| Propuesta de Rediseño y Migración a Microservicios para la Plataforma de Gestión de Proyectos de LUNARi |
| --- |
|  |

**Integrantes:**

Cristobal Aguila

Angelo Millan

Oscar Muñoz

(insertar nuevo nombre completo)

Desarrollo FullStack 1 DSY1103-0006V

Profesor Tomás Opazo

Mayo, 2025

# 1. Introducción

## 1.1 Contexto de LUNARi

LUNARi es una empresa que se dedica a la mejora y modernización de los sistemas de ventas y bodegaje de los emprendimientos de redes sociales. Dentro de nuestra labor, está el profesionalizar, a través de una página web, la forma en que dichos emprendedores llegan a sus potenciales clientes, permitiendo mejorar sus ventas y con ello hacer crecer a su empresa.

Una tarea recurrente dentro de nuestro ámbito de acción es la unificación de la página web creada con las redes sociales ya creadas por el cliente, permitiendo de este modo el aprovechamiento de dichas redes a modo de publicidad y gancho para generar las ventas online, ayudando con ello a aumentar las ventas, y a no perder la fidelización de los clientes ya establecidos

Desde el momento en que se encarga la creación de una página web el cliente tiene total acceso a ver los avances en ella a través de la página web de LUNARi. Ahí, el cliente puede ir solicitando información respecto a los trabajos hechos, tener acceso a una previsualización de la página web en construcción, así como también pedir cambios en las labores a realizar, y ejecutar algunas pruebas de funcionamiento.

## 1.2 Definición del Problema: Limitaciones del Sistema Actual y Necesidad de Evolución

LUNARi ha experimentado un crecimiento sostenido en el tiempo, consolidando su propuesta de valor para emprendedores minoristas. La demanda por profesionalizar los servicios de venta y presencia digital ha implicado la necesidad de aumentar la dotación de trabajadores y desarrolladores en sus oficinas. Si bien esto representa una muy buena noticia para la empresa y valida su modelo de negocio, también ha dejado en evidencia diversas dificultades y ha expuesto las limitaciones críticas de su infraestructura tecnológica actual.

El crecimiento ha puesto de manifiesto la **insuficiencia de los sistemas usados**, particularmente el sistema central que gestiona el ciclo de vida de los proyectos, desde el seguimiento del desarrollo hasta el monitoreo una vez desplegado el sistema. LUNARi, que incluye el portal del cliente, se ha revelado como **anticuado**. Problemas clave identificados incluyen:

1. **Falta de Escalabilidad y Rendimiento:** Los sistemas actuales carecen de la escalabilidad puntual necesaria, especialmente al iniciar nuevos proyectos o al aumentar el número de clientes cuyas aplicaciones deben ser monitoreadas simultáneamente. El software de seguimiento y monitoreo evidencia **problemas de rendimiento y disponibilidad** a medida que aumenta la carga de trabajo. Esto no solo ralentiza las operaciones internas, sino que, como señala la empresa, "pone en riesgo la integridad de la página web de nuestros clientes, su funcionamiento y seguridad de los datos que almacena".
2. **Dificultades de Mantenimiento:** El software existente se ha vuelto cada vez más difícil de mantener. Su arquitectura monolítica probablemente dificulta la implementación de cambios, la corrección de errores y la adopción de nuevas tecnologías, incrementando los costos y tiempos de desarrollo.
3. **Control de Calidad y Estandarización:** Con más desarrolladores trabajando en múltiples proyectos, surge la necesidad crítica de estandarizar la creación de código, los métodos y las herramientas utilizadas. La falta de un sistema que facilite esta estandarización pone en riesgo el mantenimiento de los altos estándares de calidad que caracterizan a LUNARi.
4. **Gestión de Accesos y Seguridad:** Un equipo más grande implica un menor control inherente sobre el acceso a la información sensible de los clientes. El sistema actual no parece contar con mecanismos robustos y granulares para gestionar permisos de manera eficiente y segura en un entorno de equipo en expansión.

El potencial crecimiento de LUNARi se ve mermado por el sistema tecnológico actual que posee, lo que dificulta agregar talento a la empresa y compromete seriamente la calidad del servicio y sobre todo su sostenibilidad operativa. Es imperativo que se genere un plan de rediseño de la plataforma para abordar así deficiencias y hacer crecer a la empresa.

# 2. Análisis de Requerimientos

## 2.1 Metodología de Recolección de Requerimientos

Con el fin de comprender a cabalidad las necesidades del nuevo sistema, se utilizó una metodología de recolección de requerimientos que se centra en entrevistas con participantes clave que interactuarán con la plataforma. De esta manera se identificaron los siguientes perfiles de usuario: El administrador de la plataforma, los Product Owners del proyecto, los Clientes (representando a emprendedores) y los ingenieros DevOps del proyecto.

Se tuvo como objetivo principal de las entrevistas el obtener una visión detallada de responsabilidades, tareas rutinarias, flujo de trabajo actuales, identificación de puntos críticos y conocer la limitaciones del sistema ya en funcionamiento, para canalizar así las expectativas y requisitos para la nueva solución basada en microservicios. La información obtenida fue fundamental para definir los requerimientos que a continuación se detallan.

## 2.2 Requerimientos Funcionales

### 2.2.1 Gestión de Usuarios (Administrador)

El sistema debe permitir al Administrador gestionar cuentas de todos los usuarios de la plataforma, tanto internos como clientes, incluyendo la creación, actualización, desactivación y eliminación de estas. Esto debe estar acompañado de la configuración de permisos de acceso de cada usuario a las diferentes funcionalidades, asegurando segregación de responsabilidades.

### 2.2.2 Gestión de Proyectos y Recursos (Product Owner)

El Product Owner necesita herramientas para administrar el equipo de LUNARi asignado a cada proyecto. Debe ser capaz de realizar operaciones CRUD sobre recursos de infraestructura cloud asociados al proyecto y gestionar hitos del mismo, lo que requiere de la aprobación tanto del Product Owner como del Cliente para hacer las validaciones pertinentes.

### 2.2.3 Gestión de Pagos e Hitos (Cliente)

El Cliente debe tener la capacidad de revisar el estado de los pagos, autorizarlos y consultar el historial de éstos. Al igual que el Product Owner, podrá realizar operaciones CRUD sobre los hitos del proyecto, con la ya mencionada condición de que ciertas acciones requieran la aprobación conjunta del Cliente y el Product Owner.

### 2.2.4 Gestión de Despliegues y Monitoreo (DevOps)

El encargado DevOps requiere funcionalidades para agendar actualizaciones de software en las aplicaciones de los clientes, así como también monitorear dichas actualizaciones. Necesita la capacidad de gestionar recursos de infraestructura cloud asociados al proyecto y monitorear de manera continua el estado y rendimiento de la aplicación del cliente desplegada, con la funcionalidad de reportería asociada.

### 2.2.5 Portal del Cliente

La plataforma web que el Cliente tiene acceso debe ser capaz de registrar y autenticar a usuarios. Una vez dentro, el Cliente podrá navegar y buscar sus proyectos, filtrándolos por estado. Se podrá hacer una gestión del proyecto en detalle a través de acciones tales como gestionar hitos o solicitar cambios. La plataforma debe permitirle al Cliente el iniciar un nuevo proyecto, consultar historial de acciones, gestionar perfil y solicitar soporte técnico o ayuda.

### 2.2.6 Funcionalidad de Formalización Empresarial

El sistema debe tener una funcionalidad específica para guiar a los clientes en el proceso de formalización legal de sus empresas. Esta guía debe presentarse como un roadmap visual donde cada etapa sea reflejo del estado actual de la empresa del cliente. Para esto el sistema deberá conectarse con servicios externos como el SII y otros organismos pertinentes para verificar el progreso del cliente en cada paso de la formalización.

## 2.3 Requerimientos No Funcionales

### 2.3.1 Rendimiento

El sistema debe responder rápidamente a las interacciones de los usuarios y procesar operaciones de manera eficiente, sin importar la carga con la cual trabaja. Esto incluye tiempos de respuesta del portal, cargas ágiles en gestión de proyectos y recursos, y capacidad del manejo de flujo de datos de monitoreo sin contratiempos.

### 2.3.2 Disponibilidad

La plataforma LUNARi debe estar operativa y accesible la mayor parte del tiempo, apuntando a un alto porcentaje de uptime para garantizar que el negocio opere de manera óptima y se le entregue a los clientes confianza en el sistema.

### 2.3.3 Escalabilidad

El sistema debe ser capaz de crecer para manejar un aumento de usuarios, clientes, proyectos, desarrolladores y/o volumen de datos sin que sea necesario rediseños fundamentales, de manera horizontal o vertical y especialmente para manejar flujos extraordinarios de demanda o inicio de nuevos proyectos de manera eficiente.

### 2.3.4 Mantenibilidad

El sistema deberá ser capaz de aceptar modificaciones, correcciones y la adición de nuevas funcionalidades sin dificultades. El código debe ser claro, documentado, modular y testeable, superando los inconvenientes del sistema monolítico actual

### 2.3.5 Seguridad

El sistema debe garantizar la confidencialidad, integridad y disponibilidad de los datos. Esto implica una serie de funcionalidades clave como lo son la autenticación robusta, autorización a base de roles, protección contra vulnerabilidades comunes, encriptación de datos sensibles y el control de acceso a infraestructura crítica.

