grandes empresas tecnológicas como Netflix, Amazon y Uber. Adoptar este paradigma no solo permite una mayor escalabilidad, sino que también fomenta el desarrollo independiente de servicios, el aislamiento de fallos y la posibilidad de desplegar nuevas funcionalidades de forma ágil y segura. Esto lo convierte en un tema sumamente relevante y atractivo tanto en el ámbito profesional como en el académico.

Además, la integración y el despliegue continuo (CI/CD), y el uso de herramientas avanzadas de observabilidad (Prometheus y Grafana), fortalecen el proyecto como una oportunidad de estudio. Se demuestra la importancia de monitorizar y analizar el rendimiento en tiempo real para garantizar la estabilidad y eficiencia de las aplicaciones.

Este proyecto resulta interesante y relevante porque representa un puente entre la teoría y la práctica, mostrando cómo los conceptos técnicos se transforman para resolver problemas de la industria, contribuyendo así al desarrollo de sistemas más robustos, flexibles y alineados con las necesidades del mundo digital actual.

2.3. Alcance del Proyecto

Incluye:

- Implementación de 3 microservicios contenerizados (Auth, WorkOrder, Tracking) con Spring Boot.
- Comunicación síncrona mediante APIs REST (Auth/Tracking) y GraphQL (WorkOrder).
- Autenticación JWT con RBAC y validaciones de seguridad centralizadas.
- Gestión de recursos mediante Docker (namespaces/cgroups) y ECR
- Pipeline CI/CD básico con GitHub Actions para construcción de imágenes Docker y push al ECR.
- Despliegue de los microservicios en AWS ECS.
- Uso de Parameter Store para almacenar variables de entorno.
- Uso de almacenamiento para archivos Binarios (Imágenes, documentos, etc.).
- Monitoreo con Prometheus, CloudWatch y Grafana.
- Bases de datos individuales para cada microservicio en instancias RDS

Excluye:

- Comunicación asincrónica con Amazon SNS.
- Alta disponibilidad multirregión y auto-scaling en la nube.
- Integración con sistemas externos (ERP, CRM)

3. Marco Teórico / Conceptos Fundamentales

Arquitecturas de Software y Microservicios

Evolución Arquitectónica

La arquitectura de software ha evolucionado desde sistemas monolíticos hacia patrones distribuidos. Richardson (2018) identifica que las arquitecturas monolíticas presentan limitaciones fundamentales de escalabilidad, donde el acoplamiento estrecho entre componentes genera cuellos de botella operacionales y tecnológicos.

Los microservicios emergen como patrón arquitectónico que descompone aplicaciones en servicios pequeños, independientemente desplegables, comunicándose via APIs bien definidas (Newman, 2021). Esta aproximación permite:

- **Escalabilidad independiente:** Cada servicio escala según demanda específica.
- **Diversidad tecnológica:** Selección de stack tecnológico óptimo por servicio.
- **Aislamiento de fallos:** Fallas localizadas no propagan al sistema completo.
- **Desarrollo paralelo:** Equipos independientes desarrollan servicios autónomos.

Event-Driven Architecture (EDA)

Michelson (2006) define EDA como patrón donde servicios reaccionan a eventos significativos del negocio, promoviendo acoplamiento temporal débil. Esta arquitectura facilita:

- **Comunicación asíncrona:** Servicios procesan eventos independientemente.
- **Escalabilidad elástica:** Procesamiento distribuido de carga variable.
- **Resiliencia:** Tolerancia a fallos mediante event sourcing y replay capabilities.

Computación en la Nube y Amazon Web Services

Paradigma Cloud Computing

Mell & Grance (2011) del NIST definen cloud computing como modelo para acceso ubicuo, conveniente y bajo demanda a recursos computacionales configurables. Las características esenciales incluyen:

- **Auto-servicio bajo demanda:** Provisioning automático sin intervención humana.
- **Acceso amplio via red:** Disponibilidad mediante protocolos estándar.
- **Pooling de recursos:** Recursos multi-tenant con asignación dinámica.
- **Elasticidad rápida:** Escalamiento automático según demanda.
- **Servicio medido:** Monitoreo, control y reporte de utilización.

Amazon Web Services (AWS)

AWS se estableció como líder del mercado cloud proporcionando más de 200 servicios completamente gestionados. Barr & Cabrera (2019) destacan servicios fundamentales relevantes al proyecto:

Amazon Elastic Container Service (ECS): Servicio de orquestación de contenedores completamente gestionado que elimina necesidad de instalar y operar infraestructura de clustering. Proporciona scheduling inteligente, auto-scaling e integración nativa con servicios AWS.

