Plataforma para algoritmos evolutivos paralelos con una arquitectura basada en microservicios

Memoria de proyecto presentada a
la Facultad de Ingeniería de
la Universidad de la República por

Luis Sergio Costela Rodio

en cumplimiento parcial de los requerimientos
para la obtención del título de
Ingeniero en Computación.

Tutores

Dr. Ing. Santiago Iturriaga ..... Universidad de la República
Dra. Ing. Laura González ..... Universidad de la República

Tribunal

MSc. Ing. Guzmán Llambías ..... Universidad de la República
Ing. Enrique Galindo ..... Universidad de la República
Dra. Ing. Jimena Ferreira ..... Universidad de la República

Montevideo
viernes 27 diciembre, 2024

Plataforma para algoritmos evolutivos paralelos con una arquitectura basada en microservicios, Luis Sergio Costela Rodio.

Esta tesis fue preparada en LaTeX.
Contiene un total de 119 páginas.
Compilada el viernes 27 diciembre, 2024.

Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher.

[Parece que la perfección no se alcanza cuando no hay nada más que agregar, sino cuando no hay nada más que quitar.]

Antoine de Saint-Exupéry

AGRADECIMIENTOS

En primer lugar, quiero agradecer a mi familia, quien siempre me ha acompañado y ha sido un apoyo imprescindible en este camino. Todo esto es por y para ellos.

Agradezco también a mis amigos y compañeros, quienes me han ayudado a recorrer las diferentes etapas de mi formación universitaria.

Dedico un saludo especial a aquellos mentores que, de forma directa o indirecta, colaboraron en potenciar mi formación y despertaron en mí el gusto por aprender y mejorar. Gracias a ellos, deseo crecer profesionalmente.

No puedo dejar de mencionar a mis tutores, Santiago y Laura, quienes aceptaron guiarme durante varios meses en la elaboración y realización de este proyecto de grado. Así mismo, agradezco al INCO, URI, y a Santiago nuevamente por brindarme un ambiente donde desplegar y probar mi software.

Finalmente, quiero expresar mi gratitud a la Universidad de la República por darme la oportunidad de formarme como Ingeniero en Computación.

Dedicado a toda mi familia, con quienes he aprendido a ser la persona que soy.

RESUMEN

En el campo de los problemas de optimización, existen diversas técnicas para hallar o aproximar soluciones. En este contexto, los algoritmos evolutivos son métodos computacionales inspirados en la evolución natural que definen una serie de pasos para buscar soluciones, las cuales no necesariamente son exactas. Los algoritmos evolutivos paralelos constituyen un caso particular, en el que se utiliza la ejecución simultánea para acelerar la búsqueda en el espacio de soluciones.

Por otro lado, la arquitectura de microservicios aporta una serie de beneficios tanto al *software* como al proceso de desarrollo, abarcando todo su ciclo de vida. Esta arquitectura es considerada uno de los pilares fundamentales del modelo *cloud native*. El enfoque *cloud native* busca que las aplicaciones estén diseñadas para ser escalables, resilientes y manejables en condiciones cambiantes y entornos dinámicos, explotando las ventajas de la computación en la nube.

Si bien existen diversas herramientas que permiten implementar algoritmos evolutivos paralelos, a nuestro leal saber y entender, ninguna de ellas se adhiere plenamente al enfoque *cloud native*. Esto representa una oportunidad para explorar nuevas soluciones que aprovechen las ventajas de esta perspectiva. En este contexto, este proyecto propone una plataforma con una arquitectura de microservicios para la ejecución de algoritmos evolutivos paralelos, siguiendo los principios fundamentales del modelo *cloud native*.

En primer lugar, se realiza una revisión de la literatura académica relacionada con las temáticas tratadas, estableciendo una base de conocimientos para tomar decisiones adecuadas al plan de este proyecto. También se consultan sitios web de profesionales con experiencia en el campo, así como la documentación oficial de productos con trayectoria en el área, como *jMetal*, *ParadisEO* y *HeuristicLab*. Simultáneamente, se estudian diversos trabajos relacionados con investigaciones alineadas con la propuesta de este proyecto.

Posteriormente, se realiza el análisis y diseño de la plataforma, evaluando la aplicabilidad de la arquitectura de microservicios en sistemas diseñados para ejecutar algoritmos evolutivos. Este proceso se centra en analizar la viabilidad de la arquitectura propuesta, estableciendo requisitos no funcionales verificables.