### 2.3.6 Usabilidad

El sistema debe poseer una interfaz de usuario intuitiva, fácil de aprender y eficiente de usar. Se debe atener a prácticas UX/UI de manera que la información sea entregada de forma clara y comprensible.

# 3 Análisis del sistema actual

## 3.1 Descripción de la Arquitectura Monolítica Existente

El sistema actual tiene la capacidad de poder llevar a cabo múltiples tareas, y entre ellas vale la pena destacar las siguientes como más importantes:

* Creación de la página web.
* Seguimiento en la creación de la página web por parte de nuestros clientes.
* Mantención de la página web ya creada.
* Seguimiento preventivo por parte de nuestro equipo de la página web.
* Poder generar tickets de solicitudes por parte de nuestros clientes, sean estos respecto a problemas detectados por ellos, solicitudes de cambios de funciones, e incluso los mismos detectados por nuestros ingenieros y técnicos.

Cómo es posible apreciar, son variados los casos en que nuestro sistema puede ser usado. En un principio aquello no acarreaba mayor problema, pues nuestros clientes eran pocos y no todos los proyectos contaban con una extrema complejidad; es después de la pandemia cuando todo esto cambia.

Nuestra empresa comenzó a recibir muchas solicitudes de contratación de servicios, lo cual nos puso contentos, pues veíamos cómo las personas confiaban en nosotros, pero aquello representó un gran desafío en lo que eran nuestros sistemas, nuestra forma de trabajar, nuestra empresa en sí misma.

Se detectaron varios problemas, las cuales pasamos a enumerar a continuación:

## 3.2 Identificación de Puntos Débiles

**La escalabilidad horizontal es limitada:** puesto que en el sistema monolítico todo se trata como una sola unidad, si quisiéramos escalar una parte concreta del sistema (como la parte asociada al seguimiento de las páginas web de nuestros clientes) no podríamos hacerlo “solamente” en ese componente, pues no hay componentes separados: es necesario escalar todo el sistema, acarreando gastos innecesarios, y pérdidas de tiempo, así como posibles problemas en la nueva implementación por incompatibilidades.

**Despliegues frecuentes:** con un equipo en crecimiento y la necesidad de implementar nuevas funcionalidades rápidamente para satisfacer la demanda de los clientes, los despliegues se volverán más frecuentes. En un sistema monolítico, cualquier cambio, incluso uno pequeño, requiere desplegar toda la aplicación. Esto aumenta el riesgo de introducir errores en otras partes del sistema que no fueron modificadas directamente.

**Aumentan los tiempos de despliegue:** la complejidad del sistema hace necesaria realizar pruebas más intensas para evitar los errores mencionados en el punto anterior. La misma complejidad del conjunto hace que resolver posibles errores tome excesivo tiempo.

**Dificultad para aislar fallos:** un problema en una parte del sistema monolítico puede implicar un bloqueo de toda la aplicación. Ello es lo indeseado, pues impediría el correcto funcionamiento de una empresa completa en sus distintas áreas de desarrollo y mantención.

**Problemas con las nuevas tecnologías:**  se haría en exceso dificultoso adoptar lo nuevo del mercado informático, pues es probable que la aplicación deba reescribirse por completo para poder brindar compatibilidad. Bajo este mismo punto es que también se dificulta poder añadir nuevas características a la aplicación, y con ello estancando las posibilidades de la empresa.

**Problemas con los nuevos colaboradores:** tomará tiempo y pérdidas de dinero que los nuevos colaboradores comprendan el funcionamiento de tan intrincado software.

**Problemas con los equipos de desarrollo:** en el sistema monolítico no está clara la frontera entre una característica del sistema y otra, por lo que el orden, la organización y los avances de los desarrollos (y fundamentalmente entre los desarrolladores) se verán afectados.

# 4. Diseño de la nueva Arquitectura de Microservicios

Esta sección detalla la propuesta para la nueva arquitectura del sistema LUNARi, basada en un enfoque de microservicios, y diseñada para satisfacer los requerimientos identificados.

## 4.1 Estrategia de Microservicios Seleccionada

Para el rediseño de la plataforma LUNARi, se ha seleccionado la estrategia de **Descomposición por Capacidad de Negocio.** Esta estrategia consiste en modelar la arquitectura descomponiendo el sistema en servicios que corresponden directamente a las distintas áreas funcionales y responsabilidades del negocio, lo que implica que el análisis de requerimientos realizado se ha de trasladar de forma íntegra a como se debe ajustar la arquitectura. Esto se traduce en confección de servicios como Gestión de Usuarios, Gestión de Proyectos, Facturación, Monitoreo, Soporte, entre otros, lo que permitirá una encapsulación de la lógica de negocio definida. Se escogió esta estructura ya que facilita la comprensión del sistema y asignación de responsabilidades a los distintos equipos de desarrollo.

El uso de la descomposición por capacidad de negocio aborda los problemas ya mencionados que tiene LUNARi actualmente. Permite escalar de forma independiente las áreas con mayor demanda sin que esto afecte el sistema completo; facilita la mantenibilidad ya que los cambios son contenidos en sus respectivos sectores; y se promueve la autonomía de funcionamiento de equipos de trabajo, usando tecnologías adecuadas para cada servicio, permitiendo la incorporación de desarrolladores especializados y el trabajo en paralelo.

## 4.2 Componentes de la Arquitectura

La nueva arquitectura está compuesta por los siguientes microservicios principales y componentes clave:

### 4.2.1 Servicio de Gestión y Autenticación de Usuarios

Este servicio es un componente central y crítico de la arquitectura, ya que actúa como única fuente de identidad de los usuarios y la gestión de accesos en toda la plataforma LUNARi. Sus responsabilidades clave incluyen registro de nuevos usuarios, login, emisión/validación de tokens de sesión, gestión de perfiles de usuario, y la administración de roles y permisos (Control de Acceso Basado en Roles) que permitirán determinar las acciones que pueden realizar actores en otros microservicios.

Para construir este servicio se ha seleccionado el siguiente stack tecnológico:

* **Node.js con Express.js:** Node.js es ideal para manejar una gran cantidad de conexiones concurrentes de manera eficiente. Express.js, como framework minimalista, facilita la creación rápida de APIs REST necesarias para el servicio, aprovechando el ecosistema de middlewares que posee.
* **Autenticación con Passport.js:** Popular librería para manejar autenticación con Node.js. Permite integrar diversas estrategias de autenticación, lo que simplifica la implementación de flujos seguros de login.
* **Base de datos PostgreSQL:** Base de datos relacional robusta, madura y ACID compatible, lo que garantiza la integridad de datos críticos.
* **Seguridad de contraseñas con bcrypt:** Librería estándar para el hashing adaptativo de contraseñas para proteger las credenciales de los usuarios, incluso si la base de datos se viese comprometida.
* **Gestión de sesiones con JWT:** Estándar ideal para manejar sesiones de forma stateless. Cada microservicio podrá verificar el token para autenticar la petición sin necesidad de consultar a este servicio constantemente.

### 4.2.2 Servicio de Gestión de Proyectos

Este servicio permite ser usado como núcleo para la información y la lógica del negocio de los proyectos de los clientes dentro de LUNARi. Tiene la capacidad de gestionar el ciclo de vida de un proyecto desde su génesis hasta su culminación, incluyendo la gestión de los miembros asociados a dicho proyecto. Cabe destacar que es en este servicio en que se implementa la lógica necesaria para la aprobación de los hitos clave por parte del PO y el cliente mismo..

Para implementar este servicio, con su énfasis en datos estructurados y flujos de trabajo definidos, se ha seleccionado el siguiente stack tecnológico:

* **Framework con Django:** Ideal para desarrollo web complejo y basado en datos de forma rápida. Su ORM simplifica la gestión de modelos y relaciones en la base de datos. El sitio de administración incorporado facilita la gestión interna. Promueve código organizado y legible gracias a su estructura y convenciones, y al uso de Python.
* **Base de datos con PostgreSQL:** Adecuada para la naturaleza relacional de los datos (proyectos, hitos, equipos, estados). Ofrece fiabilidad, consistencia transaccional (ACID) para operaciones importantes, capacidad para consultas complejas y perfecta integración con Django.
* **Desarrollo de API con Django Rest Framework:** Herramienta estándar y recomendada para construir APIs RESTful robustas y escalables con Django. Simplifica la serialización, autenticación (JWT), definición de permisos y generación automática de documentación de la API (Swagger/OpenAPI), crucial para microservicios.
* **Autenticación de API’s con drf-simple-jwt:**Permite validar de forma segura los JWT enviados en las solicitudes para integrarse con el sistema de autenticación centralizado (User Service). Facilita la identificación de usuarios y la verificación de sus permisos para las acciones solicitadas.