Amazon API Gateway: Servicio completamente gestionado para crear, publicar, mantener, monitorear y asegurar APIs REST y WebSocket. Implementa throttling, caching, autenticación y documentación automática.

Amazon RDS: Servicio de base de datos relacional gestionado que automatiza tareas administrativas como provisioning de hardware, setup de database, patching y backups.

Contenedores y Virtualización

Virtualización a Nivel de Sistema Operativo

Soltesz et al. (2007) distinguen contenedores como forma de virtualización de SO que permite múltiples instancias aisladas de userspace ejecutándose en kernel único. Esta aproximación ofrece ventajas sobre virtualización tradicional:

- **Overhead reducido:** Eliminación de hypervisor reduce latencia y consumo de recursos.
- **Densidad superior:** Mayor número de instancias por host físico.
- **Startup rápido:** Inicialización en milisegundos vs minutos de VMs tradicionales.

Docker como Plataforma de Contenedores

Merkel (2014) analiza Docker como plataforma que democratiza contenedores mediante abstracciones simples. Docker Engine implementa:

- **Layered filesystem:** Optimización de almacenamiento mediante copy-on-write.
- **Process isolation:** Namespaces y cgroups para aislamiento de recursos.
- **Portabilidad:** "Build once, run anywhere" mediante imágenes inmutables.

Comunicación Interprocesos en Sistemas Distribuidos

Patrones de Comunicación

Tanenbaum & Van Steen (2016) categorizan comunicación en sistemas distribuidos:

Comunicación Síncrona: Client bloquea hasta recibir respuesta del servidor. REST APIs implementan este patrón proporcionando semantics request-response familiares, pero introduciendo acoplamiento temporal.

Comunicación Asíncrona: Sender continúa procesamiento sin esperar respuesta inmediata. Message queues y event streams implementan este patrón, reduciendo acoplamiento, pero complicando manejo de errores y consistencia.

Message Queues y Event Streaming

Kleppmann (2017) analiza sistemas de mensajería como backbone de arquitecturas distribuidas. Amazon SNS implementa patrón publish-subscribe donde:

- **Publishers** emiten eventos sin conocimiento de subscribers.
- **Topics** actúan como canales de distribución.
- **Subscribers** reciben eventos relevantes asíncronamente.
- **Delivery guarantees** aseguran procesamiento confiable.

Monitorización y Observabilidad

Observabilidad en Sistemas Distribuidos

Majors et al. (2022) definen observabilidad como capacidad de inferir estado interno de sistema mediante outputs externos. En sistemas distribuidos, observabilidad comprende:

- **Metrics:** Agregaciones numéricas de datos a lo largo del tiempo.
- **Logs:** Registros discretos de eventos con timestamps.
- **Traces:** Representación de requests atravesando múltiples servicios.

Prometheus y Grafana

Prometheus implementa modelo pull-based para recolección de métricas, proporcionando (Godard, 2019):

- **Time-series database:** Almacenamiento eficiente de métricas temporales.
- **Query language (PromQL):** DSL para análisis y alerting.
- **Service discovery:** Descubrimiento automático de targets.

Grafana complementa Prometheus proporcionando visualización avanzada y dashboards interactivos.

Integración Continua y Despliegue Continuo (CI/CD)

DevOps y Automatización

Kim et al. (2016) en "The DevOps Handbook" establecen que CI/CD elimina toil operacional mediante automatización. GitHub Actions implementa CI/CD nativo en plataforma de desarrollo, proporcionando:

- **Workflow automation:** Pipelines declarativos vía YAML.
- **Matrix builds:** Ejecución paralela en múltiples ambientes.
- **Secrets management:** Manejo seguro de credenciales.
- **Ecosystem integration:** Conectores nativos con servicios cloud.

Relación con Sistemas Operativos

Gestión de Procesos Distribuidos

Los microservicios representan evolución natural de procesos de SO tradicionales. Silberschatz et al. (2018) establecen que procesos requieren:

- **Process Control Block (PCB):** Metadatos de estado y recursos.
- **Scheduling:** Asignación de CPU time.
- **Synchronization:** Coordinación entre procesos concurrentes.

En arquitecturas contenerizadas, ECS proporciona análogos distribuidos:

- **Task Definition:** Especificación declarativa análoga a PCB.
- **Service Mesh:** Coordinación de comunicación interprocesos.

