UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA

Plataforma de microservicios para minería de procesos de negocio

Informe de Proyecto de Grado presentado por

Damián Piccini, Gonzalo Núñez, Nicolás Luraschi

en cumplimiento parcial de los requerimientos para la graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Supervisor

Andrea Delgado

Montevideo, 8 de diciembre de 2024

Resumen

Un aspecto crucial para la mejora continua de los procesos de negocio en las organizaciones es la capacidad de obtener medidas precisas de la operativa diaria. Tener un modelo del funcionamiento de la organización permite a los expertos en el negocio examinar las definiciones actuales y descubrir oportunidades de mejora estratégica y operativa.

La minería de procesos se ha destacado como un enfoque efectivo para obtener estos modelos de la operativa del negocio utilizando los registros de eventos de la organización. Este método, además de permitir descubrir los modelos de proceso, proporciona también la capacidad de comparar la ejecución real con los modelos existentes y realizar análisis detallados para optimizar la eficiencia operativa.

A pesar de la disponibilidad de diversas herramientas y librerías que ofrecen implementaciones de técnicas y algoritmos de minería de procesos, su selección y utilización pueden representar un desafío. La integración de estas implementaciones en contextos empresariales específicos a menudo plantea dificultades, lo que destaca la necesidad de una solución que simplifique y promueva su adopción por parte de las organizaciones.

En este contexto, este proyecto tiene como objetivo central la definición e implementación de una plataforma de microservicios dedicada a proporcionar acceso a las diversas implementaciones disponibles en el ámbito de minería de procesos. Esta iniciativa tiene la intención de facilitar la integración y uso de estas técnicas y librerías, abriendo las puertas a participantes diversos y reduciendo la complejidad asociada con la implementación y aprovechamiento de estas herramientas.

Además de abordar la complejidad de la integración, la plataforma también se esfuerza por garantizar la calidad de servicio, incluyendo aspectos críticos como seguridad y privacidad de los datos.

Palabras clave: Procesos de Negocio, Minería de Procesos, Microservicios, Plataformas de Microservicios

Índice general

1. Introducción **1**
- 1.1. Motivación . . . 1
- 1.2. Objetivos . . . 2
- 1.3. Resultados esperados . . . 2
- 1.4. Contexto de trabajo . . . 3
- 1.5. Estructura del documento . . . 3

2. Marco Conceptual **5**
- 2.1. Procesos de negocio . . . 5
  - 2.1.1. Gestión de procesos de negocio . . . 6
- 2.2. Minería de procesos . . . 8
  - 2.2.1. Log de Eventos . . . 8
  - 2.2.2. Tipos de minería de procesos . . . 10
  - 2.2.3. Herramientas . . . 11
  - 2.2.4. Algoritmos . . . 12
- 2.3. Arquitectura de Microservicios . . . 13
  - 2.3.1. Patrones de diseño . . . 13
  - 2.3.2. Beneficios y desafíos de la arquitectura de Microservicios . 15

3. Problema planteado **17**
- 3.1. Requerimientos del sistema . . . 17
  - 3.1.1. Requerimientos funcionales . . . 18
  - 3.1.2. Requerimientos no funcionales . . . 20

4. Diseño de la solución **23**
- 4.1. Arquitectura general de la solución . . . 23
  - 4.1.1. Componentes del sistema . . . 23
  - 4.1.2. Arquitectura de microservicios . . . 25
  - 4.1.3. Diagrama de Entidad-Relación de Base de Datos (ERD) . 26
  - 4.1.4. Interacción entre componentes . . . 28
- 4.2. Patrones de diseño . . . 29
  - 4.2.1. Patrones de diseño generales . . . 29
  - 4.2.2. Patrones de diseño de microservicios . . . 34
- 4.3. Metodología de diseño . . . 36

4.4. Relación con los Requerimientos del Sistema 36
4.4.1. Requerimientos Funcionales 36
4.4.2. Requerimientos No Funcionales 40

5. Desarrollo del prototipo **43**
5.1. Tecnologías utilizadas 43
5.1.1. Interfaz gráfica 43
5.1.2. Componentes de gestión 44
5.1.3. Componentes de minería de procesos 45
5.2. Patrones de diseño 45
5.2.1. Implementación del patrón Strategy 46
5.2.2. Implementación del patrón Repository 47
5.2.3. Implementación del patrón Inyección de Dependencias . . 48
5.2.4. Implementación del patrón Command 51
5.2.5. Implementación del patrón Decorator 51
5.2.6. Implementación del patrón Invocación de Procedimientos Remotos 56
5.2.7. Implementación del patrón Chasis 57
5.2.8. Implementación del patrón Configuración Externalizada . 57
5.3. Despliegue en producción 60
5.4. Extensibilidad 62
5.4.1. Cómo exponer un nuevo algoritmo en el sistema 64

6. Aplicación en un caso de estudio **67**
6.1. Descripción del sistema 67
6.2. Descripción del log de eventos 68
6.2.1. Estructura del log de eventos 69
6.2.2. Flujo del proceso 69
6.3. Carga de archivos de eventos 70
6.4. Aplicación de algoritmos de Descubrimiento 70
6.4.1. Descubrimiento con librerías Java 71
6.4.2. Descubrimiento con librerías Python 71
6.5. Aplicación de algoritmos de Verificación de Conformidad 73
6.6. Aplicación de algoritmos de Extensión de Modelos 77
6.7. Discusión y evaluación final 79

7. Conclusiones y Trabajo Futuro **81**
7.1. Conclusiones 81
7.2. Desafíos encontrados 82
7.3. Trabajo futuro 83

Referencias **87**

A. Documentación del Repositorio (README) **91**

Capítulo 1

Introducción

En este capítulo se presenta la motivación para la realización de este proyecto además de mencionarse los objetivos principales que se definieron para el mismo. Por último se da una breve descripción sobre la estructura de este documento y el contenido de cada capítulo.

1.1. Motivación

En la dinámica empresarial actual, la mejora continua de los procesos de negocio se ha convertido en un imperativo para las organizaciones que buscan optimizar su eficiencia operativa y mantenerse competitivas en un entorno en constante evolución. La capacidad de obtener medidas precisas de la operativa diaria se presenta como un factor clave para este proceso de mejora, permitiendo a los expertos en el negocio analizar las definiciones actuales y descubrir oportunidades estratégicas y operativas.

La minería de procesos ha surgido como una herramienta efectiva para analizar datos de ejecución de procesos, proporcionando una visión profunda y detallada. Sin embargo, la integración de estas técnicas en el tejido empresarial a menudo plantea desafíos significativos, desde la selección de herramientas hasta su implementación en contextos específicos.

Además, la existencia de una amplia variedad de algoritmos y técnicas de minería de procesos implementados en distintos lenguajes de programación y en diversos formatos —como herramientas independientes (*standalone*), aplicaciones web y librerías— añade una capa adicional de complejidad en la adopción de estas tecnologías. A menudo, estas opciones requieren instalaciones específicas, permisos especiales y la intervención de distintos usuarios, lo cual puede fragmentar las funcionalidades entre diferentes herramientas y dificultar su adopción por parte de las organizaciones.

Por otro lado, el ecosistema de herramientas de minería de procesos incluye tanto soluciones de código abierto como software propietario, cada uno con sus propias características, ventajas y limitaciones. Las herramientas de códi-

go abierto pueden ofrecer flexibilidad y personalización, pero pueden requerir conocimientos técnicos avanzados y un esfuerzo adicional en su configuración e integración. Las herramientas propietarias, aunque a menudo presentan interfaces más amigables y soporte técnico, pueden implicar altos costos de licencia y depender de los ciclos de actualización del proveedor.

1.2. Objetivos

Los objetivos de este proyecto son los siguientes:

1. Analizar conceptos y arquitecturas de microservicios, *BPM* (*Business Process Management*), minería de procesos y datos, así como escenarios tecnológicos que permitan proveer implementaciones existentes de técnicas y algoritmos de minería de procesos y datos como microservicios para proveer soporte al análisis integrado de procesos, así como servicios asociados a la carga y evaluación de calidad de datos.

2. Seleccionar herramientas y definir una plataforma de microservicios que exponga microservicios de minería de procesos y datos considerando elementos clave de *Quality of service (QoS)*, y carga y evaluación de calidad de datos.

3. Implementar un prototipo de la plataforma de microservicios de *backend* así como posible *frontend* de integración y uso de los servicios provistos (ej. incluir solo una selección de algoritmos de minería de procesos y datos, carga y evaluación de calidad).

4. Aplicar la propuesta en un caso de estudio con base en procesos y datos reales

1.3. Resultados esperados

Los resultados esperados por este proyecto son:

1. Informe de estado del arte en microservicios y arquitecturas, *BPM*, minería de procesos y datos, herramientas y librerías existentes de minería de procesos y datos, *QoS* para servicios.

2. Informe de análisis de escenarios tecnológicos para la plataforma de microservicios, selección de herramientas y definición de la plataforma incluyendo *QoS* para servicios.

3. Informe del prototipo de la plataforma de microservicios de *backend* así como posible *frontend* de integración y uso de los servicios provistos.

4. Informe del caso de estudio de aplicación de la propuesta con el prototipo desarrollado.

5. Informe del proyecto de tesis durante todo el desarrollo por capítulos, final, correcciones y defensa.

1.4. Contexto de trabajo

Este proyecto de grado se enmarca en el proyecto de investigación financiado por el Fondo María Viñas (FMV) de ANII 2021 "Minería de procesos y datos para la mejora de procesos colaborativos aplicada a e-Government" del grupo COAL del INCO.

1.5. Estructura del documento

Este informe está compuesto por un total de siete capítulos, cada uno enfocado en un aspecto clave del proyecto. En el Capítulo 2 se abordan los fundamentos teóricos, explorando conceptos esenciales sobre minería de procesos y microservicios, además de ofrecer un panorama de las herramientas existentes en el estado del arte.

El Capítulo 3 está dedicado a la definición del problema central del proyecto, detallando los requisitos funcionales y no funcionales necesarios para su resolución.

En el Capítulo 4 se presenta el diseño conceptual de la solución propuesta, incluyendo la metodología empleada, la arquitectura general del sistema y los patrones de diseño implementados.

El Capítulo 5 se enfoca en las decisiones técnicas tomadas durante la implementación de la plataforma de microservicios, proporcionando detalles sobre las tecnologías utilizadas, el proceso de despliegue en producción y una guía para escalar el prototipo desarrollado.

En el Capítulo 6 se describe la aplicación práctica del sistema en un caso de estudio real, detallando paso a paso las funcionalidades ofrecidas por la plataforma y los resultados obtenidos.

Finalmente, el Capítulo 7 incluye las conclusiones del proyecto, discutiendo los principales desafíos enfrentados, las lecciones aprendidas y las oportunidades de mejora para futuras iteraciones.

Capítulo 2

Marco Conceptual

En este capítulo se presentan los principales conceptos para comprender el marco teórico del proyecto. Se introducen conceptos básicos sobre gestión de procesos de negocio, minería de procesos y microservicios.

2.1. Procesos de negocio

Los procesos de negocio son fundamentales para el funcionamiento y éxito de cualquier organización. Se pueden definir como conjuntos estructurados de actividades diseñadas para producir un resultado específico para un cliente o mercado particular. Un proceso de negocio es una secuencia de tareas que se encuentran interrelacionadas sistemáticamente, culminando en la entrega de un servicio o producto a los clientes (Weske, 2019). En la figura 2.1 se presenta un ejemplo de un proceso de negocio de un vendedor especificado con el estándar *BPMN 2.0* (Object Management Group, 2011).

Estos procesos se componen de varios elementos claves: entradas (recursos necesarios para realizar el proceso), transformaciones (las actividades que convierten las entradas en salidas) y salidas (los servicios o productos finales entregados a los clientes).

Los procesos de negocio constituyen la columna vertebral de la organización ya que son los que definen el funcionamiento de los distintos actores que la componen. Alinear los procesos de negocio con los objetivos estratégicos de la empresa es crucial para el éxito organizacional. Esta alineación asegura que cada proceso contribuya de manera efectiva a los objetivos a largo plazo de la organización, facilitando una mejor toma de decisiones y optimización de recursos (Weske, 2019).

A medida que una organización se vuelve mas compleja, también se vuelve mas difícil hacer un correcto seguimiento de las actividades que se llevan a cabo en el día a día y en como interactúan entre si. No tener identificado correctamente cuales son las características de los procesos utilizados puede incidir directamente en el éxito o fracaso que se tenga para alcanzar los objetivos

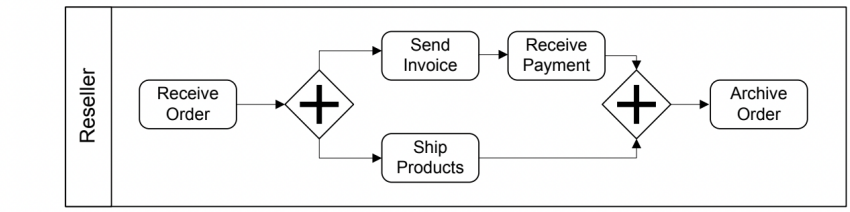


Figura 2.1: Modelo de un proceso de negocio extraído de (Weske, 2019)

previamente definidos por la organización (Weske, 2019).

Es por esto que la comprensión y optimización de los procesos de negocio se eligen como pilares fundamentales para la mejora continua y la competitividad organizacional. El estudio y análisis de los procesos de negocio no solo involucra la descripción de las actividades realizadas, sino también la comprensión de las interrelaciones entre ellas, la identificación de cuellos de botella, la detección de posibles mejoras y la adaptación a cambios en el entorno empresarial.

2.1.1. Gestión de procesos de negocio

La gestión de procesos de negocio (*BPM*, por sus siglas en inglés) es una disciplina integral que combina elementos de gestión, estrategia y tecnología de la información con el objetivo de mejorar el rendimiento de los procesos empresariales. El objetivo fundamental de *BPM* es mejorar la eficiencia y la eficacia de los procesos de negocio de una organización a través de la observación, el control y la innovación continua (Weske, 2019).

BPM no es únicamente una serie de herramientas o técnicas, sino un enfoque sistemático para hacer que los flujos de trabajo de una organización sean más efectivos, más eficientes y más capaces de adaptarse a los entornos cambiantes. Al adoptar un enfoque de este estilo y orientado a los procesos, las organizaciones pueden obtener una visión holística de sus operaciones, identificar áreas de mejora y tomar decisiones más informadas y estratégicas.

La gestión de procesos de negocio se compone de varios elementos fundamentales (Weske, 2019):

- Modelado de Procesos: Consiste en la representación gráfica de los procesos de una organización, facilitando su comprensión y análisis. Las notaciones como *BPMN (Business Process Model and Notation)* son comúnmente utilizadas para este propósito.

- Automatización de Procesos: Utiliza la tecnología para ejecutar procesos recurrentes sin intervención humana, lo cual reduce el margen de error y aumenta la eficiencia.

- Monitoreo de Procesos: Se refiere a la supervisión de los procesos para asegurar su correcto funcionamiento. Incluye el rastreo del rendimiento y

la identificación de cuellos de botella.

- Optimización de Procesos: Una vez que los procesos están en marcha y se monitorean, se analizan para identificar áreas de mejora. Esto puede involucrar el re diseño de algunos aspectos del proceso o su mejora continua.

El modelado de procesos es un aspecto clave de la gestión de procesos de negocio. Esta representación, a menudo en forma de diagramas de flujo o modelos *BPMN (Business Process Model and Notation)*, proporciona una base sólida para la comprensión y comunicación de los procesos dentro de la organización. Al capturar la lógica y la secuencia de las actividades, el modelado de procesos facilita la identificación de ineficiencias, la estandarización de las mejores prácticas y la alineación de los objetivos empresariales.

El ciclo de vida de *BPM* incluye varias fases: diseño, configuración, ejecución, monitoreo y optimización (Weske, 2019). En la fase de diseño, se identifican y definen los procesos que son cruciales para el negocio. Durante la configuración, estos procesos se implementan a través de cambios organizativos o desarrollos de software. La ejecución de los procesos se realiza en un entorno operativo real, mientras que el monitoreo se asegura de que los procesos estén funcionando dentro de los parámetros deseados y, finalmente, la optimización se enfoca en mejorar los procesos basándose en los datos recogidos durante el monitoreo.

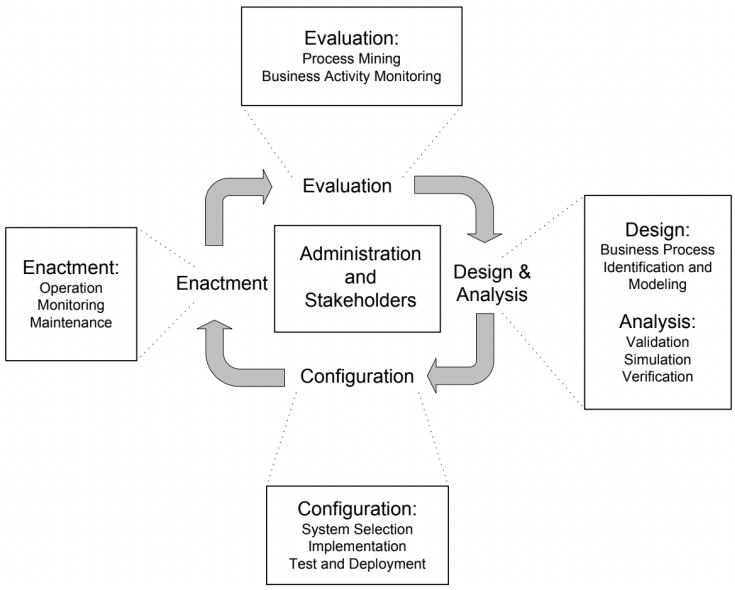


Figura 2.2: Ciclo de vida de un proceso de negocio extraído de (Weske, 2019)

Una gestión eficaz de procesos de negocio no solo se traduce en mejoras operacionales y de eficiencia, sino que también fortalece la innovación y la capacidad de una organización para ofrecer valor continuo a sus clientes. Al alinear

los procesos de negocio con las necesidades y expectativas del cliente, las empresas pueden mantenerse competitivas en un mercado cada vez más dinámico y exigente (Weske, 2019).

2.2. Minería de procesos

La minería de procesos es una técnica de análisis de procesos que se basa en la extracción de información de los registros de eventos para descubrir, monitorear y mejorar procesos reales ejecutados dentro de una organización. La minería de procesos ofrece un puente crucial entre el modelado de procesos de negocio y las tecnologías de datos. Esta técnica permite a las organizaciones analizar efectivamente cómo se están llevando a cabo sus procesos en la práctica, no solo cómo se supone que deben ejecutarse según los modelos predeterminados (W. van der Aalst, 2016). La integración de la minería de procesos en el marco de la gestión de procesos de negocio permite a las organizaciones obtener una comprensión más profunda y completa de sus operaciones, impulsando la mejora continua y la excelencia operativa.

Como explica (W. van der Aalst, 2016), la minería de procesos se sitúa en la intersección entre la ciencia de datos y la ciencia de procesos, ya que integra técnicas de ambos campos para analizar y mejorar los flujos de trabajo en las organizaciones. Su objetivo principal es extraer conocimiento útil a partir de los datos generados por la ejecución de procesos de negocio. Esta disciplina se ha convertido en una herramienta crucial para las organizaciones que buscan comprender, analizar y mejorar sus operaciones comerciales, ya que ofrece una visión detallada y objetiva de cómo se llevan a cabo los procesos en la práctica, en contraposición a cómo se supone que deberían ser ejecutados según los modelos previamente establecidos.

La minería de procesos se establece como una disciplina indispensable para cualquier organización que busque no solo mantener sino optimizar sus operaciones en un entorno empresarial que cambia rápidamente. La capacidad de entender y mejorar los procesos empresariales basados en evidencias concretas permite a las empresas mantener una ventaja competitiva y responder de manera efectiva a las exigencias del mercado y los desafíos internos (W. van der Aalst, 2016).

2.2.1. Log de Eventos

La minería de procesos sería imposible sin los *logs* de eventos, estos son registros detallados de las actividades realizadas dentro de una organización. Estos registros capturan eventos relevantes, como la ejecución de actividades, las transiciones de estados, las decisiones tomadas y otros eventos relacionados con la interacción entre los actores y los sistemas de información (W. van der Aalst, 2016).

La estructura de un *log* de eventos sigue un metamodelo específico, lo que significa que incluye elementos y relaciones definidas de manera consistente.

Típicamente, estos *logs* contienen información esencial como la marca de tiempo de cada evento, la identificación única del caso o instancia del proceso, la actividad realizada, el actor involucrado y cualquier otro dato relevante para el análisis posterior. Esta estructura estándar permite que los registros, ya sean generados automáticamente por sistemas de información o capturados manualmente, puedan ser utilizados de manera consistente en técnicas de minería de procesos.

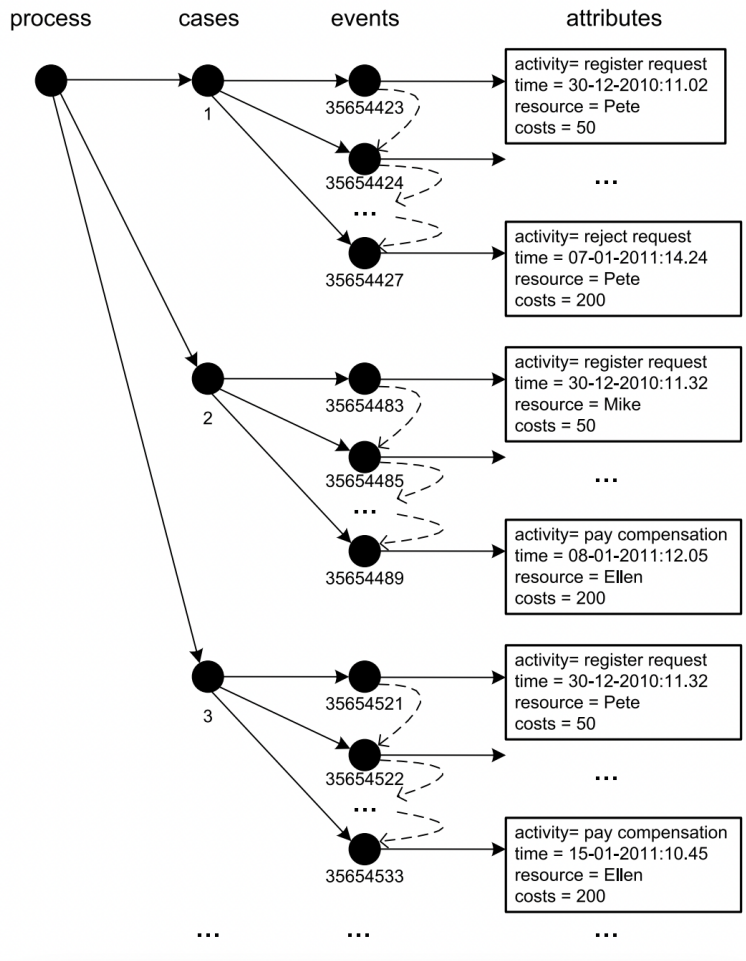


Figura 2.3: Estructura de un log de eventos extraído de (W. van der Aalst, 2016)

El análisis de *logs* de eventos en el contexto de la minería de procesos implica la extracción de conocimiento útil a partir de estos datos para comprender, modelar y mejorar los procesos empresariales. Las técnicas de análisis de *logs* de eventos pueden incluir desde la identificación de patrones y tendencias hasta

la detección de desviaciones y anomalías en la ejecución del proceso. Al analizar los datos de los *logs* de eventos, los investigadores y analistas pueden descubrir información clave sobre cómo se llevan a cabo los procesos en la práctica, identificar posibles problemas o áreas de mejora y tomar decisiones informadas para optimizar el rendimiento operativo (W. van der Aalst, 2016).

Además, los *logs* de eventos también son fundamentales para la construcción y evaluación de modelos de procesos en la minería de procesos. Al utilizar los datos de los *logs* de eventos como entrada, es posible descubrir automáticamente modelos de procesos que representen con precisión el comportamiento observado. Este proceso de descubrimiento de modelos permite a las organizaciones obtener una comprensión más profunda de sus procesos operativos y facilita la identificación de oportunidades de mejora y la implementación de cambios efectivos.

En resumen, los *logs* de eventos desempeñan un papel crucial en el contexto de la minería de procesos, proporcionando datos detallados y objetivos sobre la ejecución de procesos empresariales. Al analizar estos registros, las organizaciones pueden obtener información valiosa para comprender, modelar y mejorar sus operaciones, lo que conduce a una mayor eficiencia, calidad y competitividad en el mercado (W. van der Aalst, 2016).