Finalmente, se implementa un prototipo de la propuesta utilizando lenguajes y herramientas de código abierto y de uso libre, garantizando un producto de iguales características. Siempre que es posible, se opta por instalar productos especializados, siguiendo un enfoque orientado a la reutilización. Como tecnología de orquestación, se elige *Kubernetes*, estándar de facto en la industria, reconocido por su capacidad para automatizar el despliegue, escalado y administración de aplicaciones basadas en contenedores. Además, al utilizar *Kubernetes*, se obtiene una solución integrada a la nube, que permite tanto un enfoque *DevOps* como un desarrollo *cloud native*.

Para validar su efectividad y rendimiento, se llevan a cabo pruebas utilizando problemas conocidos en el campo de la optimización. Estos experimentos permiten identificar los beneficios y limitaciones de esta arquitectura, destacando los problemas más adecuados para este entorno y aquellos donde resulta menos favorable frente a otras soluciones.

Los resultados obtenidos son satisfactorios y confirman que esta arquitectura es viable para construir este tipo de sistemas, identificando escenarios de uso interesantes.

Palabras clave: *algoritmos evolutivos, arquitectura de microservicios, patrones de diseño, código abierto, computación distribuida y paralela, DevOps, cloud native.*

PREFACIO

El proyecto de grado surge como iniciativa propia, tras la realización de una tarea de una asignatura de esta carrera, donde se debía dar solución a un problema mediante algoritmos evolutivos.

En su momento, las ideas que surgieron eran interesantes, pero superaban el alcance de lo pedido. De optar por la realización de mi propuesta, se habría acotado el proyecto y se hubiese desaprovechado una oportunidad de indagar y analizar un diseño más completo. Entonces, ¿por qué no realizar un proyecto de grado que me permita explorar un diseño más complejo? Y por qué no ir un poco más allá y generalizar la idea y construir una plataforma.

Fue así como nació la idea, que une dos grandes temas que me parecen sumamente interesantes: los algoritmos evolutivos y la arquitectura de software.

Mi propuesta permitía modelar una solución modular, orientada a componentes independientes y escalable, digna de ser representada mediante una arquitectura de microservicios, lo que permite la flexibilidad de ejecutar algoritmos evolutivos paralelos de forma natural. Esa solución hubiese sido algo compleja y corría el riesgo de terminar siendo poco extensible o reutilizable, ya que podía estar atada a resolver el problema particular de la tarea.

Luego de profundizar más en las soluciones de microservicios construidas y destinadas para el uso en la nube, descubrí que hay mucho por explorar y aprender referente a ello, por lo que podía apuntar a su inclusión y colocar este proyecto como un camino hacia algo más grande.

Gracias a la gran receptividad de mis tutores, logré dar forma a la propuesta y elegirla por encima de otras que fueron ofrecidas. Esta decisión fue acertada, ya que disfruté la realización de este proyecto.

Me entusiasma poder explorar nuevas tecnologías, diseños de arquitecturas y soluciones integrales, así como ver funcionar aquello que modelo y someterlo a análisis para mejorarlo día a día. Espero que este proyecto entusiasme a muchas personas como a mí y que puedan descubrir y potenciar sus intereses en los diferentes campos de la ingeniería.

Luis Costela

TABLA DE CONTENIDOS

CAPÍTULO 1

INTRODUCCIÓN

Contenido

1.1. Meta del proyecto

Este proyecto es conducido por una meta más amplia, que corresponde a la creación de una metodología para adaptar el desarrollo, despliegue y administración de heurísticas de optimización a un modelo *cloud native*, acompañado de la especificación de la solución y una implementación de referencia.

El alcance de este proyecto se define abordando una parte del problema general, enfocando el esfuerzo en analizar los beneficios de la arquitectura de microservicios aplicada a un sistema que opera con algoritmos evolutivos paralelos, bajo las condiciones adecuadas para conducir la solución a una adopción nativa en la nube. Aspectos como la utilización de contenedores y la adopción de prácticas *DevOps* son estudiados e incluidos como parte de la solución, tomando estándares y buenas prácticas del diseño de una arquitectura de microservicios y computación en la nube.

1.2. Motivación

Las arquitecturas de *software*, procesos y tecnologías han evolucionado a lo largo de los años y las organizaciones han acompañado el cambio, diseñando sus aplicaciones guiadas por los estándares y recomendaciones del momento. En la actualidad, existen aspectos claves como el proceso de desarrollo, la arquitectura de las aplicaciones, el despliegue y empaquetado, y la administración de la infraestructura, que definen la forma de construir y ejecutar aplicaciones, marcando un camino hacia un modelo *cloud native* [1]. El progreso de los procesos de desarrollo tradicionales hacia los ágiles con una cultura *DevOps*, la migración de los sistemas monolíticos a microservicios, el uso de tecnologías de empaquetado basado en imágenes y desplegados en contenedores y la conducción hacia el uso de recursos en la nube son los pilares de este modelo.