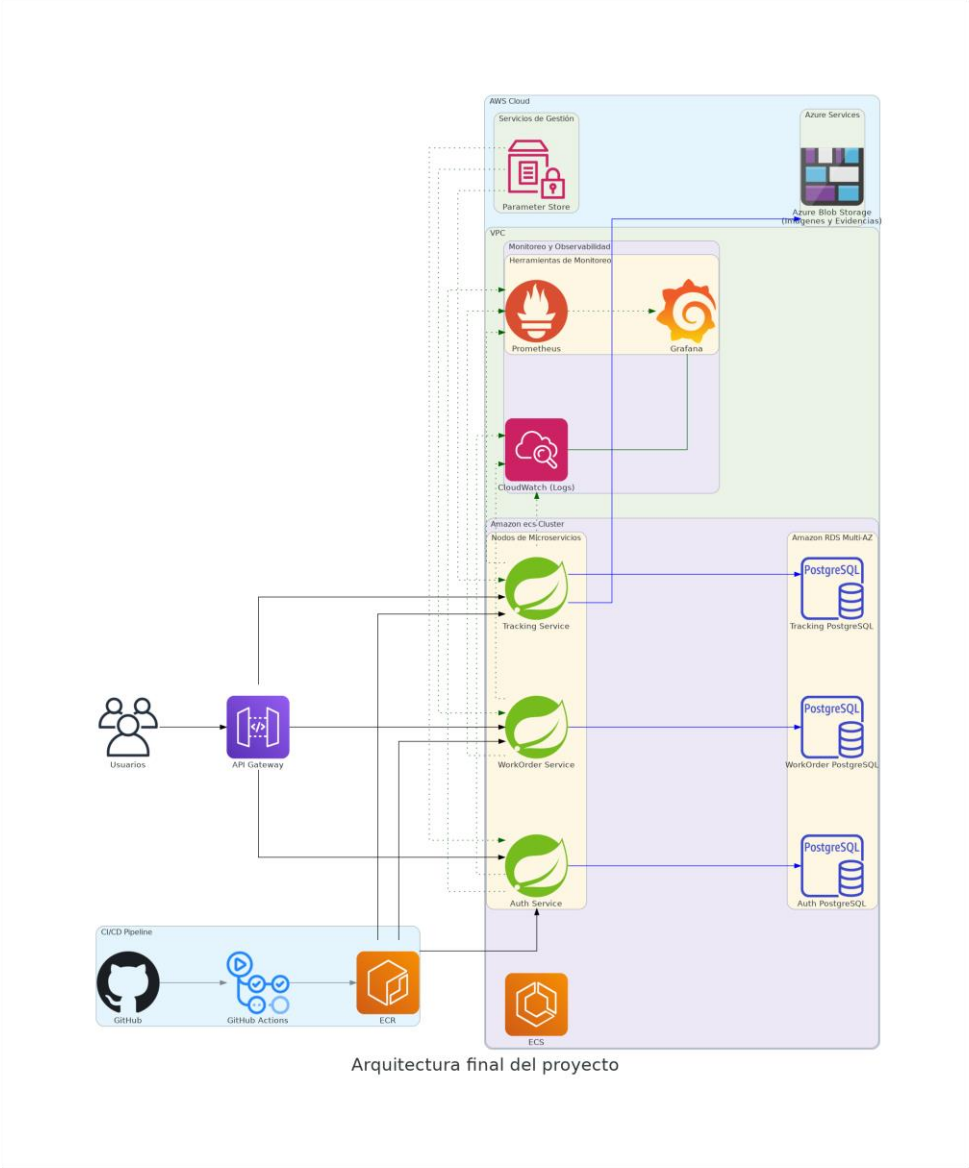
Virtualización y Aislamiento de Recursos

Contenedores implementan virtualización de SO mediante kernel features:

- **Namespaces:** Aislamiento de system resources (PID, network, filesystem).
- **Control Groups (cgroups):** Limitación y accounting de recursos.
- **Copy-on-Write filesystems:** Optimización de storage y memoria.

4. Diseño e Implementación

4.1. Diseño de la Solución



Arquitectura final del proyecto

Decisiones Clave:

- Uso de GraphQL para un microservicio:
  - Se implementó GraphQL en WorkOrder-Service para consultas flexibles (ej: `query { ordenes(estado: "ASIGNADA") }`).
    - Justificación: Facilitaba devolver los datos necesarios según lo que necesite el Frontend, ya que este era el microservicio más grande.
- Aislamiento con Contenedores:
  - Cada servicio en contenedor Docker con usuario no root por defecto y endpoint de salud del contenedor. Todo esto para darle seguridad a nuestras aplicaciones y facilitar el monitoreo de los servicios.