2.2.2. Tipos de minería de procesos

La minería de procesos abarca varios enfoques y técnicas destinados a extraer conocimiento útil de los datos de los procesos empresariales. Estos enfoques pueden clasificarse en diferentes tipos según el objetivo y la naturaleza del análisis. A continuación, se presentan algunos de los tipos más comunes de minería de procesos. (W. van der Aalst, 2016)

Descubrimiento de Modelos (Discovery)

El descubrimiento se refiere a la creación de un modelo de proceso basado únicamente en los datos existentes en los registros (*logs*) de eventos, sin necesidad de un modelo a priori. El modelado se logra mediante técnicas de análisis de datos que permiten detectar las actividades realizadas, su secuencia temporal, las relaciones entre ellas y las decisiones tomadas durante la ejecución del proceso.

Conformance Checking

El *conformance checking*, o verificación de conformidad, implica comparar el modelo de proceso esperado con el proceso real que se ha registrado en los eventos. Este análisis ayuda a identificar desviaciones, incumplimientos y posibles áreas de mejora. El *conformance checking* facilita la detección temprana de problemas y la toma de decisiones informadas para su resolución.

Extensión de modelos (Enhancement)

La extensión de modelos utiliza los datos extraídos de los *logs* de eventos para optimizar los procesos existentes. Estos datos permiten analizar aspectos específicos del proceso, como los recursos humanos y tecnológicos utilizados, los tiempos de ejecución de cada actividad y la secuencia de tareas. Gracias a esta información, es posible ajustar o rediseñar procesos para mejorar la eficiencia, reducir el tiempo de ciclo, eliminar cuellos de botella que ralentizan el flujo de trabajo y minimizar redundancias en actividades que no agregan valor.

2.2.3. Herramientas

Tal como se mencionó anteriormente, la minería de procesos es una técnica fundamental para el análisis de procesos empresariales, permitiendo la extracción de conocimientos valiosos a partir de los registros de eventos. Para llevar a cabo esta tarea de manera efectiva, se utilizan diversas herramientas especializadas. En esta sección se presentan algunas de las herramientas más utilizadas en la minería de procesos, entre las cuales se destacan *ProM*, *Disco*, *Apromore*, *PM4PY* y *BupaR*. A continuación se describen sus características principales.

ProM

ProM es una plataforma de minería de procesos de código abierto ampliamente utilizada en la academia. Ofrece una amplia variedad de *plugins* que permiten realizar múltiples tipos de análisis de procesos, desde el descubrimiento hasta la conformidad y la mejora.(Aalst, van der y cols., 2009)

Disco

Disco, desarrollado por *Fluxicon*, es una herramienta comercial de minería de procesos conocida por su facilidad de uso y potentes capacidades de análisis. Está diseñada para ofrecer una experiencia de usuario amigable, permitiendo a los analistas obtener resultados rápidos y significativos. Entre sus principales características se destaca su interfaz de usuario intuitiva, el uso de un algoritmo de descubrimiento de procesos basado en el *Fuzzy Miner* —el cual permite una visualización simplificada y comprensible de los procesos complejos— y su soporte para grandes volúmenes de datos y análisis de rendimiento (Günther y Rozinat, 2012).

Apromore

Apromore es una herramienta de código abierto que proporciona una plataforma avanzada para la minería de procesos. Es utilizada tanto por investigadores como por profesionales para realizar análisis detallados de procesos empresariales. Es una plataforma extensible con capacidades avanzadas de análisis, y con soporte para descubrimiento de procesos, análisis de conformidad y variantes de procesos.(La Rosa y cols., 2011)

Celonis

Celonis es una herramienta comercial ampliamente utilizada en la academia y la industria. Al igual que *Apromore*, ofrece una plataforma web con capacidades de minería de procesos que incluyen descubrimiento, análisis de conformidad y monitoreo del rendimiento de procesos en tiempo real (*Celonis*, 2024).

PM4PY

PM4PY es una librería de minería de procesos en *Python*, diseñada para ser fácil de usar y altamente flexible. Es adecuada para la integración en entornos de análisis de datos y para ser utilizada por desarrolladores que buscan automatizar el análisis de procesos. Además de soportar múltiples técnicas de minería de procesos, se destaca por su fácil integración con otros entornos de análisis de datos (Berti, van Zelst, y Schuster, 2023). Asimismo, *PM4PY* está disponible en su versión profesional mediante *PMTk*, una interfaz gráfica proporcionada por *Process Intelligence Solutions*, que facilita el acceso a las herramientas de análisis de procesos para usuarios no técnicos (*Process Intelligence Solutions*, 2024) .

BupaR

BupaR es un paquete de minería de procesos en *R*, diseñado para analistas de datos que trabajan en el ecosistema de *R*. Proporciona herramientas para el análisis y visualización de procesos, facilitando la extracción de conocimientos valiosos de los registros de eventos.(Janssenswillen, Depaire, Swennen, Jans, y Vanhoof, 2018)

2.2.4. Algoritmos

En el campo de la minería de procesos, existe una amplia variedad de algoritmos diseñados para descubrir modelos de procesos a partir de registros de eventos. Estos algoritmos abordan diferentes desafíos relacionados con el manejo de datos ruidosos, estructuras complejas, concurrencia, y la necesidad de garantizar un equilibrio entre precisión y simplicidad en los modelos generados.

Entre los métodos más representativos se encuentran algoritmos como *Alpha*, *Heuristics Miner* e *Inductive Miner* (W. van der Aalst, 2016), entre otros. Cada uno de estos enfoques ofrece características y ventajas específicas, dependiendo de los requisitos y del contexto en el que se apliquen.

Un análisis comparativo de los principales algoritmos de minería de procesos revela que cada técnica presenta fortalezas y limitaciones dependiendo del contexto en el que se aplique. Algunos algoritmos destacan por su capacidad de manejar grandes volúmenes de datos o comportamientos infrecuentes, mientras que otros priorizan la simplicidad y estructura de los modelos generados. Estas diferencias hacen que la selección del algoritmo adecuado sea una tarea crítica y dependiente de los objetivos específicos del análisis.

Estudios recientes han evaluado y comparado exhaustivamente estas técnicas en diferentes escenarios, proporcionando una base sólida para comprender sus características distintivas y áreas de aplicación (Augusto y cols., 2018).

2.3. Arquitectura de Microservicios

Los microservicios representan un paradigma arquitectónico para el desarrollo de aplicaciones. Este enfoque se caracteriza por la creación de servicios pequeños e independientes, cada uno operando en su propio proceso y comunicándose mediante interfaces ligeras, típicamente a través de una *API*. Estos servicios se estructuran en torno a las capacidades específicas del negocio y son desplegables de manera individual. Cada microservicio se concibe como una entidad autónoma, encargada de una única funcionalidad del negocio. Esta arquitectura permite a los equipos de desarrollo trabajar de forma independiente en distintos servicios, facilitando así la escalabilidad y la actualización eficiente de la aplicación.(Irakli Nadareishvili y Amundsen, 2016)

Este paradigma representa una evolución significativa en la arquitectura de software, ofreciendo una alternativa eficaz a los enfoques monolíticos tradicionales y proporcionando una base sólida para la construcción de sistemas distribuidos y altamente escalables en el contexto de las demandas actuales del mercado.

El valor real de la arquitectura de microservicios no radica en la tecnología, procesos o herramientas utilizados para el desarrollo del sistema, sino que se encuentra en el objetivo en sí mismo: un sistema que pueda facilitar el cambio.

2.3.1. Patrones de diseño

Los patrones de diseño son esenciales para el desarrollo y mantenimiento efectivos de una arquitectura de microservicios. Estos patrones proporcionan soluciones probadas para problemas comunes y ayudan a garantizar que los servicios sean escalables, mantenibles y confiables. En esta sección (Chris Richardson, 2024) se describen algunos de los patrones de diseño clave en la arquitectura de microservicios. Si bien estos patrones no son exhaustivos, proporcionan una base sólida para diseñar sistemas de microservicios robustos y escalables.

Patrones de descomposición del sistema

- Descomposición por Capacidad de Negocio: Este patrón sugiere definir servicios en función de las capacidades de negocio. Cada servicio representa una capacidad de negocio completa y autónoma, lo que facilita el alineamiento de los servicios con los objetivos organizacionales.

- Descomposición por Subdominio: Relacionado con el diseño dirigido por el dominio (*DDD*), este patrón implica definir servicios que correspondan a subdominios específicos dentro del modelo de dominio general. Esto ayuda a mantener la coherencia del modelo de datos y las reglas de negocio.

Patrones de gestión de datos

- Base de Datos por Servicio: Cada microservicio tiene su propia base de datos privada, lo que asegura que los servicios estén desacoplados y puedan evolucionar de manera independiente. Sin embargo, esto puede presentar desafíos en términos de consistencia de datos.

- Sagas: Las sagas son una forma de gestionar la consistencia de los datos en un sistema distribuido. Consisten en una serie de transacciones locales coordinadas que se ejecutan para completar una transacción de negocio. Si una de las transacciones falla, se ejecuta una serie de transacciones compensatorias para revertir las operaciones realizadas.

- Eventos de Dominio: Este patrón implica publicar un evento cada vez que ocurre un cambio significativo en los datos. Otros servicios pueden suscribirse a estos eventos y actuar en consecuencia, lo que facilita la coordinación y la integración de servicios.

Patrones de comunicación entre servicios

- Invocación de Procedimientos Remotos (RPI): Este mecanismo utiliza protocolos como HTTP o *RPC* para la comunicación síncrona entre servicios. Aunque es directo, puede llevar a una alta latencia y acoplamiento temporal entre servicios.

- Mensajería Asíncrona: Utiliza colas de mensajes o sistemas de mensajería para la comunicación asíncrona. Este enfoque mejora la confiabilidad y desacopla temporalmente los servicios, permitiendo que continúen operando incluso si uno de los servicios está inactivo.

Patrones de despliegue y operación

- Múltiples Instancias por Host: Este patrón implica desplegar múltiples instancias de servicios en un solo *host*. Es una forma eficiente de utilizar recursos de *hardware*, aunque puede complicar la gestión y el aislamiento de fallos.

- Instancia de Servicio por Contenedor: Cada instancia de servicio se despliega en su propio contenedor. Esto proporciona un buen aislamiento y portabilidad, facilitando el despliegue y la escalabilidad.

- Despliegue sin Servidor (Serverless): Utiliza plataformas de computación en la nube para ejecutar servicios en respuesta a eventos. Esto elimina la necesidad de gestionar servidores, aunque puede introducir latencia adicional y limitaciones en el control de la infraestructura.

Patrones de asuntos transversales

- Chasis de Microservicios: Un chasis de microservicios es un *framework* que maneja las preocupaciones transversales como el registro de *logs*, la gestión de la configuración y el manejo de errores. Esto simplifica el desarrollo de nuevos servicios al proporcionar una base común.

- Configuración Externalizada: Externalizar todas las configuraciones (por ejemplo, la ubicación de la base de datos y las credenciales) permite que los servicios sean más flexibles y fáciles de administrar en diferentes entornos.

2.3.2. Beneficios y desafíos de la arquitectura de Microservicios

La arquitectura de microservicios se ha convertido en un enfoque popular para el desarrollo de aplicaciones modernas debido a sus numerosas ventajas. Sin embargo, también presenta una serie de desafíos que deben ser considerados cuidadosamente. En esta sección se examinan las principales ventajas y desventajas de adoptar una arquitectura de microservicios.

Ventajas de la Arquitectura

- Escalabilidad independiente: Cada microservicio puede ser escalado de manera independiente según sus necesidades específicas de carga. Esto permite una utilización más eficiente de los recursos y una respuesta rápida a los cambios en la demanda. (Irakli Nadareishvili y Amundsen, 2016)

- Despliegue rápido y frecuente: Los equipos pueden desplegar actualizaciones a servicios específicos sin necesidad de modificar y volver a desplegar toda la aplicación. Esto reduce el tiempo de inactividad y acelera la entrega de nuevas funcionalidades. (Chris Richardson, 2024)

- Resiliencia y aislamiento de fallos: Los fallos en un servicio no necesariamente afectan a otros servicios. Aumenta la resiliencia del sistema global, ya que los fallos pueden ser contenidos y manejados más fácilmente.(Chris Richardson, 2024)

- Alineación con las estructuras de equipos: Los microservicios permiten que los equipos sean responsables de servicios específicos, mejorando la propiedad y la responsabilidad. Esto promueve una mayor eficiencia y motivación dentro de los equipos.(Irakli Nadareishvili y Amundsen, 2016)

Desventajas de la Arquitectura

- Complejidad de gestión: La gestión de múltiples servicios independientes aumenta la complejidad operativa. Requiere herramientas avanzadas para la orquestación, el monitoreo y el despliegue.(Chris Richardson, 2024)

- Latencia y sobrecarga de comunicación: La comunicación entre servicios a través de la red puede introducir latencia. La necesidad de manejar la comunicación de red y asegurar la consistencia de los datos puede complicar el diseño del sistema.(Irakli Nadareishvili y Amundsen, 2016)
- Consistencia de datos: Mantener la consistencia de los datos en un sistema distribuido es un desafío significativo. Requiere implementar patrones complejos como Sagas o eventos de dominio para asegurar la coherencia.(Chris Richardson, 2024)
- Requisitos de infraestructura: Los microservicios pueden requerir una infraestructura más sofisticada para soportar la orquestación y el monitoreo. Esto puede implicar costos adicionales y la necesidad de habilidades especializadas en el equipo de desarrollo.(Chris Richardson, 2024)
- Carga de mantenimiento: Mantener múltiples servicios y sus respectivas interfaces puede incrementar la carga de mantenimiento. Requiere un esfuerzo continuo para asegurar la integridad y compatibilidad de los servicios.(Irakli Nadareishvili y Amundsen, 2016)

Capítulo 3

Problema planteado

La mejora continua de los procesos de negocio en las organizaciones requiere una evaluación constante de la operativa diaria mediante la obtención y análisis de diversas métricas. La minería de procesos se presenta como una técnica clave para llevar a cabo este análisis, utilizando métodos de minería de datos para examinar los registros de eventos y descubrir patrones, comparar ejecuciones reales con modelos predefinidos y optimizar los procesos.

A pesar de la existencia de herramientas y librerías especializadas, que permiten utilizar diversos algoritmos de minería de procesos, la selección y aplicación de estas tecnologías en diferentes contextos resulta compleja y demandante. Integrar estas herramientas en los sistemas organizacionales implica desafíos significativos, lo que dificulta su adopción y aprovechamiento pleno.

Ante esta situación, surge la necesidad de una solución que facilite el acceso y la integración de técnicas de minería de procesos. Una plataforma basada en microservicios podría ofrecer una vía para reducir la complejidad asociada, promoviendo su uso y permitiendo a distintos participantes aprovechar estas tecnologías de manera más eficiente.

En este capítulo se describe el objetivo del proyecto, especificando los requerimientos funcionales y no funcionales del sistema implementado.

3.1. Requerimientos del sistema

Como se mencionó anteriormente, el principal objetivo de la plataforma desarrollada es proveer una interfaz para que usuarios (técnicos o no técnicos) puedan utilizar las distintas herramientas de minería de procesos sin tener que manejar directamente los detalles técnicos de las mismas (instalación, invocación por línea de comandos, etc.).

En esta sección se describen los principales requerimientos identificados para el desarrollo de la solución. Estos requisitos no solo definen las capacidades esenciales de la plataforma, sino que también establecen las bases para el diseño de la arquitectura que se presenta en la sección 4.1.

3.1.1. Requerimientos funcionales

En primer lugar se describen en alto nivel los requerimientos funcionales del sistema, es decir aquellos que tienen que ver con el comportamiento del mismo.

Autenticación

Como primer punto, la plataforma debe contar con un sistema de autenticación de usuarios. Este requisito funcional resulta fundamental, ya que permite obtener información específica sobre cada sesión de usuario, llevar un registro detallado de los recursos utilizados, y garantizar la personalización y seguridad de la experiencia.

El sistema de autenticación debe estar basado en un flujo básico que incluya el registro de nuevos usuarios y el inicio de sesión mediante un nombre de usuario y contraseña.

Los casos de uso principales que componen este requerimiento funcional son los siguientes:

- Registro de usuarios: este caso de uso permite que nuevos usuarios creen una cuenta en la plataforma proporcionando su información básica, como nombre de usuario y contraseña.
- Inicio de sesión: los usuarios registrados pueden autenticarse proporcionando su nombre de usuario y contraseña. Si las credenciales son correctas, se genera un *token* de sesión que permite mantener la autenticación activa durante el uso de la plataforma.
- Cierre de sesión: este caso de uso asegura que los usuarios puedan cerrar sesión de manera segura, invalidando el *token* de sesión para prevenir accesos posteriores desde dispositivos no autorizados.

Ejecución de tareas

La plataforma debe centrarse en la ejecución de algoritmos de minería de procesos, ofreciendo soporte para diversas técnicas e implementaciones. Su principal característica es la capacidad de crear, configurar y ejecutar tareas personalizadas, brindando a los usuarios la flexibilidad necesaria para adaptarse a múltiples escenarios y necesidades específicas.

La plataforma debe permitir a los usuarios crear tareas de ejecución para algoritmos de minería de procesos, configurándolas según sus requisitos específicos. Durante este proceso, el usuario podrá definir:

- Técnica de Minería de Procesos: los usuarios podrán seleccionar entre las técnicas de Descubrimiento, Conformidad y Extensión
- Librería o tecnología: se proporcionará la opción de seleccionar la tecnología o librería específica para ejecutar el algoritmo. Esto incluye herramientas compatibles disponibles en la plataforma, lo que permitirá a los usuarios trabajar con la tecnología que mejor se adapte a sus necesidades.

- Parámetros del algoritmo: cada algoritmo puede requerir configuraciones específicas. Los usuarios tendrán la posibilidad de definir estos parámetros como se especifica en detalle en el requerimiento de Configuración de parámetros

Una vez creada una tarea, la plataforma debe ofrecer funcionalidades para realizar un seguimiento detallado de su ejecución. El sistema registrará y actualizará automáticamente el estado de cada tarea creada, permitiendo a los usuarios consultar en cualquier momento el progreso y resultado. De este modo el usuario podrá visualizar en la plataforma cuando la tarea haya culminado, tanto si finalizó correctamente como si la tarea fue interrumpida por un fallo o error durante su ejecución.

Configuración de parámetros

La plataforma debe ofrecer a los usuarios la posibilidad de configurar de manera detallada los parámetros necesarios para la ejecución de algoritmos de minería de procesos. Esta funcionalidad permite una personalización profunda, asegurando que cada tarea se ajuste a las necesidades y objetivos específicos del usuario.

Durante la creación de la tarea, el sistema presentará un conjunto de parámetros configurables específicos para cada algoritmo y su correspondiente implementación. Estos parámetros deben reflejar las opciones disponibles en las librerías o tecnologías subyacentes, garantizando que los usuarios puedan interactuar con los algoritmos de la misma manera que lo harían al trabajar directamente con las herramientas originales.

La plataforma adaptará automáticamente la interfaz de configuración según el algoritmo y la técnica seleccionados, mostrando únicamente los parámetros relevantes para evitar confusiones y facilitar el proceso de personalización. Antes de ejecutar la tarea, el sistema debe validar los parámetros configurados para garantizar la coherencia y evitar errores comunes en la ejecución.

Gestión de archivos

La plataforma debe incluir una funcionalidad para la carga, almacenamiento y gestión de archivos de entrada necesarios para la ejecución de los algoritmos de minería de procesos. Esta característica garantiza una experiencia eficiente y centralizada para los usuarios, asegurando la integridad y la organización de los datos.

- Carga de archivos de entrada: los usuarios podrán cargar archivos de entrada directamente a la plataforma utilizando una interfaz intuitiva que debe soportar los formatos de datos más comunes en minería de procesos, tales como *XES* y *CSV*.

- Persistencia de archivos: todos los archivos cargados deben ser almacenados en el sistema, permitiendo su reutilización en futuras tareas sin necesidad de volver a cargarlos.

- Gestión centralizada: los usuarios dispondrán de un panel de gestión donde podrán visualizar todos los archivos cargados, se listarán únicamente los archivos cargados por el usuario. Además se tendrá la opción de eliminar archivos individuales en caso de ser necesario.

Visualización y descarga de resultados

La plataforma debe ofrecer una funcionalidad integral para la visualización y gestión de los resultados generados tras la ejecución de las tareas. Esta característica debe estar diseñada para proporcionar una experiencia completa, permitiendo a los usuarios analizar, comprender y visualizar los resultados de manera eficiente, directamente desde la interfaz web.

- Visualización de resultados: una vez finalizada la ejecución de una tarea, el usuario podrá acceder a los resultados a través de un panel de visualización interactivo. Los resultados se presentarán de manera clara y estructurada, destacando tanto los datos generados por el algoritmo como la información básica relacionada con la tarea como el nombre, algoritmo utilizado y tiempos de ejecución. Esta información proporcionará contexto al usuario y facilitará la comparación entre tareas.

- Descarga de resultados generados: el usuario tendrá la opción de descargar directamente desde la plataforma los resultados producidos por el algoritmo ejecutado.

Filtrado de tareas

El sistema debe proporcionar la capacidad de filtrar las ejecuciones disponibles según diferentes criterios, incluyendo el nombre de la tarea y las etapas específicas de la minería de procesos (descubrimiento, conformidad, extensión). Esta funcionalidad permitirá a los usuarios localizar de manera rápida y eficiente las tareas creadas en la plataforma, incluso en escenarios con un gran volumen de ejecuciones.

Los filtros estarán integrados en la interfaz de usuario de forma clara y accesible, con opciones desplegables o campos de búsqueda que permitan combinar criterios de manera sencilla.

3.1.2. Requerimientos no funcionales

Además de los requerimientos funcionales mencionados en la sección anterior, la plataforma debe cumplir con algunas características de operación y calidad que se destacan a continuación:

Arquitectura basada en microservicios (escalabilidad, flexibilidad)

El sistema debe implementarse siguiendo una arquitectura de microservicios, lo que permite escalabilidad y flexibilidad. Cada componente debe ser modular

y autónomo, con una clara separación de responsabilidades para facilitar el mantenimiento y la integración de nuevas funcionalidades.

Extensibilidad

La plataforma debe ser fácilmente extensible, permitiendo la incorporación de nuevos algoritmos y herramientas sin requerir modificaciones significativas en el código base. Esto garantiza que el sistema pueda evolucionar y adaptarse a nuevas tecnologías y métodos de minería de procesos en el futuro.

Interfaz intuitiva, segura y de alto rendimiento

Es fundamental diseñar una interfaz que sea intuitiva y accesible, de modo que tanto usuarios técnicos como no técnicos puedan navegar y utilizar la plataforma sin dificultad. La usabilidad, la seguridad y el rendimiento de la solución son aspectos clave para maximizar su adopción y efectividad, asegurando que los usuarios tengan una experiencia confiable y ágil en todo momento.

Internacionalización

La interfaz debe ser configurable para ser visualizada en múltiples idiomas, comenzando con soporte para español e inglés.

Capítulo 4

Diseño de la solución

En este capítulo se describen todas las decisiones que se tomaron para la solución al problema propuesto y se presenta su diseño, incluyendo la arquitectura del sistema, los patrones de diseño que se aplicaron y las principales características de la solución.

4.1. Arquitectura general de la solución

La solución propuesta está diseñada bajo una arquitectura basada en microservicios, lo que permite descomponer el sistema en componentes independientes, cada uno encargado de una funcionalidad específica. Este enfoque facilita el desarrollo, despliegue y escalabilidad de la plataforma, brindando flexibilidad para realizar actualizaciones y mejoras sin afectar el sistema en su totalidad. Además permite utilizar diferentes tecnologías y lenguajes de programación para los distintos servicios, eligiendo la mejor herramienta para cada tarea.

4.1.1. Componentes del sistema

La arquitectura de la solución se compone de varios microservicios que cumplen roles específicos dentro del sistema. En la figura 4.1 se presenta el diagrama de la arquitectura, en el cual se pueden observar los distintos componentes de la solución.

A continuación, se detallan los componentes más importantes en base a las responsabilidades y necesidades para los casos de uso definidos.

Por un lado se tienen los componentes que no están directamente ligados con las implementaciones de las herramientas utilizadas pero que cumplen un rol clave para el correcto funcionamiento de la plataforma. Estos componentes son los siguientes:

- Autorizador: Provee servicios de autenticación y autorización, garantizando que solo usuarios autorizados puedan acceder y utilizar la plataforma.

