Instituto Tecnológico de Buenos Aires

72.40 Ingeniería del Software II

Final - Sistema 10 (Plataforma de Desarrollo)

21/12/2023

Integrantes:

(62028) Nicolás Matías Margenat - nmargenat@itba.edu.ar

(62067) Marcos Gronda - mgronda@itba.edu.ar

(62094) Juan Burda - jburda@itba.edu.ar

(62493) Saul Ariel Castañeda - scastaneda@itba.edu.ar

(62504) Elian Paredes - eparedes@itba.edu.ar

Docentes:

Juan Martín Sotuyo Dodero

Guido Matías Mogni

Tabla de contenidos

1. Consigna

Se pide hacer un sistema SaaS para plataforma de desarrollo (i.e. Github / Bitbucket). El modelo de negocios es hacerlo gratuito para sistemas open source, a modo de capturar usuarios, y pago para sistemas comerciales, donde sus repositorios serán privados y tendrán un buen SLA.

Es fundamental que el sistema permita crear proyectos on-demand y soporte diversas herramientas en torno a los mismos (issue tracking, time tracking, release management, wiki, etc.) y permita integraciones fáciles a multiplicidad de sistemas; así como permitir que sistemas de terceros se paren encima nuestro para proveer servicios adicionales (ie: Hound.io, Travis.ci, etc.).

Es de interés a la compañía, que el sistema de control de versiones utilizado no sea único, sino que permita utilizar varios de los más comunes (SVN, git, mercurial, perforce), y no se descarta sumar nuevos a futuro conforme la demanda lo requiera. Esto no debe ser perjudicial para el resto de las herramientas ofrecidas que deben de poder integrarse con todos de forma transparente.

2. Requisitos

2.1. Requisitos funcionales

1. Crear proyectos on-demand.
2. Crear o subir un repositorio de Git, SVN, Mercurial, Perforce a un proyecto.
3. Administrar permisos para un proyecto y especificar la visibilidad del proyecto (ie. si es público o privado).
4. Permitir integración con sistemas de terceros de forma programática (ie. “usar los sistemas y que nos usen a nosotros”).
5. Poder registrarse e iniciar y cerrar sesión.
6. Pago mediante una plataforma de pagos.
7. Generar, resolver y operar con issues relacionados a un proyecto, también manejando las estimaciones de tiempo.
8. Generar releases para cierta versión de un proyecto.
9. Generar y editar una wiki para un proyecto.

2.2. Requisitos no funcionales

1. Integración transparente con distintos sistemas de control de versiones (VCS).
2. Alto nivel de seguridad para repositorios y usuarios.
3. Asegurar un 99.9% de disponibilidad.
4. Autenticación y control de acceso.
5. Integración fácil con sistemas de terceros.
6. Agregación simple de nuevos VCS.

3. Atributos de calidad

---

1. **Disponibilidad**

Al tratarse de una plataforma de desarrollo sobre la cual trabajarán tanto individuos como empresas, resulta de suma importancia proveer una alta disponibilidad. Esta disponibilidad formará parte del SLA con todos los usuarios (esto es, sin distinguirlos entre pagos y gratuitos) y será mayor al 99.9%.

2. **Seguridad**

La plataforma de desarrollo será el lugar donde se almacenarán cientos de repositorios de compañías e individuos. Resulta esencial poder asegurar un alto nivel de seguridad, tanto para usuarios como para los repositorios, pues de lo contrario se perderían años de trabajo y horas invertidas.

3. **Escalabilidad**

La solución debe ser capaz de escalar mundialmente para permitir equipos de trabajo distribuidos por el mundo. Además, se espera un gran uso de la plataforma durante la semana pero poco uso en los fines de semana, por lo que resulta deseable aumentar y achicar la escala del sistema fácilmente.

4. **Mantenibilidad**

El sistema deberá permitir distintos Version Control Systems (VCS), y en el futuro se pueden agregar nuevos sin necesidad de cambiar el sistema por completo o cambiar los sistemas complementarios como los *wikis* e *issues*.

5. **Interoperabilidad**

La plataforma debe poder integrarse fácilmente con una multiplicidad de servicios de terceros, los cuales podrán variar a lo largo del tiempo y puedan tener formatos distintos para accederlos.

6. **Usabilidad**

El sistema debe competir con alternativas como Github y Bitbucket, por lo que debe proveer una buena experiencia de usuario, de lo contrario nadie eligirá la plataforma.

7. **Otros atributos considerados**

Se consideraron otros atributos, el más importante siendo *performance*. Al final se descartó debido a dos razones: el flujo principal del sistema no es *real-time* y no se necesitan hacer operaciones computacionalmente caras. Obviamente, no significa que se hará una aplicación lenta, sino que no se pondrá un gran énfasis en la velocidad de la aplicación.

4. Supuestos

- El sistema contará con una gran cantidad de usuarios y será de alcance global.
- Cualquier sistema de control de versiones que se agregue en el futuro va a usar el File System como principal punto de almacenamiento de los archivos / diffs.
- Los sistemas de control de versiones cuentan con comunicación sobre SSH.
- No es necesario implementar un sistema propio de facturación: basta solo una API externa de Stripe para cumplir esta funcionalidad.

5. Arquitectura

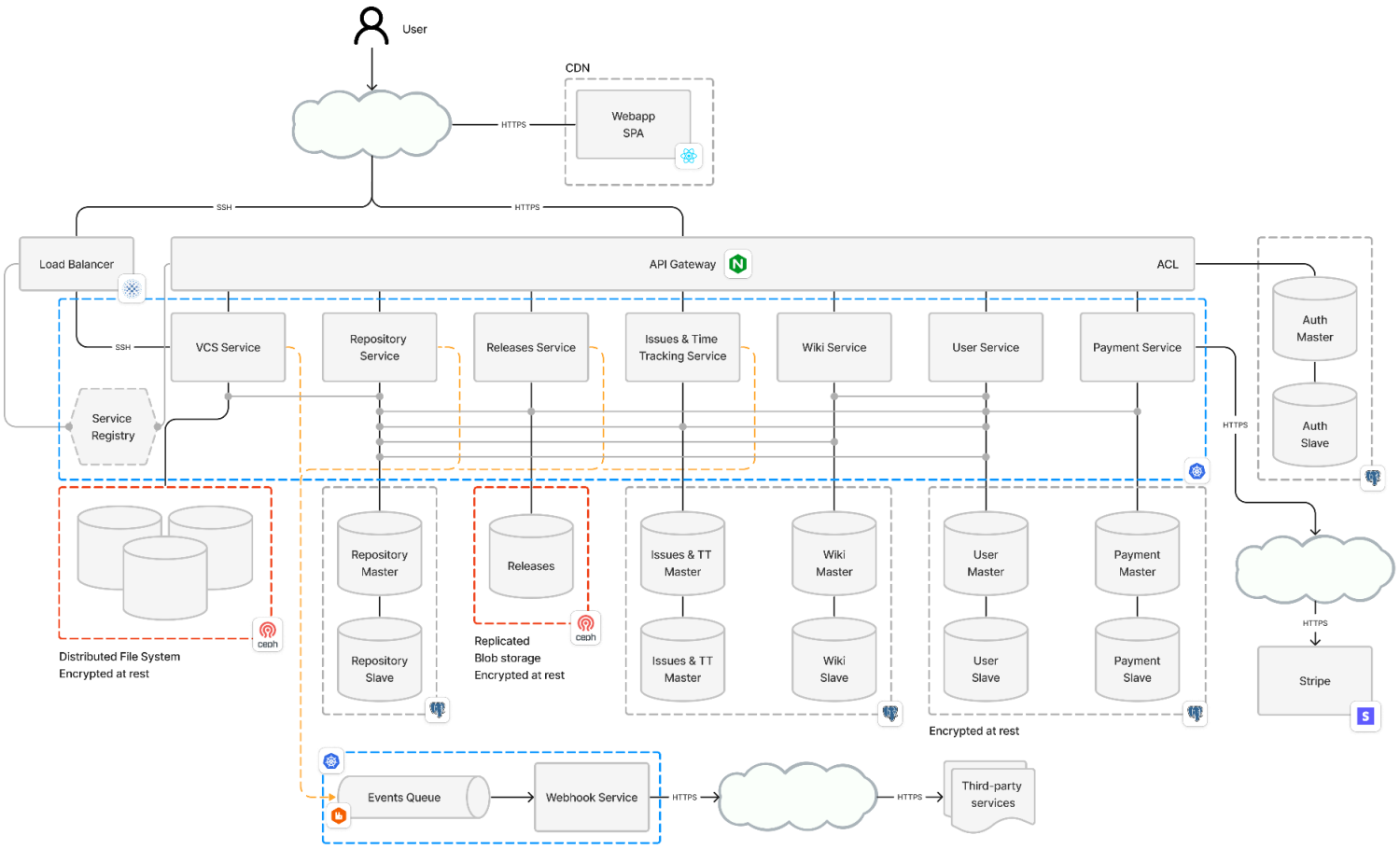


Diagrama 1: Arquitectura del sistema
Nota: se encuentra una versión más grande de la arquitectura al final del informe.

5.1. Explicación de la arquitectura

5.1.1. Uso de microservicios

Dado que el sistema será utilizado por millones de usuarios, y se espera que crezca en tamaño en los próximos años, se decidió armar la arquitectura de una manera escalable. Para ello, se optó por utilizar microservicios ya que permiten el agregado de nuevas funcionalidades de manera aislada sin impactar sobre el resto del sistema. A pesar de que la mantenibilidad del sistema se ve afectada por esta decisión, se considera que el proceso de CI/CD se ve beneficiado porque al hacer deploy de nuevos cambios estos van a tener un *blast radius* mucho menor que si se tratase de un monolito. Esto es así porque el servicio que se verá afectado solo por unos momentos, será sólo aquel que recibe los cambios.