- Monitoreo de Recursos:
    - Prometheus recolecta métricas de SO (CPU, memoria) mediante endpoints Spring Actuator.
    - Grafana visualiza datos en dashboards (ej: uso de CPU por contenedor).
- Uso de Azure Blob Storage en vez de S3.
    - Se implementó para el almacenamiento Azure Blob Storage por su capa gratuita de almacenamiento, con fines de reducir costos.
- Se uso JWT como técnica de seguridad.
    - Esto con el fin de generar microservicios stateless, disminuyendo el tiempo de respuesta y reduciendo la carga a los servidores, además de poner capas por roles a los endpoints importantes por rol.
- Se implementa un APIGATEWAY
    - Por el número considerable de endpoints y por un sistema escalable en microservicios y funcionalidades.
- Uso de CI/CD
    - Con el fin de tener las últimas versiones disponibles lo más pronto posible, una vez el código pasara por todo tipo de revisiones.
- Almacenamiento de variables de entorno centralizadas
    - Esto para manejar todas la variables y secretos en un mismo lugar centralizado.
- Base de datos independiente para cada microservicio.
    - Esto con el fin de no sobre cargar una sola base de datos y evitar la alta dependencia entre proyectos.

4.2. Tecnologías y Herramientas

Plataforma Cloud, Servicios AWS y AZURE

- **Amazon ECS.**
- **Amazon ECR.**
- **Amazon API Gateway.**
- **Amazon RDS PostgreSQL.**
- **Azure Blob Storage.**
- **Parameter Store.**
- **Amazon CloudWatch.**
- **Prometheus + Grafana.**

Desarrollo

- **Java 21 + Spring Boot.**
- **Docker y Docker Compose**
- **GitHub Actions.**
- **Devcontainers**
- **Visual Studio Code**
- **Codespaces**

APIs:

- **REST**
- **GrapgQL**

Seguridad:

- **JWT**
- **RBAC**
- **Spring Security**

4.3. Detalles de Implementación

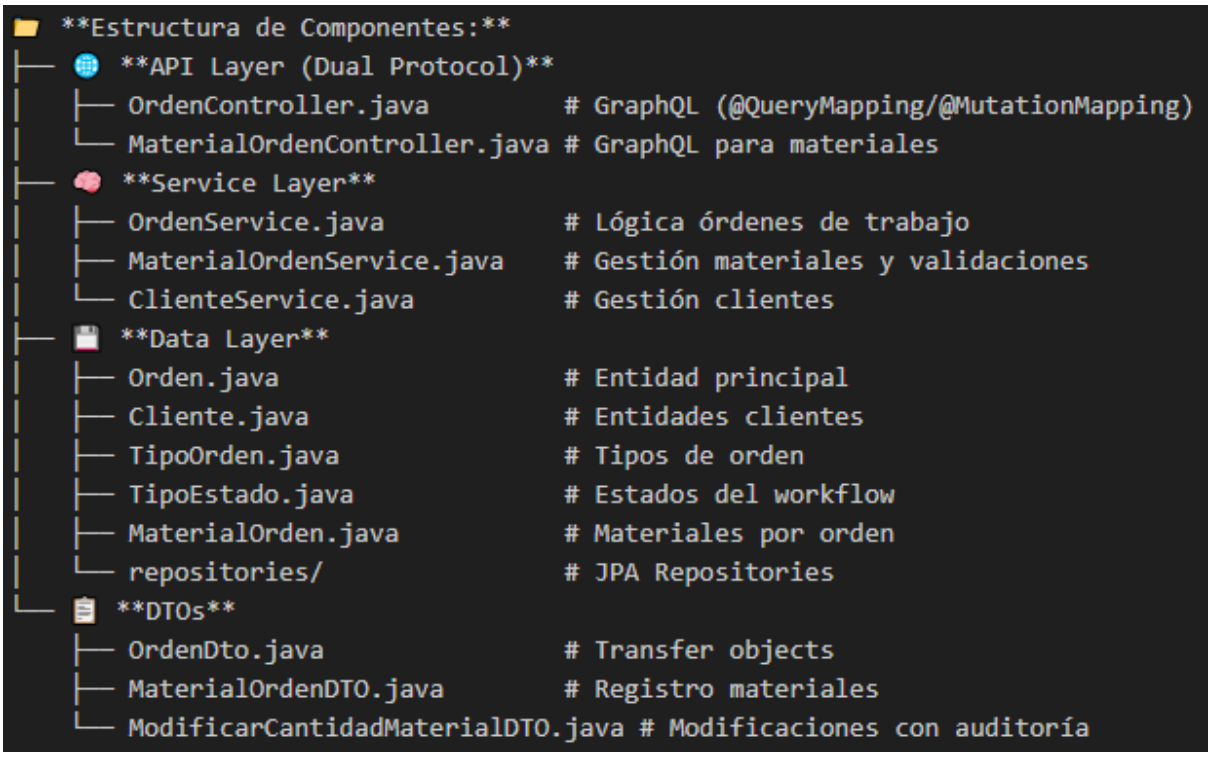
- **Estructura Servicio de Autenticación:**