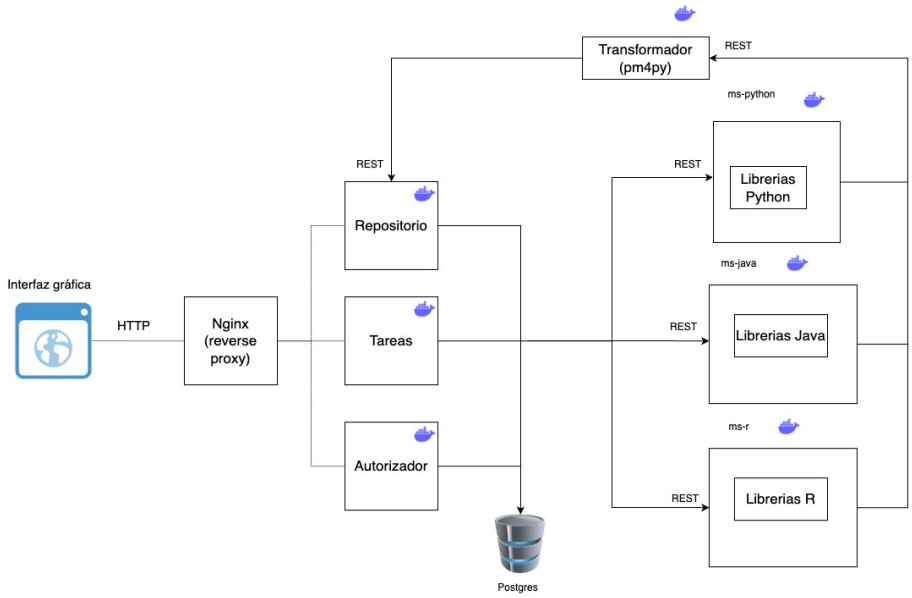


Figura 4.1: Diagrama de componentes de la arquitectura

- Tareas: Maneja la ejecución de las tareas de minería de procesos, incluyendo la coordinación y orquestación de las llamadas a los microservicios específicos de análisis.
- Repositorio: Gestiona el almacenamiento y recuperación de los archivos manejados por la plataforma, tanto los ingresados por el usuario (archivo de *logs*) como los archivos obtenidos como resultados de las tareas.

Por otro lado se definieron los microservicios que contienen las herramientas encargadas de ejecutar las tareas de minería de procesos. Dado que uno de los principales objetivos de la plataforma es proveer al usuario de diferentes implementaciones de los principales algoritmos utilizados en la minería de procesos, se utilizó un diseño de la arquitectura donde se dividen a los microservicios en base a las tecnologías soportadas por la plataforma. De esta manera se definió e implementó un microservicio por cada tecnología utilizada para el prototipo presentado. Como se puede observar en la figura del diagrama, se tienen los siguientes componentes:

- Microservicio de Python: componente que contiene y ejecuta las distintas herramientas implementadas en (*Python*, 2024).
- Microservicio de Java: componente que contiene y ejecuta las distintas herramientas implementadas en (*Java*, 2024).
- Microservicio de R: componente que contiene y ejecuta las distintas herramientas implementadas en (*R*, 2024).

Otros componentes que forman parte de la arquitectura:

- Interfaz Gráfica: Provee la interfaz de usuario mediante la cual los distintos participantes pueden interactuar con la plataforma.
- Transformador: Provee servicios para transformar los archivos resultados de las ejecuciones.
- (*PostgreSQL*, 2024): Base de datos relacional utilizada para almacenar datos críticos de la plataforma, incluyendo información de usuarios y registros de ejecuciones.
- (*Nginx*, 2024) (*Reverse Proxy*): Actúa como un *proxy* inverso, gestionando y distribuyendo las solicitudes HTTP hacia los diferentes microservicios de la plataforma.

4.1.2. Arquitectura de microservicios

La arquitectura de microservicios aporta varios beneficios en términos de escalabilidad, despliegue continuo y mantenibilidad. Cada microservicio es independiente y puede ser desarrollado, desplegado y actualizado sin interferir en el funcionamiento de otros servicios.

Para la comunicación entre microservicios se utilizan *APIs REST* (Fielding y Taylor, 2000), lo que permite una interfaz estandarizada basada en el protocolo HTTP. Este enfoque ofrece flexibilidad en la integración con diferentes tecnologías y lenguajes de programación. Cada servicio expone sus funcionalidades mediante *endpoints* específicos, lo que facilita la interoperabilidad entre ellos.

Entre los principales motivos por los cuales se decidió utilizar *APIs REST* se tienen los siguientes:

- Interfaz estandarizada: *REST* proporciona una interfaz estandarizada que facilita la integración entre diferentes componentes del sistema y con servicios externos. Esto simplifica la comunicación y reduce la complejidad en la interacción entre microservicios.
- Independencia de plataforma: Al ser independiente del lenguaje de programación y la plataforma, las *APIs RESTful* permiten que los microservicios desarrollados en diversas tecnologías puedan comunicarse sin problemas, lo que aumenta la flexibilidad en la elección de herramientas y tecnologías.
- Escalabilidad y rendimiento: Al utilizar el protocolo HTTP, las *APIs RESTful* son ligeras y fácilmente escalables. Además, permiten el almacenamiento en caché de respuestas, optimizando aún más el rendimiento general del sistema.
- Facilidad de consumo: Estas *APIs* utilizan formatos estándar como *JSON* o *XML*, lo que facilita su consumo tanto por aplicaciones como por clientes

externos, ya sean usuarios humanos o máquinas, permitiendo una interacción rápida y sencilla con los servicios.

- Simplicidad y mantenimiento: Los principios y convenciones bien definidos de las *APIs RESTful* fomentan la simplicidad, promoviendo la consistencia en su diseño. Esto no solo facilita el desarrollo, sino que también reduce los esfuerzos de mantenimiento a largo plazo, permitiendo un ciclo de vida más eficiente para las *APIs*.

Además, la plataforma se beneficia de contenedores mediante el uso de *Docker* (*Docker*, 2024), lo que asegura un entorno consistente y facilita la portabilidad y el despliegue. *Docker* permite empaquetar una aplicación y todas sus dependencias en un contenedor ligero y reproducible. Esto garantiza que la aplicación se ejecute de la misma manera en cualquier entorno, eliminando problemas de incompatibilidad y simplificando el proceso de despliegue y escalado.

Por otro lado, el sistema hace uso de un *Reverse proxy*. El uso de *NGINX* como *reverse proxy* permite gestionar las solicitudes entrantes de manera eficiente y segura, distribuyéndolas a los microservicios adecuados. *NGINX* puede balancear la carga entre múltiples instancias de microservicios, mejorar la seguridad mediante la terminación *SSL* y proteger contra ataques comunes como *DDoS*. Además, *NGINX* proporciona *caching* y compresión, mejorando así el rendimiento general del sistema.

4.1.3. Diagrama de Entidad-Relación de Base de Datos (ERD)

En la figura 4.2 se expone el *ERD* del diseño de base de datos que se utilizó. En él se pueden observar tres tablas:

- *Users*: Representa a los usuarios del sistema. Contiene las siguientes columnas:
  - `id`: la clave primaria de las filas de esta tabla.
  - `name`: nombre completo del usuario.
  - `username`: nombre del usuario utilizado para iniciar sesión en el sistema.
  - `password`: contraseña del usuario en base de datos. Cabe destacar que la misma se encripta antes de ser almacenada.
- *Tasks*: Representa las tareas de un usuario en el sistema. Contiene las siguientes columnas:
  - `id`: la clave primaria de las filas de esta tabla.
  - `user_id`: clave foránea a la tabla `users`, para poder acceder desde un usuario a todas sus tareas.
  - `name`: nombre de la tarea especificada por el usuario, para visualizar en la interfaz gráfica.

- `pid`: identificador del proceso en el contenedor del microservicio correspondiente. No se llegó a utilizar en la implementación actual, pero véase 7.3 para una descripción detallada de su propósito.
- `started_at`: fecha de comienzo de ejecución de la tarea.
- `ended_at`: fecha de fin de ejecución de la tarea.
- `algorithm`: clave del algoritmo ejecutado. Véase la sección 5.4 para más detalles sobre el rol de este campo.
- `algorithm_readable_name`: nombre completo del algoritmo, que se muestra en la interfaz gráfica.
- `service`: identificador del microservicio en el cual se ejecutó la tarea. Véase la sección 5.4.1 para más detalles sobre este campo.
- `parameters`: objecto con los parámetros y configuraciones escogidas por el usuario para el algoritmo. Véase la sección 5.4 para más detalles sobre los valores que puede tomar este objeto.
- `state`: valor enumerado que describe el estado actual en que se encuentra la tarea. Puede ser "pendiente" (`pending`), "ejecutando" (`running`), "finalizada" (`finished`), o "fallida" (`failed`) debido a algún error durante la ejecución (`error`).
- `category`: valor eumerado que describe la categoría del algoritmo ejecutado. Puede tomar los valores `discovery`, para los algoritmos de descubrimiento de modelos de negocio (2.2.2), `conformance_checking`, para los algoritmos de conformidad ejecutados sobre un archivo de modelo de negocio (2.2.2), o `enhancement`, para los algoritmos de extensión de un modelo de negocio (2.2.2).
- `output_extension`: extensión del archivo de salida producida por el algoritmo
- `created_at`: fecha en que la tarea fue creada.
- `updated_at`: fecha en que la tarea fue actualizada.

- *TokensBlacklist*: Representa los *tokens* de sesión *JWT* (4.4.1) que fueron previamente utilizados en el sistema, con el fin de evitar producir *tokens* iguales en el tiempo, lo cual introduciría problemas de seguridad. Un *token* se considera previamente utilizado si el usuario lo descartó al cerrar sesión, o si el mismo caduca.

  - `token`: la clave primaria de las filas de esta tabla, que también coincide con el valor del *JWT* que fue previamente descartado.
  - `created_at`: fecha en la cual el *token* fue marcado como expirado.

Para las claves primarias de las tablas se optó por el tipo *UUID* (Leach, Salz, y Mealling, 2005), el cual garantiza unicidad en el espacio y en el tiempo. Esto brinda una capa más de seguridad al sistema, ya que dificulta a potenciales atacantes de acceder a los datos de los usuarios debido a la complejidad que supone obtener, al menos mediante intento y error, un identificador que corresponda a un registro en el sistema.

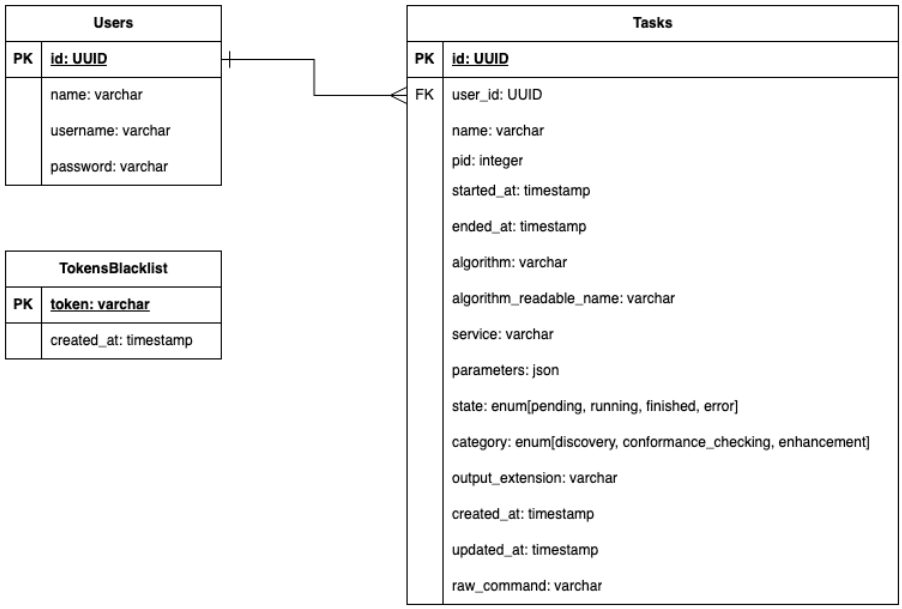


Figura 4.2: Diagrama Entidad-Relación del esquema de Base de Datos utilizado

4.1.4. Interacción entre componentes

A continuación, se presentan las principales interacciones entre los componentes del sistema.

Los usuarios interactúan con la plataforma a través de la interfaz gráfica, enviando solicitudes HTTP. El *reverse proxy* recibe estas solicitudes y las redirige al microservicio adecuado, ya sea para autenticación (servicio Autorizador), carga de archivos (servicio Repositorio) o ejecución de tareas (servicio Tareas).

Cuando el servicio de Tareas recibe una nueva tarea ingresada por el usuario, se encarga de coordinar la invocación del algoritmo correspondiente en base a la selección del usuario. Esto lo realiza mediante un llamado a la *API RESTful* definida en el servicio específico que contiene la implementación del algoritmo seleccionado para ejecutar. A su vez, el servicio de Tareas gestiona y realiza el seguimiento del estado de la ejecución, comunicándose con el servicio específico que está ejecutando el algoritmo.

Una vez que la ejecución finaliza, el servicio se encarga de subir el archivo obtenido por el algoritmo mediante el servicio de Repositorio. Además, como parte del procesamiento de los resultados de la ejecución, se hace uso del servicio de Transformador para generar una imagen a partir del modelo generado por el algoritmo. Esta imagen es la que se le presenta al usuario cuando selecciona la tarea finalizada en la interfaz web.

4.2. Patrones de diseño

En el diseño de la solución se emplearon diversos patrones de diseño que permitieron estructurar el sistema de manera modular, escalable y fácil de mantener. Estos patrones abarcan tanto el diseño general del software como los enfoques específicos utilizados en la arquitectura de microservicios. A continuación, se describen desde una perspectiva conceptual, los principales patrones de diseño empleados. La relación con las tecnologías empleadas se detallan con mayor profundidad en el Capítulo 5.

4.2.1. Patrones de diseño generales

Los patrones de diseño clásicos aplicados en esta solución fueron clave para modularizar la lógica de negocio, promover la reutilización de código y facilitar el mantenimiento. Entre los más destacados, se encuentran los siguientes:

Patrón Strategy

El patrón de diseño *Strategy* (Gamma, Helm, Johnson, y Vlissides, 1994, Chapter 5) es un patrón de comportamiento que permite definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables. El patrón *Strategy* permite que el algoritmo varíe independientemente de los clientes que lo utilizan.

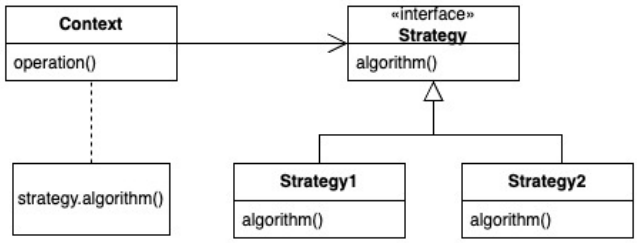


Figura 4.3: Diagrama del patrón Strategy

El patrón cuenta con los siguientes componentes:

1. Estrategia: Una interfaz común para todas las estrategias concretas. El contexto utiliza esta interfaz para llamar al algoritmo definido por una estrategia concreta.

2. Estrategia concreta: Implementa el algoritmo utilizando la interfaz Estrategia.

3. Contexto: Mantiene una referencia a un objeto Estrategia y se comunica con este objeto solo a través de la interfaz Estrategia.

Utilizar este patrón presenta varios beneficios entre los cuales se encuentran los siguientes:

- Flexibilidad: Permite cambiar el algoritmo o comportamiento en tiempo de ejecución.
- Mantenimiento: Facilita la adición de nuevas estrategias sin modificar el contexto.
- Reutilización: Las estrategias pueden ser reutilizadas en diferentes contextos.

Este patrón fue implementado para gestionar las diferentes estrategias de almacenamiento de archivos en la plataforma. Dicho patrón permite encapsular las opciones de almacenamiento dentro de estrategias concretas que se pueden seleccionar de forma dinámica durante la ejecución. Así, el sistema ofrece la flexibilidad de elegir entre distintos enfoques de almacenamiento, según las necesidades del usuario o del entorno de despliegue.

La ventaja de usar el patrón *Strategy* en este contexto es que facilita la extensibilidad del sistema. Nuevos tipos de almacenamiento se pueden incorporar como estrategias adicionales sin afectar la lógica existente ni los demás componentes del sistema. De esta manera, cada nueva opción de almacenamiento se implementa como una estrategia independiente, lo que asegura una solución modular, adaptable y fácil de mantener.

Patrón Repository

El patrón de diseño (*Repository*, 2024) es un patrón de acceso a datos que proporciona una abstracción sobre la capa de persistencia. Su objetivo es separar la lógica de negocio de la lógica de acceso a datos, permitiendo que la lógica de negocio interactúe con los datos de una manera más limpia y mantenible.

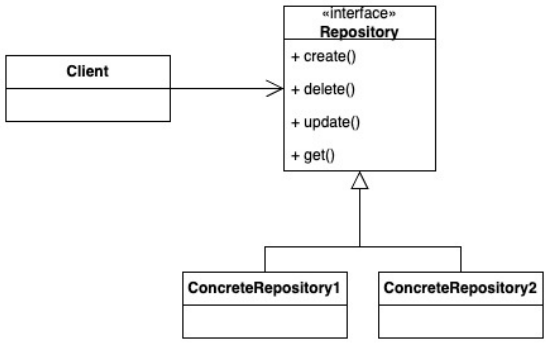


Figura 4.4: Diagrama del patrón Repository

Los componentes del patrón son los siguientes:

1. Interfaz repositorio: Define los métodos que un repositorio debe implementar para interactuar con los datos.

2. Repositorio concreto: Implementa la interfaz del repositorio y contiene la lógica para interactuar con los datos.

Los principales beneficios del patrón *Repository* se listan a continuación:

- Abstracción del acceso a datos: Proporciona una capa de abstracción sobre la lógica de acceso a datos, lo que permite cambiar la implementación de la persistencia sin afectar a la lógica de negocio.
- Mantenimiento: Centraliza la lógica de acceso a datos en un solo lugar, lo que facilita el mantenimiento y la actualización del código.
- Reutilización de código: Los métodos genéricos en el repositorio base pueden ser reutilizados por múltiples repositorios específicos, reduciendo la duplicación de código.
- Separación de responsabilidades: Promueve la separación de responsabilidades al mantener la lógica de acceso a datos separada de la lógica de negocio.

Este patrón se utilizó en el diseño de la solución para el acceso a la persistencia de datos. El patrón *Repository* actúa como una capa intermedia entre la aplicación y la estrategia de persistencia. Este patrón abstrae las operaciones de acceso a datos, facilitando la interacción con las fuentes de datos sin exponer los detalles de implementación.

Con el uso de *Repository*, es posible realizar operaciones como la consulta, inserción, actualización y eliminación de datos de manera centralizada, proporcionando un punto único para el manejo de la persistencia. Esto también permite cambiar o escalar la solución sin afectar las capas superiores de la aplicación.

Patrón Inyección de Dependencias

El patrón de diseño de (*Inyección de Dependencias*, 2024) es un patrón de diseño que permite a un objeto recibir sus dependencias de una fuente externa en lugar de crearlas por sí mismo. Esto promueve la separación de responsabilidades y facilita la prueba y el mantenimiento del código.

Los componentes del patrón son los siguientes:

1. Cliente: El objeto que tiene dependencias.

2. Servicio: La dependencia que el cliente necesita.

3. Inyector: El componente que inyecta las dependencias en el cliente.

A continuación se listan los principales beneficios de la inyección de dependencias:

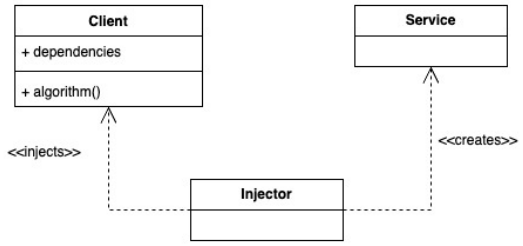


Figura 4.5: Patrón Inyección de Dependencias

- Desacoplamiento: Reduce el acoplamiento entre las clases, facilitando el mantenimiento y la evolución del código.
- Reutilización de Código: Las dependencias pueden ser reutilizadas en diferentes contextos.
- Flexibilidad: Facilita el cambio de implementaciones de dependencias sin modificar el código del cliente.

El patrón de Inyección de Dependencias fue utilizado para desacoplar las dependencias entre los componentes de la plataforma. Este enfoque permite que las dependencias se proporcionen a los objetos en lugar de ser gestionadas directamente dentro de los mismos, lo que facilita el reemplazo de componentes y mejora la *testeabilidad* del sistema.

Gracias a este patrón, se logra una mayor modularidad en el sistema, permitiendo inyectar nuevas implementaciones de componentes sin necesidad de alterar el código que las consume.

Patrón de diseño Command

El patrón de diseño *Command* (Gamma y cols., 1994, Chapter 5) es un patrón de comportamiento en el cual un objeto es utilizado para encapsular toda la información necesaria para realizar una acción. Este patrón facilita la implementación de componentes que requieran delegar o ejecutar métodos en el momento en que lo necesiten.

Los componentes del patrón son los siguientes:

1. Comando (*Command*): Una interfaz que declara un método para ejecutar una operación.
2. Comando concreto (*ConcreteCommand*): Implementa la interfaz Comando y define la asociación entre el receptor y una acción.
3. Invocador (*Invoker*): Pide al objeto Comando que ejecute la solicitud.
4. Receptor (*Receiver*): Conoce cómo realizar las operaciones asociadas a llevar a cabo una solicitud.

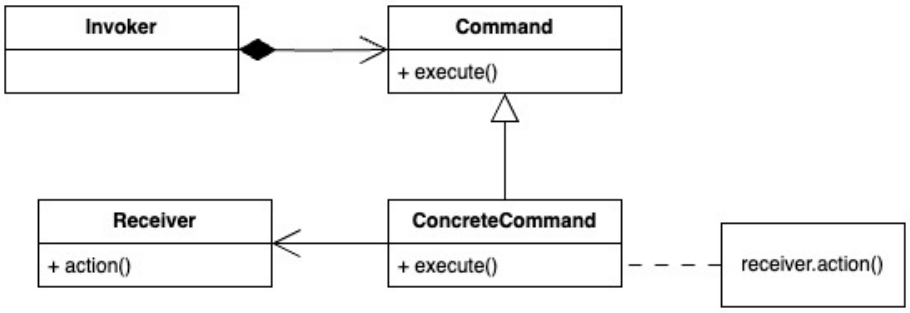


Figura 4.6: Patrón Command

Los principales beneficios de aplicar el patrón *Command* son los siguientes:

- Desacoplamiento: El Patrón *Command* desacopla el objeto que invoca la operación del que sabe cómo realizarla.
- Flexibilidad: Permite el ensamblaje y ejecución dinámica de comandos, lo que puede extenderse fácilmente para incluir nuevos algoritmos u operaciones.
- Deshacer/Rehacer: el Patrón *Command* puede soportar operaciones que se pueden deshacer almacenando el estado antes de la ejecución.

En el diseño de la solución se aplicó el patrón *Command* para la ejecución de los algoritmos provistos por las librerías de minería de procesos. Cada algoritmo puede ser visto como un comando que se ejecuta con ciertos parámetros.

Patrón decorator

El patrón de diseño *Decorator* (Gamma y cols., 1994, Chapter 3) es un patrón estructural que permite añadir funcionalidades a objetos de manera dinámica. Este patrón es útil cuando se desea extender las capacidades de las clases de manera flexible y sin modificar el código existente.

A continuación se listan sus principales características:

- Composición sobre herencia: El patrón *Decorator* se basa en la composición en lugar de la herencia. Esto significa que se pueden añadir funcionalidades a un objeto envolviéndolo con uno o más decoradores.
- Flexibilidad: Permite añadir o quitar responsabilidades a los objetos de manera flexible y dinámica. Los decoradores pueden ser apilados, es decir, se pueden aplicar múltiples decoradores a un mismo objeto.
- Interfaz consistente: Los decoradores implementan la misma interfaz que los objetos que decoran, lo que asegura que los objetos decorados puedan ser utilizados en lugar de los objetos originales sin cambios en el código cliente.

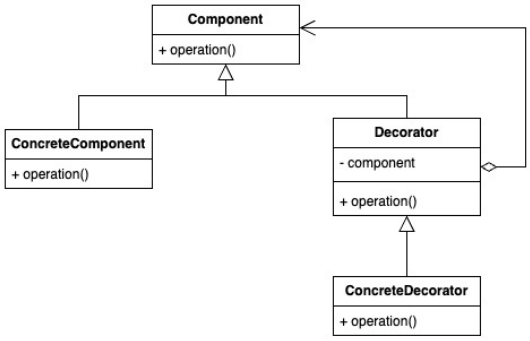


Figura 4.7: Patrón Decorator

- Transparencia: Idealmente, el uso de decoradores debería ser transparente para el código cliente. El cliente no debería necesitar saber si está tratando con un objeto decorado o no.
- Responsabilidad Única: Cada decorador tiene una responsabilidad específica, lo que permite una separación clara de las preocupaciones.

En el diseño de la solución se aplica el patrón *Decorator* para manejar la autenticación y la validación de parámetros en los distintos flujos de la plataforma.