### 4.2.3 Servicio de Gestión de Recursos

Este servicio actúa como una capa de abstracción y orquestación sobre la infraestructura cloud subyacente utilizada por los proyectos de LUNARi. Este microservicio es crucial para las acciones CRUD en la nube sobre las apps de los clientes. Tiene contacto directo con las API del proveedor de Cloud seleccionado, y mantiene un registro detallado sobre los recursos que le corresponden a cada proyecto.

* **Framework con FastAPI:** Ideal para construir APIs de alto rendimiento gracias a su soporte nativo para programación asíncrona (async/await), crucial para manejar la latencia de las APIs de proveedores cloud. Ofrece validación de datos automática y robusta con Pydantic y generación automática de documentación interactiva (Swagger UI / ReDoc), acelerando el desarrollo y la integración.
* **Base de Datos con MongoDB:** Ofrece la flexibilidad de esquema necesaria para almacenar la información heterogénea y semi-estructurada de los recursos cloud de manera eficiente, sin la rigidez de un esquema relacional. Su capacidad de escalar horizontalmente es beneficiosa para un gran número de recursos gestionados.
* **Interacción con Cloud con la SDK del proveedor cloud boto3:** Esencial para interactuar de forma programática y fiable con los servicios del proveedor cloud, encapsulando la complejidad de las APIs REST subyacentes y proporcionando una interfaz Pythonic para la gestión de recursos.
* **Modelado y validación de datos con Pydantic:** Permite definir modelos de datos claros basados en type hints para las solicitudes/respuestas de la API y para estructurar la información en MongoDB, garantizando la consistencia de los datos.
* **Cliente HTTP asíncrono con hhttpx:** Se utiliza para realizar llamadas asíncronas a APIs externas (si no se usan SDKs o para otras APIs internas), lo que se alinea con la naturaleza asíncrona de FastAPI y mejora el rendimiento.

### 4.2.4 Servicio de Monitoreo

Este servicio es esencial para asegurar la salud operativa y el rendimiento óptimo de las aplicaciones de los clientes desplegadas por LUNARi, así como para monitorear la propia plataforma LUNARi. Recolecta, procesa y almacena métricas de rendimiento y salud de los servicios desplegados. En base a esos datos es que se puede generar los avisos respectivos al detectar problemas.

* **Lenguaje de Programación Go:** Ideal para construir sistemas de red concurrentes y de alto rendimiento gracias a sus goroutines y canales, que permiten manejar eficientemente muchas conexiones y procesar datos en paralelo. Ofrece excelente rendimiento con bajo consumo de recursos al ser compilado a código nativo, crucial para procesar grandes cantidades de métricas. Su despliegue es sencillo (binarios estáticos) y cuenta con un fuerte ecosistema en monitoreo (ej. Prometheus).
* **Base de Datos de Series Temporales en InfluxDB:** Específicamente optimizada para la ingesta masiva, compresión eficiente y consulta rápida de datos de series temporales, que son la naturaleza de los datos de monitoreo. Proporciona un motor de almacenamiento de alto rendimiento y lenguajes de consulta potentes para analizar tendencias y visualizar patrones a lo largo del tiempo. Es compatible con Prometheus para almacenamiento a largo plazo y análisis complejos.

### 4.2.5 Servicio de Facturación

Servicio encargado de administrar todo lo relacionado a lo financiero en LUNARi. Puede manejar la generación de facturas (como las emitidas desde el Servicio de Proyectos al aprobarse un hito), seguimiento de pagos, y los estados del mismo. También puede conservar un historial de todas las transacciones financieras por cliente y proyecto.

* **Framework con Django:** Ofrece un entorno robusto y seguro, ideal para lógica financiera. Su ORM facilita la modelación y gestión de datos financieros. El sitio de administración incorporado es útil para la gestión interna. Las características de seguridad de Django protegen la información de pagos.
* **Base de datos con PostgreSQL:** Garantiza la integridad de los datos financieros gracias a su conformidad ACID, asegurando transacciones atómicas y consistentes. Es fiable y robusto para almacenar de forma segura registros financieros y datos de clientes.
* **Procesamiento asíncrono con Celery:** Permite manejar tareas de facturación que no deben bloquear la respuesta principal o requieren ejecutarse en segundo plano (generación de PDF, envío de correos, comunicación con pasarelas de pago, tareas periódicas), asegurando la capacidad de respuesta y la fiabilidad.
* **Integración con pasarelas de pago (tales como Fintoc, MercadoPago, Flow…):** Permite procesar pagos en línea de forma segura mediante la integración con proveedores relevantes en Chile (ej: Fintoc, Mercadopago, Flow) utilizando sus bibliotecas oficiales y gestionando las credenciales de forma segura.
* **Generación de documentos en PDF con bibliotecas específicas (ReportLab, WeasyPrint):** Permite crear facturas profesionales en formato PDF de forma programática, incluyendo el logo y el formato legal requerido, utilizando bibliotecas como ReportLab o WeasyPrint.
* **Desarrollo de API’s con Django REST Framework:** Permite construir APIs internas estandarizadas y seguras si el servicio necesita exponer funcionalidades para el frontend o para comunicarse con otros servicios (ej: el Servicio de Proyectos).
* **Testeo con Pytest,** pytest-django: Es crucial para verificar exhaustivamente la lógica financiera, la correcta aplicación de pagos, los cambios de estado, la generación de documentos y la interacción con las pasarelas de pago, utilizando pruebas unitarias y de integración.

### 4.2.6 Servicio de Roadmap Legal

Este servicio distintivo de LUNARi tiene como objetivo empoderar a los clientes minoristas guiándolos a través del complejo proceso de formalización legal y tributaria de sus negocios en Chile. A través de nuestros servicios nuestros clientes podrán ver las etapas clave necesarias gracias a un Roadmap personalizado. Tendrá la capacidad de interactuar directamente con sitios de interés, como por ejemplo el SII.

* **Lenguaje principal Python:** Python tiene un ecosistema robusto para interactuar con servicios web externos (con la librería requests) y es líder en web scraping (con BeautifulSoup y lxml) como alternativa si las APIs oficiales son limitadas. Su legibilidad facilita el mantenimiento del código de integración.
* **Base de Datos con PostgreSQL:** Ideal para almacenar la estructura del roadmap y el estado verificado de cada cliente en cada etapa, permitiendo consultas eficientes y manteniendo un historial confiable de su progreso.
* **Procesamiento asíncrono con Celery:** Permite ejecutar las verificaciones de APIs externas o web scraping de forma asíncrona en segundo plano para evitar bloquear el servicio, ya que estas tareas pueden ser lentas o tener límites de frecuencia. Celery Beat permite programar estas verificaciones periódicamente para mantener el estado del roadmap actualizado.
* **Framework API de FastAPI o Flask:** Se necesita una API REST mínima para que el Frontend visualice el estado del roadmap del cliente y, potencialmente, para la comunicación con otros servicios. FastAPI o Flask son frameworks ligeros eficientes para esto, con FastAPI ofreciendo validación automática y documentación OpenAPI.
* **Testeo con Pytest:** Es crucial para probar la lógica interna y la interacción con servicios externos. Pytest es el estándar en Python, y herramientas como responses o VCR.py son necesarias para crear tests fiables y rápidos sin depender de la disponibilidad real de los servicios externos.
* **Logging con el Módulo logging de Python:** Es vital registrar detalladamente las interacciones con servicios externos, los resultados de las verificaciones y cualquier error durante el proceso de scraping o llamado a API, utilizando la configuración estándar de logging de Python.

### 4.2.7 Servicio de Soporte

Dedicado a gestionar las interacciones de soporte entre los clientes y el equipo de LUNARi. Maneja el ciclo de vida de los tickets de soporte (creación, asignación, seguimiento de estado, historial de tickets) y también un chat para comunicaciones en tiempo real.

* **Runtime de Node.js:** Node.js es ideal para aplicaciones con alta carga de I/O y tiempo real como el chat, gracias a su modelo asíncrono y orientado a eventos que permite manejar muchas conexiones WebSocket eficientemente. Express.js se usa complementariamente para construir las APIs REST para la gestión de tickets.
* **Comunicación en tiempo real con WebSockets:** WebSockets es el protocolo estándar para una comunicación bidireccional persistente necesaria para una experiencia de chat fluida. Socket.IO simplifica el uso de WebSockets con funcionalidades como reconexión automática, detección de desconexión, fallbacks y manejo de salas para segmentar conversaciones.
* **Base de datos con MongoDB:** MongoDB, como base de datos NoSQL orientada a documentos, es adecuada para la naturaleza menos estructurada de los mensajes de chat y ofrece flexibilidad para almacenar datos de tickets (estado, descripción, historial). También proporciona buena escalabilidad horizontal.
* **Object Document Mapper:** Mongoose facilita la interacción con MongoDB desde Node.js de forma estructurada y mantenible, permitiendo definir esquemas para tickets y mensajes, aplicar validaciones, definir middleware y simplificar las operaciones de consulta y actualización.
* **Autenticación con validación de JWT:** Se requiere autenticar tanto las conexiones WebSocket como las llamadas a las APIs REST utilizando los JWT emitidos por el User Service. Existen bibliotecas específicas para integrar la validación de JWT en Socket.IO y Express.
* **Logueo con una biblioteca de Logging:** Es importante registrar eventos clave como conexiones/desconexiones de chat, creación/actualización de tickets y posibles errores utilizando una biblioteca de logging estructurado para facilitar el seguimiento y la depuración.
* **Testeo con un Framework de testing (Jest):** Se necesitan tests para validar tanto las APIs REST (con herramientas como supertest) como la lógica del chat sobre WebSockets (con socket.io-client para simular clientes). Jest proporciona un entorno completo para realizar estas pruebas en aplicaciones Node.js.