El servicio de autenticación gestiona la seguridad, validación de tokens JWT y manejo de usuarios en el sistema.

```
📁 **Estructura de Componentes:**
├── 🔐 **Security Layer**
│   ├── JwtAuthFilter.java          # Filtro validación JWT
│   ├── JwtUtils.java               # Generación/validación tokens
│   └── SecurityConfig.java         # Configuración Spring Security
├── 🌐 **API Layer**
│   └── AuthController.java         # REST: /api/v1/auth/*
├── 🧠 **Service Layer**
│   ├── AuthService.java            # Lógica autenticación
│   ├── UserService.java            # Gestión usuarios/roles
│   └── JwtService.java             # Operaciones JWT
├── 💾 **Data Layer**
│   ├── User.java                   # Entidad Usuario
│   ├── Role.java                   # Entidad Rol
│   └── UserRepository.java         # Acceso a datos
└── 📋 **DTOs**
    ├── LoginRequest.java           # Request autenticación
    ├── AuthResponse.java           # Response con token
    └── UserRegistrationDto.java    # Registro usuarios
```

- **Estructura Servicio de Órdenes**

Este servicio gestiona el ciclo de vida de las órdenes de trabajo, incluyendo materiales asociados y estados del workflow.



- **Estructura Servicio de Seguimiento de Órdenes**

Este servicio se encarga del seguimiento de avances técnicos, gestión de evidencias y auditoría de cambios en las órdenes.

```
**Estructura de Componentes:**
├── **API Layer**
│   ├── AvancesController.java       # REST: /api/tracking/avances/*
│   └── EvidenciaController.java     # REST: /api/tracking/evidencias/*
├── **Service Layer**
│   ├── AvanceServiceImpl.java       # Lógica avances técnicos
│   ├── EvidenciaServiceImpl.java    # Validación y almacenamiento archivos
│   └── storage/
│       └── StorageService.java      # Strategy pattern (S3/Azure/MinIO)
├── **Data Layer**
│   ├── Avance.java                  # Avances técnicos
│   ├── Evidencia.java               # Metadatos archivos adjuntos
│   ├── AvanceEdicion.java           # Auditoría de cambios
│   └── repositories/                # JPA Repositories
├── **Security Layer**
│   └── SecurityConfig.java          # Validación JWT + roles
└── **DTOs**
    ├── AvanceDto.java               # Transfer objects
    ├── EvidenciaDto.java            # Metadata evidencias
    └── AvanceEdicionDto.java        # Historial cambios
```

- **Configuración de seguridad en Docker.**

```
RUN addgroup --system javauser && adduser --system --ingroup javauser
javauser
RUN mkdir -p /app/logs && chown -R javauser:javauser /app/logs
USER javauser
```

- **Configuración de salud del contenedor.**

```
HEALTHCHECK --interval=30s --timeout=3s \
  CMD wget -q --spider http://localhost:8080/actuator/health || exit 1
```

- **Validación de Estado de orden**

```
public Orden cambiarEstadoOrdenOrThrow(Long id, String nuevoEstado) {
  Orden orden = ordenRepository.findById(id)
    .orElseThrow(() -> new RuntimeException("Orden no encontrada"));

  if (orden.getEstado() != null && "Finalizada".equalsIgnoreCase(orden.getEstado().getNombre())) {
    throw new RuntimeException("No se puede modificar una orden finalizada");
  }

  TipoEstado tipoEstado = tipoEstadoRepository.findByNombre(nuevoEstado)
    .orElseThrow(() -> new RuntimeException("Estado no encontrado"));
```

```
    orden.setEstado(tipoEstado);
    return ordenRepository.save(orden);
}
```