4.2.2. Patrones de diseño de microservicios

En cuanto a los microservicios, se adoptaron algunos patrones de diseño que permitieron organizar de manera eficiente la arquitectura de microservicios de la plataforma, optimizando la escalabilidad y la resiliencia del sistema.

Descomposición por Capacidad de Negocio

Se adoptó el patrón (*Descomposición por Capacidad de Negocio*, 2024) para definir y organizar los microservicios que conforman la plataforma. Al diseñar los microservicios, se consideraron las capacidades del negocio con el objetivo de delimitar los alcances de cada servicio, asegurando que cada uno sea responsable de una función de negocio claramente definida.

Como se mencionó en la sección 4.1.1, se desarrolló un microservicio *Autorizador*, encargado de gestionar la autenticación y autorización de usuarios, y un microservicio *Tareas*, responsable de manejar la lógica asociada a las tareas de los usuarios, incluyendo la ejecución de algoritmos. Cada microservicio está alineado de manera precisa con una capacidad de negocio específica, lo que facilita tanto su gestión como su evolución de forma independiente.

Este enfoque de descomposición ofrece beneficios significativos, como una mayor escalabilidad, mejor mantenibilidad y una alineación más estrecha con los objetivos estratégicos del negocio.

Invocación de Procedimientos Remotos (RPI)

Para el diseño del patrón de comunicación entre servicios, se decidió que toda la comunicación entre microservicios se realice de manera síncrona (*RPI*, 2024), utilizando el protocolo HTTP. Como se menciona en la sección 4.1.2, cada servicio del sistema expone una interfaz estandarizada mediante una *API REST*.

Este enfoque proporciona varios beneficios clave. En primer lugar, la comunicación síncrona permite mantener un control más directo sobre las interacciones entre microservicios, asegurando la consistencia de los datos en tiempo real, ya que las respuestas se reciben de manera inmediata tras cada solicitud. Esto resulta especialmente útil en escenarios donde las operaciones dependen de datos actualizados provenientes de múltiples servicios.

Además, el uso de *APIs REST* estandarizadas simplifica la integración entre servicios y facilita la interoperabilidad. Al estar basado en HTTP, un protocolo ampliamente adoptado, el sistema se beneficia de una arquitectura ampliamente soportada y documentada, lo que reduce la complejidad al interactuar con servicios externos o al incorporar nuevos microservicios en la plataforma.

Patrón Chasis de Microservicios

Se implementó el patrón (*Chasis*, 2024) de Microservicios para estandarizar y simplificar la construcción de los microservicios. Este patrón proporciona un conjunto básico de funcionalidades comunes a todos los microservicios, como manejo de errores, monitoreo, seguridad y configuración.

Gracias a este enfoque, cada microservicio hereda automáticamente las capacidades esenciales del chasis, reduciendo la duplicación de código y mejorando la coherencia entre los servicios. Esto permite acelerar el desarrollo de nuevos microservicios sin comprometer la calidad o la integridad de la arquitectura general.

Patrón Configuración Externalizada

Se utilizó el patrón de (*Configuración Externalizada*, 2024) para gestionar las configuraciones del sistema de manera centralizada y fuera del código fuente. Este patrón facilita la modificación de configuraciones en tiempo de ejecución sin necesidad de realizar despliegues o cambios en el código.

Las configuraciones, como las credenciales de bases de datos o las rutas de acceso a recursos externos, se almacenan en un servicio de configuración separado, permitiendo que los microservicios consuman estas configuraciones dinámicamente. Esto también mejora la seguridad, al evitar la exposición de información sensible dentro del código fuente.

Patrón Instancia de Servicio por Contenedor

En cuanto al despliegue de microservicios, se adoptó el patrón (*Instancia de Servicio por Contenedor*, 2024). Cada microservicio se ejecuta en su propio

contenedor aislado, lo que proporciona un entorno de ejecución consistente y garantiza que las dependencias específicas de cada servicio no interfieran entre sí.

Este patrón también permite escalar individualmente cada microservicio, dado que cada instancia de servicio puede ser replicada según la demanda del sistema, asegurando una alta disponibilidad y optimización de recursos.

4.3. Metodología de diseño

La metodología utilizada en el desarrollo de la solución se basó principalmente en enfoques iterativos. Este enfoque de diseño se utilizó para refinar el sistema de forma continua, lo que permitió una evolución progresiva de la arquitectura y las funcionalidades. Este enfoque se centró en la mejora continua y la experimentación, integrando cambios de diseño en respuesta a pruebas y validaciones.

En las primeras etapas del proyecto, se desarrolló un prototipo básico para validar los conceptos clave y asegurar que las herramientas de minería de procesos seleccionadas fueran adecuadas para los objetivos planteados. Este prototipo permitió identificar posibles limitaciones tempranas en la arquitectura y la funcionalidad, que luego se abordaron en iteraciones posteriores.

En cada iteración, el diseño conceptual y los patrones arquitectónicos se revisaron y ajustaron. Esto permitió probar nuevas configuraciones del sistema, obtener retroalimentación sobre su rendimiento y hacer ajustes antes de pasar a la siguiente fase.

Esta metodología de diseño facilitó la evolución del sistema, corrigiendo y mejorando su arquitectura y funcionalidades en cada ciclo. Esto aseguró que el sistema final fuera no solo funcional, sino también flexible, escalable y adecuado para los usuarios previstos.

4.4. Relación con los Requerimientos del Sistema

El diseño de la solución propuesta y el desarrollo del prototipo responde directamente a los requerimientos funcionales y no funcionales del sistema descritos en la Sección 3.1. A continuación, se detalla cómo se alinean las decisiones de diseño con estos requerimientos clave, asegurando que la plataforma cumple con las expectativas de operación y calidad.

4.4.1. Requerimientos Funcionales

- Autenticación: El sistema implementa un mecanismo de autenticación, garantizando que cada usuario tenga acceso personalizado y seguro. Esto cumple con el requerimiento de autenticación presentado en 3.1.1, y asegura que solo usuarios autorizados puedan ejecutar tareas o acceder a

los resultados generados. Para la implementación de este requerimiento se utilizó un esquema de autenticación basado en el intercambio de *Json Web Tokens (JWT)* (Jones, Bradley, y Sakimura, 2015) entre el cliente y el servidor, el cual funciona de la siguiente manera:

1. El usuario inicia sesión en el cliente introduciendo su nombre de usuario y contraseña.
2. Las credenciales del usuario son enviadas al método correspondiente expuesto mediante la interfaz *REST* en el microservicio *Autorizador*.
3. El microservicio *Autorizador* lee las credenciales del usuario, busca su registro en base de datos, corrobora que las credenciales sean correctas, y construye un *JWT* utilizando el algoritmo *HS256* (of Standards y Technology, 2015) para la codificación del mismo.
4. El *JWT* es retornado al cliente, de manera que cualquier solicitud HTTP enviada a cualquiera de los métodos expuestos en los microservicios que el cliente realice desde este punto en adelante puedan ser autenticadas.

La lógica del lado del servidor responsable de leer y decodificar el *JWT* para obtener el contexto del usuario en el que se ejecuta un método específico fue abstraída mediante el uso del patrón *Decorator* (4.2.1). Este patrón permite agregar anotaciones a los métodos expuestos en los controladores de los microservicios de manera tal que la lógica de autenticación presenta una implementación legible, centralizada y portable. En la sección 5.2.5 se muestra cómo se utiliza este patrón en un escenario dentro de la aplicación.

- Gestión de archivos: Se utilizó el patrón *Strategy* para gestionar las distintas estrategias de almacenamiento. Dependiendo en qué entorno la aplicación ejecute (desarrollo o producción), se decide en tiempo de ejecución qué estrategia utilizar para la gestión de archivos. En el ambiente de desarrollo se instancia una estrategia concreta que utiliza el sistema de archivos del sistema operativo anfitrión, mientras que en el despliegue en producción propuesto se instancia una estrategia concreta que utiliza soluciones de almacenamiento de archivos en la nube (véase la sección 5.3 para una explicación detallada del despliegue en un ambiente de producción). Esta capacidad está directamente alineada con el requerimiento funcional de gestión de archivos presentado en 3.1.1. Una explicación detallada de la utilización de este patrón en el contexto de gestión de archivos se provee en la sección 5.2.1. El proceso de subir un archivo a la aplicación consta de los siguientes pasos:

  1. El usuario especifica el archivo que desee subir en el cliente.
  2. Se comienza a enviar al archivo mediante solicitudes HTTP al microservicio *Repositorio*. Se destaca que el archivo se envía en paquetes de como máximo 100MB. Esto es especialmente importante para poder

proveer un mecanismo de subida de archivos eficiente para instancias donde se utilicen archivos muy grandes, como puede ser el caso con algunos *logs* de eventos. Cabe destacar que el envío de estos paquetes se realiza en forma secuencial, pero una optimización que sería interesante evaluar es la implementación de un mecanismo para el envío concurrente de dichos paquetes como se menciona en la sección de trabajo futuro 7.3.

3. El servidor almacena los paquetes que componen al archivo en un directorio temporal, para reensamblar el archivo más tarde.
4. Cuando el último paquete del archivo es enviado al microservicio *Repositorio* exitosamente, el cliente computa el código MD5 (Rivest, 1992) del archivo, y envía una orden al microservicio *Repositorio* para que proceda al ensamblaje de los paquetes para recrear el archivo original, y también le envía el código MD5 generado para que el servidor valide la integridad del archivo reensamblado.
5. El microservicio *Repositorio* ensambla el archivo, computa el código MD5 y valida la integridad del archivo ensamblado. En caso en que tal validación no sea exitosa, se retorna un error al cliente, y se borran los paquetes del archivo del sistema.

En el capítulo 6 se muestra un ejemplo más detallado de este proceso.

- Ejecución de tareas: Para facilitar la ejecución de algoritmos de minería de procesos, el diseño incluye la capacidad de seleccionar y configurar algoritmos específicos, los cuales son ejecutados dentro de microservicios aislados e independientes. Este enfoque modular asegura que se puedan ejecutar múltiples tareas simultáneamente, lo que responde al requerimiento de ejecución de tareas presentado en 3.1.1, proporcionando flexibilidad en la selección de técnicas y tecnologías. La creación y ejecución de una tarea consiste en los siguientes pasos:

  1. El usuario selecciona el algoritmo que desee ejecutar, y especifica las configuraciones y parámetros correspondientes. Este paso requiere que el archivo de entrada del algoritmo, ya sea un archivo de *logs* de eventos, o de un modelo de procesos de negocio retornado por la ejecución previa de otro algoritmo, estén previamente cargados en el sistema, como se menciona en el punto anterior. En el capítulo 6 se explica en detalle cómo se realiza el proceso en la aplicación.
  2. El cliente de la aplicación envía una solicitud HTTP al microservicio de tareas con la información detallada en el punto 1.
  3. El microservicio de tareas crea un registro en base de datos con la información detallada en el punto 1.
  4. El microservicio de tareas se comunica con el microservicio concreto que contiene a la librería que implementa el algoritmo seleccionado mediante una solicitud HTTP entre microservicios, siguiendo el

patrón de microservicios “Invocación de Procedimientos Remotos” (4.2.2), ordenándole así al microservicio concreto que inicie la ejecución del algoritmo dado con la configuración especificada por el usuario.

5. El microservicio concreto ensambla el comando necesario para la ejecución del algoritmo (véase 5.2.4 para una explicación detallada de este proceso) utilizando los parámetros recibidos por el microservicio de tareas.
6. El microservicio concreto ejecuta el algoritmo, y luego de finalizado sube los archivos resultantes al microservicio *Repositorio*, mediante una solicitud HTTP. Dichos archivos son almacenados bajo la colección de archivos que el usuario ya posee.
7. El microservicio concreto se comunica mediante una solicitud HTTP con el servicio *Transformador*, pidiéndole que genere una representación visual del modelo. Dicha representación visual típicamente se crea bajo el formato *PNG*.
8. El microservicio *Transformador* sube el archivo al repositorio de archivos del usuario mediante una solicitud HTTP al microservicio *Repositorio*.

Del lado del cliente, se implementó un sistema de sondeo a intervalos de 10 segundos que consulta con el microservicio *Tareas* por el estado de una tarea concreta. Cuando el proceso anteriormente descrito finaliza, el cliente actualiza la interfaz gráfica acorde a los resultados.

La figura 4.8 muestra un diagrama del proceso de creación y ejecución de una tarea, con todos los componentes relevantes y las interacciones entre ellos.

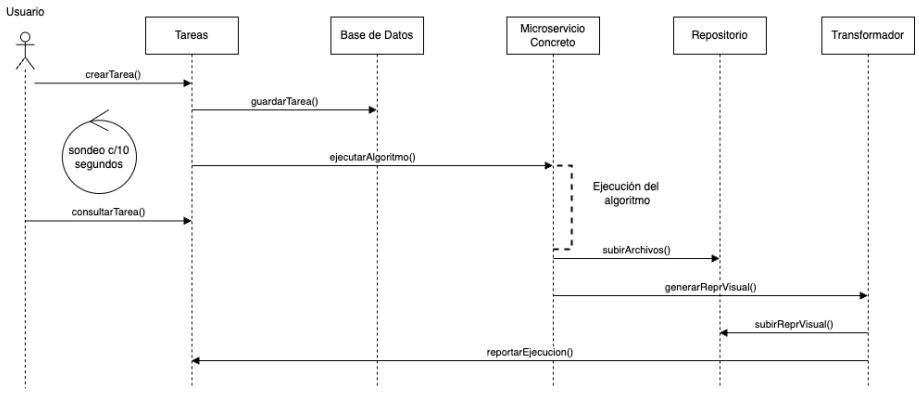


Figura 4.8: Flujo del proceso de creación y ejecución de una tarea

- Visualización y descarga de resultados: La arquitectura incluye una interfaz que permite al usuario visualizar los modelos generados por los

algoritmos y descargar los archivos resultantes, cumpliendo con el requisito de visualización y descarga de resultados presentado en 3.1.1. Esta funcionalidad es crucial para brindar a los usuarios un acceso directo y detallado a los datos procesados. La descarga de los archivos se realiza desde el microservicio *Repositorio*, mediante una solicitud HTTP que retorna el archivo, el cual luego es descargado en el sistema operativo del usuario mediante las facilidades que provee el navegador.

4.4.2. Requerimientos No Funcionales

- Arquitectura basada en microservicios: Al utilizar una arquitectura de microservicios, basada en el patrón Instancia de servicio por contenedor, se garantiza que el sistema pueda escalar horizontalmente de forma eficiente. Esto permite que el sistema maneje un aumento en la cantidad de usuarios y tareas sin comprometer el rendimiento, cumpliendo con el requerimiento de escalabilidad mencionado en la Sección 3.1.2.
- Extensibilidad: La elección de patrones de diseño como Inyección de Dependencias facilita la adición de nuevas funcionalidades sin afectar el sistema existente. También se hizo uso del patrón Chasis (*Chasis*, 2024) para agrupar y centralizar la lógica de uso común, de modo de facilitar la incorporación de nuevos servicios en la plataforma. La arquitectura permite integrar nuevas herramientas y algoritmos de minería de procesos con cambios mínimos, satisfaciendo así el requerimiento de extensibilidad presentado en 3.1.2.
- Interfaz intuitiva, segura y de alto rendimiento: Las decisiones sobre las librerías utilizadas para implementar la interfaz web de la plataforma se tomaron teniendo en cuenta el requerimiento no funcional descrito en 3.1.2. Los detalles técnicos de la implementación se encuentran en la sección 5.1.1. A continuación, se resumen los principales aspectos considerados para el diseño de la interfaz:

    1. Usabilidad: Se emplearon componentes que permiten crear una interfaz consistente y reutilizable, facilitando la navegación y el uso de la plataforma por parte de los usuarios. Además, el uso de etiquetas claras y mensajes de error específicos ayuda a los usuarios a comprender rápidamente cómo interactuar con la plataforma.
    2. Seguridad: Como se detalla en la sección 4.4.1, se utiliza un sistema de autenticación basado en *tokens* y una gestión robusta de sesiones, garantizando que solo usuarios autorizados puedan acceder a ciertas funcionalidades. Asimismo, los mensajes de error están diseñados para no revelar información sensible, contribuyendo a prevenir ataques de enumeración de usuarios.
    3. Rendimiento: Se seleccionaron librerías que optimizan las actualizaciones de la interfaz, mejorando el rendimiento general de la aplicación (ver 5.1.1 para detalles técnicos). Además, las llamadas a las

APIs de los microservicios se gestionan de manera asíncrona, permitiendo que la interfaz se mantenga receptiva mientras los datos se cargan en segundo plano.

- Internacionalización: La plataforma incorpora una funcionalidad que permite al usuario cambiar el idioma de la aplicación de manera sencilla. Para ello, se emplean archivos *JSON* que almacenan las traducciones de las cadenas de texto en diferentes idiomas. Estos archivos contienen claves asociadas a sus respectivas traducciones, lo que facilita la adaptación de la interfaz de usuario a distintos idiomas. Se ha implementado un mecanismo dinámico que permite al usuario seleccionar el idioma deseado entre las opciones disponibles, aplicando el cambio en tiempo real. Este enfoque, junto con las decisiones de diseño adoptadas, garantiza el cumplimiento del requerimiento de internacionalización descrito en 3.1.2.

Capítulo 5

Desarrollo del prototipo

En este capítulo se describe, en términos técnicos, como se implementó la solución presentada en el capítulo 4. Se mencionan las tecnologías utilizadas para los componentes de la arquitectura, cómo se implementaron los patrones de diseño mencionados anteriormente y las decisiones técnicas más relevantes que se tomaron para el diseño de la solución.

5.1. Tecnologías utilizadas

Como se observa en la figura 4.1, la plataforma está conformada por varios componentes siguiendo una arquitectura de microservicios. Este tipo de arquitectura brinda más flexibilidad a la hora de seleccionar la tecnología de cada componente. No es necesario que todos los componentes estén implementados en el mismo lenguaje, lo que permite adaptar la tecnología según las necesidades y características de cada componente. A continuación se presentan las tecnologías utilizadas para cada componente de la plataforma.

5.1.1. Interfaz gráfica

La interfaz web (capa de *UI*) de la plataforma está implementada como una aplicación (*React*, 2024), que es una librería de *Javascript* ampliamente utilizada para el desarrollo de aplicación web del estilo *SPA (Single Page Application)*. Se optó por esta tecnología por ser una de las más populares y recomendadas para el desarrollo web.

Entre los beneficios que se tuvieron en cuenta para esta elección, se encuentran los siguientes:

- Componentes reutilizables: *React* permite crear componentes que son independientes y reutilizables. Esto posibilita construir bloques modulares de código que se pueden reutilizar en diferentes partes de la aplicación, lo que reduce la duplicación de código y facilita el mantenimiento.

- Virtual DOM: *React* utiliza un *Virtual DOM (Document Object Model)* que mejora el rendimiento. En lugar de actualizar el DOM real directamente, *React* crea una representación en memoria del *DOM* y solo actualiza los componentes que han cambiado, lo que hace que las actualizaciones de la interfaz de usuario sean rápidas y eficientes.
- Flujo de datos unidireccional: *React* promueve un flujo de datos unidireccional, lo que significa que los datos fluyen en una sola dirección, desde los componentes padres hacia los hijos. Esto simplifica el seguimiento de los cambios y hace que la aplicación sea más predecible y fácil de depurar.
- React Hooks: Los *Hooks* permiten la reutilización del estado y otras características de *React* sin escribir grandes porciones de código.
- Amplia comunidad y ecosistema: *React* tiene una comunidad muy grande y activa, lo que significa que hay una gran cantidad de recursos, librerías, y herramientas disponibles para facilitar el desarrollo.
- Facilidad de Integración: *React* se puede integrar fácilmente con otras librerías o *frameworks*.

Junto con la aplicación de *React* se usó (*Typescript*, 2024), que es una extensión de *Javascript* que añade el uso de tipado estático en la aplicación de *frontend*. Esto mejora la mantenibilidad y robustez del código, ayudando a detectar errores en tiempo de desarrollo y generando un código que es más fácil de entender y mantener.

5.1.2. Componentes de gestión

Estos son los componentes que se observan en la parte central de la figura 4.1: Repositorio, Tareas y Autorizador. Dichos componentes contienen la lógica de negocio encargada de la gestión de los usuarios y su autenticación, tareas y archivos manejados por la plataforma.

Para la implementación de estos componentes se utiliza el lenguaje de programación (*Python*, 2024) mediante la aplicación del *framework* (*Flask*, 2024), que permite desarrollar aplicaciones web en este lenguaje. Al igual que como se mencionó en el punto anterior, se optó por un lenguaje ampliamente utilizado y recomendado para el desarrollo web. Todos los microservicios mencionados en este grupo están implementados como una aplicación *Flask* independiente.
Dentro de los beneficios y puntos que se consideraron para fundamentar esta elección de tecnologías se tienen los siguientes:

- Simplicidad y legibilidad: *Python* se destaca por ser un lenguaje que produce un código fácil de leer y mantener, ideal para equipos de desarrollo.
- Framework ligero: *Flask* es un *framework* ligero que facilita la creación de microservicios de manera rápida y eficientes, con un conjunto de dependencias mínimo que permita a la aplicación ejecutar.

- Desarrollo: *Python* permite un desarrollo ágil con muchas librerías disponibles para usar.
- Ecosistema: Amplia gama de librerías para integración con bases de datos, autenticación, y más.
- Comunidades activas: estas tecnologías cuentan con un amplio soporte y documentación disponible.

5.1.3. Componentes de minería de procesos

Estos son los componentes que se pueden observar a la derecha de la figura 4.1. Cada uno de ellos expone las distintas librerías de minería de procesos para un lenguaje específico. Estos componentes exponen dichas librerías mediante una *API Rest* implementada en *Flask*, con la cual se pueden invocar los distintos algoritmos con sus respectivos parámetros:

- Componente Python (microservicio `ms-python`): provee algoritmos de la librería (Berti y cols., 2023). En la implementación actual expone los siguientes algoritmos:
  - Descubrimiento:
    - *Alpha Miner* (W. M. P. van der Aalst, Weijters, y Maruster, 2004)
    - *Inductive Miner* (Leemans, Fahland, y van der Aalst, 2013)
    - *Heuristics Miner* (Weijters, van der Aalst, y de Medeiros, 2006)
    - *Directly-Follows Graph* (W. van der Aalst, 2016)
  - Chequeo de conformidad:
    - *Diagnostics Alignments* (Adriansyah, van Dongen, y van der Aalst, 2011)
- Componente (*Java*, 2024) (microservicio `ms-java`): provee algoritmos de la librería *StructuredMiner*. En la implementación actual expone los algoritmos *Fodina Miner* (vanden Broucke, Weerdt, Vanthienen, y Baesens, 2017) y *Heuristics Miner* (Weijters y cols., 2006), ambos de Descubrimiento.
- Componente (*R*, 2024) (microservicio `ms-r`): provee algoritmos de la librería *BupaR*. En la implementación actual expone únicamente el algoritmo de extensión de modelos *Process Map* (W. van der Aalst, 2016)

5.2. Patrones de diseño

En esta sección se presenta, desde un enfoque técnico, como se aplicaron e implementaron los patrones de diseño de la sección 4.2.

5.2.1. Implementación del patrón Strategy

Como se mencionó en la sección 4.2.1 el patrón *Strategy* se aplica en la solución para manejar las diferentes opciones de almacenamiento que se tienen disponibles para la gestión de los archivos en la plataforma.

La implementación de este patrón se encuentra dentro del microservicio Repositorio. Siguiendo los componentes del patrón como se presentaron en 4.2.1, se tiene por un lado la implementación de la estrategia base en la clase `BaseStorageStrategy`. Esta clase representa la interfaz común que deben seguir todas las estrategias concretas que se utilizan. En el contexto de esta plataforma, cada una de las estrategias concretas representa una estrategia de almacenamiento para los archivos utilizados en la plataforma.

Las estrategias concretas que se implementan son las siguientes:

- Sistema de archivos: la clase `FsStorageStrategy` implementa la estrategia para utilizar el sistema de archivos local como medio de almacenamiento.
- Servicio S3 de AWS: la clase `S3StorageStrategy` implementa la estrategia que hace uso del servicio *S3* de *Amazon (AWS)* para el almacenamiento de los archivos.