### 4.2.8 Aplicación Frontend

Es la interfaz de usuario principal a través de la cual todos los involucrados interactúan con la plataforma. Será un Single Page Application, moderna, intuitiva y clara, que permitirá la ejecución de acciones correspondientes a cada rol y también la comunicación con los microservicios de Backend.

* **Biblioteca UI de React:** Su arquitectura basada en componentes es ideal para vistas complejas y funcionalidades diversas, permitiendo crear componentes reutilizables, acelerando el desarrollo, mejorando la mantenibilidad y la consistencia visual. Su Virtual DOM optimiza las actualizaciones de la UI.
* **Framework con Next.js:** Framework de producción sobre React que aporta enrutamiento basado en el sistema de archivos, flexibilidad en renderizado (SSR, SSG, CSR para optimizar distintas secciones), optimizaciones automáticas y una excelente experiencia de desarrollo.
* **Estilos de CSS con Tailwind CSS:** Framework CSS utility-first que permite construir diseños personalizados rápidamente escribiendo clases directamente en el HTML/JSX, acelerando el desarrollo, asegurando consistencia visual y generando archivos CSS optimizados. Facilita la creación de diseños responsivos.
* **Lenguaje TypeScript:** Añade tipado estático a JavaScript, permitiendo detectar errores en desarrollo, mejorar la autocompletación y refactorización, y hacer el código más mantenible a largo plazo, especialmente en equipos. Next.js tiene soporte nativo para TypeScript.
* **Gestion de estado con bibliotecas (como Zustand, Redux, Toolkit…):** Necesario para manejar el estado global de la aplicación (información del usuario, datos de proyectos, configuraciones) de manera eficiente y predecible entre componentes no relacionados. Se sugieren Zustand (simple) o Redux Toolkit (robusto).
* **Obtencion de Datos con bibliotecas (SWR, React Query/ TanStack Query):** Simplifica la lógica para obtener, cachear, sincronizar y manejar errores al interactuar con las APIs backend, gestionando estados de carga, reintentos y cacheo automáticamente, mejorando la experiencia del usuario y reduciendo código boilerplate. Se sugieren SWR o React Query / TanStack Query.
* **Testing con Jest y React Testing Library:** Fundamental para probar los componentes de la UI y los flujos de usuario. Jest es el corredor de pruebas y React Testing Library promueve tests que interactúan con los componentes como lo haría un usuario real, asegurando la funcionalidad desde la perspectiva del usuario.

### 4.2.9 API Gateway

Se utilizará AWS API Gateway que funcionará como única puerta de acceso para todas las solicitudes que la aplicación frontend realice hacia los diversos microservicios de la plataforma. El rol principal que cumple este componente es actuar como un proxy que recibe las peticiones del cliente, las autentica, y las enruta de manera segura y eficiente al microservicio correspondiente. También, AWS API Gateway se encargará de tareas críticas como la terminación SSL, la aplicación de políticas de seguridad y limitación de tasa para proteger el sistema, asi como la centralización del monitoreo y logging del tráfico de datos. De esta forma, se facilita la integración de la aplicación con el backend, simplificando la gestión de seguridad y proporcionando un punto único de control para las interacciones API del sistema.

### 4.2.10 Message Broker

Si bien la comunicación síncrona vía API REST es adecuada para solicitudes que requieren una respuesta inmediata, la arquitectura propuesta se beneficia de un patrón de comunicación asíncrona. Este patrón evita que un servicio tenga que esperar la respuesta de otro, mejora la resiliencia ante fallos temporales y permite un mayor desacoplamiento. Con este fin, se utilizará **RabbitMQ** como Message Broker central.

RabbitMQ implementa el protocolo AMQP y funciona como un intermediario inteligente para los mensajes. A grandes rasgos los microservicios notificarán eventos que son comunicados a través de estos mensajes con la información relevante y serán enrutados por RabbitMQ a una o más colas, para que posteriormente microservicios relevantes en el proceso puedan suscribirse a dichas colas y procesar los mensajes cuando estos estén disponibles para su uso. Este modelo conectará los microservicios de una manera flexible y desacoplada.

En el contexto de LUNARi, este componente será crucial para flujos como: la notificación desde el Servicio de Proyectos al Servicio de Facturación tras la aprobación de un hito; el envío de eventos desde el Servicio de Usuarios al Servicio de Notificaciones para enviar correos de bienvenida o de restablecimiento de contraseña; o la propagación de eventos de dominio importantes que puedan interesar a múltiples servicios sin crear dependencias directas entre ellos.

### 4.2.11 Plataforma de Monitoreo y Observabilidad

Dada la naturaleza del proyecto, es necesario contar con una solución robusta de monitoreo y observabilidad para comprender el estado del sistema LUNARi, asi como los proyectos gestionados. Es por esto que se implementará una plataforma centralizada basada en la combinación de **Prometheus** y **Grafana.**

Prometheus actuará como el sistema de recolección y la base de datos de series temporales. Cada microservicio se conectará para exponer métricas relevantes sobre su operación en un formato estándar a través de un endpoint HTTP. De esta manera Prometheus recopilará estas métricas de forma periódica desde todos los servicios y las almacenará eficientemente. Se complementará con **AlertManager** para notificar de forma oportuna a los interesados (como DevOps) sobre condiciones anómalas o fallos potenciales del sistema.

Grafana, por su parte, funcionará como plataforma de visualización y dashboarding. Al conectarse con Prometheus como fuente de datos, se crearán dashboards interactivos y personalizados que mostrarán la información de monitoreo pertinente, lo que permitirá a los equipos de LUNARi visualizar la salud general del sistema y derivar acciones sobre este conocimiento.

### 4.2.12 Gestión Centralizada de Logs

Complementando el monitoreo basado en métricas, es esencial contar con un sistema para la gestión centralizada de logs provenientes de todos los microservicios y componentes de la infraestructura. Por esta razón se implementará el **ELK Stack**, compuesto por Elasticsearch, Logstash y Kibana. Los logs son vitales para la depuración de errores, el análisis de causa raíz y la auditoría de eventos específicos.

* **Logstash** recogerá los logs generados por cada microservicio, los interpretará para extraer información estructurada y los complementará si es necesario, enviándolos posteriormente a Elasticsearch.
* **Elasticsearch** almacenará los logs procesados de manera eficiente, permitiendo búsquedas rápidas y complejas, crucial para encontrar mensajes de error específicos y seguir el rastro de una petición a través de todo el sistema, o para analizar patrones de eventos.
* Finalmente, **Kibana** será la herramienta encargada de permitir la visualización y exploración de los logs almacenados. Los equipos interesados podrán buscar de forma interactiva información relevante, presentar dashboards y comprender el comportamiento detallado del sistema.

## 4.4 Estrategia de Despliegue y Operaciones en la nube

Para asegurar un despliegue robusto, escalable, seguro y automatizado de la arquitectura de microservicios de LUNARi, se adoptará una estrategia nativa de nube utilizando **Amazon Web Services (AWS)** como proveedor principal. El código fuente de todos los microservicios y la configuración de infraestructura residirá en repositorios privados de **GitHub**.

### 4.4.1. Containerización y Orquestación

* **Docker:** Cada microservicio, junto con sus dependencias, será empaquetado como una imagen de contenedor **Docker**. Esto asegura la portabilidad y la consistencia del entorno de ejecución de cada servicio, desde el desarrollo local hasta la producción.
* **AWS Elastic Container Registry (ECR):** Las imágenes Docker construidas serán almacenadas de forma segura en ECR, el registro de contenedores privado y gestionado de AWS.
* **AWS Elastic Kubernetes Service (EKS):** La orquestación de los contenedores se realizará mediante **Kubernetes**, utilizando el servicio gestionado **EKS** de AWS. EKS administrará el plano de control de Kubernetes, reduciendo la carga operativa.

### 4.4.2. Infraestructura como Código (IaC)

Toda la infraestructura necesaria en AWS (VPC, subredes, clúster EKS, bases de datos RDS, instancias de ElastiCache, colas de MQ, repositorios ECR, roles IAM, API Gateway, etc.) será definida y gestionada utilizando **Terraform**. Esto permite aprovisionar y modificar la infraestructura de forma programática, versionada (código en GitHub), repetible y consistente, reduciendo errores manuales y facilitando la gestión de múltiples entornos.