Por otra parte, estos microservicios serán hosteados en EC2. La elección de tecnología será explicada más adelante, pero se busca con ello lograr una alta escalabilidad y permitir así hacer crecer el sistema de una manera simple y según sea necesario a medida que se expande la plataforma.

5.1.2. Servicio de usuarios

Este servicio se encarga de almacenar, modificar y recuperar información de los usuarios de la plataforma. Además, se encarga de realizar la distinción entre usuarios pagos y no pagos, para que luego el servicio de repositorio pueda consultar esta información y habilitar la opción de repositorios privados o no.

Los usuarios se registran mediante la webapp y mediante este servicio impactan las credenciales y la información del usuario en una base de datos que utiliza PostgreSQL.

Por otro lado, en el proceso de autenticación, se validan las credenciales del usuario mediante el llamado a este servicio para que luego el sistema de control de acceso se encargue de proveer el JWToken.

5.1.3. Servicio de VCS

El servicio de VCS es el centro de la aplicación y tiene como objetivo proveer acceso a los distintos repositorios almacenados en la plataforma, de la misma manera que otros servicios como Github, GitLab o Bitbucket lo hacen, pero con el agregado de que permite interactuar con múltiples tipos de repositorios distintos (esto es, que utilizan distintos VCS).

A este fin, el servicio se encarga de ejecutar el flujo para el acceso: validando que el usuario tiene acceso a dicho repositorio y redireccionando el comando apropiado. También debe soportar los distintos patrones de accesos, ya sea por usuarios y servicios de terceros.

5.1.3.1. Manejo de los repositorios

Al investigar los distintos sistemas de control de versiones, en específico Mercurial, se encontró una herramienta que permitiría traducir, *sin pérdida de información*, las operaciones de Mercurial a Git. Esta extensión, llamada *hg-git* [1], nos evitaría tener que almacenar los repositorios en formato Mercurial, al poder reemplazarlos con Git. Esto reduciría la complejidad de la aplicación y facilitaria el almacenado de los repositorios, al poder usar herramientas específicas para Git como *Gitaly* [2].

Sin embargo, al continuar investigando, se debió descartar esta idea dado que no todas las VCS tenían herramientas análogas a *hg-git*. Por ejemplo, para Subversion existe una herramienta llamada *SubGit* [3], que permite mantener dos repositorios, uno de SVN y uno de Git actualizados con el mismo contenido, es decir, tener un mirror entre los dos. Pero esta herramienta no elimina la necesidad de tener un repositorio de Subversion. No solo eso, sino que también se debía comprar una licencia para dicha herramienta.

Finalmente, se descartó la idea de solo almacenar repositorios Git, por lo que se debió pensar una manera de almacenar y operar con los distintos VCS de forma separada.

5.1.3.2. Mantenibilidad a largo plazo

En una primera instancia, se consideró hacer que el servicio VCS sea un orquestador que reaccionaba los pedidos a los servicios de repositorios, como Git y SVN. Esto involucra generar servicios para cada uno de los repositorios y tendría este esquema:

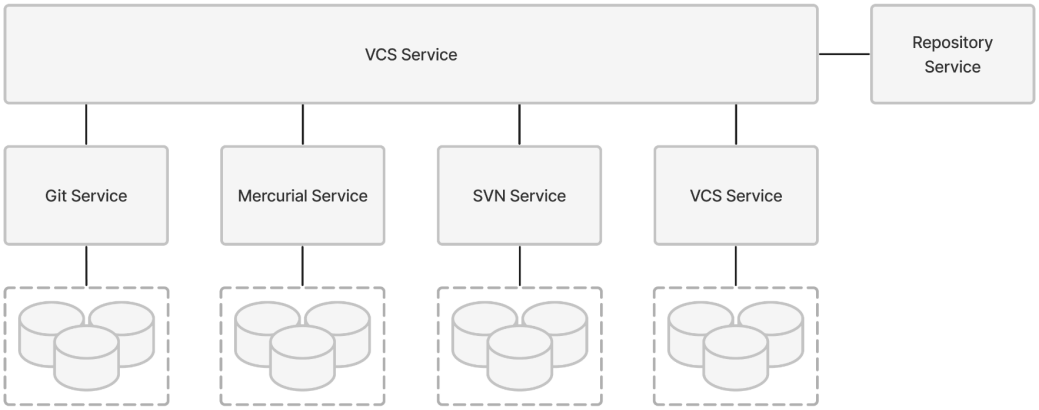


Diagrama 2: esquema inicial de servicio de VCS

Teniendo en cuenta que el uso extensivo de microservicios atenta contra la mantenibilidad del sistema y considerando que se eligió como atributo de calidad la mantenibilidad, se consideró mejor empaquetar los 5 microservicios en uno. Esto permitiría más fácilmente agregar nuevos tipos de repositorios, conforme a la demanda lo requiera, dado que solo se tiene que correr un nuevo servidor dentro del servicio de VCS, en vez de generar un nuevo microservicio entero.

De esta forma, generamos un nuevo esquema:

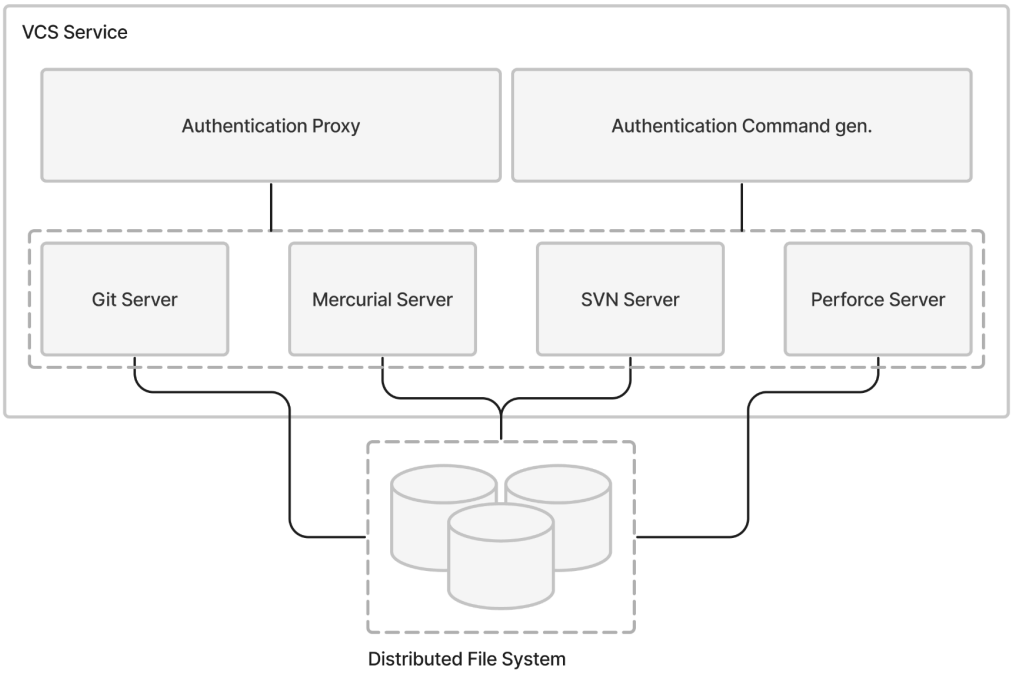


Diagrama 3: esquema final de servicio de VCS

5.1.3.3. Flujo de acceso a repositorios

Se consideraron 2 patrones de acceso para los repositorios:

1. Un usuario accediendo al repositorio para plasmar cambios en este (ej. pushear un commit a un repositorio Git).
2. Un servicio externo obteniendo el código y/o commits/updates de los repositorios para proveer un servicio adicional (ej. code reviews).

A continuación se mostrará la arquitectura usada para estos componentes de la aplicación:

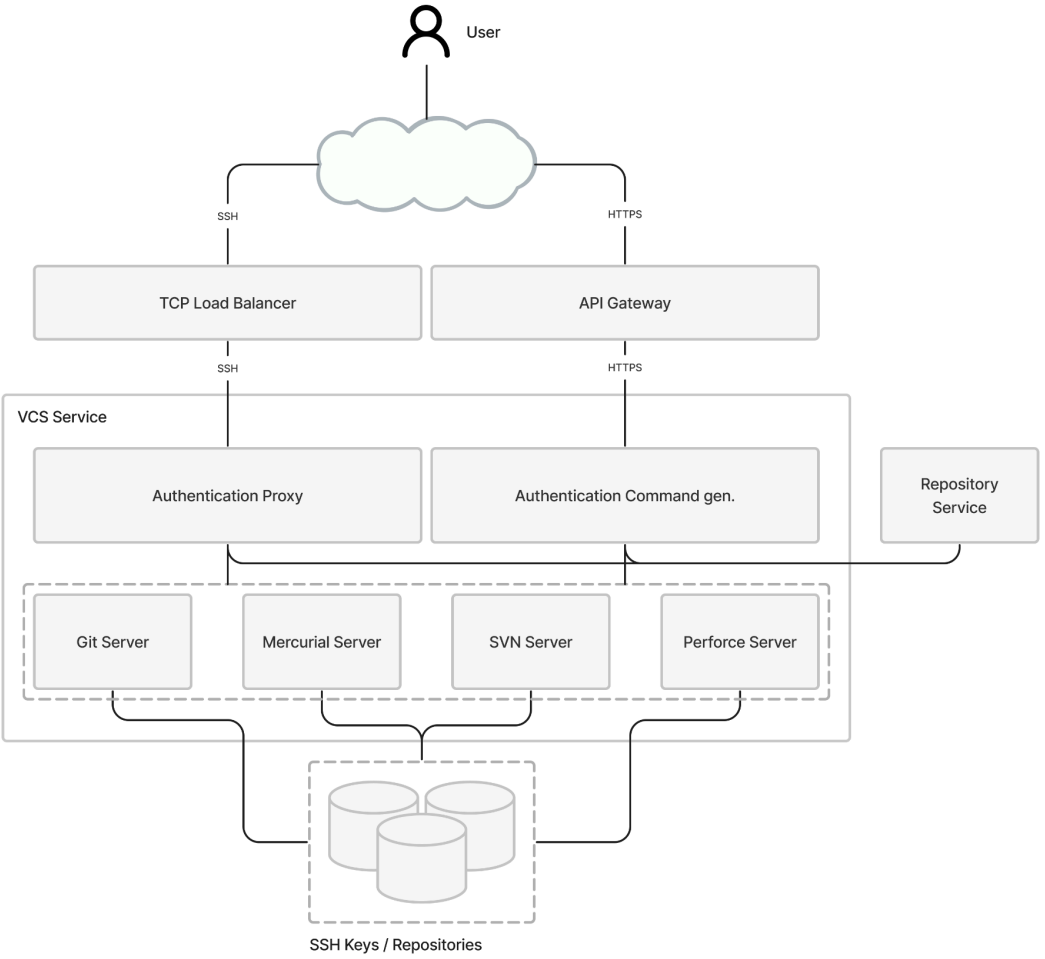


Diagrama 4: Flujo de acceso a repositorios

Comenzando por el acceso por parte de un usuario, se decidió permitir este acceso mediante el uso del protocolo SSH, dado que es un estándar establecido y aceptado por los servidores de Git, Mercurial, SVN y Perforce que se usarían para interactuar con los repositorios. Se usará un SSH Key para validar la identidad del usuario. A este fin, el usuario debe:

1. Generar una clave SSH, almacenando localmente la clave privada.

2. Acceder a la webapp y agregar la clave pública como clave aceptada, la cual asocia la clave pública SSH con el usuario y con sus repositorios. El almacenamiento de las SSH keys se hará en el *Distributed File System* (DFS), al cual tendrá acceso el servidor de *Authentication Proxy*. También se almacenará en el servicio de *repository* la asociación entre el repositorio, usuario y SSH key.

Teniendo ambas claves SSH almacenadas correctamente, el usuario podrá acceder mediante una conexión SSH a los repositorios:

1. Inicializar una sesión SSH con el *load balancer*, la cual se hace generando una conexión con la URL correspondiente al repositorio, que redirecciona el pedido a una de las instancias del servicio de VCS.
2. El servicio de VCS, específicamente el servidor *Authentication Proxy*, que internamente tiene un servidor SSH, aceptara la sesión SSH (en el caso que sea válida la SSH key).
3. Teniendo la sesión abierta, el cliente SSH del usuario enviará el comando que se quiere ejecutar, el cual tiene el repositorio al que se quiere acceder. El servidor de *Authentication Proxy* luego podrá verificar, mediante el uso del servicio de *repository*, si el usuario asociado a esa SSH key puede acceder al repositorio.
4. En el caso positivo, el *Authentication Proxy* generará una nueva conexión SSH, usando una clave interna del servicio, pero ahora con el servidor de repositorio (Git, Mercurial, SVN, etc) y luego enviará el comando inicialmente enviado por el usuario.
5. Finalmente, teniendo la respuesta del servidor del repositorio, se enviará mediante la sesión SSH al cliente.

Por otro lado, se debió permitir que servicios externos obtengan el código y/o commits/updates de forma programática. En este caso, en vez de usar SSH para acceder directamente al repositorio, se realiza el acceso mediante HTTPS, de modo que el API Gateway será el encargado de redirigir la petición al servicio de VCS. De todas formas, se debe autenticar el servicio externo mediante el uso de una API token, que cumple un uso similar al de usar un SSH key para validar un usuario.

Continuando, se mantiene el siguiente flujo:

1. Teniendo esta API token asociada a un repositorio y servicio externo, para poder acceder a dicho repositorio, el servicio externo debe hacer un pedido al API Gateway, el cual usa el servicio de ACL para validar que el token realmente es válido.
2. El API Gateway hace el pedido al servicio de VCS.
3. Al igual que con el flujo anterior, el servicio autentica, usando el servicio de *repository*, que el servicio de tercero tiene acceso al repositorio.
4. Y aquí difiere con el flujo anterior: en vez de actuar como proxy, forwardeando el pedido a los servidores de alguno de los VCS (Git, SVN, etc.), el servicio genera la operación correspondiente ya que solo se debe permitir obtener código y commits/updates, y no alterarlo.

Teniendo en cuenta este último punto, se definieron los comandos que se usarán para cada tipo de repositorio:

- Git: *pull*, *fetch*
- SVN: *update*
- Mercurial: *pull*
- Perforce: *sync*

5. Habiendo enviado el comando apropiado al servidor correspondiente, se obtiene la respuesta que luego se envía devuelta al API Gateway y luego al servicio externo.

5.1.4. Servicio de Webhooks

El servicio de Webhooks permite disparar un callback HTTP ante cierta acción ejecutada en el repositorio de código. Los webhooks abren una amplia variedad de escenarios en los cuales pueden ser utilizados:

- Disparar procesos de CI (*Continuous Integration*) en servicios externos
- Enviar notificaciones a sistemas de mensajería
- Vincularse con herramientas de gestión de equipos
- Realizar despliegues en servidores
- Utilizarlos para propósitos de auditabilidad

Más en detalle, cuando el usuario realiza alguna acción sobre el sistema, el servicio encargado de dicho dominio, obtendrá información pertinente desde el servicio de repositorios. El servicio de repositorios tendrá a su vez registrado qué eventos dentro de ese repositorio tienen registrados una URL de webhook. Si la acción que se acaba de registrar corresponde a una suscrita para ser notificada mediante webhooks, se emitirá un evento a la cola de mensajes para que el servicio de webhooks se encargue de hacer la petición correspondiente.

La cola de mensajes permitirá desacoplar a cada uno de los servicios del servicio de webhooks, el cual atenderá a los eventos publicados a medida que los vaya concretando.

5.1.4.1. Flujo de webhooks

Antes de que se pueda ejecutar el flujo de los webhooks, el usuario debe configurarlo. El usuario deberá registrar en la sección de configuración de su repositorio a qué URL la plataforma deberá emitir una operación POST, y ante cuál de los eventos. Esta URL puede ser proporcionada por un servicio propietario del usuario o uno tercerizado en la nube.

Teniendo esto, se sigue el siguiente flujo:

1. Se ejecuta cierta acción en un repositorio administrado por el servicio de VCS (ej. *push* a un repositorio Git).
2. El servicio de VCS verifica si existen webhooks asociados a dicho repositorio y si la acción ejecutada cumple con la configuración dictada por el usuario.

3. El evento se publica en la cola de mensajes.
4. Eventualmente, el servicio de webhook, recibe el evento de la cola de mensajes y envía del POST al URL especificado, junto al payload requerido.

5.1.4.2. Patrón adapter

En un primer instante, se consideró tener un único tipo de payload propietario para los webhook. Esto simplifica el desarrollo, pero obligaría que los sistemas de terceros tengan que generar lógica específica para operar con nuestros webhooks. Este *trade-off* se consideró inaceptable dado que, como rige uno de los atributos no funcionales definidos, se debe permitir una fácil integración con servicios de terceros. Por lo que se debe tener una forma de amoldar el *payload* generado para poder soportar otros estándares usados por terceros.

Se terminó eligiendo implementar un patrón adapter, el cual facilita la incorporación e integración con sistemas de terceros. Usando esto, se podría implementar fácilmente adaptadores para varios de los webhooks más usados (como los provistos por Atlassian Jira, SNS AWS, Slack, entre otros), y expandir la oferta de manera transparente a medida que la demanda lo requiera.

5.1.5. Servicios de issues y time tracking

Este servicio se encarga de proveer la funcionalidad de *issues* y *time tracking*. Cabe aclarar que se consideran directamente relacionados estos dos y por ende, se debían incorporar en un único servicio.

Se decidió implementar un sistema de *issues* similar al que tiene GitLab. Se permite que los usuarios habilitados generen issues (los cuales figuran como creados por este usuario). Dentro del *issue* se define su título, un texto de descripción, su estado actual y finalmente una estimación de tiempo, que luego permite marcar cuánto tiempo tardó en resolverse. Este último punto es el *time tracking*, que permite manejar estimaciones de tiempo y luego ver el uso total.

El servicio almacenará los datos correspondientes a los *issues* en una base de datos SQL y será accedida mediante la API Gateway.

5.1.6. Servicio de wiki

El servicio de wiki maneja las wikis asociadas a los repositorios. Las wikis permiten describir la arquitectura de los repositorios, generar guías de instalación, lineamientos de código, tener FAQs, entre otras cosas. Se define que cada repositorio puede tener una wiki asociada y usará una estructura jerárquica, junto a contenido Markdown, para representar la wiki.

El servicio permite crear y editar páginas, asociando a ellas una posición en la estructura jerárquica de la wiki, cuyo contenido viene en formato Markdown y pueden incluir imágenes.

Al igual que los otros servicios, este se accede mediante la API Gateway. Además, deberá comunicarse con el servicio de repositorio y usuarios para verificar que el usuario puede obtener y hacer cambios a la wiki.

5.1.7. Servicio de releases

El servicio de *releases* permitirá almacenar y luego distribuir los binarios correspondientes a un repositorio, junto a notas de lanzamiento y la versión de *release*. Esto permite distribuir versiones estables de una forma simple.

Para generar un nuevo *release*, un usuario autorizado deberá subir los binarios compilados que, como el resto de los servicios, deberán pasar primero por la API Gateway. Los binarios luego se persisten en una base de datos Blob. Dependiendo de la visibilidad del repositorio, estos binarios se podrán obtener por todos los usuarios (en el caso de los proyectos *open source*) o solo aquellos que estén autorizados (en el caso de los repositorios privados). Además, debido a que pueden existir releases privados se decidió encriptar la base de datos de *releases*.

5.1.8. Servicio de pagos

Para implementar la facturación de los sistemas comerciales se decidió establecer un convenio con Stripe. Las empresas contarán con una suscripción a los servicios corporativos del sistema.