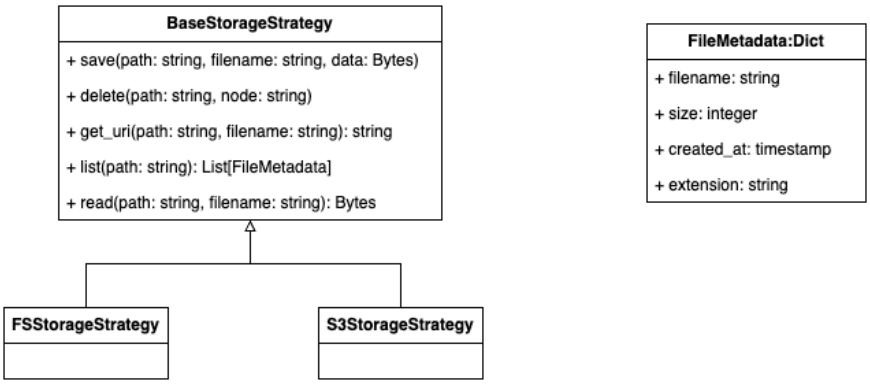


Figura 5.1: Diagrama del patrón Strategy

El contexto en el que se utiliza la estrategia se encuentra en la función `file_managemenmt_deps_factory` dentro del archivo `controllers/helpers.py` del microservicio *Repositorio*. La selección se realiza en tiempo de ejecución dependiendo del entorno (desarrollo, o producción, utilizando la variable de entorno `PYTHON_ENV`).

Para el ambiente de desarrollo se utiliza la estrategia de almacenamiento en el sistema de archivos y para el ambiente de producción se utiliza el servicio *S3* de *AWS*. El patrón *Strategy* en el contexto de esta plataforma permite que el

sistema sea flexible y extensible, ya que se pueden agregar nuevas estrategias de almacenamiento de archivos sin modificar el código que utiliza estas estrategias, y permite fácilmente extender la aplicación para soportar otras soluciones de almacenamiento de archivos *(e.g: Google Cloud Storage, Azure Blob Storage, etc)*.

```
file_storage_strategy = (
    FsStorageStrategy()
    if os.getenv("PYTHON_ENV") is None or os.getenv("
        PYTHON_ENV") == "development"
    else S3StorageStrategy()
)

def file_managemenmt_deps_factory():
    @dataclass(frozen=True, slots=True)
    class _Dependencies:
        storage_strategy: BaseStorageStrategyProto =
            file_storage_strategy

    return _Dependencies()
```

Listing 5.2.1: Ejemplo de cómo se instancia la estrategia para el almacenamiento de archivos

5.2.2. Implementación del patrón Repository

El patrón (*Repository*, 2024) se utiliza para abstraer la lógica de acceso a datos y proporcionar una interfaz más limpia y desacoplada para interactuar con la base de datos. El uso de este patrón permite que la lógica de negocio no dependa de los detalles de la persistencia, facilitando así el mantenimiento y las pruebas.

La interfaz repositorio se implementa en la clase `BaseRepository`. Esta clase es una implementación genérica que proporciona métodos comunes para interactuar con la base de datos. Este repositorio es abstracto y puede ser utilizado para cualquier entidad de la plataforma.

Por otro lado se tienen clases que contienen implementaciones concretas del patrón *Repository*. La clase `UsersRepository` extiende la clase `BaseRepository` para manejar la persistencia de la entidad `User` (que representa los usuarios de la plataforma). De la misma forma, se tiene una clase `TasksRepository` que gestiona la persistencia de la entidad `Task` (relativa a las tareas que se definen y ejecutan en la plataforma). Por último se tiene la clase `TokensBlacklistRepository` que se utiliza para gestionar la lista negra de *tokens* de autenticación en la aplicación. Esto es útil para invalidar *tokens* de autenticación, por ejemplo, cuando un usuario cierra sesión, una sesión expira, o cuando se detecta un *token* comprometido.

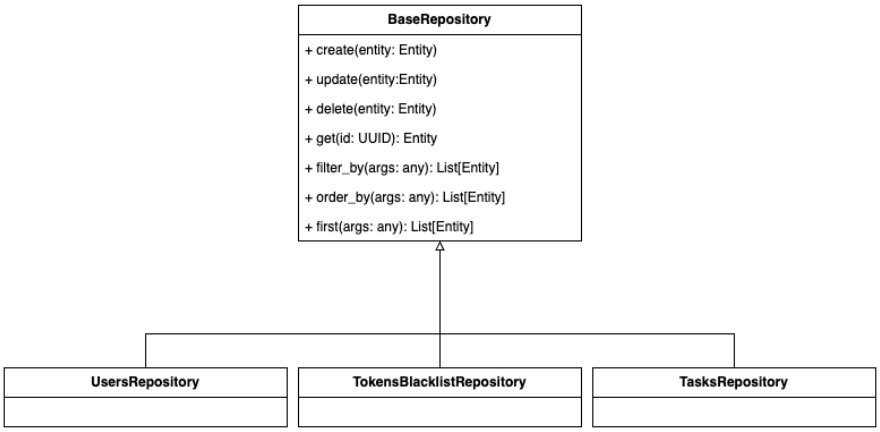


Figura 5.2: Diagrama del patrón Repository

Un ejemplo de uso de los repositorios se puede encontrar en la función `execute_task.py`, demostrada en el Listing 5.2.2. En este contexto se utiliza el repositorio `UsersRepository` para obtener el usuario asociado a la tarea a ejecutar y el repositorio `TasksRepository` para actualizar el estado de una tarea después de su ejecución.

La aplicación del patrón *Repository* en la solución proporciona una capa de abstracción sobre la lógica de acceso a datos, permitiendo que la lógica de negocio interactúe con los datos de manera más limpia y desacoplada. Esto facilita el mantenimiento del código y mejora la capacidad de realizar pruebas unitarias, ya que los repositorios pueden ser fácilmente simulados o reemplazados en un entorno de prueba.

5.2.3. Implementación del patrón Inyección de Dependencias

El patrón de (*Inyección de Dependencias*, 2024) se utiliza para proporcionar las dependencias necesarias para llevar a cabo las distintas tareas a las funciones que implementan los distintos flujos del sistema. Estas dependencias pueden ser los repositorios para el manejo de datos, las estrategias de almacenamiento, variables de entorno, etc.

Para la definición de las dependencias se utilizan objectos que encapsulan las dependencias, y estos se pasan como argumentos a las funciones del sistema. Los funciones reciben estas dependencias encapsuladas al momento de ser invocadas, y las utilizan para realizar sus operaciones específicas. Esto permite que las dependencias sean inyectadas desde fuera y en el momento de su uso.

Un ejemplo del uso de este patrón se puede observar en la función `sign_in_user`, demostrada en el Listing 5.2.3. Junto con la función se define un objecto con

```python
def execute_task(deps: Dependencies):
    def runner(task: Task) -> Task:
        service_url = _map_service_url(task.service)

        user = deps.users_repository.get(task.user_id)

        if user is None:
            raise ValueError(f"User not found: {task.user_id}")

        response = requests.post(
            f"{service_url}/algorithms/{task.algorithm}/{task.id}/execute",
            headers={
                "Service-Authorization": os.getenv("SERVICE_SECRET") or "",
                "Content-Type": "application/json",
                "Accept": "application/json",
            },
            json={
                "parameters": task.parameters,
                "user_id": str(user.id),
                "task_id": str(task.id),
            },
        )

        if not response.ok:
            raise Exception(
                f"Failed to execute task {task.id}: {response.status_code} {response.text}"
            )

        task.started_at = datetime.now()
        task.ended_at = None
        task.pid = response.json()["pid"]
        task.raw_command = response.json()["raw_command"]
        task.state = State.running
        deps.tasks_repository.update(task)

        return task

    return runner
```

Listing 5.2.2: Ejemplo de la utilización del patrón Repository

las dependencias necesarias para la ejecución.

```python
@final
@dataclass(frozen=True, slots=True)
class Dependencies(object):
    password_hash_checker: Callable[[str, str], bool]
    users_repository: UsersRepository
    token_generator: Callable[..., str]


def sign_in_user(dependencies: Dependencies):
    def runner(username: str, password: str) -> tuple[User,
        str]:
        user = dependencies.users_repository.filter_by(
            username=username).first()
        if not user:
            raise InvalidAuth("Invalid username")
        if not dependencies.password_hash_checker(user.
            password, password):
            raise InvalidAuth("Invalid password")

        token = dependencies.token_generator(
            user_id=str(user.id),
            name=user.name,
            username=user.username,
            minutes_to_expire=SESSION_DURATION,
        )

        return (user, token)

    return runner
```

Listing 5.2.3: Ejemplo de la utilización del patrón Inyección de Dependencias

En este ejemplo se pueden observar los siguientes componentes:

- La clase `Dependencies`, que define las propiedades del objeto que contiene las dependencias que la función requiere.
- La función `sign_in_user`, que retorna una *función*, la cual es instanciada ya tomando en su contexto las dependencias especificadas.
- La función `runner`, la cual ejecuta el caso de uso (en este ejemplo, realizar el inicio de sesión de un usuario), en cuyo contexto de ejecución se tiene acceso a las dependencias.

Este enfoque permite una implementación desacoplada de los casos de uso en el sistema, donde flujos mas complejos se pueden implementar mediante

composición de funciones, estando cada función instanciada en el contexto de las dependencias que requiere.

5.2.4. Implementación del patrón Command

Como se mencionó en 4.2.1, el patrón *Command* (Gamma y cols., 1994, Chapter 5) se utiliza en la solución para ejecutar los algoritmos de minería de procesos que provee la plataforma. Se tiene por un lado la definición de los comandos a ejecutar que se encuentran en los archivos `algorithms.json`, ubicados dentro de la carpeta de cada microservicio que importa las librerías de minería de procesos que le corresponden. Cada uno de estos archivos *JSON* actúa como una configuración que define los algoritmos disponibles en el microservicio, sus parámetros, y cómo deben ser invocados. Cada algoritmo tiene un comando asociado que se ejecuta mediante una llamada al sistema. El comando está parametrizado para que pueda ser invocado con los valores específicos seleccionados por el usuario en la plataforma para cada parámetro disponible. En el Listing 5.2.4 se muestra parte de un archivo *JSON* que se utiliza como parte de este patrón:

En ella se puede observar que cada algoritmo tiene un campo `executable` que especifica el comando a ejecutar, con marcadores de posición para los parámetros.

Algunas de las librerías de algoritmos que se utilizaron no proveen una interfaz por línea de comandos para invocar los algoritmos. Para estos casos el problema se solucionó creando un *script* con permisos de ejecución que funcione como dicha interfaz, que luego la aplicación puede consumir mediante una llamada al sistema. En el Listing 5.2.5 se puede ver un fragmento del *script* `pm4pycli.py` que es responsable de exponer los algoritmos expuestos por la librería *PM4PY* como métodos invocables por línea de comando. Este *script* se encarga de interpretar configuraciones y parámetros correspondientes, y ejecutar el comando solicitado, actuando como el invocador en el patrón *Command*.

En el *script* `pm4pycli.py` (Listing 5.2.5) se utiliza la librería `argparse` para analizar los argumentos de la línea de comandos, que corresponden a los parámetros definidos en los archivos *JSON* que contienen los algoritmos. El *script* mapea los argumentos analizados al método correspondiente implementado en la librería *PM4PY*. Para ello utiliza un diccionario (llamado `ALGORITHMS`) para asociar las claves que identifican cada algoritmo con sus respectivas funciones de ejecución. Luego los resultandos son almacenados en un archivo de salida utilizando la función de escritura correspondiente, que depende de los tipos de salida que el algoritmo en cuestión maneje.

5.2.5. Implementación del patrón Decorator

En la solución propuesta se utiliza el patrón *Decorator* (Gamma y cols., 1994, Chapter 3) para manejar aspectos transversales a todos los microservicios de la aplicación como la autenticación y la validación de parámetros en las rutas de los controladores de los microservicios.

```
{
    "algorithms": [
        {
            "name": "Alpha Miner - Petri Net",
            "key": "alpha",
            "category": "discovery",
            "description": {
                "es": "Descubre una Red de Petri utilizando el
                    algoritmo Alpha Miner",
                "en": "Discover a Petri Net using the Alpha Miner
                    algorithm"
            },
            "link": "https://pm4py.fit.fraunhofer.de/static/assets
                /api/2.7.8/generated/pm4py.discovery.
                discover_petri_net_alpha.html#pm4py-discovery-
                discover-petri-net-alpha",
            "executable": "$python_path $scripts_path/pm4py-cli.py
                 --algorithm alpha-miner-petri $input_file
                $output_file $activity_key $timestamp_key
                $case_id_key",
            "library": "pm4py",
            "outputExtension": "pnml",
            "parameters": {
            "input_file": {
                "name": {
                    "es": "Archivo de entrada",
                    "en": "Input file"
                },
                "description": {
                    "es": "Archivo de logs de entrada",
                    "en": "Input log file"
                },
                "type": "File",
                "required": true,
                "flag": "--input"
            },
            "output_file": {
              "name": {
                "es": "Archivo de salida",
                "en": "Output file"
              },
              "description": {
                "es": "Archivo del modelo generado",
                "en": "Generated model file"
              },
              "type": "String",
              "required": true,
              "flag": "--output"
            }
        }
    ]
}
```

Listing 5.2.4: Ejemplo de `algorithms.json`

```python
from collections.abc import Iterable
import pm4py
import argparse
import matplotlib.pyplot as plt
from pandas.plotting import table

ALGORITHMS = {
    "alpha-miner-petri": {
        "algo_runner": pm4py.discovery.
            discover_petri_net_alpha,
        "model_writer": pm4py.write.write_pnml,
    },
    "inductive-miner-petri": {
        "algo_runner": pm4py.discovery.
            discover_petri_net_inductive,
        "model_writer": pm4py.write.write_pnml,
    },
    "heuristics-miner-petri": {
        "algo_runner": pm4py.discovery.
            discover_petri_net_heuristics,
        "model_writer": pm4py.write.write_pnml,
    },
    "dfg": {
        "algo_runner": pm4py.discovery.discover_dfg,
        "model_writer": pm4py.write.write_dfg,
    },
    "conformance_diagnostics_alignments": {
        "algo_runner": pm4py.
            conformance_diagnostics_alignments,
        "model_reader": pm4py.read_pnml,
        "model_writer": pm4py.save_vis_alignments,
    },
}

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "--algorithm",
        dest="algorithm",
        help="Algorithm to use",
        type=str,
        choices=ALGORITHMS.keys(),
        required=True,
    )
    parser.add_argument(
        "--input", dest="input_file", help="Input file",
            type=str, required=True
    )
    # ... otros argumentos ...
```

Listing 5.2.5: Ejemplo de script `pm4pycli.py`

A continuación se mencionan los decoradores aplicados en la solución:

- Decoradores de autenticación: Se utilizan para asegurar que ciertas rutas solo sean accesibles para usuarios autenticados. Esto se logra mediante la verificación de *tokens* de autenticación antes de permitir el acceso a la función de la ruta.

- Decoradores de validación de parámetros: Se utilizan para validar que los parámetros requeridos estén presentes en las solicitudes HTTP antes de que se ejecute la lógica principal de la función.

- Otros decoradores provistos por librerías de terceros, tales como los provistos por el *framework web Flask* (*Flask*, 2024), el cual se utilizó para exponer los métodos de la aplicación en una interfaz *REST*.

Dentro del archivo `tareas/controllers/tasks_controller.py`, que es parte del microservicio "Tareas"(4.1), se puede observar la aplicación de los decoradores mencionados anteriormente. Un ejemplo de esto es la función `create_task`, mostrada en el Listing 5.2.7, que se utiliza para crear tareas en la plataforma.

El decorador `@params_required` se encarga de validar que los parámetros requeridos por la función estén presentes en la solicitud *JSON* antes de que se ejecute la lógica principal.

Por otro lado se tiene el decorador `@auth_required` que asegura que la función `create_task` solo se ejecute si el usuario está autenticado correctamente en la plataforma. La implementación del mismo se muestra en el Listing 5.2.6

Como se puede observar en la implementación, este decorador se encarga de obtener el *token* de autenticación presente en el cabezal de la solicitud, chequear que exista, que sea válido, y decodificarlo para obtener la información del usuario autenticado.

Por último, se puede observar el decorador `@tasks_controller.route`, provisto por el *framework Flask*, que se encarga de exponer el método `create_task` en una interfaz *REST*.

La aplicación de este patrón ofrece varios beneficios entre los que se encuentran:

- Mantenimiento: Al encapsular la lógica de autenticación y validación de parámetros en decoradores, se evita la repetición de código en múltiples lugares, facilitando el mantenimiento y las actualizaciones de seguridad.

- Claridad del Código: Las funciones que requieren autenticación no necesitan incluir lógica de autenticación explícita, lo que las hace más legibles y enfocadas en su propósito principal. De la misma forma, las funciones pueden centrarse en su lógica principal sin preocuparse por la validación de parámetros, lo que mejora la claridad y la organización del código.

```python
def auth_required(f):
@functools.wraps(f)
def decorated(*args, **kwargs):
    raw_token = request.headers.get("Authorization")

    if not raw_token:
        return make_response(jsonify({"message": "
            Unauthorized"}), 401)

    jwt_token = raw_token.split(" ")[1]

    try:
        token_is_blacklisted = (
            tokens_blacklist_repository.get(jwt_token) is
                not None
        )

        if token_is_blacklisted:
            raise jwt.exceptions.ExpiredSignatureError

        data = jwt.decode(
            jwt_token, cast(str, os.getenv("SECRET_KEY")),
                algorithms=["HS256"]
        )
        current_user = users_repository.filter_by(username=
            data["username"]).first()
        if current_user is None:
            raise jwt.exceptions.ExpiredSignatureError
    except jwt.exceptions.ExpiredSignatureError:
        response = make_response(jsonify({"message": "
            Session expired"}), 401)
        return response

    return f(current_user, *args, **kwargs)

return decorated
```

Listing 5.2.6: Implementación del decorador `auth_required`, utilizado para forzar autenticación en los controladores

```python
@tasks_controller.route("/", methods=["POST"])
@params_required(json=["taskName", "service", "algorithm", "execute", "parameters"])
@auth_required
def create_task(current_user, json: dict[str, str]):
    task_name = json.pop("taskName")
    service = json.pop("service")
    algorithm = json.pop("algorithm")
    execute_task = json.pop("execute", False)

    task = create_task_interactor(create_task_deps)(
        user=current_user,
        task_name=task_name,
        service=service,
        algorithm=algorithm,
        parameters=cast(AlgoParameters, json),
    )

    if execute_task:
        task = execute_task_interactor(execute_task_deps)(task)

    return make_response(jsonify(serialize(task=task)), 201)
```

Listing 5.2.7: Ejemplo del uso de decoradores en la función `create_task`

5.2.6. Implementación del patrón Invocación de Procedimientos Remotos

Como se mencionó en la sección 4.2.2, todas las comunicaciones entre servicios se realizan de manera síncrona mediante solicitudes HTTP. Esto implica que, cuando un servicio envía una solicitud HTTP a otro, el flujo de ejecución se detiene hasta recibir la respuesta correspondiente. Este comportamiento puede observarse en el Listing 5.2.8, que muestra la implementación de la función `execute_task` en el archivo `tareas/interactors/execute_task.py`. En dicha función, se realiza una solicitud HTTP *POST* para ejecutar un algoritmo en el contexto de una tarea dentro de la plataforma. Al tratarse de una comunicación síncrona, el flujo de ejecución queda bloqueado mientras se espera la respuesta del servicio remoto.

Las respuestas de los microservicios se gestionan de manera que el servicio solicitante pueda actualizar su estado o manejar errores según corresponda. Esto se refleja en cómo se actualiza el estado de una tarea tras recibir la respuesta del servicio que ejecuta el algoritmo.

Cada microservicio expone sus funcionalidades a través de rutas HTTP, las cuales son invocadas por otros servicios para ejecutar procedimientos remotos. Por ejemplo, el Listing 5.2.9 muestra algunas de las rutas definidas en el mi-

croservicio `Tareas`, utilizadas para gestionar las tareas creadas dentro de la plataforma. Estas rutas fueron diseñadas siguiendo los principios *REST*, lo cual se aplica de manera uniforme en todos los microservicios de la arquitectura. Este enfoque garantiza una comunicación estandarizada, facilitando la interoperabilidad y el mantenimiento del sistema.

5.2.7. Implementación del patrón Chasis

El patrón (*Chasis*, 2024) se implementó en la solución mediante el desarrollo y adopción de librerías compartidas y prácticas consistentes en todos los microservicios. Se definió un conjunto de funcionalidades y componentes comunes que son utilizados por varios servicios para garantizar la consistencia y reducir la duplicación de código.

El directorio `lib/pgradodeps` contiene las librerías y funcionalidades compartidas que son utilizadas por múltiples microservicios. Esto incluye clases comunes, decoradores y otras utilidades que aseguran una funcionalidad uniforme en toda la plataforma. Por ejemplo, en `decorators/controller.py`, se encuentran decoradores que son referenciados y utilizados por diversos microservicios para implementar funcionalidades comunes, como la gestión de autenticación y el control de parámetros.

Además, se implementó un mecanismo de registro de *logs* consistente que permite a los microservicios capturar y gestionar los registros generados durante las ejecuciones. Esta implementación se encuentra en el directorio `logger` y asegura un manejo unificado de los *logs* en toda la arquitectura.

El uso de (*Flask Blueprints*, 2024) es otro ejemplo de cómo se aplicó el patrón Chasis. En el contexto de esta solución, los *blueprints* de *Flask* se emplean para definir y registrar rutas de manera consistente en los diferentes microservicios. Esto proporciona una estructura común y reutilizable para la gestión de las rutas HTTP, facilitando la interoperabilidad y el mantenimiento en toda la plataforma.

5.2.8. Implementación del patrón Configuración Externalizada

Se implementó el patrón de (*Configuración Externalizada*, 2024) al separar la configuración de la aplicación del código fuente, lo que permite gestionarla de manera independiente y modificarla fácilmente sin necesidad de realizar despliegues adicionales.

Para almacenar las configuraciones, se utilizan archivos `.env`, en los que se definen variables de entorno que son requeridas y utilizadas por los servicios. Estas configuraciones incluyen credenciales de bases de datos, definición del tipo de entorno (desarrollo, pruebas, producción), credenciales de servicios externos y rutas utilizadas por la aplicación, entre otros parámetros clave.

Estas variables de entorno se referencian dentro del código para configurar aspectos de la aplicación como las *URLs* de los servicios, claves secretas y otras dependencias críticas. En el Listing 5.2.10 se puede observar un ejemplo de uso

```python
def execute_task(deps: Dependencies):
    def runner(task: Task) -> Task:
        service_url = _map_service_url(task.service)

        user = deps.users_repository.get(task.user_id)

        if user is None:
            raise ValueError(f"User not found: {task.user_id}")

        response = requests.post(
            f"{service_url}/algorithms/{task.algorithm}/{task.id}/execute",
            headers={
                "Service-Authorization": os.getenv("SERVICE_SECRET") or "",
                "Content-Type": "application/json",
                "Accept": "application/json",
            },
            json={
                "parameters": task.parameters,
                "user_id": str(user.id),
                "task_id": str(task.id),
            },
        )

        if not response.ok:
            raise Exception(
                f"Failed to execute task {task.id}: {response.status_code} {response.text}"
            )

        task.started_at = datetime.now()
        task.ended_at = None
        task.pid = response.json()["pid"]
        task.raw_command = response.json()["raw_command"]
        task.state = State.running
        deps.tasks_repository.update(task)

        return task

    return runner
```

Listing 5.2.8: Ejemplo de una solicitud HTTP POST en la función `execute_task`

```python
@tasks_controller.route("/", methods=["GET"])
@auth_required
def lists_tasks(current_user):
    tasks = list_tasks_interactor(list_tasks_deps)(user_id=
        current_user.id)
    return make_response(jsonify(tasks), 200)


@tasks_controller.route("/<string:task_id>/execute", methods
    =["POST"])
@auth_required
def execute_task(_current_user, task_id):
    task = TasksRepository().get(task_id)

    if task is None:
        return make_response(jsonify({"errors": ["Task not
            found"]}), 404)

    task = execute_task_interactor(execute_task_deps)(task)

    return make_response(jsonify(serialize(task=task)), 200)


@tasks_controller.route("/<string:task_id>", methods=["
    DELETE"])
@auth_required
def delete_task(current_user, task_id):
    task = TasksRepository().get(task_id)

    if task is None:
        return make_response(jsonify({"errors": ["Task not
            found"]}), 404)

    TasksRepository().delete(task)

    return make_response(jsonify({}), 200)


@tasks_controller.route("/<string:task_id>", methods=["GET"
    ])
@auth_required
def show_task(_current_user, task_id: str):
    task = TasksRepository().get(task_id)
    serialized_task = serialize(task) if task is not None
        else ""
    return make_response(jsonify(serialized_task), 200)
```

Listing 5.2.9: Ejemplo de rutas HTTP definidas en el archivo `tareas/controllers/tasks_controller.py`

de variables de entorno para obtener la *URL* de un servicio. Se hace uso de `os.getenv` para acceder a los valores de las diferentes variables de entorno, lo que permite que las configuraciones sean modificables sin alterar el código fuente, promoviendo flexibilidad y adaptabilidad.

```
match service:
    case "ms-java":
        service_url = os.getenv("JAVA_MS_URL")
    case "ms-python":
        service_url = os.getenv("PYTHON_MS_URL")
    case "ms-r":
        service_url = os.getenv("R_MS_URL")
    case _:
        raise TypeError(f"Invalid service name: {service}")
```

Listing 5.2.10: Ejemplo de uso de variables de entorno para obtener la *URL* de un microservicio. Código extraído de la función `get_service_algos`

Adicionalmente, se definen variables de entorno específicas en el archivo `docker-compose.yml`, lo que facilita que cada servicio cuente con su propia configuración personalizada. Esto resulta especialmente útil para especificar *URLs* de servicios, credenciales y otros parámetros necesarios para el correcto funcionamiento de los microservicios dentro de los entornos definidos.