### 4.4.3. Integración y Despliegue Continuos (CI/CD)

Se establecerá un pipeline de CI/CD robusto utilizando **GitHub Actions**. El flujo típico será:

* Un desarrollador envía código a una rama específica en GitHub.
* GitHub Actions se dispara automáticamente: ejecuta linters, análisis estático, tests unitarios y de integración.
* Si las pruebas pasan (especialmente en ramas de staging o main), la Acción construye la imagen Docker del microservicio.
* La imagen Docker se etiqueta y se sube a AWS ECR.
* Otra Acción (o paso) se encarga de desplegar la nueva versión de la imagen en el clúster EKS correspondiente (entorno de Staging o Producción). Esto se hará aplicando manifiestos de Kubernetes (YAML), posiblemente gestionados con herramientas como Kustomize o Helm, desde el workflow de GitHub Actions.

### 4.4.4. Servicios Gestionados de AWS

Para minimizar la carga operativa y aprovechar la escalabilidad y fiabilidad de AWS, se utilizarán servicios gestionados siempre que sea posible:

* **Bases de Datos:** **AWS RDS** para las instancias de PostgreSQL y **AWS DocumentDB** (compatible con MongoDB API) o **MongoDB Atlas** (partner de AWS) para las bases de datos MongoDB.
* **Caching:** **AWS ElastiCache** para instancias de Redis.
* **Message Broker:** **AWS MQ** (compatible con RabbitMQ/ActiveMQ) o un servicio gestionado de RabbitMQ.
* **API Management:** **AWS API Gateway**.

### 4.4.5. Redes, Seguridad y Gestión de Secretos

* **AWS VPC (Virtual Private Cloud):** Se crearán VPCs para aislar la red de los diferentes entornos (ej: Desarrollo, Staging, Producción). Se usarán Subredes, Grupos de Seguridad y Network ACLs para controlar finamente el tráfico entre servicios y hacia/desde internet.
* **AWS IAM (Identity and Access Management):** Se configurarán roles y políticas IAM precisas para otorgar los permisos mínimos necesarios a los usuarios, servicios y recursos de AWS.
* **AWS Secrets Manager:** Las credenciales sensibles (contraseñas de bases de datos, claves API de terceros, secretos de JWT) se almacenarán y gestionarán de forma segura utilizando AWS Secrets Manager, y se inyectarán en los contenedores en tiempo de ejecución de forma segura.

### 4.4.6. Activos Estáticos y DNS

* **AWS S3 (Simple Storage Service):** Los artefactos de build de la aplicación Frontend (Next.js) se almacenarán en S3.
* **AWS CloudFront:** Se utilizará CloudFront como CDN (Content Delivery Network) para distribuir los activos estáticos del frontend desde S3, mejorando el rendimiento global y reduciendo la latencia para los usuarios finales.
* **AWS Route 53:** Gestionará los registros DNS para los dominios de LUNARi, apuntando al API Gateway, a la distribución de CloudFront y a otros endpoints públicos necesarios.

## 4.5 Ejemplos de Caso de Uso del Sistema

### 4.5.1 Registro de un Nuevo Cliente

**Actor:** Cliente Potencial (Usuario no autenticado).

**Objetivo:** Crear una cuenta de usuario en la plataforma LUNARi.

**Flujo de Eventos e Interacciones:**

1. **(Frontend)** El usuario completa el formulario de registro (nombre, email, contraseña, quizás nombre de la empresa) y hace clic en "Registrar".
2. **(Frontend -> API Gateway)** Se envía una solicitud POST a un endpoint como /api/users/register con los datos del formulario.
3. **(API Gateway -> User Service)** El Gateway valida la solicitud inicial y la enruta directamente al Servicio de Usuarios.
4. **(User Service)**:
   * Valida los datos recibidos (formato de email, complejidad de contraseña, unicidad del email).
   * Hashea la contraseña de forma segura (usando bcrypt).
   * Guarda la información del nuevo usuario (incluyendo los datos básicos de la empresa si se recolectaron) en su base de datos (PostgreSQL). Genera un ID único de usuario.
   * (Opcional/Para Auditoría) Podría publicar un evento UserRegistered (con ID de usuario) en el exchange user\_events de **RabbitMQ**, aunque no haya un servicio de notificación consumiéndolo directamente para el usuario.
   * Devuelve una respuesta de éxito (ej: 201 Created con algunos datos del usuario o simplemente confirmación) al API Gateway.
5. **(API Gateway -> Frontend)** El Gateway reenvía la respuesta de éxito.
6. **(Frontend)** Muestra un mensaje de confirmación al usuario, indicándole que ya puede iniciar sesión (ej: "¡Registro completado!").

### 4.5.2 Creación de un Nuevo Proyecto por un Cliente

**Actor:** Cliente (Autenticado).

**Objetivo:** Iniciar un nuevo proyecto de desarrollo web/app en la plataforma.

**Flujo de Eventos e Interacciones:**

1. **(Frontend)** Asumiendo que una de las partes ya aprobó, la segunda parte (ej: el PO) revisa el trabajo asociado al hito y hace clic en "Aprobar Hito".
2. **(Frontend -> API Gateway)** Se envía una solicitud PUT o PATCH (ej: /api/projects/{projectId}/milestones/{milestoneId}/approve) con el token JWT del actor.
3. **(API Gateway -> Project Service)** El Gateway valida el JWT y enruta la solicitud al Servicio de Gestión de Proyectos.
4. **(Project Service)**:
   * Verifica los permisos del actor (PO en este caso) sobre el proyecto y el hito.
   * Consulta el estado actual del hito en su base de datos (PostgreSQL) y confirma la aprobación previa de la otra parte (el Cliente).
   * Actualiza el estado de aprobación del PO en la base de datos.
   * Al detectar que ambas partes han aprobado, cambia el estado general del hito a 'Aprobado'.
   * **Publica** un evento MilestoneApproved (conteniendo ID del hito, ID del proyecto, monto asociado si existe, etc.) en el exchange project\_events de **RabbitMQ**.
   * Devuelve una respuesta de éxito (ej: 200 OK con el estado actualizado del hito) al API Gateway.
5. **(API Gateway -> Frontend)** El Gateway reenvía la respuesta de éxito.
6. **(Frontend)** La interfaz de usuario se actualiza para reflejar que el hito está completamente "Aprobado".
7. **(RabbitMQ -> Billing Service)** El Servicio de Facturación está suscrito a la cola correspondiente y consume el evento MilestoneApproved.
8. **(Billing Service)**:
   * Utiliza la información del mensaje para generar la entrada de facturación correspondiente en su base de datos (PostgreSQL). Puede realizar llamadas internas al Project Service si necesita detalles adicionales no incluidos en el evento.
   * (Opcional) Podría publicar un evento InvoiceGenerated para consumo interno por otros servicios (ej: reportería) o auditoría, pero no para notificar al usuario directamente.

### 4.5.3 Cliente Solicita Soporte vía Chat

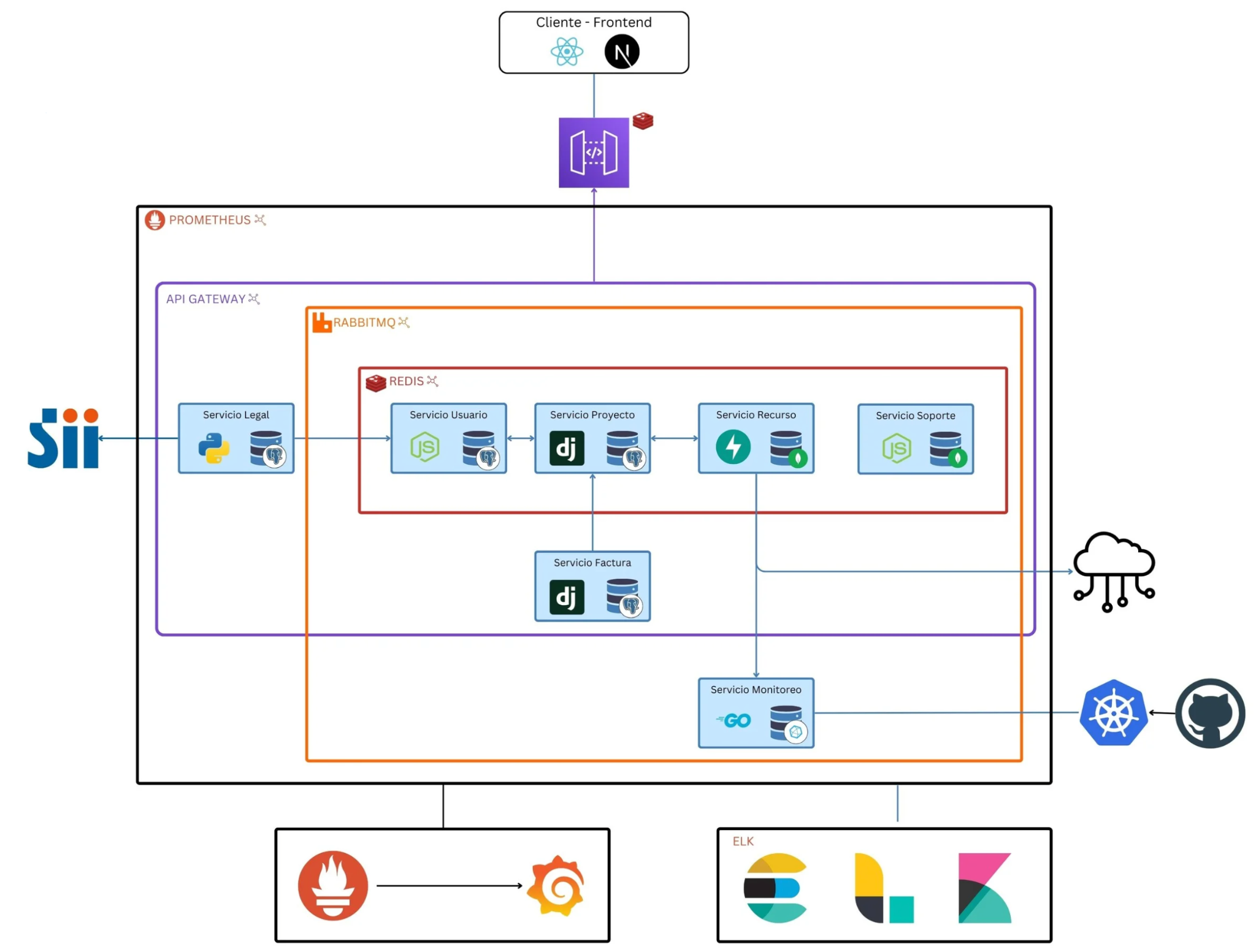
**Actor:** Cliente (Autenticado).