- **Validación de Avances**

```
@Service
@Transactional
public class EvidenciaServiceImpl implements EvidenciaService {

    private static final List<String> ALLOWED_CONTENT_TYPES =
            Arrays.asList("image/jpeg", "image/png", "image/gif", "application/pdf", "text/plain");
    private static final long MAX_FILE_SIZE = 10 * 1024 * 1024; // 10MB
    private static final int MAX_FILES_PER_AVANCE = 3;

    /**
     * Valida que el archivo cumpla con las restricciones
     */
    private void validateFile(MultipartFile file) throws InvalidFileException {
        // Verificar si el archivo está vacío
        if (file.isEmpty()) {
            throw new InvalidFileException("No se puede subir un archivo vacío");
        }

        // Verificar tipo de archivo
        String contentType = file.getContentType();
        if (contentType == null || !ALLOWED_CONTENT_TYPES.contains(contentType)) {
            throw new InvalidFileException("Tipo de archivo no permitido: " + contentType);
        }

        // Verificar tamaño
        if (file.getSize() > MAX_FILE_SIZE) {
            throw new InvalidFileException(
                    String.format("El tamaño del archivo excede el máximo permitido de %dMB",
                            MAX_FILE_SIZE / (1024 * 1024)));
        }
    }
}
```

- **Workflow de Github action**

```
name: Build and Push to ECR
```

on: push: branches: [main, develop, featere/aws-registry] tags: ['v*'] pull_request: branches: [main] workflow_dispatch:

env: AWS_REGION: ${{ secrets.AWS_REGION }} ECR_REGISTRY: ${{ secrets.ECR_REGISTRY }} ECR_REPOSITORY: ${{ secrets.ECR_REPOSITORY }}

jobs: build-and-push: runs-on: ubuntu-latest

```
steps:
-                      name:                       Checkout                       code
  uses:                                                          actions/checkout@v4

-              name:              Configure              AWS              credentials
  uses:                                      aws-actions/configure-aws-credentials@v4
  with:
    aws-access-key-id:           ${{           secrets.AWS_ACCESS_KEY_ID           }}
    aws-secret-access-key:        ${{        secrets.AWS_SECRET_ACCESS_KEY        }}
    aws-region:                                                          us-east-1

-          name:          Login          to          Amazon          ECR          Public
  run:                                                                             |
    aws ecr-public get-login-password --region us-east-1 | docker login --username
AWS                     --password-stdin                      public.ecr.aws/a2c8l1h5

-                name:                Build                Docker                image
  run:                                                                             |
    docker  build  -t  public.ecr.aws/a2c8l1h5/auth-service-so:${{  github.sha  }}  .
    docker    tag    public.ecr.aws/a2c8l1h5/auth-service-so:${{    github.sha    }}
public.ecr.aws/a2c8l1h5/auth-service-so:latest

-                name:                Push                Docker                image
  run:                                                                             |
    docker    push    public.ecr.aws/a2c8l1h5/auth-service-so:${{    github.sha    }}
    docker            push            public.ecr.aws/a2c8l1h5/auth-service-so:latest

-                name:                Output                image                details
  run:                                                                             |
    echo  "Image  pushed:  public.ecr.aws/a2c8l1h5/auth-service-so:${{  github.sha  }}"
    echo     "Image     pushed:     public.ecr.aws/a2c8l1h5/auth-service-so:latest"
```

5. Pruebas y Evaluación

5.1. Metodología de Pruebas

Se siguió el siguiente enfoque:

- **Pruebas manuales**: Uso de Swagger UI (REST), GraphiQL (GraphQL) y Postman.
- **Validación de seguridad**: Endpoints protegidos con JWT y roles RBAC (TECNICO, SUPERVISOR).

5.2. Casos de Prueba y Resultados

ID Caso de prueba	Descripción del caso de prueba	Resultado esperado	Resultado obtenido	Éxito/Fallo
CP-001	Subir evidencia PNG (válida)	HTTP 200 OK	HTTP 200 OK	Éxito
CP-002	Acceder a un endpoints sin permisos	HTTP 403 Forbidden	HTTP 403 Forbidden	Éxito
CP-003	Orden de trabajo, evicencia, avance o usuario no encontrado.	HTTP 404 Not Found	HTTP 404 Not Found	Éxito
CP-004	Acceso sin token JWT	HTTP 401 Unauthorized	HTTP 401 Unauthorized	Éxito

5.3. Problemas Encontrados y Soluciones

Problema 1: Inconsistencia en puertos entre servicios.
Solución: Estandarización en `docker-compose.yml`:
```yaml
services:
  ms-tracking:
    ports: ["8080:8080"]
  ms-auth:
    ports: ["8081:8081"]
```

Problema 2: los actions que se ofrecían preconstruido para subir la imagen de Docker no funcionaba.
Solución: Usar los comandos típicos por consola:

- name: Build Docker image

run: |
docker build -t public.ecr.aws/a2c8l1h5/auth-service-so:${{ github.sha }} .
docker tag public.ecr.aws/a2c8l1h5/auth-service-so:${{ github.sha }} public.ecr.aws/a2c8l1h5/auth-service-so:latest

- name: Push Docker image
  run: |
  docker push public.ecr.aws/a2c8l1h5/auth-service-so:${{ github.sha }}
  docker push public.ecr.aws/a2c8l1h5/auth-service-so:latest

Problema 3: No se podía validar la autenticidad del token con JWT Secret en común. **Solución**: Estandarizar el proceso de encriptación y desencriptación:

```
return Jwts.builder()
        .setClaims(claims)
        .setSubject(user.getUsername())
        .setIssuedAt(now)
        .setExpiration(expiryDate)
        .setId(tokenId)                                  // JWT ID único
        .setIssuer(applicationName)                      // Emisor del token
        .setAudience("telconova-apis")                   // Audiencia prevista
        .setNotBefore(now)                              // No válido antes de ahora
        .signWith(
            Keys.hmacShaKeyFor(jwtSecret.getBytes(StandardCharsets.UTF_8)),
            SignatureAlgorithm.HS512
        )
        .compact();
```

6. Conclusiones

Logros:

1. Implementación funcional de 3 microservicios con gestión autónoma de recursos.
2. Validación práctica de conceptos de SO:
    2.1. Virtualización: Contenedores Docker (namespaces/cgroups) como unidades de aislamiento.
        2.1.1. Gestión de procesos: ECS ≈ scheduler distribuido (Task Definition ≈ PCB).

2.1.2. IPC: Comunicación REST/GraphQL como extensión de IPC en sistemas distribuidos.

3. Pipeline CI/CD básico con GitHub Actions.

7. Trabajo Futuro

1. Implementar Amazon SNS para comunicación de eventos.
2. Migrar a AWS ECS con auto-scaling basado en métricas.
3. Usar Herramientas de código como Infraesctructura (Terraform u otro).
    a. Implementar los otros 4 microservicios que remplazan algunas herramientas de AWS reduciendo costos y dándonos más dominio del negocio.,

8. Referencias

Prometheus Authors. (n.d.). Prometheus Monitoring System. https://prometheus.io/

Grafana Labs. (n.d.). Grafana: The open observability platform. https://grafana.com/

GraphQL Foundation. (n.d.). GraphQL: A query language for your API. https://graphql.org/

Docker Inc. (n.d.). Docker Documentation. https://docs.docker.com/

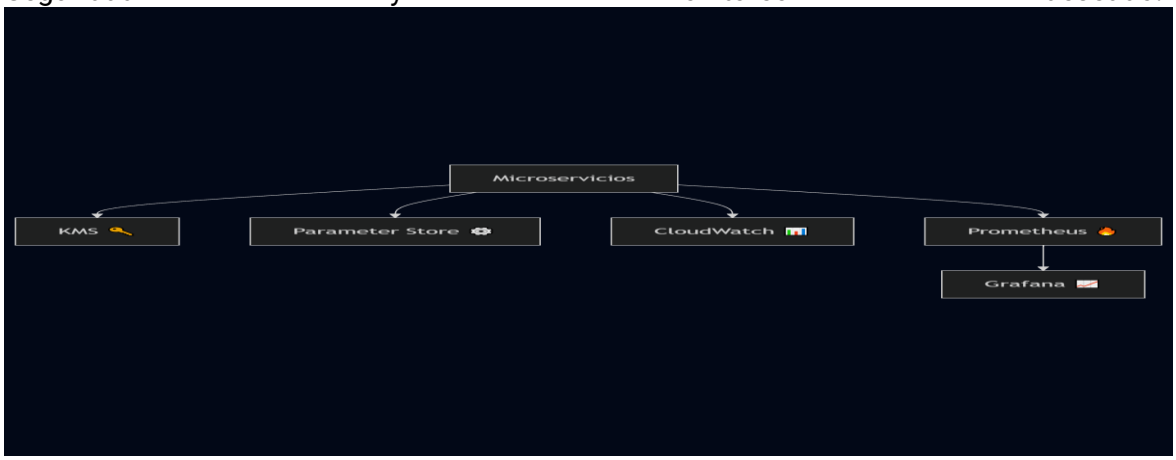
9. Anexos

- Configuración Prometheus a nivel local(prometheus.yml):