5.3. Despliegue en producción

En esta sección se explica como se realizó el despliegue en producción de la plataforma. Se utilizó el servicio en la nube *Amazon Web Services* (*Amazon Web Services*, 2024) para la creación del ambiente y todos los recursos necesarios por la plataforma.

En el diagrama 5.3 se puede observar como está organizado el entorno y los servicios utilizados. Se optó por este esquema de despliegue por una cuestión de costos, ya que la solución de *AWS* para desplegar contenedores (*AWS Fargate*) no tiene una versión gratuita. Sin embargo el ambiente de desarrollo si utiliza contenedores para los microservicios, y existen soluciones en la nube (por lo general pagas) capaces de desplegar contenedores de *Docker*.

El despliegue en la nube cuenta con los siguientes recursos de *AWS*:

- Elastic Load Balancer (ELB): cumple la función de *proxy* inverso, el cual permite que la interfaz gráfica se comunique con un solo punto, y luego se agregan condiciones para redirigir las solicitudes HTTP al microservicio correspondiente, en función del *path* de la solicitud.
- Amazon EC2 (*primary* y *secondary*): instancias de máquinas virtuales de AWS donde se montan los microservicios. Como ya se mencionó, dado que se utilizó la versión gratuita de los servicios que *AWS* provee, las

capacidades de cómputo de una sola máquina virtual resultaron ser insuficientes para soportar todos los microservicios, por tanto los mismos fueron distribuidos en dos instancias de máquina virtual distintas

- Amazon RDS: sistema para el manejo de la base de datos *Postgres*.
- Amazon S3: solución para el almacenamiento de archivos. Se utilizó para guardar los archivos relacionados con la ejecución de algoritmos por un usuario dado, tales como los registros de eventos que reciben los algoritmos, archivos de modelo producidos, y sus representaciones visuales. Cabe destacar también que la interfaz gráfica se provee mediante este sistema, el cual puede funcionar como una red de distribución de contenido (o *CDN* por sus siglas en inglés) para servir los archivos *HTML*, *CSS* y *Javascript* que componen a la interfaz gráfica del sistema.
- AWS Secrets Manager: repositorio para almacenar variables de entorno y otras configuraciones que luego son leídas desde las instancias EC2 al momento de inicializar los microservicios que contienen.

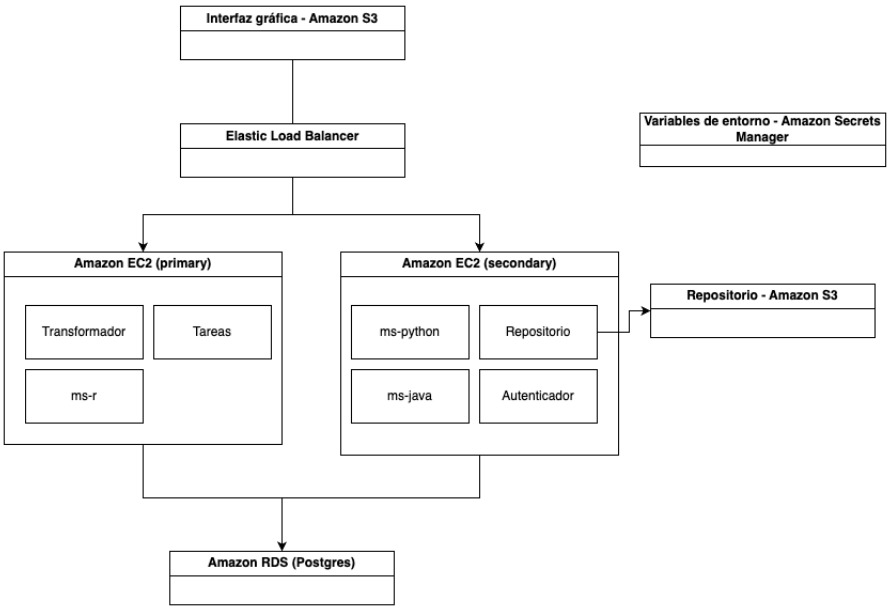


Figura 5.3: Diagrama de despliegue en AWS

A su vez se proveen los siguientes scripts para el manejo de los recursos ya mencionados en la nube:

- `scripts/aws/deploy`: Despliega los microservicios en las instancias de *AWS*.

- `scripts/aws/deploy_ui`: Compila y despliega el código para la interfaz gráfica utilizando *Amazon S3*.
- `scripts/aws/infra`: Gestiona los recursos de infraestructura de *AWS* utilizada por la aplicación.
- `scripts/aws/logs`: Obtiene los *logs* del sistema de los distintos microservicios.
- `scripts/aws/secret`: Gestiona el repositorio de variables de entorno de la aplicación.
- `scripts/aws/ssh`: Conecta a una instancia *Ec2* de *AWS* utilizando *SSH*. Útil para instalaciones de dependencias y otras librerías del sistema.

5.4. Extensibilidad

Uno de los requerimientos más importantes a la hora del diseño de esta aplicación fue su extensibilidad. Se consideró de especial importancia el diseño de una arquitectura que permitiera, con relativa facilidad, exponer nuevos algoritmos para ser usados en la plataforma, ya sean implementaciones en algunas de las tecnologías que la aplicación ya maneja, o implementaciones en tecnologías de momento no soportadas por la aplicación.

En este sentido, la arquitectura de microservicios provee una solución que permite disponer de servicios independientes, cada uno con las librerías y dependencias requeridas para la ejecución de un conjunto de algoritmos implementados en una tecnología dada.

Cada uno de los tres microservicios mencionados carga los algoritmos disponibles para su tecnología a través de un archivo de configuración *JSON*. Este archivo especifica el formato de los parámetros que recibe cada algoritmo. Véase el Listing 5.4.1 a modo de ejemplo de como está estructurado este archivo

Se destacan los siguientes campos:

- `key`: Identificador único para el algoritmo
- `executable`: Comando a ejecutar para invocar al algoritmo. Utiliza variables con el formato `$variable` que luego se interpolarán con los parámetros requeridos.
- `outputExtension`: Extensión de los archivos de salida generados por el algoritmo.
- `parameters`: Lista de parámetros que recibe el algoritmo, incluyendo tipo de dato, descripción, rango de valores que puede tomar (si aplica), etc.

Es importante que el nombre de un parámetro (por ejemplo, `input_file`) coincida exactamente con las variables utilizadas en `executable`. Esto es necesario porque, al ensamblar el comando de invocación del algoritmo, se reemplazan

```
{
  "algorithms": [
    {
      "name": "Process Map",
      "key": "process_map",
      "category": "enhancement",
      "description": {
        "en": "Allows to visualize the process' performance.",
        "es": "Permite visualizar el rendimiento del proceso."
      },
      "link": "https://bupaverse.github.io/docs/performance_maps.
          html",
      "executable": "Rscript $scripts_path/process-map.R
          $input_file $output_file $performance_function
          $frequency_type",
      "library": "bupar",
      "outputExtension": "svg",
      "parameters": {
        "input_file": {
          "name": {
            "en": "Input file",
            "es": "Archivo de entrada"
          },
          "description": {
            "en": "Input Logs file",
            "es": "Archivo de logs de entrada"
          },
          "type": "File",
          "required": true,
          "flag": "--input"
        },
        "output_file": {
          "name": {
            "en": "Output file",
            "es": "Archivo de salida"
          },
          "description": {
            "en": "Output image file",
            "es": "Archivo de la imagen generada"
          },
          "type": "String",
          "required": true,
          "flag": "--output"
        }
      }
    }
  ]
}
```

Listing 5.4.1: Ejemplo del archivo de configuración `algorithms.json` para el microservicio en R

las variables `$variable` por los valores de los parámetros indicados en el *JSON* de entrada. En este ejemplo concreto, se remplazará la variable `$input_file` por la ruta del archivo de entrada especificado.

5.4.1. Cómo exponer un nuevo algoritmo en el sistema

Para agregar un nuevo algoritmo, consideremos los siguientes escenarios:

1. El algoritmo ya existe en una librería utilizada por alguno de los microservicios (e.g: `ms-python`)
    - Se deben agregar los parámetros del algoritmo al archivo de configuración `algorithms.json` del microservicio correspondiente.
2. El algoritmo no existe en las librerías utilizadas, pero está implementado en un lenguaje ya soportado. En este escenario, se deben agregar los parámetros del nuevo algoritmo al archivo `algorithms.json` del microservicio correspondiente, y de ser necesario proveer una interfaz por línea de comando para el mismo, la cual se puede implementar como un *script* con permisos de ejecución que sea expuesto en el directorio especificado en la variable de entorno `SCRIPTS_PATH`, o bien exponiendo el *script* en algún directorio que se encuentre en la variable `PATH` del sistema. Esto se debe a que, como se mencionó en 5.2.4, la implementación propuesta requiere que los algoritmos sean ejecutables por interfaz de línea de comando. Véase los archivos `ms-python/pm4py-cli.py` o `ms-r/process-map.R` a modo de ejemplo.
3. El algoritmo está implementado en un lenguaje no soportado por los microservicios existentes. En este caso se debe crear un nuevo microservicio que exponga el algoritmo deseado, y montar dicho microservicio en el contexto de la aplicación. Esto se realiza de la siguiente manera:
    - Crear un nuevo microservicio para el lenguaje correspondiente, lo cual consiste en implementar los siguientes archivos:
        - `app.py`: Lógica para inicializar el servidor web que expone la interfaz REST del microservicio.
        - `requirements.txt`: Dependencias del microservicio.
        - `algorithms.json`: Declaración del nuevo algoritmo.
        - `Dockerfile`: Define la imagen del contenedor del microservicio.
    - Extender `tareas/app.py` (véase listings 5.4.2 y 5.4.3) para que en el momento en que se inicializa el microservicio de tareas, pueda cargar los algoritmos provistos por el nuevo microservicio. Hay que asegurarse que se agrega una nueva variable de entorno con la *URL* correspondiente al nuevo microservicio, y agregar la correspondiente cláusula al mapeo descrito en el código presentado.

```
for service in ["ms-java", "ms-python", "ms-r"]: # Agregar el nombre del nuevo microservicio, puede ser cualquier identificador que no colisione con alguno de los ya existentes
    service = cast(ServiceIdentity, service)
    services_algos_store[service] = get_service_algos(service)

    ...
```

Listing 5.4.2: Ejemplo de cómo extender el arreglo de nombres de servicios en `tareas/app.py` para agregar un nuevo microservicio

```
def get_service_algos(service: ServiceIdentity) -> AlgorithmsStore:
    match service:
        case "ms-java":
            service_url = os.getenv("JAVA_MS_URL")
        case "ms-python":
            service_url = os.getenv("PYTHON_MS_URL")
        case "ms-r":
            service_url = os.getenv("R_MS_URL")
        """
        A modo de ejemplo, si se agregase un nuevo microservicio con el identificador 'ms-javascript':
        case 'ms-javascript':
            service_url = os.getenv('JAVASCRIPT_MS_URL')
        """
        case _:
            raise TypeError(f"Invalid service name: {service}")
    ...
```

Listing 5.4.3: Ejemplo de cómo extender la función `get_service_algos` en `tareas/app.py` para cargar los algoritmos expuestos por el nuevo microservicio

Capítulo 6

Aplicación en un caso de estudio

En este capítulo, se presenta una guía de uso de la plataforma, mostrando paso a paso cómo cargar un archivo de eventos, seleccionar y configurar algoritmos de minería de procesos, y visualizar los resultados obtenidos. Se utiliza el archivo de eventos *DomesticDeclaration.xes* del sitio de *BPI Challenge* (*BPI Challenge*, 2024) como ejemplo, con el propósito de explorar las funcionalidades de la plataforma en una situación práctica.

6.1. Descripción del sistema

En esta sección se describen brevemente las principales funcionalidades del sistema para dar un mayor entendimiento del mismo.

Para utilizar el sistema, es necesario que el usuario se registre, creando un nombre de usuario y una contraseña que le permitirá acceder a todas las funcionalidades. Tras iniciar sesión, el usuario podrá empezar a utilizar las distintas herramientas ofrecidas por la plataforma.

El primer paso consiste en cargar un archivo de *logs* al sistema, lo cual se realiza mediante un botón de carga de archivos. Esta acción permite al usuario no solo agregar nuevos archivos, sino también listar los archivos previamente subidos y, si lo desea, eliminarlos.

Una vez que el archivo ha sido cargado con éxito, se puede proceder a crear una nueva ejecución en el sistema. Este proceso se estructura en cuatro pasos:

1. Nombre de la ejecución: cada ejecución creada en el sistema debe tener un nombre asociado.

2. Seleccionar algoritmo: el usuario podrá elegir entre los distintos algoritmos disponibles, clasificados por la tecnología subyacente (*Java*, *Python* o *R*) y por su rol en la minería de procesos (*Discovery*, *Conformance Checking* o *Enhancement*).

3. Configurar algoritmo: tras seleccionar el algoritmo, es necesario ajustar sus parámetros. Dependiendo del algoritmo elegido, se deberán especificar ciertos detalles, como el archivo de entrada a procesar y el nombre del archivo de salida.

4. Confirmar: en este último paso, se presenta un resumen de toda la configuración de la ejecución por crear y se permiten dos opciones: crearla y ejecutarla en el mismo momento, o crear la ejecución para poder ser ejecutada en un momento posterior.

Una vez que la ejecución ha sido creada, quedará visible en el panel de ejecuciones. En este panel, el usuario podrá visualizar todas las ejecuciones que ha creado, con detalles como el nombre, la fecha de inicio y finalización, y el estado de cada una. El sistema permite filtrar las ejecuciones por nombre o por las diferentes etapas de la minería de procesos. Al seleccionar una ejecución, se mostrará en el panel central la imagen del modelo generado por el algoritmo seleccionado, la cual se puede descargar en formato *PNG*. Adicionalmente, se podrán consultar detalles adicionales sobre la ejecución y descargar tanto el archivo de entrada como el archivo de salida.

6.2. Descripción del log de eventos

El *log* de eventos utilizado en este caso de estudio proviene de los datos recopilados en el *BPI Challenge 2020* (*BPI Challenge*, 2024), que se centra en el proceso de reembolso de gastos de viaje en la Universidad Tecnológica de Eindhoven (TU/e). El conjunto de datos incluye información de los años 2017 y 2018, con datos organizados en diferentes tipos de solicitudes y permisos de viaje.

El *log* de eventos está compuesto por varios archivos que documentan distintos aspectos del proceso de reembolso, incluyendo:

- Solicitudes de pago: Registra gastos que no están relacionados con viajes, como costos de representación o la compra de *hardware* para el trabajo.

- Declaraciones nacionales: Registra los reembolsos solicitados por viajes dentro del país, que no requieren de un permiso previo.

- Costos de viaje prepagos: Incluye los costos que fueron pagados antes del viaje, tales como pasajes de avión o costos de inscripción a conferencias.

- Declaraciones internacionales: Registra los reembolsos solicitados por viajes internacionales, que requieren la aprobación de un supervisor antes de ser realizados.

- Permisos de viaje: Incluye todos los eventos relacionados con las solicitudes de permisos para viajar, los costos prepagos y las declaraciones correspondientes.

El log de eventos está anonimizado para proteger la identidad de los empleados, reemplazando los identificadores internos con nuevos identificadores generados específicamente para este conjunto de datos. Además, los montos económicos registrados no son exactos, pero los totales sumados en muestras suficientemente grandes deberían aproximarse a los valores reales.

6.2.1. Estructura del log de eventos

El *log* de eventos está dividido en cinco archivos principales, cada uno representando un aspecto específico del proceso:

1. **RequestForPayment.xes:** Contiene 6.886 casos y 36.796 eventos relacionados con solicitudes de pago que no deberían estar vinculadas a viajes.

2. **DomesticDeclarations.xes:** Contiene 10.500 casos y 56.437 eventos relacionados con declaraciones de viajes nacionales.

3. **PrepaidTravelCost.xes:** Contiene 2.099 casos y 18.246 eventos relacionados con costos de viaje prepagos.

4. **InternationalDeclarations.xes:** Contiene 6.449 casos y 72.151 eventos relacionados con declaraciones de viajes internacionales.

5. **PermitLog.xes:** Contiene 7.065 casos y 86.581 eventos relacionados con permisos de viaje y eventos asociados a costos prepagados y declaraciones de viaje.

Cada archivo documenta las acciones realizadas por los empleados o el sistema en el proceso de reembolso, desde la solicitud inicial hasta la aprobación y el eventual pago (si corresponde).

6.2.2. Flujo del proceso

El flujo de proceso registrado en el *log* de eventos varía ligeramente según el tipo de solicitud, pero en general sigue una secuencia similar que se presenta a continuación:

1. Presentación de la solicitud: El empleado presenta una solicitud de reembolso o permiso de viaje.

2. Aprobación por la administración de viajes: La solicitud es enviada para su aprobación a la administración de viajes de la TU/e.

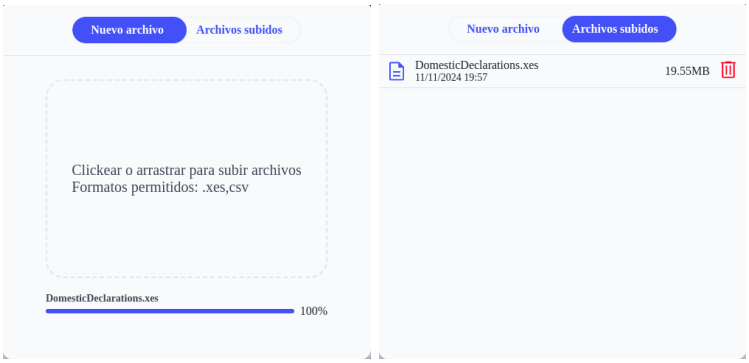
3. Revisión por el responsable del presupuesto y el supervisor: Si la solicitud es aprobada por la administración de viajes, se envía al responsable del presupuesto y luego al supervisor. Si ambas funciones las ejerce la misma persona, se omite un paso. En algunos casos, también se requiere la aprobación del director.

4. Resultado de la aprobación: Si la solicitud es rechazada en algún paso, el empleado puede revisarla y reenviarla o rechazarla también. Si la solicitud es aprobada, se procede con el reembolso o, en el caso de permisos de viaje, se permite la realización del viaje.

5. Reembolsos y declaraciones posteriores: Después de la aprobación y la realización del viaje, el empleado puede presentar solicitudes adicionales para reembolsar los costos incurridos durante el viaje.

6.3. Carga de archivos de eventos

Para comenzar, el usuario debe cargar el archivo de eventos que se analizará. La figura 6.1 muestra la interfaz de la plataforma dedicada a la gestión de archivos. La ventana de carga de archivos se presenta en la figura 6.1a. En este ejemplo, el usuario selecciona el archivo *DomesticDeclaration.xes* desde el sistema de archivos local para subirlo a la plataforma, quedando así disponible para su procesamiento. La figura 6.1b muestra la lista de archivos ya cargados en el sistema, desde donde también es posible eliminarlos si es necesario.



(a) Nuevo archivo (b) Lista de archivos

Figura 6.1: Gestión de archivos del sistema.

6.4. Aplicación de algoritmos de Descubrimiento

En esta sección se muestra cómo utilizar algunos de los algoritmos de descubrimiento integrados en el prototipo desarrollado. Se explica el proceso de selección y configuración de los distintos algoritmos en la plataforma, seguido de su ejecución y visualización de los resultados obtenidos.

6.4.1. Descubrimiento con librerías Java

La plataforma ofrece dos algoritmos de descubrimiento de la librería *BPMN Miner*: *Fodina Miner* (vanden Broucke y cols., 2017) y *Heuristics Miner* (Weijters y cols., 2006). En este ejemplo, se detalla el proceso de selección y configuración del algoritmo *Heuristics Miner*, aplicado al *log* de eventos previamente cargado en el sistema.

Para iniciar, en el panel de ejecuciones se crea una nueva ejecución, lo que despliega un asistente para guiar el proceso. El primer paso consiste en asignar un nombre a la ejecución, como se muestra en la figura 6.2.

Figura 6.2: Nombre de la ejecución.

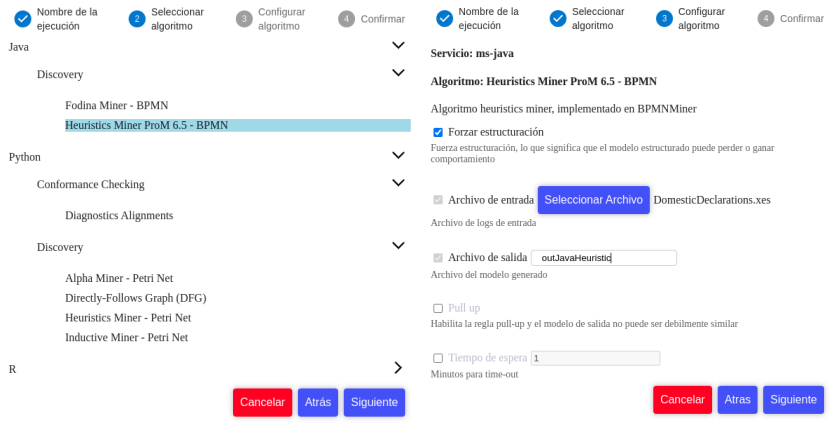
En la figura 6.3a se ilustra el paso de selección del algoritmo, en este caso, eligiendo *Heuristic Miner*. La plataforma permite configurar parámetros específicos, proporcionando flexibilidad para adaptar la ejecución a las necesidades del análisis. En esta ejecución se optó la opción de forzar una estructura en el modelo para una mayor claridad, como se observa en la figura 6.3b.

Finalmente, el último paso del asistente presenta un resumen de la ejecución que se va a crear, tal como se muestra en la figura 6.4.

Cuando el usuario inicia la ejecución del algoritmo de descubrimiento, el progreso se muestra en tiempo real, brindando una experiencia interactiva. Al finalizar la ejecución, la plataforma genera un resumen detallado que incluye el tiempo de procesamiento y ofrece la posibilidad de descargar tanto el archivo de eventos utilizado como el modelo generado, que en este caso se encuentra en formato *BPMN*. Además, se presenta una representación gráfica del modelo, destacando las transiciones principales y las actividades más frecuentes del proceso. En este ejemplo, como se muestra en la figura 6.5, el algoritmo completó la ejecución en 15 segundos.

6.4.2. Descubrimiento con librerías Python

El sistema también ofrece la posibilidad de ejecutar algoritmos de descubrimiento proporcionados por la librería *PM4PY*, ampliando la capacidad analítica



(a) Selección del algoritmo. (b) Configuración del algoritmo.

Figura 6.3: Selección y configuración del algoritmo.