**Objetivo:** Iniciar una conversación de chat en tiempo real con el equipo de soporte de LUNARi.

**Flujo de Eventos e Interacciones:**

1. **(Frontend)** El cliente, dentro de su portal, hace clic en el botón o widget de "Chat de Soporte".
2. **(Frontend <-> Support Service via WebSocket)** El Frontend inicia una conexión **WebSocket** segura con el Servicio de Soporte. Durante el establecimiento de la conexión, envía el token JWT del cliente para la autenticación.
3. **(Support Service)**:
   * Recibe la solicitud de conexión WebSocket, valida el JWT para autenticar al usuario.
   * Una vez autenticado, establece la conexión persistente.
   * Identifica al usuario y lo asocia a una conversación (nueva o existente). Asigna la conversación a una "sala" lógica.
   * Notifica internamente (ej: a través de WebSockets a los agentes conectados) la disponibilidad de un nuevo chat o mensaje.
4. **(Frontend)** El cliente escribe su consulta en la interfaz de chat y presiona "Enviar".
5. **(Frontend -> Support Service via WebSocket)** El mensaje de texto se envía a través de la conexión WebSocket establecida.
6. **(Support Service)**:
   * Recibe el mensaje del cliente.
   * Persiste el mensaje en su base de datos (MongoDB), asociado a la conversación/ticket correspondiente.
   * Reenvía (broadcast) el mensaje a través de WebSocket a todos los participantes de esa "sala" (el cliente y el/los agente/s de soporte asignado/s).
7. **(Frontend Agente/Cliente)** Los clientes WebSocket del agente y del propio cliente reciben el mensaje y lo renderizan en su respectiva interfaz de chat casi instantáneamente.
8. **(Flujo Inverso)** Cuando un agente de soporte responde, el proceso se repite: el mensaje viaja por WebSocket al servicio, se guarda en MongoDB y se reenvía al cliente.

## 4.6 Diagrama de Arquitectura

****

En el diagrama se puede apreciar las interacciones entre los diferentes componentes del sistema, tal como fueron descritos en la sección anterior. Cada caja corresponde a un sector del sistema que está asociado por la herramienta en común utilizada en los microservicios.

## 4.7 Diagrama de Despliegue

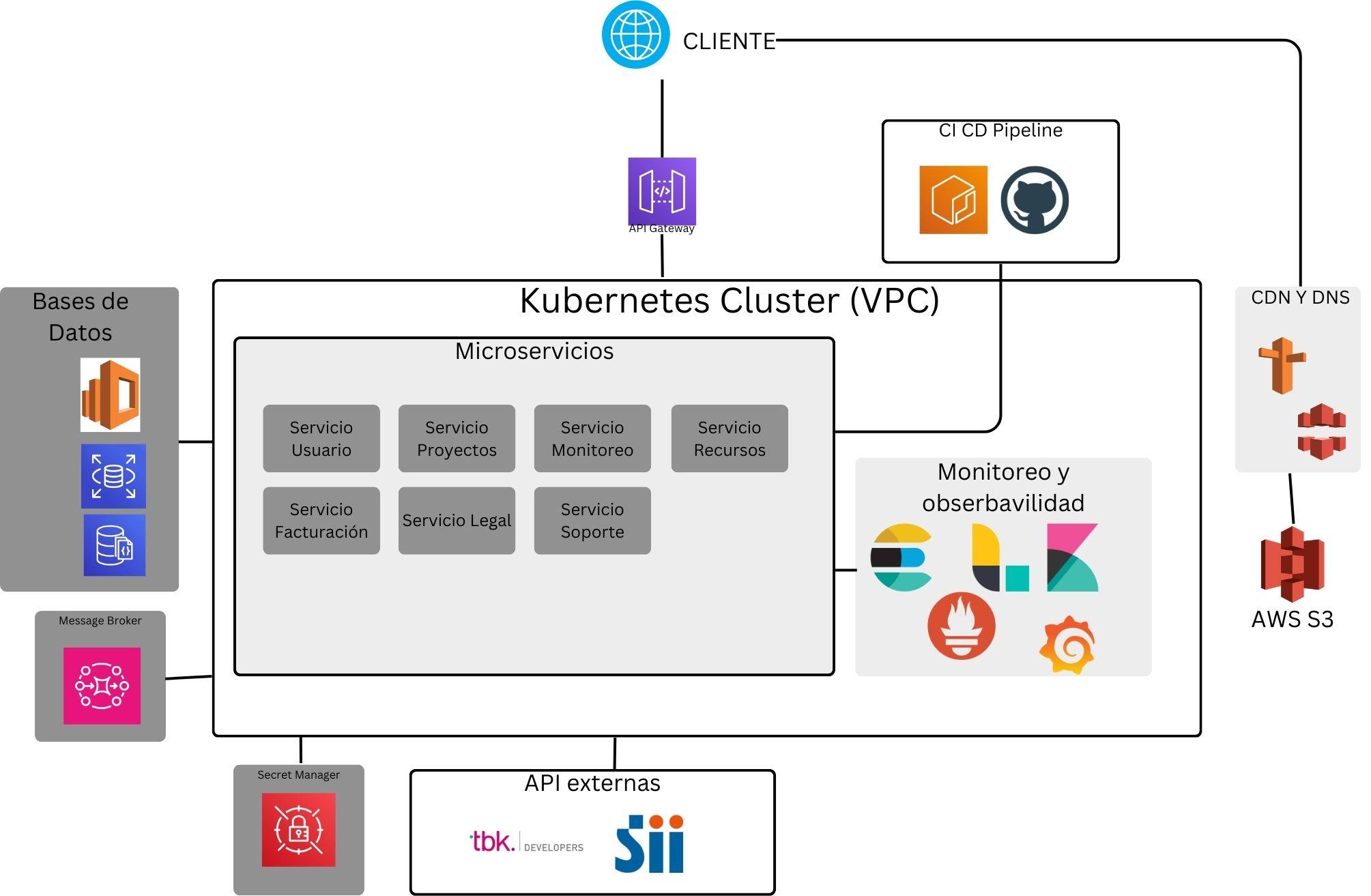


Diagrama que demuestra el despliegue de la arquitectura en la nube, específicamente AWS, junto con sus servicios respectivos

# 5. Planificación de la migración

## 5.1 Estrategia y Fases de Migración

La migración nos proporcionará múltiples beneficios, siendo el mayor de ellos poder poner a la empresa a la vanguardia del mercado, asegurando su funcionamiento y sostenibilidad a lo largo del tiempo.

Dentro de dichos beneficios encontramos los siguientes:

**Escalabilidad Independiente:** Con microservicios, puedes escalar individualmente los componentes que experimentan mayor carga. Por ejemplo, si la creación de nuevas páginas web tiene un máximo de demanda entonces puedes escalar solo el microservicio encargado de esta función, sin necesidad de escalar todo el sistema de seguimiento o mantenimiento. Esto optimiza el uso de recursos y reduce costos.

**Despliegues más rápidos y menos riesgosos:** Si ocurre un problema en un microservicio durante un despliegue el impacto se limita a ese servicio específico, sin afectar la funcionalidad general de la creación, mantenimiento o seguimiento de las páginas web. Esto minimiza las interrupciones para nuestros clientes.

**Flexibilidad para adaptarnos a los cambios del mercado,** y también para aquellos cambios que nuestros clientes requieren en la mejora constante de sus plataformas web.

**Organización:** pues cada grupo de desarrolladores podrá dedicarse exclusivamente a su área, sin interferir con los demás desarrollos que se pudieran estar llevando a cabo. Nos da posibilidades reales de acelerar nuestras implementaciones, mejorando la confianza hacia nuestros clientes, ganando una posición sólida en el mercado.