El término *cloud native* refiere a los conceptos de creación y ejecución de aplicaciones para tomar las ventajas de la computación distribuida en un modelo de la nube. Estas aplicaciones están diseñadas para explotar características como escalado, elasticidad, resiliencia, entre otras [1]. Existen muchas definiciones de *cloud native*, y en general se reducen al aprovechamiento de los servicios y modelos de entrega basados en la nube.

Por otra parte, en diversas disciplinas se presenta la necesidad de abordar problemas complejos de optimización, por lo que existen diferentes productos que asisten al modelado de problemas y búsqueda de soluciones mediante metaheurísticas. Algunos ejemplos de este tipo de *software* son *jMetal*[1], *HeuristicLab*[2] y *ParadisEO*[3].

La diversidad de herramientas es útil para que los usuarios puedan optar por aquella que se ajuste mejor a sus necesidades. Sin embargo, a nuestro leal saber y entender, ninguna de ellas se adhiere al enfoque *cloud native*, lo que evidencia una oportunidad para explorar esta temática dentro del desarrollo de aplicaciones basadas en metaheurísticas. Esto tiene el potencial de otorgar valor agregado a esta familia de *software*, ya que al optar por un modelo *cloud native*, las aplicaciones se ven fortalecidas al ser escalables, explotando el uso de contenedores y microservicios [2].

Además, el enfoque *cloud native* puede aportar a estas soluciones aspectos tales como bajo acoplamiento, resiliencia, y la obtención de productos administrables, automatizables y observables, facilitando y fomentando los cambios frecuentes en el *software* con un mínimo esfuerzo [2]. Entonces, esto establece un interesante punto de partida desde donde empezar a analizar el enfoque *cloud native* para el entorno de los sistemas que modelan problemas de optimización, como algoritmos evolutivos.

1.3. Objetivos

El objetivo general del proyecto es avanzar hacia un modelo *cloud native* para la ejecución de algoritmos evolutivos, mediante la definición de una plataforma basada en una arquitectura de microservicios.

Para lograr la conducción del proyecto, se establecen diferentes objetivos específicos, interconectados y organizados en una estructura jerárquica. Esta organización permite establecer un camino en la ejecución de las diferentes etapas del proyecto.

Los primeros objetivos componen tareas de investigación y análisis, con el fin de establecer una base de conocimiento.

Los siguientes objetivos se vinculan por su carácter de elaboración y creación, siendo consecuencia de los esfuerzos de estudio, ya que se nutren de los aportes anteriores.

Como parte final, se presenta la evaluación como principal mecanismo de comprensión y validación de las actividades anteriores.

En la figura 1.1 se aprecia la organización de los diferentes objetivos, de acuerdo a los criterios explicitados.

[1] `https://jmetal.readthedocs.io/en/latest/`
[2] `https://dev.heuristiclab.com/trac.fcgi/`
[3] `https://nojhan.github.io/paradiseo/`

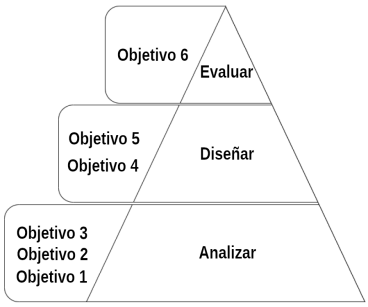

Figura 1.1: Esquema de organización de los objetivos.

Objetivo 1: Analizar las características de las soluciones existentes que asisten la creación de algoritmos evolutivos

Existen diversos *frameworks* tales como *jMetal*, *ParadisEO* y *HeuristicLab*, que son reconocidos y utilizados en la actualidad para facilitar el desarrollo de algoritmos evolutivos. Además, hay numerosos informes y publicaciones que abordan la temática del diseño y desarrollo de estos sistemas bajo diferentes enfoques, tales como el de microservicios o la migración a tecnologías en la nube.

La realización de un relevamiento de conocimiento existente es de suma importancia durante el proyecto, ya que extiende el conocimiento de partida. Dado que se espera como resultado la construcción de una plataforma capaz de trabajar con algoritmos evolutivos paralelos, es de suma importancia nutrir el diseño con la experiencia y conocimiento de otros productos y autores.

Objetivo 2: Analizar la aplicabilidad de una arquitectura de microservicios en el ámbito de algoritmos evolutivos paralelos

La utilización de una arquitectura basada en microservicios es uno de los puntos fundamentales que introducen la realización de este proyecto.