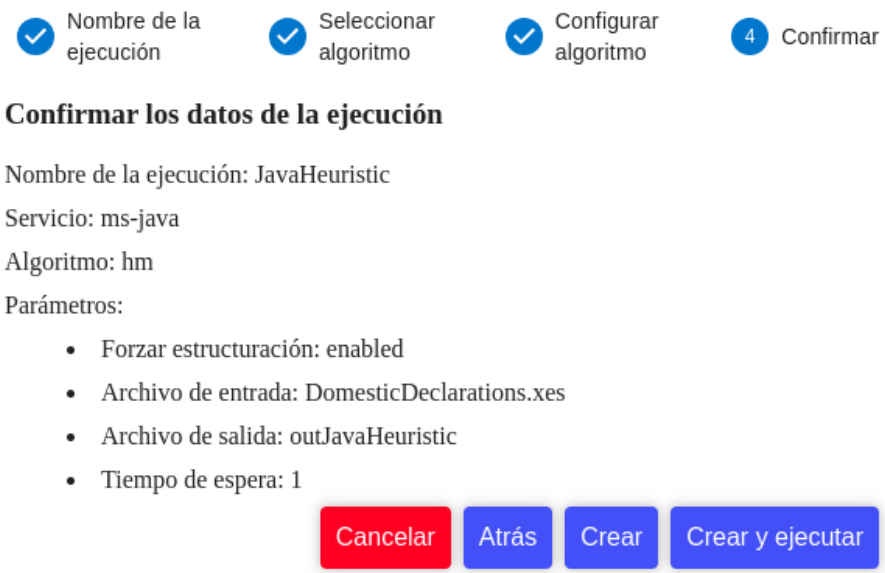


Figura 6.4: Confirmar ejecución.

de la plataforma. En este contexto, los usuarios pueden seleccionar entre los algoritmos *Alpha Miner* (W. M. P. van der Aalst y cols., 2004), *Heuristic Miner* (Weijters y cols., 2006) e *Inductive Miner* (Leemans y cols., 2013), que generan modelos en formato de Red de Petri. Adicionalmente, se incluye el algoritmo *Directly Follows Graph (DFG)*, que utiliza una notación basada en grafos de secuencias directas para representar los procesos. Esta integración de tecnologías

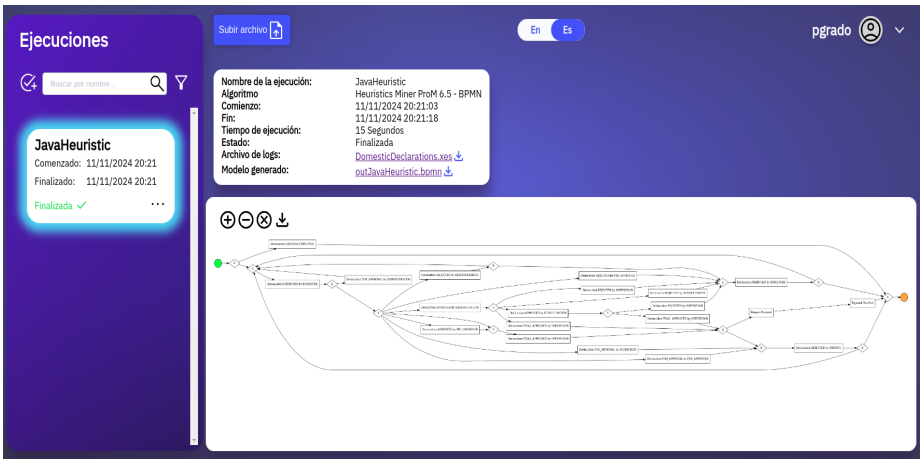


Figura 6.5: Vista de una ejecución finalizada.

permite ajustar el análisis según las características específicas de cada algoritmo y las necesidades del usuario.

El flujo para crear ejecuciones con estos algoritmos sigue un esquema similar al descrito anteriormente. La diferencia está en la configuración de los parámetros específicos para cada algoritmo. Por ejemplo, el *Inductive Miner* (Leemans y cols., 2013) permite ajustar varios parámetros, tal como se muestra en la figura 6.6. Entre sus opciones configurables, se encuentran la selección de los atributos del *log* que servirán como clave de actividad, identificador de caso y clave de tiempo. También es posible definir un umbral de ruido para filtrar comportamientos poco frecuentes, habilitar el uso de multiprocesamiento para optimizar el rendimiento y deshabilitar la regla de *fallthroughs*, logrando así modelos más refinados y específicos.

En este caso, como se muestra en la figura 6.7, la ejecución del algoritmo aplicado al *log* de eventos *DomesticDeclarations.xes* tuvo una duración de 24 segundos. El modelo resultante, denominado *InductiveModel.pnml*, fue generado en formato de Red de Petri y está listo para ser utilizado en análisis posteriores. Por ejemplo, este modelo puede seleccionarse para realizar un análisis de conformidad.

6.5. Aplicación de algoritmos de Verificación de Conformidad

Es importante poder evaluar la conformidad entre los eventos registrados en el *log* de eventos y el modelo de proceso descubierto por el algoritmo. La plataforma permite evaluar la conformidad utilizando el algoritmo *Diagnostic Alignments* de librería *PM4PY*. Este método de conformidad permite identificar desviaciones específicas entre el comportamiento registrado y el modelo

Figura 6.6: Configuración para Inductive Miner.

descubierto, proporcionando información sobre cuántos casos siguen el modelo y cuántos presentan discrepancias.

Para utilizar este algoritmo, el usuario debe disponer de un modelo de pro-

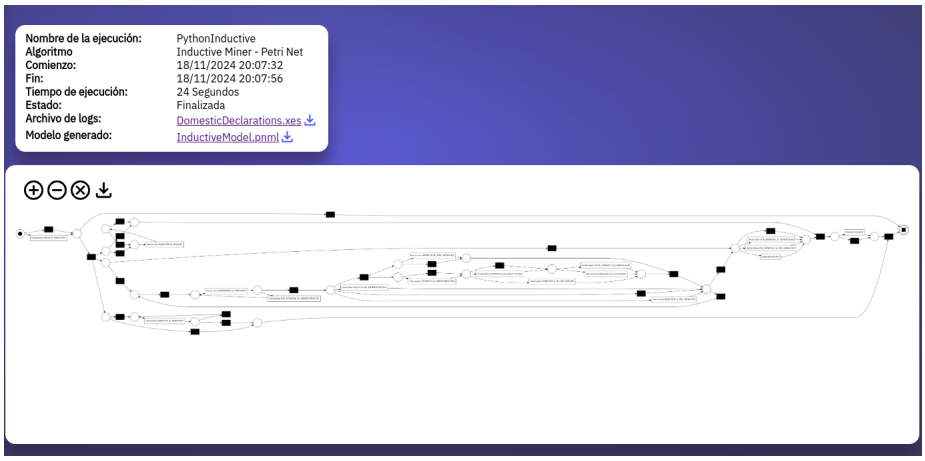


Figura 6.7: Visualización del modelo InductiveModel.pnml.

ceso en formato de Red de Petri generado previamente. Una vez creado, dicho modelo queda disponible en el sistema para su selección en el algoritmo *Diagnostic Alignments*.

En este ejemplo, retomando el modelo *InductiveModel.pnml* generado previamente, se analiza su conformidad con el *log* de eventos *DomesticDeclarations.xes*. El procedimiento inicia con la creación de una nueva ejecución en la plataforma, seleccionando el algoritmo *Diagnostic Alignments*. Como se muestra en la figura 6.8, la configuración del algoritmo requiere cargar dos archivos de entrada: el *log* de eventos y el modelo de proceso en formato de Red de Petri. Adicionalmente, la plataforma permite activar una opción para incluir diagnósticos detallados en los resultados, modificando el formato de la salida generada.

Si la opción de incluir diagnósticos en el resultado no está habilitada, la salida generada es una imagen que ilustra las alineaciones entre el modelo de proceso y las trazas del *log* de eventos. La imagen presenta dos columnas principales:

- En la primera columna, se enumeran las variantes del *log* de eventos, que corresponden a secuencias únicas de actividades observadas en las trazas. Cada variante incluye el número de ocurrencias correspondientes en el *log*.

- En la segunda columna, se visualiza la alineación de cada variante con el modelo de proceso. Los pasos que coinciden correctamente entre el modelo y el *log* se resaltan en verde, mientras que los desajustes (o errores de alineación) se marcan en rojo.

En la figura 6.9, se presentan los primeros resultados obtenidos al ejecutar el algoritmo.

Por otro lado, si se habilita la opción de incluir diagnósticos, el resultado será una tabla que muestra, para cada caso del *log* de eventos, su nivel de ajuste con respecto al modelo. La tabla incluye las siguientes columnas:

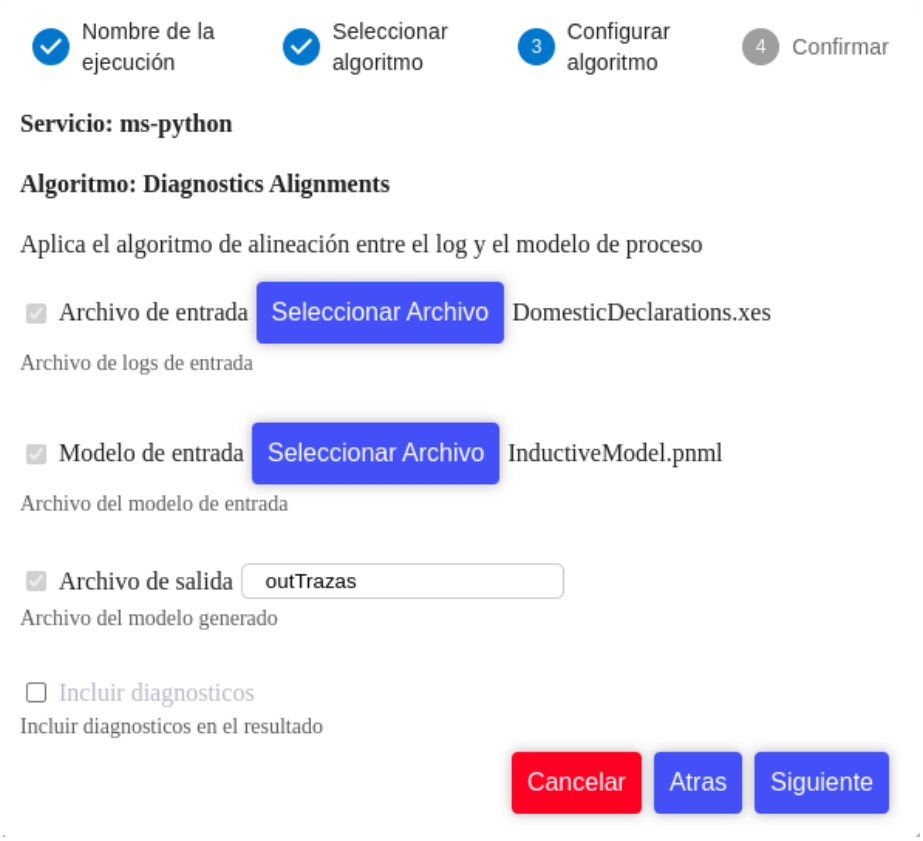


Figura 6.8: Configuración del algoritmo Diagnostic Alignments.

Figura 6.9: Vista parcial de las alineaciones entre el modelo de proceso y las trazas del log de eventos.

- Case ID: identifica de manera única cada traza en el *log* de eventos.
- Cost: Indica el número total de desajustes entre la traza y el modelo, ya sea por actividades del modelo que no ocurren en el *log* (movimientos en el modelo) o actividades del *log* que no están en el modelo (movimientos en el *log*).

- Fitness: Representa el grado de ajuste de la traza al modelo, con valores entre 0 y 1, donde 1 indica un ajuste perfecto.
- Is Fit: Es un indicador booleano que señala si la traza está completamente alineada con el modelo (true) o si presenta algún desajuste (false).

Un ejemplo de este formato de salida puede verse en la figura 6.10.

	case_id	cost	fitness	is_fit
0	declaration 86791	15	1.0	True
1	declaration 86795	14	1.0	True
2	declaration 86800	14	1.0	True
3	declaration 86731	15	1.0	True
4	declaration 86735	15	1.0	True
5	declaration 86805	15	1.0	True
6	declaration 86809	24	1.0	True
7	declaration 86816	15	1.0	True
8	declaration 86716	15	1.0	True
9	declaration 86720	27	1.0	True
10	declaration 86820	15	1.0	True
11	declaration 86824	15	1.0	True
12	declaration 86828	15	1.0	True
13	declaration 86739	24	1.0	True
14	declaration 86746	15	1.0	True
15	declaration 86750	1	1.0	True
16	declaration 86752	15	1.0	True
17	declaration 86756	15	1.0	True
18	declaration 86832	15	1.0	True
19	declaration 86836	15	1.0	True

Figura 6.10: Tabla de diagnósticos.

Ambos enfoques permiten obtener información sobre el comportamiento del proceso, ayudando a identificar posibles problemas y áreas de mejora.

6.6. Aplicación de algoritmos de Extensión de Modelos

La última funcionalidad presentada por la plataforma es la aplicación del algoritmo de extensión *Process map* de la librería *BupaR*. Este algoritmo ofrece una representación gráfica del flujo de trabajo y las relaciones entre las actividades dentro del proceso, permitiendo entender cómo se desarrollan los casos a lo largo del tiempo. El resultado es un mapa de procesos que ilustra cómo las actividades se conectan entre sí, cuántas veces se repiten y la frecuencia con la que suceden, proporcionando una visión clara del comportamiento general del

proceso. Una de las principales ventajas de esta técnica es su capacidad para identificar cuellos de botella y puntos de decisión dentro del flujo del proceso, lo que facilita la optimización y mejora del rendimiento.

La exposición de este algoritmo en la plataforma permite generar mapas de rendimiento, donde se visualiza la eficiencia del proceso mediante una función de agregación aplicada a los tiempos de procesamiento. En este caso particular, se muestran los resultados de aplicar el algoritmo *Process map* sobre el *log* de eventos *DomesticDeclarations.xes*.

En la figura 6.11 se muestra la configuración del algoritmo para obtener como resultado un mapa de rendimiento. En este caso será necesario completar el parámetro «Función de agregación» especificando la función de agregación que se desea utilizar. En este caso se optó por utilizar la media, de modo que los resultados presentan el tiempo promedio que cada actividad requiere para completarse, así como el tiempo que transcurre entre actividades.

Figura 6.11: Configuración para mapa de rendimiento promedio.

La figura 6.12 muestra el mapa de rendimiento obtenido al aplicar el algoritmo *Process map* al mismo *log* de eventos.

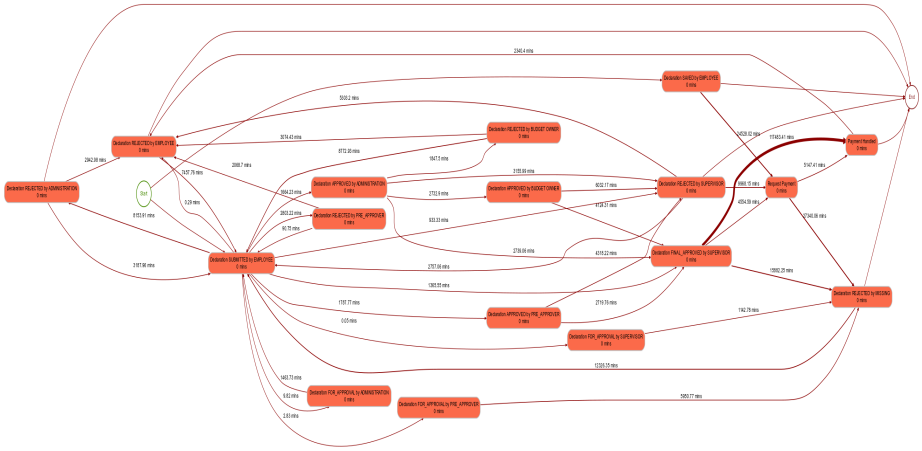


Figura 6.12: Mapa de rendimiento promedio.

6.7. Discusión y evaluación final

La plataforma desarrollada ha permitido demostrar su funcionalidad y flexibilidad mediante la ejecución de un caso de estudio basado en el *log* de eventos *DomesticDeclarations.xes*. A continuación, se realiza una discusión final en base a la experiencia obtenida, destacando los aspectos positivos y señalando posibles áreas de mejora.

Fortalezas de la plataforma

- Integración de múltiples algoritmos: La plataforma soporta algoritmos de descubrimiento, análisis de conformidad y extensión de modelos, tanto de tecnologías basadas en *Java* como *Python* o como *R*, lo que amplía las posibilidades de análisis al permitir el uso de herramientas como *PM4PY*, *BPMN Miner* y *BupaR*. Esta flexibilidad es un claro punto fuerte, ya que adapta la solución a distintos contextos y necesidades.

- Interfaz intuitiva y configuración personalizable: La interfaz gráfica facilita la creación y configuración de ejecuciones. La posibilidad de ajustar parámetros específicos de los algoritmos, como umbrales de ruido, tiempos de espera o la habilitación de reglas avanzadas, proporciona un alto grado de control al usuario.

- Gestión de archivos centralizada: La funcionalidad de gestión de archivos centralizada es eficiente y simplifica el proceso de carga, eliminación y selección de *logs* y modelos, lo que reduce errores operativos.

Áreas de mejora

- Visualización de resultados: Si bien las visualizaciones generadas ofrecen una representación clara y accesible, se podría mejorar los resultados integrando a la plataforma distintas herramientas de visualización que faciliten aún más el análisis y la toma de decisiones.
- Validación de parámetros de configuración: Algunos usuarios pueden experimentar dificultades al configurar parámetros avanzados de los algoritmos. Una validación en tiempo real de los valores ingresados, junto con sugerencias automáticas, podría mejorar la experiencia y reducir errores.
- Soporte para más formatos de exportación: Si bien los resultados se presentan en formatos estándar como *BPMN* y *PNML*, ampliar las opciones de exportación podría ser útil, agregando conversores de formatos para pasar de uno a otro de manera sencilla dentro de la plataforma.

Capítulo 7

Conclusiones y Trabajo Futuro

En este capítulo se realiza una evaluación integral de los resultados obtenidos a lo largo del proyecto, analizando el grado de cumplimiento de los objetivos planteados inicialmente y reflexionando sobre los desafíos enfrentados durante su desarrollo. Asimismo, se destacan los principales aportes logrados, se identifican las limitaciones encontradas y se proponen posibles líneas de trabajo futuro para extender y consolidar el proyecto.

7.1. Conclusiones

En este proyecto se logró desarrollar una plataforma de microservicios que brinda la capacidad de ejecutar los diversos algoritmos relacionados a las distintas técnicas que comúnmente se utilizan en la Minería de Procesos, demostrando su utilidad y aplicabilidad mediante un caso de estudio real (6). A lo largo del trabajo, se cumplieron los objetivos propuestos en (1.2):

1. Analizar conceptos y arquitecturas de microservicios, *BPM*, minería de procesos y datos: durante el desarrollo de la aplicación, se realizó un análisis exhaustivo de diversos patrones de arquitectura de microservicios, así como un estudio detallado de las implementaciones existentes de los algoritmos de minería de procesos y las tecnologías asociadas a estas. Como resultado de este objetivo, se elaboró un informe sobre el estado del arte (2), que presenta los fundamentos principales relacionados con los procesos de negocio, la minería de procesos y las arquitecturas de microservicios.

2. Seleccionar herramientas y definir una plataforma de microservicios que exponga microservicios de minería de procesos y datos considerando elementos clave de *Quality of service (QoS)*, y carga y evaluación de calidad de datos: se seleccionó un conjunto de herramientas que, a nuestro juicio, son ampliamente utilizadas en el ámbito académico y que cuentan con

implementaciones en diversas tecnologías. Esto permitió demostrar la viabilidad de integrar dichas herramientas en una única plataforma cohesiva. Sin embargo, al momento de definir la arquitectura de la plataforma, no se tomaron en consideración aspectos relacionados a la calidad del servicio. En la sección 7.3 mencionamos posibles incorporaciones para cumplir con este objetivo.

3. Implementación del prototipo de la plataforma de microservicios e interfaz gráfica: como parte del desarrollo de este proyecto, se implementó un prototipo funcional de la plataforma de microservicios, cumpliendo con todos los requerimientos funcionales y no funcionales descritos en la sección 3.1. Este prototipo integra las librerías de minería de procesos seleccionadas, presentadas en la sección 5.1.3. Además, se desarrolló una interfaz gráfica intuitiva que permite a los usuarios acceder a estas herramientas directamente desde la web. La plataforma cumple con el objetivo planteado de facilitar el acceso a las implementaciones de los algoritmos de minería de procesos seleccionados. Esto reduce la complejidad técnica que implica la utilización de estas herramientas, logrando un impacto significativo en su utilización.

4. Aplicación de la propuesta en un caso de estudio con procesos y datos reales: la utilidad y funcionalidad del prototipo desarrollado se ilustró mediante su aplicación en un caso de uso real, descrito en detalle en el capítulo 6. Para este caso de estudio, se utilizó un archivo de eventos real obtenido del sitio *BPI Challenge* (*BPI Challenge*, 2024). En él, se presentaron los principales flujos que la plataforma permite ejecutar, destacando el uso de las librerías y algoritmos integrados en el prototipo. El caso de estudio incluye una descripción detallada del proceso, que abarca desde la carga de archivos hasta la selección y aplicación de los algoritmos disponibles, así como la visualización de los resultados obtenidos. Este enfoque paso a paso demuestra la capacidad de la plataforma para facilitar el análisis y procesamiento de eventos reales de manera eficiente y accesible.

7.2. Desafíos encontrados

Uno de los primeros y principales desafíos encontrados fue cómo exponer las distintas librerías bajo una interfaz unificada. El problema radica en que algunas librerías, como *BPMN Miner* (ICPM, 2024), ofrecen su implementación mediante un archivo *jar*, lo que permite su invocación directa a través de la línea de comandos. En contraste, otras implementaciones, como *PM4PY* (Berti y cols., 2023) o *BupaR* (Janssenswillen y cols., 2018), son librerías de código que deben integrarse y ejecutarse dentro del contexto de un programa.

Para abordar este problema, se decidió que todos los algoritmos expuestos en la plataforma estuvieran disponibles a través de una interfaz de línea de comandos (5.2.4). Esto implica que, si una librería ya cuenta con una versión ejecutable, esta puede integrarse directamente en la plataforma siguiendo lo

descrito en la sección 5.4. En caso contrario, se debe desarrollar un ejecutable que permita exponer los algoritmos mediante una interfaz de línea de comandos.

Otro desafío significativo estuvo relacionado con la gestión de archivos de eventos. Los archivos de eventos provenientes de aplicaciones reales suelen ser de gran tamaño, lo que hace inviable enviarlos al servidor en una única solicitud HTTP. Para solucionar este problema, se implementó un mecanismo de carga de archivos por lotes: cada archivo se divide en fragmentos de tamaño fijo, que son enviados individualmente al servidor. Posteriormente, los fragmentos son reensamblados para reconstruir el archivo completo. Una vez finalizado el proceso, se valida la integridad del archivo reensamblado comparando su código MD5 con el original (4.4.1).

Hacia la etapa final del desarrollo del prototipo, se identificaron varias limitaciones al momento de desplegar la aplicación en un entorno de producción. Para el despliegue, se seleccionó la plataforma *Amazon Web Services (AWS)*. Sin embargo, se encontró que el servicio *Amazon Fargate* (Amazon Web Services, 2024), que facilita el despliegue de contenedores en la nube, no está disponible dentro de las opciones de prueba gratuita ofrecidas por *AWS*. En consecuencia, se optó por la solución descrita en (5.3), que consiste en el despliegue de múltiples microservicios dentro de una única máquina virtual.

Otro de los problemas relacionados con el despliegue en producción fue la gestión del *token* de sesión *JWT*. Debido a que el cliente web y el balanceador de carga, que actúa como punto de acceso al *backend*, se encuentran en dominios diferentes (ambos asignados por defecto por *AWS*), no fue posible utilizar *cookies* para transmitir el *JWT*. Esto se debe a que los navegadores, por razones de seguridad, generalmente no permiten el intercambio de *cookies* entre dominios distintos. Como solución, se implementó un mecanismo alternativo para transmitir el *JWT* mediante el uso de encabezados HTTP en cada solicitud realizada al *backend*.

7.3. Trabajo futuro

En esta sección se identifican varias líneas de trabajo futuro que no se llevaron a cabo en esta instancia, ya sea porque no formaban parte del alcance inicial del proyecto o porque surgieron a partir de las conclusiones y limitaciones detectadas durante su desarrollo.

- Visualización de resultados: tal como se mencionó en el capítulo 6, los resultados podrían mejorarse integrando a la plataforma herramientas adicionales de visualización que optimicen el análisis y faciliten la toma de decisiones.
- Conversión de formatos:
  - Logs de eventos: actualmente, la aplicación asume que el usuario dispone de *logs* de eventos en formato *XES*, aunque también es común el uso de archivos en formato *CSV*. Si bien la plataforma admite

ambos formatos, se propone implementar un mecanismo que permita convertir archivos de *logs* de eventos entre estas dos extensiones de manera integrada en la plataforma.

- Modelos generados: La librería *PM4PY* incluye funciones para convertir entre diferentes tipos de modelos generados. Se sugiere incorporar esta funcionalidad directamente en la plataforma, facilitando la transformación entre los distintos tipos de modelos.

- Seguridad:

  - Comunicación y autenticación: se propone implementar el protocolo HTTPS para las comunicaciones entre el cliente y los distintos microservicios, garantizando una mayor seguridad en la transmisión de datos. Asimismo, se sugiere utilizar *cookies* cifradas para el manejo del *token* de sesión *JWT*, en lugar de transmitirlo mediante encabezados HTTP, como se discutió en la sección 7.2.
  - Gestión de *tokens JWT* caducados: actualmente, no se cuenta con un mecanismo para la eliminación o reutilización de *tokens JWT* caducados, los cuales permanecen almacenados indefinidamente en la base de datos. Se propone implementar una solución que permita liberar estos *tokens* y reutilizarlos de manera eficiente en el futuro.