**Menores pérdidas:** asociadas a la menor curva de aprendizaje de los nuevos colaboradores.

Es posible realizarla en los siguientes pasos:

1. Identificar componentes lógicos en el monolito: (ej. lo lógico, lo relativo a logueos, lo relativo a ventas, lo relativo a usuarios, etc).
2. Aplanar y refactorizar componentes: (es posible encontrar componentes que realizan funciones similares, pudiendo combinarse).
3. Identificar las dependencias de los componentes: (ej. la forma en que “logueos” y “usuarios” se interrelacionan).
4. Identificar grupos de componentes: (puede haber un grupo de componentes que se comuniquen entre sí, pero solo uno de ellos se comunica con otro componente de otro grupo).
5. Crear una API para la interfaz de usuario remota: (que es la vía de conexión entre el software y el usuario).
6. Migrar grupos de componentes a macroservicios: (mover grupos de componentes a proyectos separados y realizar implementaciones separadas).
7. Migrar macroservicios a microservicios.

De todas formas, es necesario aclarar que puede haber pasos que se pueden saltar. Por ejemplo, el paso 7 puede omitirse si es que el “macroservicio” no es lo suficientemente grande como para necesitar ser descompuesto en trozos más pequeños.

## 5.2 Riesgos asociados a la migración de los sistemas

Aunque la migración a microservicios siempre se ve con buenos ojos, también conlleva problemas asociados, de los cuales podemos nombrar los siguientes como más importantes:

* En cuanto a la complejidad: pasar de monolito a microservicios significa la descomposición de todo el software (o gran parte de él), por lo que encontrar todos los puntos en que un microservicio interactúa con otro de forma similar a como lo hacía en el código original de modo de mantener la compatibilidad con el anterior sistema es una tarea titánica si el software original es demasiado grande.
* Cada microservicio depende de su propio lugar de almacenamiento de información.
* Tener muchas fuentes de información conlleva un problema de seguridad, pues hay más fuentes desde donde poder robar esa información.
* Si cada microservicio se conecta con otro entonces existen más puertas de una posible fuga de datos, o de poder inyectar código malicioso que pudiera destruir un software basado en microservicios. Lo anterior es una delicia para los hackers, que siempre andan buscando formas de ingresar a un sistema.
* Se debe garantizar la compatibilidad con los datos del sistema anterior en cuanto a la forma en que se procesan los datos, la forma en que los datos son guardados, leídos, importados o exportados, aunque dependiendo del proyecto podría no representar un problema (como cuando un sistema pretende mejorar por completo y deliberadamente no desear la retrocompatibilidad).

# 6. Implementación

## 6.1 Base de Datos

Como base fundamental del desarrollo de cualquier API está el desarrollo de su base de datos. Es primordial para el correcto funcionamiento de todo el sistema.Esta base de datos puede gestionar usuarios con roles y permisos granularizados, administrar empresas, clientes y sus relaciones con usuarios, organizar un catálogo de servicios con sus categorías, recursos necesarios y opciones adicionales, y también implementa un carrito de compras con seguimiento de estado y expiración.

A continuación se pasa a explicar la estructura de la base de datos basándose en el archivo presente en la rama main de GitHub **script\_creacion\_tablas.sql**

### 6.1.1 Configuración Inicial y Eliminación de Tablas Existentes

* Habilita la extensión para generar UUIDs (identificadores únicos universales).
* Elimina tablas existentes en orden inverso a sus dependencias (usando CASCADE para eliminar también las relaciones).

### 6.1.2 Estructura de Tablas de Usuarios y Permisos

Sus tablas principales:

* **Rol**: Define roles de usuario (ADMINISTRADOR, PRODUCT\_OWNER, etc.)
* **Permiso**: Lista de permisos disponibles en el sistema.
* **Rol\_Permiso**: Relación muchos-a-muchos entre roles y permisos.
* **Empresa\_Cliente**: Almacena información de empresas de clientes.
* **Usuario**: Contiene datos de usuarios del sistema.
* **Membresia\_Equipo**: Relaciona usuarios con proyectos externos.

Características que podemos destacar:

* Uso de UUIDs como claves primarias para Usuario y Empresa\_Cliente.
* Campos de auditoría (creado\_el, actualizado\_el) en todas las tablas. Aquello permite hacer seguimiento de los movimientos realizados.
* Relaciones bien definidas con acciones ON DELETE apropiadas.
* Índices para optimizar búsquedas frecuentes, lo cual ayuda a operar de mejor manera.

### 6.1.3. Datos Iniciales (Seed)

Inserta datos básicos para:

* Los 4 roles principales.
* 22 permisos diferentes.
* Asignación de permisos a cada rol

### 6.1.4. Módulo de Inventario.

Sus tablas principales:

* **Categoria**: Clasificación de servicios.
* **Catalogo**: Servicios ofrecidos por nuestra empresa.
* **TipoRecurso**: Tipos de recursos necesarios para los servicios ofrecidos.
* **PaqueteRecursoServicio**: Relaciona servicios con los recursos que requieren.
* **ServicioAdicional**: Opciones adicionales para servicios.
* **ServicioServicioAdicional**: Relación muchos-a-muchos entre servicios y sus adicionales.

Características notables:

* Restricciones CHECK para validar datos (precios positivos, duraciones válidas).
* Un sistema completo para definir qué recursos (tiempo, materiales) componen cada servicio.

### 6.1.5. Módulo de Carrito

Tablas principales:

* **Carrito**: Contenedor principal del carrito de compras del sistema.
* **Carrito\_Item**: Servicios agregados al carrito.
* **Carrito\_Servicio\_Adicional**: Servicios adicionales asociados a items del carrito.

Algunas características notables:

* Estados del carrito (ACTIVO, PROCESADO, etc.).
* Lógica de expiración automática (30 días por defecto).
* Cálculo de subtotales y totales.
* Capacidad para personalizaciones.

Se buscó que la base de datos contenga en su esencia las buenas prácticas incorporadas, como por ejemplo **Restricciones de integridad** (FOREIGN KEY, CHECK), índices para mejorar el rendimiento, que haya campos de auditoría para rastreo de cambios, y en caso de ser necesario una eliminación limpia de datos con políticas CASCADE/RESTRICT/SET NULL según corresponda.

## 6.2 Microservicios

Se han creado 3 microservicios para el desarrollo del sistema. Estos microservicios tienen por tarea fundamental manejar todos los datos de la aplicación respecto a los Usuarios, Inventario y Carrito de la plataforma, teniendo los microservicios nombres congruentes con las funciones y datos que manejan, y con ello fomentar un espacio de buenas prácticas y calidad de software.

Estas tareas pueden ser divididas en 4 grupos:

* GET: devuelve un dato solicitado.
* POST: guarda información en la base de datos correspondiente.
* PUT: modifica datos de la base de acuerdo a algún parámetro.
* DELETE: elimina datos de la base de acuerdo a algún parámetro.

El microservicio de **Usuario** es el encargado de manejar todas la operaciones CRUD de los usuarios del sistema. Este microservicio no solamente se centra en el manejo de los usuarios-clientes, sino que también maneja a los usuarios- desarrolladores de LUNARi, creando de esta manera un espacio único y centralizado de los usuarios en general, permitiendo una mejor mantenibilidad y evitando posibles incompatibilidades con el resto de los microservicios.

Las características de este microservicio son las siguientes:

* En la construcción del microservicio se hizo la distinción entre los distintos roles del usuario potencial del sistema, diferenciando entre 4: *Administrator, Product\_Owner, Client, DevOp.*
* Su base de datos contiene información fundamental de cada usuario, tales como un ID generado automáticamente, Nombre, Email, Password y Rol. Cabe destacar que se añadieron manejos de errores para los campos que deben ser necesariamente llenados con datos en la DB, y se muestran mensajes de advertencias asociados al error.
* **UserController** contiene todos los métodos CRUD del microservicio.
* Se crea un **SwaggerUI** pensado en visualizar las distintas operaciones de consulta de los usuarios a través de una interfaz gráfica amigable.

El microservicio de **Inventario** está pensado para hacer el manejo de todos los datos referente a los servicios prestados.

Nuestra empresa se dedica a vender servicios de creación de páginas web, lo que en nuestro caso complejiza ligeramente generar un “inventario” de lo que hacemos.

En primera instancia se pensó que podríamos tener una cantidad limitada de páginas web para vender por mes (supongamos unas 15 unidades), pero aquello crea problemas, pues no valoriza de forma correcta el tiempo usado en el desarrollo de dichas páginas web. Además, si estas páginas web fueran desarrolladas en poco tiempo entonces tendríamos tiempo muerto por delante hasta el siguiente mes, lo que también generaría pérdidas de dinero para la empresa, y es por eso que se decidió vender **horas-hombre**. Lo anterior implica que el cliente debe pasar por un proceso de cotización, y en base a la complejidad de su proyecto es que esas horas-hombre son paquetizadas y vendidas como “packs” de desarrollo. Se cuentan con desarrolladores limitados, siendo sus horas de trabajo el principal activo que se puede agotar, y por consiguiente “**inventariar”**.

Cabe destacar que estas horas-hombre no son directamente comprables a través del sistema, pues no se espera que sea el cliente quien decida *en* *cuantas horas se realizará el desarrollo*.