Se decidió tercerizar este servicio debido a la gran cantidad de desafíos que conlleva implementar estos sistemas de cero: seguridad, detección de fraude, respuesta ante fallos en los métodos de pagos, etc.

Se optó por Stripe como método de pago principal debido a su éxito en multitud de países y su madurez en el mercado, además de ofrecer 99,999% de uptime [35]. Adicionalmente, esta plataforma cuenta con soluciones e integraciones con multitud de servicios existentes (incluyendo SaaS) y librerías para React (tecnología utilizada para el SPA).

En cuanto a funcionamiento, el microservicio de pagos se comunicará con Stripe para realizar la facturación a la empresa. La comunicación se hará a través de HTTPS para proteger la información en tránsito. Stripe provee el almacenamiento de los métodos de pago, por lo que estos datos no serán guardados en la base de datos del sistema.

A su vez, los detalles de la transacción se guardarán en la base de datos de facturación. De esa manera, en caso de un error, la información quedará almacenada para determinar cuál fue la causa del mismo. De todas formas, Stripe cuenta con servicios para recuperación y reintento de la facturación.

5.1.9. Servicio de repository

Los usuarios tendrán la posibilidad de crear repositorios donde los usuarios pueden no solo almacenar su código, sino también abordar y dar seguimiento a problemas específicos, mantener un historial de cambios detallado y vincular información a través de una wiki asociada. El servicio de *repository*, se encargará de la gestión de los repositorios de código fuente. Conocerá todos los metadatos correspondientes a un determinado repositorio, como pueden ser descripciones, configuraciones, quiénes son los colaboradores y desarrolladores del proyecto.

Permitirá también la búsqueda de repositorios, lo cual será especialmente útil para proyectos *open source*, que quizás lo requieran para ser encontrados por posibles contribuyentes.

5.1.10. API Gateway y ACL

Para el acceso a los distintos servidores backend y endpoints de nuestros microservicios, se antepone una API Gateway a fines de orquestar los distintos *request* y aportar mecanismos de seguridad en el acceso a los mismos. Para ello, se implementa una capa de seguridad con un ACL que utiliza Json Web Tokens (JWT) a fines de proporcionar un mecanismo de autenticación. Este mecanismo permite trabajar de manera *stateless* con los servidores backend de la plataforma, permitiendo tener una mejor distribución de carga.

Por otro lado, se generan API keys del lado de la API Gateway para proporcionar control de acceso y autorización a servicios de terceros.

5.1.11. Load balancer SSH

Para que múltiples usuarios puedan trabajar en sus repositorios y actualizar u obtener el código fuente de sus proyectos directamente desde su terminal, se podrán conectar con la plataforma de forma segura mediante SSH. Dado que se espera una amplia cantidad de usuarios conectados en simultáneo, se deberá dividir la carga entre las distintas instancias del servicio de VCS. Con este objetivo, se decidió complementar el flujo con un load balancer.

Sin embargo es importante decidir el tipo de load balancer que se utilizará, y la configuración del mismo, ya que los load balancers utilizados para HTTP no sirven para cumplir con estos requerimientos. Es por esto que se utilizará un load balancer de TCP, el cual soporte el protocolo PROXY.

5.1.12. Webapp

La aplicación web proporcionará una interfaz gráfica intuitiva y accesible que permitirá a los equipos aprovechar plenamente las diversas funciones de gestión y colaboración ofrecidas por la plataforma. En la misma se podrán crear, modificar y eliminar repositorios, así como también añadir colaboradores, analizar versiones anteriores y nuevos cambios y consultar los nuevos *issues* que se agregaron, además de realizar un seguimiento de tiempo. Adicionalmente, los usuarios podrán explorar la wiki asociada a sus repositorios. Esta wiki estará formateada en HTML, con secciones escritas en lenguaje Markdown, ofreciendo una experiencia de lectura clara y organizada.

Es crucial que el diseño de la interfaz asegure la accesibilidad para una amplia gama de usuarios, minimizando cualquier fricción. La simplicidad y la claridad serán principios centrales para garantizar que los usuarios se sientan capacitados y cómodos al interactuar con la plataforma. La interfaz se estructurará de manera que los usuarios siempre tengan el control, brindando una experiencia fluida y sin complicaciones mientras exploran, colaboran y administran proyectos en la plataforma web. Dicha atención a la usabilidad garantizará que la plataforma sea eficaz para equipos con diversos niveles de experiencia y habilidades.

5.1.13. Service Discovery

Al implementar una arquitectura de microservicios, es importante que el sistema conozca cuáles servicios se encuentran disponibles y listos para recibir peticiones. Es por esto que se decide implementar el patrón de *service discovery*, el cual facilitará la identificación dinámica y la comunicación entre servicios que se encuentran en constante cambio.

Para llevarlo a cabo, se creará un servicio denominado "Service Registry" que contendrá información sobre los servicios que se encuentran disponibles en el sistema, como su nombre, dirección IP, puerto entre otra metadata útil. Cada una de las instancias de los distintos servicios que componen la arquitectura, tendrán la responsabilidad de registrarse en dicho servicio en cuanto estén disponibles. Esto permitirá que el API Gateway consulte el registro para verificar la disponibilidad del servicio al que se intenta acceder y, al mismo tiempo, proporcionarle a los propios servicios conocimiento acerca de los demás servicios que puedan necesitar.

5.2. Elección de tecnologías

5.2.1. Servidores de sistemas de control de versiones

Para poder acceder y operar con los distintos repositorios, elegimos usar distintos servidores:

- Git: usaremos un servidor SSH, en este caso la implementación de *OpenSSH*, junto al comando *git* de línea de comandos.
- Mercurial: usaremos la herramienta open source llamada *HGKeeper* [19] que provee un servidor simple para manejar repositorios de Mercurial.
- Subversion: se usará el daemon *svnserve* [20], un servidor para operar con repositorios SVN, provista por la herramienta Subversion.
- Perforce: se usará Perforce Helix Core [21], una herramienta propietaria que también permite acceder mediante SSH a los repositorios. Cabe destacar que esta es la única solución closed-source y paga.

El criterio principal de elección que se usó es que se debía poder acceder mediante SSH, dado que esta sería la principal manera que se utilizaría para acceder a los servidores en los distintos flujos de acceso.

Los servidores también debían ser lo más simples y livianos posibles, para no malgastar los recursos computacionales. Se menciona esto dado que también se consideraron herramientas como Gitea y GitLab Community Edition en el caso de Git, las cuales proveen mayor funcionalidades como un UI integrado o integración con Github Actions preexistentes, pero con la desventaja de ser demasiado costosos computacionalmente [16].

5.2.2. Cola de mensajes de los webhooks

En los webhooks se investigó acerca de las posibles colas de mensajes a utilizar. En particular, se analizaron dos alternativas *open source* con el objetivo de abaratar costos: Apache Kafka y RabbitMQ. Lo que más se valoró a la hora de tomar la decisión fue que garantice una alta disponibilidad.

En primer lugar, se comenzó analizando Apache Kafka [9]. Esta cola de mensajes provee escalabilidad, alta disponibilidad y tolerancia a fallas, por lo que es muy utilizada para soluciones *real-time*. Dentro de las cosas buenas de Kafka está su comunidad y su extensa documentación. Por otro lado, resulta una solución compleja de implementar dado que es un sistema distribuido con muchos componentes. Además, requiere un *overhead* operacional para mantener y administrar el sistema, por lo que se requiere de un equipo especializado en esta tarea. Por lo tanto, si bien es una solución efectiva, resulta demasiado compleja para el presente caso de uso que no requiere de procesamiento en tiempo real [8].

La otra alternativa que se analizó es RabbitMQ [10]. Esta alternativa asegura alta disponibilidad, pero la documentación que existe es escasa y pobre. Esto sería un trade-off que habría que hacer entre mantenibilidad y disponibilidad, pero dado que la alternativa de Kafka resulta demasiado compleja para el caso de uso que se busca atacar se cree que vale la pena hacerla [11].

Se buscaron otras alternativas, pero se descartaron por ser costosas frente a las previamente mencionadas.

Consecuentemente, se eligió RabbitMQ como cola de mensajes para los Webhooks.

5.2.3. Persistencia

5.2.3.1. Almacenamiento de los repositorios

Sabiendo que los servidores de SVN, Git, Perforce y Mercurial usan el file system para almacenar los repositorios, y también considerando que se debían tener múltiples instancias de dichos servidores para garantizar la disponibilidad, se debió encontrar una manera de acceder el sistema de archivos de forma no local. Se decidió usar un sistema de archivos distribuido (DFS), que permitiría acceder al sistema de archivos de forma transparente desde múltiples instancias.

Se consideraron múltiples alternativas: GlusterFS, CephFS y JuiceFS.

Comenzando por GlusterFS, creado por Gluster Inc. y mantenido por Red Hat, este DFS permite generar pools de almacenamiento, que permiten ser fácilmente extendidos, y pueden ser usados como sistemas de archivos. Entre los features destacados se incluye correr en hardware “commodity” y que tiene sistemas para failover automático.

Una de las razones por las cuales se decidió no usar GlusterFS es la compañía que mantiene el proyecto. Hace ya varios meses Red Hat eliminó el libre acceso a Red Hat Enterprise Linux, una distribución de Linux [25]. Por esto, se consideró más riesgoso usar una herramienta que también es mantenida por Red Hat y que podría caer en el mismo problema.

Luego, JuiceFS es un DFS open source que tiene como objetivo ser fácil de usar, performante, altamente disponible (el cual especifica que tiene una uptime de 99.95%) y con

facilidad de escalar [4]. Es compatible con los protocolos POSIX, HDFS y S3, lo cual facilita el desarrollo de sistemas que lo usan, dado que no se tiene que aprender un sistema propietario.