- Experiencia de usuario: se propone añadir documentación detallada sobre los algoritmos disponibles y su uso en la interfaz gráfica, incluyendo una descripción de los parámetros aceptados, su significado y las validaciones necesarias para garantizar su correcto uso, así como enlaces a la documentación oficial de cada algoritmo. Asimismo, se identifican oportunidades de mejora en las opciones de filtrado y agrupación de tareas existentes en el sistema, con el objetivo de optimizar la interacción del usuario y la usabilidad general de la plataforma.

- Optimizaciones generales:

  - Subida de archivos: como se mencionó en la sección 4.4.1, los archivos actualmente se envían al *backend* en lotes de forma secuencial. Una posible optimización para este proceso sería realizar los envíos en paralelo, lo que reduciría significativamente el tiempo necesario para completar la operación.
  - Caché de algoritmos expuestos por los microservicios: actualmente, el microservicio *Tareas* almacena en memoria la colección de algoritmos expuestos por todos los microservicios. Se sugiere implementar una solución más robusta, utilizando por ejemplo *Redis* (Redis Ltd., 2024), para desacoplar esta funcionalidad y mejorar la escalabilidad y eficiencia del sistema.

- Quality of Service:

    - Realizar pruebas de estrés en el sistema, para determinar cuales algoritmos son más exigentes en la utilización de recursos, a fin de poder escalar los microservicios que más lo necesiten.
    - Evaluar herramientas para escalar horizontalmente los microservicios más críticos (e.g: *Tareas*) en función del volumen de solicitudes que reciben.
    - Gestión de ejecuciones en curso: en el sistema actual no existe un mecanismo para identificar ejecuciones de algoritmos que se detengan inesperadamente o se prolonguen indefinidamente. Se propone implementar un sistema de monitoreo que detecte estas situaciones y permita recuperarse de manera adecuada, ya sea reintentando la ejecución o terminándola de forma controlada.

- Roles de usuario: incorporar diferentes roles de usuario en el sistema, como administradores, usuarios con permisos de ejecución limitados, y usuarios técnicos o no técnicos, para ofrecer un control más granular y adaptado a las necesidades específicas de cada perfil.

- Despliegue a producción: implementar el uso de contenedores para el despliegue de los microservicios en producción, lo que facilitaría la portabilidad, escalabilidad y gestión del entorno.

- Documentación: mejorar los registros de *logs* y las trazas de eventos del sistema, y documentar los métodos de los controladores utilizando herramientas como *Swagger* (SmartBear Software, 2024), con el objetivo de proporcionar una referencia clara y accesible para desarrolladores y usuarios técnicos.

- Tests: actualizar y ampliar la cobertura de los *tests* unitarios, asegurando que abarquen una gama más amplia de escenarios y mejoren la fiabilidad del sistema.

Referencias

Aalst, van der, W., Dongen, van, B., Günther, C., Rozinat, A., Verbeek, H., y Weijters, A. (2009). Prom : the process mining toolkit. En *Proceedings of the bpm 2009 demonstration track* (pp. 1–4). Ulm, Germany: CEUR-WS.org.

Adriansyah, A., van Dongen, B. F., y van der Aalst, W. M. P. (2011). Conformance checking using cost-based fitness analysis. *International Conference on Business Process Management (BPM)*, 262–276. doi: 10.1007/978-3-642-23059-2_21

Amazon Web Services. (2024). *Aws fargate - run containers without managing servers or clusters.* Descargado de https://aws.amazon.com/fargate/ (Accessed: 2024)

Amazon web services. (2024). https://aws.amazon.com. (Accessed: 2024-10-26)

Augusto, A., Conforti, R., Dumas, M., Rosa, M. L., Maggi, F. M., Marrella, A., ... Soo, A. (2018). Automated discovery of process models from event logs: Review and benchmark. *IEEE Transactions on Knowledge and Data Engineering*, *31*(4), 686–705. Descargado de https://ieeexplore.ieee.org/document/8368306 doi: 10.1109/TKDE.2018.2841877

Berti, A., van Zelst, S., y Schuster, D. (2023). Pm4py: A process mining library for python. *Software Impacts*, 100556. doi: 10.1016/j.simpa.2023.100556

Bpi challenge. (2024). https://icpmconference.org/2020/bpi-challenge/. (Accessed: 2024-08-29)

Celonis. (2024). https://www.celonis.com/. (Accessed: 2024-10-26)

Chasis. (2024). https://microservices.io/patterns/microservice-chassis.html. (Accessed: 2024-11-14)

Chris Richardson. (2024). *A pattern language for microservices.* https://microservices.io/patterns/index.html. (Accessed: 2024-07-18)

Configuración externalizada. (2024). https://microservices.io/patterns/externalized-configuration.html. (Accessed: 2024-11-16)

Descomposición por capacidad de negocio. (2024). https://microservices.io/patterns/decomposition/decompose-by-business-capability.html. (Accessed: 2024-11-16)

Docker. (2024). https://www.docker.com/. (Accessed: 2024-12-08)

Fielding, R. T., y Taylor, R. N. (2000). *Architectural styles and the design*

of network-based software architectures (Tesis Doctoral no publicada). (AAI9980887)

Flask. (2024). https://flask.palletsprojects.com/en/stable/. (Accessed: 2024-10-26)

Flask blueprints. (2024). https://flask.palletsprojects.com/es/main/blueprints/. (Accessed: 2024-11-18)

Gamma, E., Helm, R., Johnson, R., y Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software.* Addison-Wesley.

Günther, C., y Rozinat, A. (2012). Disco: discover your processes. En *Proceedings of the demonstration track of the 10th international conference on business process management (bpm 2012)* (pp. 40–44). CEUR-WS.org.

ICPM. (2024). *Bpmn miner.* https://sep.cs.ut.ee/Main/BPMNMiner/. (Accessed: 2024-09-18)

Instancia de servicio por contenedor. (2024). https://microservices.io/patterns/deployment/service-per-container.html. (Accessed: 2024-11-16)

Inyección de dependencias. (2024). https://en.wikipedia.org/wiki/Dependency_injection. (Accessed: 2024-10-26)

Irakli Nadareishvili, M. M., Ronnie Mitra, y Amundsen, M. (2016). *Microservices architecture: Aligning principles, practices, and culture.* O'Reilly Media.

Janssenswillen, G., Depaire, B., Swennen, M., Jans, M., y Vanhoof, K. (2018). bupar: Enabling reproducible business process analysis. *Knowledge-Based Systems*. doi: 10.1016/j.knosys.2018.10.018

Java. (2024). https://www.java.com/es/. (Accessed: 2024-10-26)

Jones, M. B., Bradley, J., y Sakimura, N. (2015, mayo). *JSON Web Token (JWT)* (n.º 7519). RFC 7519. RFC Editor. Descargado de https://www.rfc-editor.org/info/rfc7519 doi: 10.17487/RFC7519

La Rosa, M., Reijers, H., Aalst, van der, W., Dijkman, R., Mendling, J., Dumas, M., y García-Bañuelos, L. (2011). Apromore : an advanced process model repository. *Expert Systems with Applications*, 7029–7040. doi: 10.1016/j.eswa.2010.12.012

Leach, P. J., Salz, R., y Mealling, M. H. (2005, julio). *A Universally Unique IDentifier (UUID) URN Namespace* (n.º 4122). RFC 4122. RFC Editor. Descargado de https://www.rfc-editor.org/info/rfc4122 doi: 10.17487/RFC4122

Leemans, S. J. J., Fahland, D., y van der Aalst, W. M. P. (2013). Discovering block-structured process models from event logs – a constructive approach. En *International conference on application and theory of petri nets and concurrency (icatpn)* (Vol. 7927, pp. 311–329). Springer. doi: 10.1007/978-3-642-38697-8_17

Nginx. (2024). https://nginx.org/en/. (Accessed: 2024-10-26)

Object Management Group. (2011). *Business Process Model and Notation (BPMN) Version 2.0.* Descargado de https://www.omg.org/spec/BPMN/2.0 (Retrieved from https://www.omg.org/spec/BPMN/2.0)

of Standards, N. I., y Technology. (2015, August). Secure hash standard (shs).

Federal Information Processing Standards Publication, *180*(4). Descargado de https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf doi: 10.6028/NIST.FIPS.180-4

Postgresql. (2024). https://www.postgresql.org/. (Accessed: 2024-10-26)

Process intelligence solutions. (2024). https://processintelligence.solutions/. (Accessed: 2024-10-26)

Python. (2024). https://www.python.org/. (Accessed: 2024-10-26)

R. (2024). https://www.r-project.org/. (Accessed: 2024-10-26)

React. (2024). https://es.react.dev/. (Accessed: 2024-10-26)

Redis Ltd. (2024). *Redis.* https://redis.io/es/. Descargado de https://redis.io/es/ (Accessed: 8 de diciembre de 2024)

Repositorio de gitlab. (2024). https://gitlab.fing.edu.uy/damian.piccini/tesis-2022-plataforma-de-microservicios-para-procesos-de-mineria-de-negocios. (Accessed: 2024-11-26)

Repository. (2024). https://deviq.com/design-patterns/repository-pattern. (Accessed: 2024-10-26)

Rivest, R. L. (1992, 1 de abril). *The MD5 Message-Digest Algorithm* (n.º 1321). RFC 1321. RFC Editor. Descargado de https://www.rfc-editor.org/info/rfc1321 doi: 10.17487/RFC1321

Rpi. (2024). https://microservices.io/patterns/communication-style/rpi.html. (Accessed: 2024-11-16)

SmartBear Software. (2024). *Swagger: Api development for everyone.* Descargado de https://swagger.io/ (Accessed: 2024)

Typescript. (2024). https://www.typescriptlang.org/. (Accessed: 2024-10-26)

vanden Broucke, S. K. L. M., Weerdt, J. D., Vanthienen, J., y Baesens, B. (2017). Fodina: A robust and flexible heuristic process discovery technique. *Decision Support Systems*, *100*, 109–118. Descargado de https://www.sciencedirect.com/science/article/pii/S0167923617300728 doi: 10.1016/j.dss.2017.04.005

van der Aalst, W. (2016). *Process mining: Data science in action.* Springer.

van der Aalst, W. M. P., Weijters, T., y Maruster, L. (2004). Workflow mining: Discovering process models from event logs. En *Ieee transactions on knowledge and data engineering* (Vol. 16, pp. 1128–1142). IEEE. doi: 10.1109/TKDE.2004.47

Weijters, A. J. M. M., van der Aalst, W. M. P., y de Medeiros, A. K. A. (2006). Process mining with the heuristicsminer algorithm. *BETA Working Paper Series*, *166*.

Weske, M. (2019). *Bpm: Concepts, languages, architectures.* Springer.

Anexo A

Documentación del Repositorio (README)

En este anexo se presenta el contenido del archivo `README.md` del repositorio asociado al proyecto (*Repositorio de Gitlab*, 2024), que documenta de manera concisa los aspectos esenciales del desarrollo. Se incluye información general sobre la implementación, su estructura, dependencias, instrucciones de instalación, uso, y otros detalles relevantes que facilitan la comprensión y utilización del código fuente. Este documento es una referencia clave para usuarios y desarrolladores interesados en replicar o extender los resultados obtenidos en este trabajo.

Requisitos

- Docker
- Docker Compose
- Python 3.11 o superior

Cómo inicializar la aplicación

1. Clona el repositorio.
2. Ejecuta `pip install -r requirements.txt` para instalar las dependencias del proyecto. Si bien cada servicio corre dentro de un contenedor propio, esto es útil para:
    - Ejecutar los tests.
    - Permitir que IntelliSense pueda inferir los tipos de los objetos en el código, en caso de que se use VSCode para el desarrollo local.

Alternativamente, se puede abrir VSCode desde el contenedor en ejecución para cada microservicio, aunque esto puede ser más incómodo si se quiere tener todas las carpetas de los microservicios abiertas dentro de la misma ventana de VSCode.

3. Ejecuta el binario `process-miner`, localizado en la carpeta raíz del proyecto, y selecciona la opción `"build"` (4). Esto construirá los contenedores de Docker para cada servicio que compone a la aplicación.

4. Para iniciar la aplicación, ejecuta el binario `process-miner` nuevamente y selecciona la opción `"boot"` (1). Esto levantará todos los servicios definidos en el archivo `docker-compose.yml`. La aplicación estará disponible en `http://localhost:3000`.

Cómo ejecutar los tests

1. Ejecuta `sh ./tests.sh`. Se reportarán en la terminal los resultados de los tests, junto con la cobertura de los mismos.

Arquitectura

El proyecto implementa una arquitectura de microservicios, como se ilustra en el siguiente diagrama:

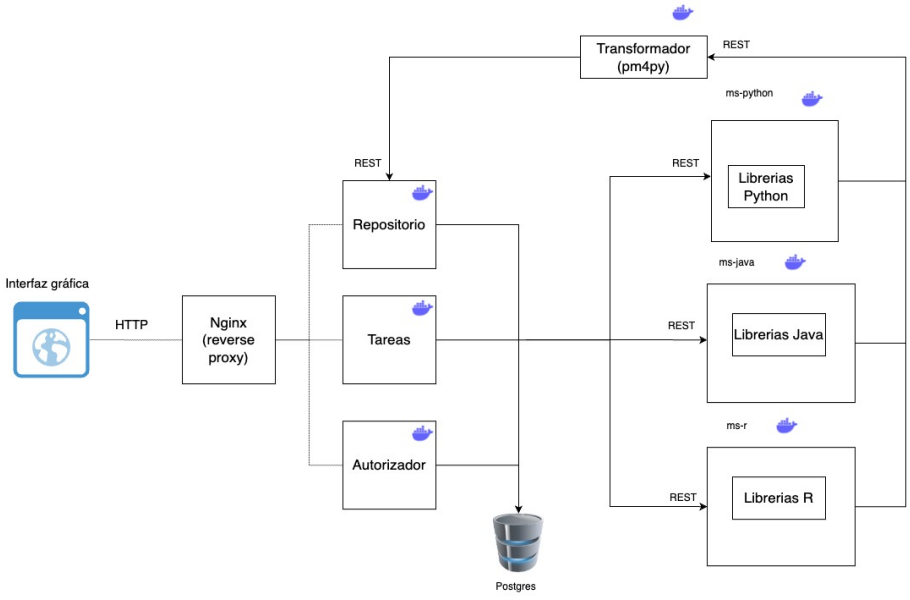


Figura A.1: Diagrama de componentes de la arquitectura

A continuación, se describen los servicios que componen la aplicación:

Nginx - API Gateway

Reverse proxy que redirige las solicitudes HTTP entrantes al microservicio correspondiente, permitiendo que la interfaz gráfica interactúe con un único endpoint.

Autorizador

Implementa la lógica de autenticación de usuarios.

Tareas

Gestiona la lógica de tareas (ejecuciones de algoritmos) para los usuarios.

Repositorio

Abstrae la lógica de acceso y manipulación de archivos de usuarios, utilizados en la ejecución de algoritmos.

MS-Java

Microservicio que implementa algoritmos de minería de procesos de negocio utilizando librerías Java. Actualmente incluye la librería `StructuredMiner`.

MS-Python

Microservicio que implementa algoritmos de minería de procesos de negocio utilizando librerías Python. Actualmente incluye la librería pm4py.

MS-R

Microservicio que implementa algoritmos de minería de procesos de negocio utilizando librerías R. Actualmente incluye la librería bupaR.

Transformador

Servicio que implementa funciones para la transformación de objetos relacionados con los algoritmos de minería de procesos. Actualmente soporta la conversión de archivos de modelo de procesos de negocio a imágenes SVG para su visualización, utilizando `pm4py`.

Postgres

Base de datos utilizada para la persistencia de datos del sistema, compartida por todos los microservicios que la requieran.

Organización del proyecto

El proyecto está organizado de la siguiente manera:

- `db-scripts`: Scripts de inicialización de la base de datos.
- `frontend`: Código de la interfaz gráfica de la aplicación.
- `lib`: Bibliotecas compartidas por los microservicios del sistema.
- `ms-python`: Microservicio de Python.
- `ms-java`: Microservicio de Java.
- `ms-r`: Microservicio de R.
- `nginx`: Configuración del API Gateway (Nginx).
- `tareas`: Microservicio de tareas.
- `autorizador`: Microservicio de autorización.
- `repositorio`: Microservicio de repositorio.
- `transformador`: Microservicio de transformación.

Consideraciones generales

Cada microservicio se inicializa en un contenedor Docker independiente. El archivo `docker-compose.yml` especifica todos los artefactos necesarios para su ejecución (volúmenes, redes, imágenes, etc.).

Microservicios de algoritmos: `ms-java`, `ms-python` y `ms-r`

Estos tres microservicios exponen los distintos algoritmos implementados en sus respectivos lenguajes. Es importante destacar que no están expuestos directamente por el API Gateway, sino que se invocan durante la ejecución de una tarea.

Cuando un usuario crea una tarea, debe especificar el algoritmo a ejecutar. Dependiendo del algoritmo escogido, se invoca al microservicio correspondiente en el momento de la ejecución.

Estos microservicios consisten en un servidor web Flask y contienen el código necesario para interpretar los parámetros de una tarea dada, ensamblar el comando de invocación del algoritmo y realizar una **llamada al sistema** para su ejecución.

Consideraciones sobre la ejecución de tareas

Para la ejecución de tareas, se utiliza un esquema de trabajos distribuidos. El proceso es el siguiente:

1. Se crea un registro en la base de datos cuando se ejecuta una tarea.
2. Se invoca al microservicio correspondiente.
3. El microservicio ejecuta el algoritmo.
4. El microservicio envía los archivos de salida al repositorio.
5. El microservicio envía el modelo de salida al transformador para generar una representación gráfica del modelo retornado.
6. Se notifica al usuario a través de la interfaz gráfica.

Consideraciones sobre la ejecución de algoritmos

Cada microservicio implementa un esquema de invocación de comandos del sistema operativo mediante el uso de procesos. Esto implica que, para ejecutar un algoritmo dado, este debe estar disponible en el `PATH` del sistema operativo del contenedor para ser invocado por interfaz de línea de comandos.

Extensibilidad

Declaración de algoritmos

Cada uno de los tres microservicios mencionados carga los algoritmos disponibles para su tecnología a través de un archivo de configuración JSON. Este archivo especifica el formato de los parámetros que recibe cada algoritmo.

Por ejemplo, un fragmento del archivo `algorithms.json` del microservicio `ms-r` se puede observar en el Listing A.0.1.

Campos destacados:

- `key`: Identificador único para el algoritmo.
- `executable`: Comando a ejecutar para invocar al algoritmo. Utiliza variables con el formato `$var` que luego se interpolarán con los parámetros requeridos.
- `outputExtension`: Extensión de los archivos de salida generados por el algoritmo.
- `parameters`: Lista de parámetros que recibe el algoritmo, incluyendo tipo, descripción, etc.

```
{
  "algorithms": [
    {
      "name": "Process Map",
      "key": "process_map",
      "category": "enhancement",
      "description": {
        "en": "Allows to visualize the process' performance.",
        "es": "Permite visualizar el rendimiento del proceso."
      },
      "link": "https://bupaverse.github.io/docs/performance_maps.
          html",
      "executable": "Rscript $scripts_path/process-map.R
          $input_file $output_file $performance_function
          $frequency_type",
      "library": "bupar",
      "outputExtension": "svg",
      "parameters": {
        "input_file": {
          "name": {
            "en": "Input file",
            "es": "Archivo de entrada"
          },
          "description": {
            "en": "Input Logs file",
            "es": "Archivo de logs de entrada"
          },
          "type": "File",
          "required": true,
          "flag": "--input"
        },
        "output_file": {
          "name": {
            "en": "Output file",
            "es": "Archivo de salida"
          },
          "description": {
            "en": "Output image file",
            "es": "Archivo de la imagen generada"
          },
          "type": "String",
          "required": true,
          "flag": "--output"
        }
      }
    }
  ]
}
```

Listing A.0.1: Ejemplo del archivo de configuración `algorithms.json` para el microservicio en R

Es importante que el nombre de un parámetro (por ejemplo, `input_file"`) coincida exactamente con las variables utilizadas en `executable`. Esto es necesario porque, al ensamblar el comando de invocación del algoritmo, se reemplazan las variables `$var` por los valores de los parámetros indicados en el JSON de entrada. En este ejemplo concreto, se remplazará la variable `$input_file` con la ruta del archivo de entrada especificado.

Cómo agregar un nuevo algoritmo

Para agregar un nuevo algoritmo, consideremos los siguientes escenarios:

1. **El algoritmo ya existe en una biblioteca utilizada por alguno de los microservicios** (por ejemplo, en `ms-python`):
   - Simplemente se debe agregar el algoritmo al archivo `algorithms.json` del microservicio correspondiente.
2. **El algoritmo no existe en las bibliotecas utilizadas, pero está implementado en un lenguaje ya soportado**:
   - Extender el archivo `algorithms.json` del microservicio correspondiente para agregar el nuevo algoritmo.
   - Asegurarse de que el algoritmo esté disponible en el `PATH` del sistema operativo del contenedor para ser invocado por línea de comandos.
   - Si la biblioteca no proporciona una interfaz de línea de comandos, se deberá implementar un script con permisos de ejecución y exponerlo en el `PATH` del contenedor.
3. **El algoritmo está implementado en un lenguaje no soportado por los microservicios existentes**:
   - Crear un nuevo microservicio para el lenguaje correspondiente.
   - Implementar los siguientes archivos:
     - `app.py`: Lógica para inicializar el servidor web que expone la interfaz REST.
     - `requirements.txt`: Dependencias del microservicio.
     - `algorithms.json`: Declaración del nuevo algoritmo.
     - `Dockerfile`: Define la imagen del contenedor del microservicio.
   - Extender `tareas/app.py` para que pueda cargar los algoritmos expuestos en el nuevo microservicio.

```
for service in ["ms-java", "ms-python", "ms-r"]:
    service = cast(ServiceIdentity, service)
    services_algos_store[service] = get_service_algos(
        service)
```

```python
match service:
    case "ms-java":
        service_url = os.getenv("JAVA_MS_URL")
    case "ms-python":
        service_url = os.getenv("PYTHON_MS_URL")
    case "ms-r":
        service_url = os.getenv("R_MS_URL")
    case _:
        raise TypeError(f"Invalid service name: {service}")
```

- Agregar la variable de entorno que contiene la URL del nuevo microservicio en el archivo `docker-compose.yml`.
- Asegurarse de que el algoritmo esté disponible en el PATH del sistema operativo del contenedor del nuevo microservicio.

Despliegue en producción

Para el despliegue en producción se utiliza AWS. En la carpeta `scripts/aws` se encuentran los scripts necesarios para declarar los artefactos utilizados en AWS, así como los scripts para el despliegue de la aplicación.

El entorno de producción está organizado de la siguiente manera:

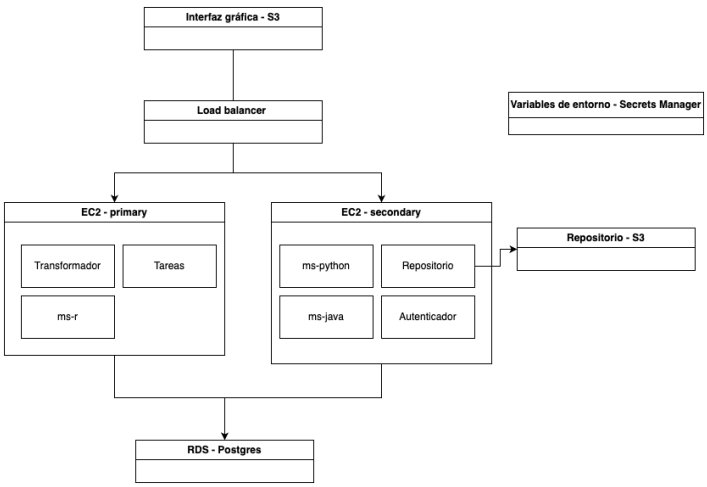


Figura A.2: Arquitectura del entorno de producción en AWS.

Se decidió utilizar este esquema de despliegue por una cuestión de costos, ya que la solución de AWS para desplegar contenedores, AWS Fargate, no tiene una versión gratuita.

Se proveen los siguientes scripts:

- `scripts/aws/deploy`: Despliega los microservicios en las instancias de AWS.
- `scripts/aws/deploy_ui`: Compila y despliega el código para la interfaz gráfica en un bucket de S3.
- `scripts/aws/infra`: Gestiona la infraestructura de AWS utilizada por la aplicación.
- `scripts/aws/logs`: Obtiene los logs de los distintos microservicios.
- `scripts/aws/secret`: Gestiona las variables de entorno de la aplicación.
- `scripts/aws/ssh`: Conecta a una instancia de AWS utilizando SSH.