Las características de este microservicio son las siguientes:

* Implementa un **SwaggerUI** para facilitar las visualizaciones a la base de datos del inventario, haciéndolas más visuales y amigables.
* Dentro de su carpeta **controller** se encuentra **CustomErrorController,** pensado en el manejo de errores del microservicio.
* Existen diferentes clases JAVA para diferentes necesidades de la base de datos del microservicio.
* **InventoryController** nos da acceso a la revisión del Catálogo de servicios ofrecidos por nuestra empresa. Es posible ver todos los servicios del catálogo, o encontrar el requerido por ID, Nombre o la categoría a la que pertenecen.
* Dentro del mismo controlador es posible hacer las otras operaciones CRUD del sistema, tales como agregar servicios, eliminarlos, modificarlos, y tambien activarlos o desactivarlos.

El microservicio de **Carrito** es la encargada de manejar todas las operaciones de compra o contratación de los servicios ofrecidos por la empresa a sus clientes. Este microservicio representa una complejidad distinta a los demás servicios explicados con anterioridad, pues **Carrito** debe hacer llamadas a los otros dos microservicios a manera de hacer consultas respecto de sus datos contenidos.

Es imposible que un carrito pueda funcionar si no se cumplen las siguientes condiciones:

* Carrito debe contener un usuario válido.
* Carrito debe contener productos que existan en la base de datos del Inventario.
* El producto inventariado debe tener existencias para poder ser ingresado al Carrito.

Las características de este microservicio son las siguientes.

* **CarritoController** puede manejar las distintas solicitudes hechas al microservicio.
* Estas consultas pueden ser: crear un carrito, obtener un carrito por su ID, obtener todos los carritos de un usuario, agregar items al carrito, modificar la cantidad de un ítem en ese carrito, eliminar un ítem del carrito, dar por finalizada la compra, o simplemente vaciar el carrito.
* El carrito contiene estados útiles para el resto del microservicio siendo estos **ACTIVO, PROCESADO, ABANDONADO, EXPIRADO.**

7. Consideraciones Éticas

La industria de la informática está en constantes avances y desarrollos. Lo vertiginoso de estos avances y las nuevas tecnologías nos hacen olvidar todo aquello que está detrás de dichos avances. Nada es gratis, y el progreso tiene muchas vías por las cuales nos puede cobrar, no solamente en dinero, sino que también con nuestra propia seguridad.

Por ello es lógico pensar en preguntas como las siguientes:

* ¿De donde se obtienen todos esos kilowatts que se consumen al desarrollar tecnología?
* ¿Dónde van a parar todos los datos que diariamente ingresamos en nuestros dispositivos y plataformas de internet?
* ¿Cómo sé que la información ingresada no será parte de un gran conjunto de datos que se venderán a otras compañías con fines inescrupulosos?
* ¿Está en peligro mi trabajo por culpa de estos avances?
* ¿Estoy realmente seguro en internet?

Todas estas preguntas son aplicables a nuestro caso de estudio, y son preguntas que bien vale la pena responder.

Por parte de LUNARi, existe el compromiso de poner énfasis en la reducción en su mayor porcentaje posible a todos los efectos adversos que pudieran aparecer por el hecho de la migración a los microservicios.

Se han reconocido tres ejes principales como preocupantes: **privacidad, empleo y medioambiente.**

La **privacidad** como eje fundamental de la implementación de la arquitectura de microservicios implica una serie de puntos a tener en consideración. Puesto que cada microservicio da un uso distinto a distintos datos, y sumado a que existe constante intercomunicación entre esos microservicios, resulta imperativo tener total refuerzo sobre el uso de esos datos, protegiendo a nuestros clientes a través de los mejores sistemas de seguridad del mercado.

Entre los métodos a utilizar en pro de mejorar la seguridad debemos comenzar por la principal debilidad que tiene el sistema de microservicios: las API. Es posible usar tokens en las API de manera que las credenciales de los usuarios no sean visibles ni compartidas entre los microservicios, pudiendo centralizar el acceso a dichas credenciales, y que sean los tokens los encargados de la autenticación posterior. Con ello le damos al cliente la seguridad de que sus datos no podrán ser interceptados ni usados para suplantaciones de identidad.

Otra forma de proteger la información será apegandonos a la norma ISO 27001, la cual se pasa a describir a continuación:

*“La norma ISO 27001 es un estándar internacional que establece los requisitos para la implementación, mantenimiento y mejora continua de un Sistema de Gestión de la Seguridad de la Información (SGSI). Este sistema se utiliza para proteger la confidencialidad, integridad y disponibilidad de la información. La norma proporciona un marco para la seguridad de la información que ayuda a las organizaciones a identificar y gestionar sus riesgos de seguridad de la información de manera efectiva.”*

Además, por el ámbito legal, nuestra empresa debe hacer apego irrestricto a aquellas normas que regulan el tratamiento, almacenamiento, difusión y eliminación de los datos personales. Tales normativas son:

* **La Ley Orgánica de Protección de Datos Personales de Chile**, que protege la privacidad y el tratamiento de datos personales.
* **La nueva Ley de Protección de Datos (LPD)** regula el tratamiento de datos personales y crea la Agencia de Protección de Datos Personales (APPD).
* **La Ley 21096** establece el derecho a protección de los datos personales.
* **La Ley 19.628** establece el derecho de las personas a la información, modificación, cancelación o bloqueo de sus datos personales.

El **empleo** es también un tema sensible al momento de las migraciones a microservicios, pues muchas veces involucra nuevas formas de trabajo para las cuales las personas, y para no ver mermada su capacidad de desarrollar, su bienestar laboral ni tampoco dejar obsoleto a algún colaborador es que es necesario:

* Capacitar a las personas en la nueva forma de trabajo.
* Enseñar las nuevas tecnologías que se usarán.
* Implementar las mejoras y cambios (si es posible) de manera progresiva.
* Establecer claramente las nuevas jefaturas o estructuras organizativas que nacen del nuevo sistema de trabajo, con el objetivo de que cada colaborador reconozca su espacio de trabajo, tenga claridad en sus labores y con ello evitar malos entendidos, así como promover la estabilidad laboral.

Y como punto fundamental es necesario evitar al máximo despedir personal. Esto es posible lograrlo a través de la reubicación de colaboradores en distintas funciones. La mejora de una empresa no debe estar nunca por encima de quienes la hacen grande.

Año a año vemos como el calentamiento global causa estragos en nuestro planeta. Gran parte de esa culpa proviene de nosotros, los humanos, y nuestra insaciable sed de progreso. El uso de los combustibles fósiles, sumado a un uso indiscriminado del agua, así como la constante erosión de nuestros bosques y la tala de los mismos ha causado que el **medio ambiente** se encuentre en graves problemas.

Según datos de la ONU l[a humanidad generó 62 millones de toneladas de desechos electrónicos en 2022](https://news.un.org/es/story/2024/03/1528476). Sumado a ese dato hay que considerar que el consumo eléctrico de los centros de datos también corresponde a cifras estratosféricas. Solamente [Google en 2024 consumió 24 TWh de energía](https://elperiodicodelaenergia.com/google-consume-el-10-de-la-electricidad-de-todos-los-centros-de-datos-del-mundo/), lo cual representa el 10% de toda la energía consumida por todos los centros de datos repartidos por el mundo.

En un intento de no ser un mal aporte al medio ambiente, es que consideramos pertinente que nuestro avance también vaya de la mano con un cuidado hacia nuestro entorno. El uso de dispositivos con buena eficiencia energética y hacer un correcto proceso de reciclaje de aquellos dispositivos que ya no son útiles podrían ayudar en dicho cometido, así como la implementación local de paneles solares, implementar prácticas de desarrollo y despliegue de aplicaciones eficientes, y también la optimización en todos los procesos que puedan ser objeto de mejora.

**Otros tópicos que merecen la atención:** nuestros clientes merecen tener la mayor transparencia respecto a los movimientos que se están haciendo dentro de la empresa, ello con el fin de generar confianza, y de evitar las fugas de clientes. La transparencia también ayuda en el momento en que los cambios están siendo llevados a cabo, pues se les puede poner en conocimiento respecto a momentos en que el servicio entregado podría estar off-line, pudiendo ellos organizarse con lo necesario respecto a sus emprendimientos, y así no generar pérdidas.

# 8. Conclusiones

LUNARi busca modernizar su infraestructura mediante la migración a microservicios para superar las limitaciones de su sistema actual, impulsadas por su crecimiento. Esta estrategia promete escalabilidad, agilidad y mejor organización, aunque implica desafíos en complejidad y seguridad. La migración se planifica en etapas, apoyándose en herramientas de colaboración y considerando los riesgos asociados.

Para esta entrega se ha añadido la información referente al funcionamiento e implementación de los microservicios asociados a la aplicación, destacando sus puntos importantes, y cómo estos operan.

LUNARi se compromete a abordar las implicaciones éticas en privacidad, empleo y medioambiente, buscando una transición exitosa y sostenible que beneficie a sus clientes y a la empresa a largo plazo.