Dicho eso, tuvo su primer release estable en agosto de 2022 [5]. Por esta razón, se decidió descartar esta opción. Por más completa que sea su funcionalidad, teniendo en cuenta que se almacenarán repositorios de compañías y proyectos open source, se consideró demasiado riesgoso usar una herramienta con poca trayectoria como es JuiceFS.

Finalmente, se consideró CephFS, un DFS construido usando Ceph, una herramienta de almacenamiento de objetos distribuidos. Esta, como las otras herramientas, es compatible con los protocolos POSIX, y asegura una alta disponibilidad y performance [6]. Permite ser deployado sobre *commodity hardware*. Cabe destacar que a diferencia de JuiceFS, su primer release estable fue en el 2013 [7], por lo que presenta más tiempo en el mercado.

Otro aspecto que aporta a la fiabilidad de la herramienta es su uso en OpenStack, una herramienta open source para generar plataformas cloud [17], desarrollada por Rackspace Computing y NASA.

Habiendo elegido CephFS y con el fin de tener un cierto nivel de disponibilidad y resiliencia, se eligió tener un factor de replicación de 3, es decir, el sistema mantendrá 3 réplicas en todo momento. Nos parece que esto provee un balance entre la disponibilidad y el costo total de usar dicha cantidad de réplicas, y es la cantidad recomendada por varios proyectos que usan Ceph y CephFS por debajo [18].

Cabe también mencionar que el CephFS correrá sobre un container orquestado por Kubernetes, de la misma manera que el servicio de VCS, que lo usará.

5.2.3.2. Almacenamiento de Releases

Los releases son archivos comprimidos del repositorio, por lo que para el almacenamiento de estos datos se precisará una base de datos que guarde el tipo de dato Blob. Se evaluaron las distintas opciones ponderando principalmente la performance. el costo de adquirirlas y la facilidad de uso. Las opciones evaluadas fueron: PostgreSQL, Amazon S3, Minio y Ceph.

En primer lugar, si bien PostgreSQL tiene un tipo de dato parecido al Blob llamado ByteA, no se recomienda hacer un uso intensivo de este tipo de dato. En el caso de la plataforma de desarrollo, se espera un uso frecuente de la funcionalidad de releases, por lo que se descartó esta opción [12].

Como alternativa IaaS se analizó Amazon S3 [32]. Este servicio es extremadamente fácil de usar y provee un buen nivel de seguridad. Además, es altamente escalable, posee baja latencia y provee almacenamiento prácticamente ilimitado. Este almacenamiento es del tipo *add-as-you-grow*, por lo que se pagaría solamente por lo que se usa. La API REST que provee también es muy conveniente y práctica para interactuar con el sistema; y provee una gran documentación y comunidad [15]. En cualquier caso, el costo asociado a contratar este servicio resulta superior al de otras alternativas, aún teniendo en cuenta los beneficios que trae, por lo que se descartó esta opción.

Minio es una alternativa *open source* con licencia de uso libre y comercial [13]. Se utiliza principalmente para aplicaciones de Inteligencia Artificial o Machine Learning, por lo

que pone mucho énfasis en la performance de las lecturas. Además, es capaz de correr sobre cualquier infraestructura *on-premise* o en la nube (aunque en el caso de este proyecto se utilizaría la primera), y es extremadamente fácil ponerla en producción. La comunidad que la rodea es muy grande y la documentación es extensa [14]. De todos modos, se terminó descartando esta opción por Ceph por las razones que se verán a continuación.

Ceph es un sistema *open source* de almacenamiento de objetos. Al igual que Minio, cuenta con una gran comunidad y detallada documentación. Es muy performante, provee durabilidad, tolerancia a fallos y es escalable. Asimismo, se puede correr sobre *commodity hardware* y provee una API para interactuar programáticamente con ella. A diferencia de Minio donde el setup era muy simple, aquí esto resulta una tarea compleja [14]. De todos modos, se debe tener en cuenta que ya se eligió el sistema de archivos de Ceph (CephFS) para los repositorios, por lo que ya se contará con desarrolladores especializados y no supondrá un esfuerzo extra el implementar esta solución. Además, es importante mantener la cantidad de sistemas distintos al mínimo dentro de la arquitectura para aumentar la mantenibilidad.

Es por todo esto que se optó por Ceph como sistema de almacenamiento de los releases.

5.2.3.3. Almacenamiento de Wikis

Las Wikis de un repositorio contienen toda la documentación del mismo. Están conformadas por directorios y subdirectorios, además de archivos de texto como Markdown, imágenes, etcétera.

Para elegir una opción de almacenamiento y siguiendo los lineamientos de la sección anterior, se decidió por no extender el framework tecnológico, es decir, trabajar con PostgreSQL o CephFS para no complejizar el proyecto.

Debido a que la funcionalidad de Wikis es secundaria y no tendrá demasiados accesos comparado al repositorio central, se descartó el sistema distribuido de CephFS para abaratar costos. Entonces, la información se guardará en una base de datos PostgreSQL, que tendrá una estructura recursiva para definir directorios y subdirectorios, mientras que los otros tipos de archivos se guardarán como ByteA. Como la Wiki se encuentra definida de forma recursiva se degradará la performance, pero se consideró despreciable en relación a la cantidad de lecturas que tendrá la misma.

5.2.3.4. Almacenamiento de Facturas

Si bien se terciarizará el servicio de pagos, se guardarán en la base de datos de facturación los detalles de la transacción a modo de registro, buscando asegurar un alto nivel de seguridad. Este punto será explicado más adelante (ver Sección 5.3.2) cuando se hable de los distintos atributos de calidad y la manera de asegurarlos.

Para esta funcionalidad se utilizará una base de datos PostgreSQL pues es suficiente para lo que se busca resolver.

5.2.3.5. Almacenamiento de issues y time tracking

Se decidió proveer una funcionalidad de *time tracking* similar a la provista por Gitlab [22] y asociada solamente a los *issues*. De esta manera, se puede introducir un valor estimado

de tiempo que tardará un issue en ser resuelto, y a medida que avanza la resolución del *issue* se puede actualizar dicho valor manualmente. Para esta funcionalidad se eligió una base de datos PostgreSQL pues es suficiente para el caso de uso que se busca resolver.

5.2.3.6. Almacenamiento de usuarios

El almacenamiento de usuarios requiere de una atención especial dado que se busca brindar un alto nivel seguridad para evitar accesos maliciosos a los repositorios, posiblemente confidenciales o que llevaron años de trabajo, y así arruinar compañías enteras. Por esta razón se decidió utilizar SHA-256 como algoritmo de encriptación, junto con Salt para evitar ataques de "Rainbow Table" [23]. Asimismo, dado que se busca el máximo nivel de seguridad posible en este punto, se hará *encryption-at-rest* de esta base de datos, por lo que utilizar Pepper no será necesario. El sistema de base de datos utilizado será PostgreSQL, y tendrá una replicación maestro-esclavo pasivo.

5.2.5. Lenguajes para los microservicios

Para la elección de los lenguajes que utilizarán los microservicios se decidió tener en cuenta la mantenibilidad y la posibilidad de utilizar librerías con los sistemas que estamos utilizando.

RabbitMQ y PostgreSQL proveen librerías para C, Java, Python, Rust y Go, entre otros lenguajes [26][27]. Por otro lado, Ceph tiene librerías para C, Java y Python [28]. Considerando que se busca la mantenibilidad del sistema, se optó por elegir Java, ya que es un lenguaje de alto nivel, fuertemente tipado y ampliamente utilizado en la industria, por lo que se podrá encontrar desarrolladores fácilmente.

5.2.6. Hosting y containers

Se decidió utilizar Kubernetes para manejar los contenedores de Docker donde correrán los microservicios pues es la alternativa más conocida. A pesar de ser compleja de administrar y configurar, es utilizada extensamente en la industria y es la solución *de facto* para este tipo de arquitecturas.

Al momento de elegir donde *hostear* los servidores se tuvieron en cuenta diversas opciones de IaaS. La razón por la cual no se consideraron opciones propias es que se cree que la mantenibilidad del proyecto, al tener tantos componentes, se complejiza enormemente. Asimismo, el TTM se vería enormemente retrasado. Además, las alternativas de IaaS son variadas y muy utilizadas en la industria, por lo que los servicios provistos suelen ser maduros y cuentan con la documentación necesaria. Se consideraron como aspectos importantes al evaluar estas soluciones las siguientes características: precio, escalabilidad, facilidad de uso, documentación disponible, tamaño de la comunidad de desarrollo. Con esto, se busca lograr un buen balance entre mantenibilidad, escalabilidad y costos. Se evaluaron dos opciones en profundidad dada su popularidad entre desarrolladores: Amazon EC2 y Google Cloud Engine.

Amazon EC2 es una herramienta que viene siendo utilizada desde 2008, mientras que Google Cloud Engine (GCE) surge como una alternativa a esta plataforma en el 2012. EC2 está muy instalada en la industria, disponiendo de una extensa documentación y comunidad

de desarrollo. Si bien GCE es también muy utilizada, la comunidad de EC2 es superior y se puede ver en sitios como “StackShare” [30] donde los seguidores de EC2 triplican a los de la alternativa de Google. Asimismo, EC2 ofrece distintos modelos de precios, de los cuales el más interesante para la plataforma de desarrollo es el de “instancias reservadas”, ya que se maximiza el ahorro pues busca ofrecer el servicio por un largo período de tiempo [29]. Además, ofrece descuentos del 10% si se utiliza un uptime entre el 99% y 99.95% como es nuestro caso, aunque esto es algo que también ofrece GCE. Otro punto en común que tienen ambas soluciones es la posibilidad de tener *whitelists* y maneras de manejar tanto el tráfico tanto entrante como saliente [29]. Por otro lado, EC2 provee funcionalidad de Enhanced Networking que mejora el *network throughput* que no ofrece GCE, aunque finalmente EC2 pierde en la relación costo-performance igualmente pierde frente a dicho sistema [30].