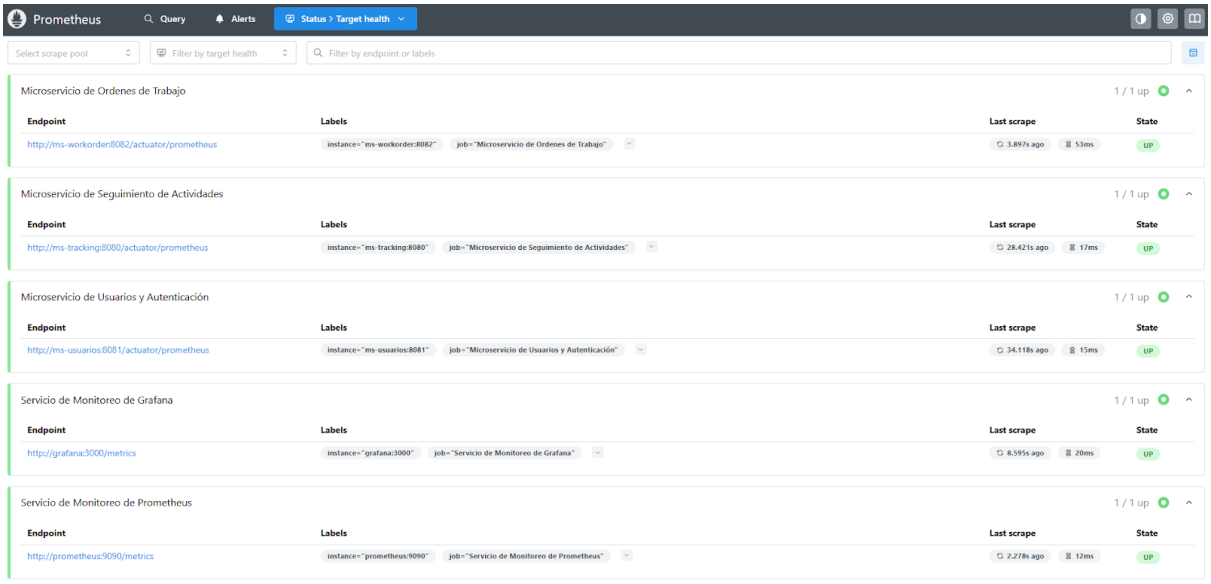
```
scrape_configs:
  - job_name: 'Microservicio de Seguimiento de Actividades'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['ms-tracking:8080']

  - job_name: 'Microservicio de Usuarios y Autenticación'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['ms-usuarios:8081']

  - job_name: 'Microservicio de Ordenes de Trabajo'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['ms-workorder:8082']

  - job_name: 'Servicio de Monitoreo de Prometheus'
```

```
    scrape_interval: 10s
    static_configs:
      - targets: ['prometheus:9090']

  - job_name: "Servicio de Monitoreo de Grafana"
    scrape_interval: 10s
    static_configs:
      - targets: ['grafana:3000']
```

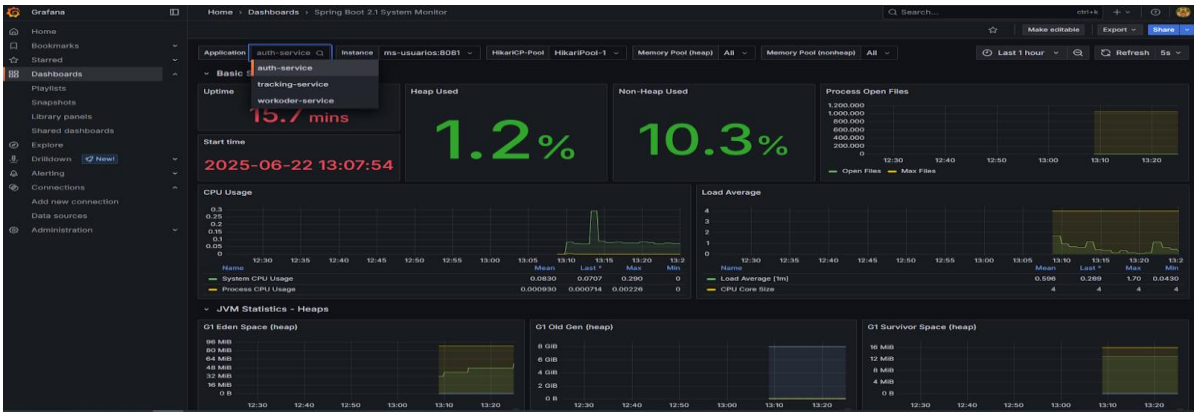
- Seguridad y Monitoreo deseado:



- Prometheus:



- Grafana:



- Repositorios:
  - Servicio de Órdenes: link
  - Servicio de Autenticación: link
  - Servicio de Seguimiento de Órdenes: link