En conclusión, se considera que la documentación y comunidad que rodea a Amazon EC2 supera la relación costo-performance en la que pierde frente a GCE. Se debe recordar que la performance no es un atributo que se está buscando asegurar, mientras que la mantenibilidad sí lo es. Es por ello que se eligió Amazon EC2 como plataforma para hostear nuestra plataforma.

5.2.6. Load balancer

Para implementar el load balancer para las peticiones SSH se investigaron dos posibles alternativas: NginX y HAProxy.

NginX ofrece varias soluciones para servidores, incluyendo load balancing. Además, cuenta con amplia documentación y tiene una gran cantidad de empresas que lo usan, incluyendo Netflix, Dropbox, etc. [31]

Por otro lado, también se consideró la herramienta *open source* HAProxy. En principio, ofrece las mismas ventajas que NginX: se trata de un producto consolidado y maduro en su desarrollo. Sin embargo, se encuentra altamente especializado en equilibrio de carga.

Finalmente, se optó por el servicio de HAProxy debido a que solo se requería un load balancer robusto en el sistema, sin funcionalidades extra, sumado a que el mismo es utilizado por servicios como Github [33] y ofrece una configuración directa para SSH [34].

5.2.7 Webapp

Para que los usuarios de la plataforma puedan registrarse, ingresar, crear proyectos y analizar cambios en los repositorios, obtener información sobre los problemas, realizar seguimiento del tiempo, entre otras funciones, se desarrollará una aplicación web consistirá en una SPA desarrollada en React. Se seleccionó este framework dado que el mismo posee una amplia comunidad open source, por lo que se encuentran disponibles muchos paquetes que ayudarán a solucionar los problemas más comunes durante el desarrollo, disminuyendo el time to market.

En lo que conlleva ofrecer una cómoda experiencia de usuario, se optará por utilizar una librería de componentes como Mantine [39], la cual no sólo ofrece componentes testeados en múltiples dispositivos si no que también posee consideraciones con respecto a la accesibilidad de los mismos. Mantine posee una documentación clara, y un soporte de la

comunidad constante. Esta librería permitirá llegar a una estructura intuitiva de la interfaz, facilitando la creación de repositorios y sobre todo la configuración de los entornos de trabajo. Por otra parte, al ser un sistema global, es importante considerar ambos modos de lectura (LTR y RTL), los cuales ya vienen implementados listos para utilizar. Además, posee implementada la navegación por los componentes con teclado y, entre otras funcionalidades, podemos encontrar la reducción de movimiento, esquemas de colores, la cual será configurable para el usuario.

En particular para el caso de las wikis de cada repositorio, se optó por utilizar el parser de Markdown llamado Marked.js. Esta librería se encargará de convertir el lenguaje Markdown en HTML para poder ser renderizada en la aplicación web, y obtener una visualización más clara, pero habiendo escrito la documentación en un lenguaje más cómodo como Markdown. Entre las cualidades de esta librería, podemos encontrar que acepta flavors de Markdown de otras plataformas como GitHub, lo cual evitaría que los desarrolladores tengan que adaptarse a otras sintaxis en caso de migrar a esta plataforma.

Adicionalmente, este módulo estará hosteado en una CDN para mejorar la experiencia de usuario al reducir tiempos de respuesta y tener un sistema distribuido para que no se interrumpa el servicio ante grandes cantidades de tráfico. La tecnología para CDN elegida es Cloudflare, debido a su alta popularidad ya que ofrece un 100% de uptime estipulado por el SLA en sus versiones Business y Enterprise (en caso de incumplimiento del mismo, al cliente se le devolverá parte del crédito acorde a minutos sin servicio y cantidad de usuarios afectados) [37].

5.3. Resolución de atributos

5.3.1. Disponibilidad

Al momento de elegir la mayoría de las tecnologías, se tuvo en cuenta la disponibilidad como uno de los atributos a garantizar. Se eligió entonces RabbitMQ como cola para los webhooks, y CephFS para sostener la funcionalidad principal de la plataforma: el almacenamiento de los repositorios. Este último permite tener replicación de los datos almacenados, lo cual elimina un single point of failure.

Asimismo, cada una de las bases de datos PostgreSQL contará con un sistema de replicación activo-pasivo, por lo que si una instancia falla, otra tomará su lugar inmediatamente asegurando así la continuidad del servicio.

Dado que el sistema se hosteará en EC2, se tiene asegurado un nivel de disponibilidad altísimo a través del SLA.

Por el lado de los servicios, se usaría Kubernetes para orquestar los contenedores que los corren, junto a 2 load balancers (uno por el lado del acceso SSH a los repositorios y otro por el lado de la API), para aumentar aún más la disponibilidad del sistema.

Finalmente, se hosteo la webapp en React con la CDN de Cloudflare, ya que permite recuperarse si una instancia falla debido a su sistema distribuido.

5.3.2. Seguridad

Para el acceso al sistema desde la SPA se implementará ACL y 2FA para todos los usuarios con el objetivo de proveer seguridad a las cuentas y sus repositorios, en especial si se trata de usuarios que tienen acceso a repositorios privados. Las contraseñas de los usuarios se almacenarán utilizando SHA-256, un algoritmo de encriptación robusto y ampliamente usado. Junto con esto se utilizará Salt para garantizar una altísima seguridad en las contraseñas. Además, los sistemas de almacenamiento de usuarios (PostgreSQL) y de repositorios (CephFS) tendrán *encryption-at-rest*.

Las comunicaciones se realizan por HTTPS tanto para los sistemas de pago como para la comunicación con la SPA, reduciendo el riesgo de ataques MITM a un mínimo. Este riesgo se verá atacado también por la implementación de "Certificate Pinning" [36] en la webapp. La única consideración adicional de tomar esta medida es que se debe recordar cambiar el certificado fijado en la Webapp en caso de que se cambie el que se utiliza, lo que reduce un poco la mantenibilidad del sistema (esto sería un *trade-off*), pero dado que es algo infrecuente no se considera un problema. De esta manera, se elimina completamente la posibilidad de realizar ataques MITM y robar datos de facturación. Además, el tráfico relacionado a las acciones sobre los repositorios (push, pull, etc.) se hará por SSH con el objetivo de aislar completamente esas operaciones.

Con respecto al uso de la aplicación web, se sanitizarán todos los documentos Markdown para wikis de repositorios almacenados en el sistema con el fin de evitar prácticas de XSS, dado que Markdown admite tags de HTML en su contenido y al ser renderizado, los usuarios podrían verse afectados. Además, se verificarán los datos que se envían desde el frontend al backend con esquemas de tipado en runtime (utilizando una librería como Zod [38]) y por otra parte, se sanitizarán los ingresos del usuario en inputs, evitando SQL Injection.

5.3.3. Escalabilidad

La escalabilidad del sistema busca asegurarse tanto por la elección de arquitectura como por la elección de las tecnologías subyacentes.

En primer lugar, se optó por una arquitectura de microservicios que permite crecer los servicios de manera independiente y según la demanda lo requiera. En este punto, es importante notar que el mayor cuello de botella se encontrará en el servicio de repositorios y de VCS. Sin embargo, esto no es un problema pues la elección de una arquitectura de microservicios permite escalar el sistema solo en aquellos servicios que lo requieran. Además, la elección de RabbitMQ para la cola de webhooks es altamente escalable, por lo que no se encontrarán cuellos en este punto tampoco.

CephFS es también una tecnología que permite ser escalada sin *downtime.* Además, al tratarse de un sistema de archivos distribuido, las lecturas se distribuyen a lo largo de varios servidores por lo que la performance ante un gran número de accesos prácticamente no se ve afectada.

Para manejar un incremento en la cantidad de *requests* SSH a los distintos servidores de VCS, se optó por un loadbalancer HAProxy que se encarga de distribuir las peticiones a lo largo de las instancias.

5.3.4. Mantenibilidad

La mantenibilidad del sistema fue uno de los puntos más difíciles de asegurar. Siendo uno de los atributos de calidad menos prioritarios, en varios puntos se tuvieron que hacer *trade-offs* entre este y otros más importantes.

En primer lugar, si bien la elección de Ceph y CephFS como sistemas de almacenamiento va en detrimento de este atributo dada la dificultad de configuración y mantenimiento de las herramientos, la decisión de utilizar ambos sistemas iguales en lugares distintos de la arquitectura busca minimizar la heterogeneidad de la misma.

En cuanto al código, se tomarán diversas medidas que buscarán asegurar la calidad, y por lo tanto mantenibilidad, del mismo.

Se implementará un sistema de *code-review* con énfasis en las piezas de código más importantes, pero no limitado a ellas. Además, con el objetivo de no entorpecer esta práctica, se hará hincapié en mantener la cantidad de líneas en un *pull request* en valores manejables (idealmente menor a 600 líneas de código). De esta manera, se estará logrando que los desarrolladores tengan un mejor conocimiento de la *codebase* a la par que se detectan errores antes de poner el código en producción.

Asimismo, se buscarán realizar tests unitarios de los distintos microservicios con el objetivo de asegurar que, en el largo plazo, no se rompan cosas que previamente funcionaban. En este punto, se implementará Continuous Integration (CI) así como herramientas de análisis estático de código para minimizar el error humano, garantizando así una altísima calidad de código.

5.3.5. Interoperabilidad

El servicio de webhooks permitirá integraciones con múltiples herramientas, las cuales son diversas, por lo que se implementó un patrón adapter en dicho servicio. Esto reduce la complejidad de agregar un nuevo webhook aceptado de un servicio externo, por lo que se puede aumentar la cantidad de servicios soportados inicialmente y luego expandir en base a la demanda.

5.3.6. Usabilidad

Al tratarse de una plataforma especializada en desarrollo y trabajo en equipo, resulta esencial mantener buenas prácticas de usabilidad. Esto facilitará que los usuarios de la plataforma sean más ágiles al analizar cambios en los repositorios, obtener información sobre los problemas, realizar seguimiento del tiempo, entre otras funciones. Al utilizar la librería de Mantine, se podrá desarrollar estos distintos flujos del usuario de una forma más rápida y sencilla. Además, al seguir un modelo de negocios tipo SaaS, la plataforma atraerá usuarios con diversos orígenes y necesidades, por lo que resulta crucial adaptarse a estas variadas necesidades y Mantine ofrece una amplia gama de componentes configurables.

Adicionalmente, la CDN utilizada para la webapp reduce los tiempos de carga, mejorando la experiencia de usuario.

6. Puntos críticos

6.1. Acceso fácil, seguro y transparente de los repositorios remotos

Se considera que el acceso fácil y seguro de los repositorios remotos es uno de los principales puntos críticos de la aplicación. Se resuelve este punto crítico mediante el uso de varios sistemas y tecnologías (explicados en detalle anteriormente en el informe):

- El uso de SSH para el acceso de los usuarios (el cual es un estándar aceptado por los principales VCS para la sincronización y manejo de servidores de repositorios), y HTTPS, junto a una API token para el acceso de los servicios externos.
- El uso de una autorización para poder acceder a los repositorios, otorgada por los dueños/administradores de los repositorios.
- El uso de un proxy que permite redireccionar a los comandos de forma transparente a los servidores de los repositorios, que no requiere de programas adicionales para usar.
- El uso de microservicios, junto a un sistema de archivos distribuidos para los servidores de repositorios, lo cual permite mantener la consistencia y disponibilidad del sistema.

6.2. Integración simple con sistemas externos

Otro pilar de la aplicación consiste en permitir interactuar con sistemas externos, y por lo tanto, poder proveer aún más valor al cliente. Esto se ataca con 2 sistemas centrales:

- La creación de una API, junto a un sistema de validación de identidad usando API tokens, que puede ser consumida por sistemas externos.
- La provisión de un sistema de Webhooks, que ante un evento disparador configurado por el usuario, permite enviar un payload a una URL específica.

Adicionalmente, para mejorar la interoperabilidad con sistemas externos, y por ende facilitar el agregado a futuro de nuevos sistemas externos permitidos, se implementa un patrón adapter que permitirá adecuar el payload para los distintos webhooks de los servicios externos.

6.3. Disponibilidad de servicios

Es de vital importancia que los servicios provistos estén disponibles ya que compañías e individuos dependen de ellos para trabajar. Para mantener la disponibilidad del sistema se utilizaron las siguientes tecnologías y arquitecturas:

- El uso de microservicios administrados por Kubernetes, junto a un load balancer para distribuir el tráfico.
- El *hosteo* de dichos microservicios sobre una plataforma IaaS como es AWS EC2, que tiene un SLA que dicta un 99.99% de disponibilidad.

- La replicación de todas las base de datos, ya sean PostgreSQL o Ceph, y del sistema de archivos distribuido, CephFS.
- El *hosteo* de la SPA en un CDN, el cual garantiza una alta disponibilidad.

6.4. Uso de herramientas complementarias a los repositorios

Los servicios complementarios, como el de issues y time tracking, las wikis y los releases pueden mejorar las experiencia de desarrollo, unificando mucha de la funcionalidad necesaria para llevar a cabo un proyecto en una única herramienta. Todas las herramientas se pueden acceder mediante una única aplicación Web, lo cual ayuda a la usabilidad y simplicidad del sistema. En esta, no solo se podrá acceder a los servicios complementarios de un único lugar, sino que también manejar las autorizaciones de los usuarios y servicios externos, la edición de los metadatos del repositorio y también la búsqueda de repositorios por su nombre.

Para proveer estos servicios, se generaron microservicios para cada uno de ellos, las cuales podrán ser escaladas conforme se requiera.

7. Escenarios

7.1. Disponibilidad

- Escenario: Se cae la base de datos de *issues*/wiki/repositorios/usuarios/facturación (PostgreSQL).
  Solución: No hay problema porque se realiza replicación master-slave, por lo que no se pierde el acceso a los datos.

- Escenario: Se cae alguno de los nodos de CephFS.
  Solución: CephFS trabaja en cluster y con replicación, por lo que la caída de un nodo no significa ni la interrupción del servicio del sistema, ni la pérdida de datos.

- Escenario: Se cae algún microservicio.
  Solución: Los microservicios son stateless, por lo que la caída de alguno de ellos solo provocará que se levante otro que tome su trabajo.

- Escenario: Se cae la base de datos de Releases (Ceph).
  Solución: Ceph trabaja en cluster y con replicación para asegurar alta disponibilidad, por lo que la caída de un nodo no significa ni la interrupción del servicio del sistema.

7.2. Seguridad

- Escenario: A través de ingeniería social se consigue la contraseña de un usuario.
  Solución: Se provee 2FA evitando la suplantación de identidad.

- Escenario: Se intenta hacer un ataque “Rainbow Table” para robar contraseñas a los usuarios.
  Solución: Se cuenta con una encriptación con Salts, por lo que para efectivamente robar la contraseña se debe conocer la Salt, que es un número generado de manera aleatoria.

- Escenario: Se compromete la seguridad de la base de datos de usuarios y se intenta robar los datos de los usuarios.
  Solución: La base de datos de usuarios se encuentra encriptada.

- Escenario: Se intenta hacer un ataque MITM cuando un usuario realiza algún *push.*
  Solución: Este tipo de acciones se realizan por SSH, por lo que no es posible realizar dicho ataque de manera exitosa.

- Escenario: Se compromete la seguridad del *file system* donde están los repositorios (CephFS).
  Solución: Los repositorios se encuentran en CephFS y se encuentran encriptados.

- Escenario: Se realiza un ataque MITM para robar datos de facturación.
  Solución: Se realiza “Certificate Pinning” por lo que no es posible realizar el ataque de manera exitosa.

7.3. Escalabilidad

- Escenario: El sistema supera las expectativas y recibe más usuarios de los esperados.
  Solución: Se pueden escalar todos los servicios horizontalmente para soportar la demanda inesperada, mediante el uso de un orquestador como es Kubernetes y load balancer.

- Escenario: La webapp recibe un pico de tráfico.
  Solución: Al estar hosteado en un CDN que permite escalar los servicios que provee la Web app, se puede abarcar ese pico de tráfico.

7.4. Mantenibilidad

- Escenario: Se quiere agregar un nuevo VCS.
  Solución: Se agrega un nuevo módulo al servicio de VCS con las directivas de este nuevo sistema de control de versiones, sin necesidad de generar un nuevo microservicio y sin alterar el resto del sistema.

- Escenario: Se suma un nuevo programador al equipo y debe agregar funcionalidad a alguno de los microservicios.
  Solución: Los microservicios están escritos en Java y siguen buenas prácticas de código. Además la funcionalidad de cada microservicio está aislada del resto.

7.5. Interoperabilidad

- Escenario: Se decide soportar una nueva herramienta para webhooks.
  Solución: Se utiliza el patrón adapter por lo que no supone un problema, al poder acomodarse el payload para la herramienta.

7.6. Usabilidad

- Escenario: Un desarrollador quiere agregar *issues* a un repositorio *open source* al que está contribuyendo.
  Solución: La interfaz es intuitiva por lo que encontrar la funcionalidad es fácil.

- Escenario: Un desarrollador con poca experiencia quiere crear su primer repositorio.
  Solución: La interfaz provee buena experiencia de usuario por lo que logra hacerlo.

8. Riesgos y no riesgos

8.1. Riesgos

- Se cae el servicio de pagos de Stripe, por lo que no se podrán aceptar pagos.
- Se cae un servicio externo, para el cual se había configurado un webhook y al ocurrir el evento disparador, no se ejecutara la acción requerida en el servicio externo.
- El CDN en el cual se hostea la SPA se cae o se degrada la performance, lo cual no permitiría acceder a la webapp.
- Se cae el load balancer o la API Gateway al ser un *single point of failure*. En este caso, no se podría acceder a ningún servicio.

8.2. No riesgos

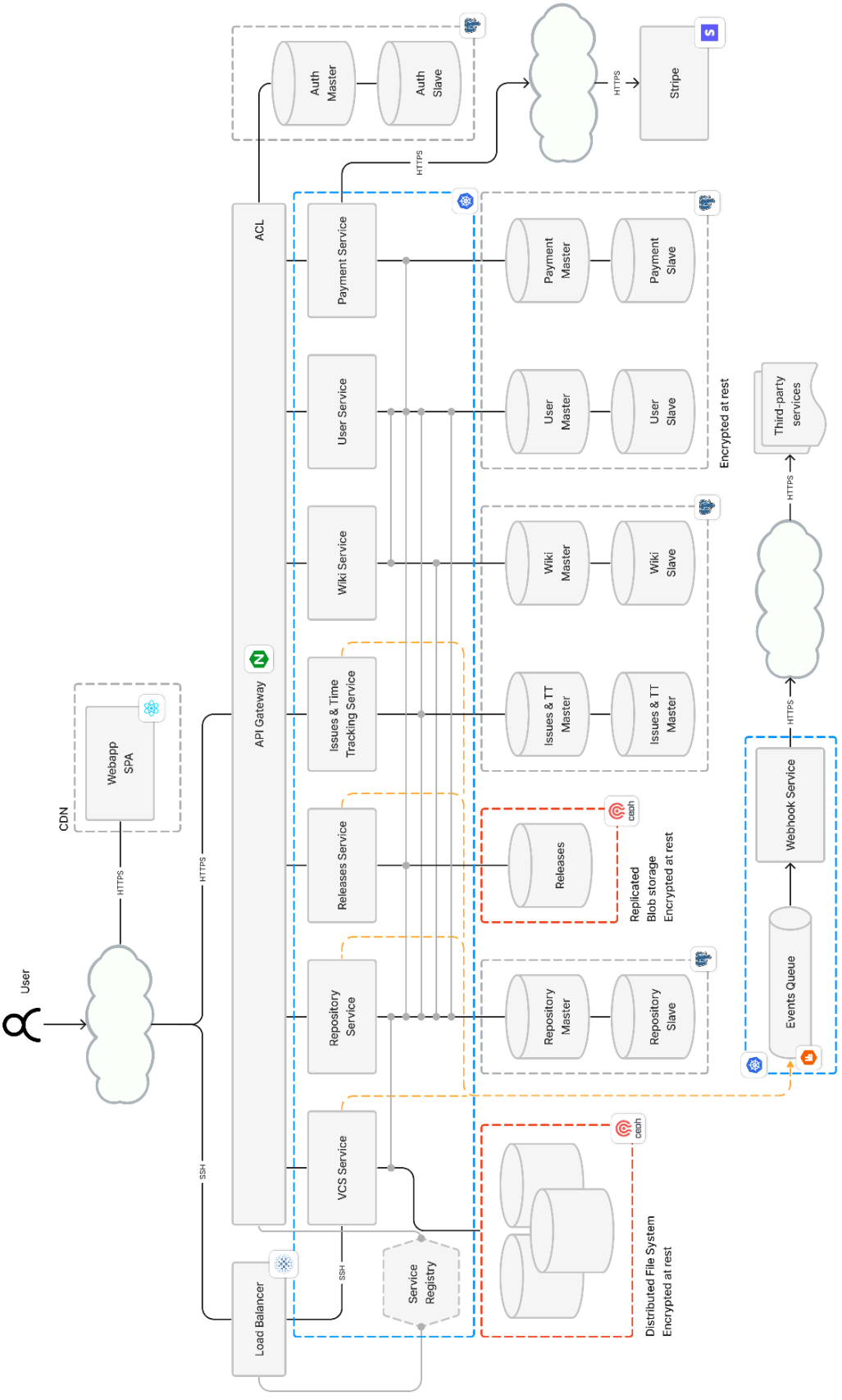
- El robo de credenciales no es un riesgo pues utilizamos 2FA, el cual agrega un paso adicional que se debe hacer al iniciar la sesión.
- Accesos maliciosos al *file system* de repositorios no son un riesgo pues se usa un esquema de *encryption-at-rest*.
- Accesos maliciosos a la base de datos de usuarios no son un riesgo pues también se usa un esquema de *encryption-at-rest*.
- Ataques MITM no son posibles cuando se transportan datos de los repositorios pues está información va por SSH, el cual encripta los datos.
- Ataques MITM no son posibles para el robo de datos de facturación.
- Ataques “Rainbow Table” para robar contraseñas de usuarios, dado que se utiliza un algoritmo de encriptación junto con Salts.
- Agregar un nuevo VCS al no tenerse que deployar un nuevo microservicio y solo tener que agregar el servidor al servicio de VCS.
- Agregar un nuevo servicio que provee webhooks, al usar un patrón adapter para los payloads de los webhooks.
- La caída de un microservicio, al tener múltiples instancias que son orquestadas por Kubernetes.
- Ante un aumento importante en la cantidad de pedidos, el uso de microservicios permite escalar fácilmente para aceptar el nuevo aumento.

9. Trade-offs

- Escalabilidad vs. Mantenibilidad: Los microservicios hacen que el mantenimiento se vuelva una tarea compleja, pero nos provee una altísima escalabilidad. Se eligió priorizar la escalabilidad, y mitigar la pérdida de mantenibilidad siguiendo buenas prácticas de desarrollo.
- Seguridad vs. Mantenibilidad: El uso de “Certificate Pinning” requiere que se modifique el certificado aceptado por la Web app cada vez que se cambia el certificado utilizado.
- Seguridad vs. Usabilidad: La introducción de 2FA para los usuarios ayuda enormemente a la seguridad de las cuentas, pero va en detrimento a la experiencia que tiene el usuario al interactuar con el sistema.
- Escalabilidad vs. Costos: La arquitectura de microservicios asociada al atributo de escalabilidad implica un gran costo monetario asociado a mantener los servidores y sus contenedores, pero es la manera más confiable de poder asegurar dicho atributo. Consecuentemente, se eligió priorizar la escalabilidad.
- Escalabilidad vs. TTM: El desarrollo de microservicios tiene un gran costo temporal inicial, al tener que definir la manera en que trabajan los contenedores y configurarlos. Esta configuración inicial retrasa enormemente la salida al mercado del proyecto.
- Disponibilidad vs. Costos: Con el objetivo de asegurar una altísima disponibilidad, se debieron duplicar la gran mayoría de bases de datos aumentando así los costos del proyecto. También el uso de load balancers y microservicios aumenta los costos operacionales.

10. Anexo

A. Arquitectura (formato horizontal)



11. Referencias

[1] “HgGit,” Mercurial, https://wiki.mercurial-scm.org/HgGit.
[2] “Gitaly and Gitaly Cluster,” GitLab, https://docs.gitlab.com/ee/administration/gitaly/.
[3] “The ultimate solution for SVN to Git Migration,” SVN to Git Migration - TMate SubGit, https://subgit.com/.
[4] “A high-performance, cloud-native, Distributed File System,” JuiceFS, https://juicefs.com/en/.
[5] J. Team, “JuiceFS v1.0 officially released,” JuiceFS, https://juicefs.com/en/blog/release-notes/juicefs-release-v1.
[6] “Discover,” Ceph, https://ceph.io/en/discover/.
[7] “Ceph releases (index),” Ceph Releases (index) - Ceph Documentation, https://docs.ceph.com/en/latest/releases/.
[8] HarshSingh, “D of Kafka,” Medium, https://medium.com/@harsh.b26/d-of-kafka-fcb459a50ee5.
[9] “Apache kafka,” Apache Kafka, https://kafka.apache.org/.
[10] “Quorum queues,” RabbitMQ, https://www.rabbitmq.com/#community.
[11] E. B. Abid, “Rabbitmq vs kafka - message brokers (pros and cons),” Cloud Infrastructure Services, https://cloudinfrastructureservices.co.uk/rabbitmq-vs-kafka-message-brokers/.
[12] A. Zanini, “Blob data type: Everything you can do with it,” DbVisualizer, https://www.dbvis.com/thetable/blob-data-type-everything-you-can-do-with-it/.
[13] Inc. MinIO, “High performance, kubernetes native object storage,” MinIO, https://min.io/.
[14] “Ceph vs Minio: What are the differences?,” StackShare, https://stackshare.io/stackups/ceph-vs-minio.
[15] O. Weis, “Pros and cons of Amazon S3 2023,” CloudMounter, https://cloudmounter.net/pros-and-cons-of-amazon-s3/.
[16] “Gitea official website,” Gitea Official Website, https://about.gitea.com/.
[17] “Open source cloud computing platform software,” OpenStack, https://www.openstack.org/software/.
[18] “Additional Ceph considerations,” Mirantis Documentation: Additional Ceph considerations, https://docs.mirantis.com/mcp/q4-18/mcp-ref-arch/openstack-environment-plan/storage-plan/ceph-plan/ceph-cluster-considerations.html.
[19] HGKeeper, https://keep.imfreedom.org/grim/hgkeeper/.
[20] “svnserve, a Custom Server,” Svnserve, a custom server, https://svnbook.red-bean.com/en/1.7/svn.serverconfig.svnserve.html.
[21] “Perforce helix core,” Perforce, https://www.perforce.com/products/helix-core.
[22] “Time Tracking,” GitLab, https://docs.gitlab.com/ee/user/project/time_tracking.html.
[23] “Rainbow table attack: Beyond identity,” What is a Rainbow Table Attack? | Beyond Identity, https://www.beyondidentity.com/glossary/rainbow-table-attack.
[24] Jacco et al., “Password hashing: Add salt + pepper or is salt enough?,” Information Security Stack Exchange,

https://security.stackexchange.com/questions/3272/password-hashing-add-salt-pepper-or-is-salt-enough.
[25] 2023 3:53 pm UTC Kevin Purdy - Jun 30, “Red Hat’s new source code policy and the intense pushback, explained,” Ars Technica, https://arstechnica.com/information-technology/2023/06/red-hats-new-source-code-policy-and-the-intense-pushback-explained/.
[26] “Clients libraries and developer Tools,” RabbitMQ, https://www.rabbitmq.com/devtools.html.
[27] “List of drivers,” List of drivers - PostgreSQL wiki, https://wiki.postgresql.org/wiki/List_of_drivers.
[28] “API documentation,” API Documentation - Ceph Documentation, https://docs.ceph.com/en/latest/api/.
[29] S. Dangi, “EC2 vs google compute engine: Comparing the big players in iaas,” Cloud Academy, https://cloudacademy.com/blog/ec2-vs-google-compute-engine/.
[30] “Amazon EC2 vs Google Compute engine: What are the differences?,” StackShare, https://stackshare.io/stackups/amazon-ec2-vs-google-compute-engine.
[31] “Google Compute engine vs Amazon’s AWS EC2: Upguard,” RSS, https://www.upguard.com/blog/google-compute-engine-vs-amazons-aws-ec2.
[31] Nginx, https://nginx.org/en/.
[32] techopedia, What is Amazon Simple Storage Service (Amazon S3), https://www.techopedia.com/definition/13565/amazon-simple-storage-service-amazon-s3.
[33] T. Preston-Werner, “How we made github fast,” The GitHub Blog, https://github.blog/2009-10-20-how-we-made-github-fast/.
[34] M. Mayen, “Route SSH connections with HAProxy (in-depth configuration),” HAProxy Technologies, https://www.haproxy.com/blog/route-ssh-connections-with-haproxy.
[35] “Financial infrastructure for the internet,” Stripe, https://stripe.com/.
[36] *How certificate pinning helps thwart mobile MITM attacks* (2023) *Mobile App Protection*. Available at: https://approov.io/blog/how-certificate-pinning-helps-thwart-mobile-mitm-attacks.
[37] “Business Service Level Agreement,” Cloudflare, https://www.cloudflare.com/business-sla/
[38] “Zod” https://zod.dev/
[39] “Mantine” https://mantine.dev/