De acuerdo al escenario de aplicación de algoritmos evolutivos paralelos, se puede discutir si este estilo arquitectónico es el más adecuado o no, y por ello es importante determinar cuáles son las problemáticas en las que esta arquitectura ofrece beneficios.

En otras palabras, se apunta a identificar cuáles son los tipos de problemas en los que esta arquitectura resulta más adecuada, ya que de acuerdo a las características de los problemas, puede ser más conveniente utilizar, por ejemplo, un *framework* orientado a objetos.

Objetivo 3: Especificar los requerimientos funcionales y atributos de calidad del sistema a construir

Está claro que la solución debe tener la capacidad de realizar el modelado y ejecución de algoritmos evolutivos paralelos, pero es necesario especificar el conjunto de requerimientos funcionales del *software* a construir. A grandes rasgos, debe contemplar la operación del ciclo de vida de desarrollo de un algoritmo evolutivo, pero es necesario definir claramente el alcance y forma que toman.

Por otra parte, las características no funcionales deben ser contempladas y especificadas. En particular, son de interés atributos de calidad vinculados a la escalabilidad, evolucionabilidad, resiliencia, interoperabilidad y modificabilidad.

Estas características deben ser correctamente descritas y cuantificadas para su evaluación.

Objetivo 4: Diseñar una plataforma para ejecutar algoritmos evolutivos paralelos basada en la arquitectura de microservicios

La definición de la arquitectura puede convertirse en un camino de descubrimientos para el encargado del diseño. En etapas tempranas del proyecto se debe optar por una arquitectura que sea capaz de afrontar los requerimientos de los interesados. Estas inquietudes se convierten en factores determinantes para la toma de decisiones [3].

Las influencias que puede recibir una arquitectura se pueden agrupar en dos categorías: las centradas en el problema y las centradas en la solución [3]. La categoría que se centra en el problema aborda el diseño para solucionar un problema e intenta responder por qué o qué debe poder hacer el sistema. Por otra parte, las que se centran en la solución influencian el diseño, sugiriendo o especificando la manera de cómo construir el sistema, respondiendo preguntas de cómo o con qué desarrollar el *software*, incluyendo directivas tecnológicas, estándares o políticas concretas [3].

En este proyecto, el diseño está principalmente centrado en la solución y el objetivo planeado corresponde a diseñar una solución bajo una arquitectura de microservicios, que contemple los requerimientos funcionales y atributos de calidad especificados. Este diseño debe apuntar a que la solución sea capaz de oficiar de *framework* en términos de definición de algoritmos evolutivos, a la vez de tender a una solución de plataforma como servicio.

Objetivo 5: Establecer un proceso de desarrollo para algoritmos evolutivos paralelos en el ámbito de la plataforma propuesta

Es necesario definir y documentar el proceso de desarrollo para integrar componentes a la plataforma propuesta.

Esto se refiere al proceso de desarrollo de componentes reutilizables como parte de un algoritmo evolutivo.

Objetivo 6: Evaluar el diseño de la plataforma y el proceso de desarrollo mediante la implementación del sistema

Una vez obtenido un diseño, es importante comprobar que es capaz de cumplir con las especificaciones deseadas. El proceso de evaluación establece la comprobación de las decisiones tomadas, en conjunto con una evaluación de cuán adecuadas son.

Como mecanismo de evaluación establecido para este objetivo, se apunta a la implementación del sistema, en conjunto con la verificación de sus capacidades.

1.4. Aportes del proyecto

La realización de este proyecto conlleva una serie de aportes que pueden ser utilizados por quien lo necesite. Estas contribuciones tienen relación directa con el cumplimiento de los objetivos propuestos.

Aporte 1: Análisis de soluciones existentes que sirven para ejecutar algoritmos evolutivos

Producto del estudio de los diferentes *frameworks* y de las diversas propuestas de diseño provenientes de publicaciones, se logra obtener un panorama de cómo se utilizan este tipo de soluciones y cuáles son sus características.

Aporte 2: Análisis de aplicabilidad de una arquitectura de microservicios para sistemas que ejecutan algoritmos evolutivos

La aplicabilidad de la arquitectura de microservicios para los sistemas que operan con algoritmos evolutivos puede ofrecer tanto beneficios como dificultades.

En este análisis se identifican las situaciones en las que este modelado resulta más adecuado y se comprenden los motivos; al tiempo que se reconocen los escenarios donde otro tipo de implementación sería más natural.

Aporte 3: Diseño de una plataforma que opera con algoritmos evolutivos utilizando microservicios

Se ofrece el diseño de una plataforma capaz de implementar algoritmos evolutivos bajo una arquitectura de microservicios.

Esto se da por medio de la especificación del sistema propuesto, un conjunto de diagramas que lo definen y la documentación asociada.

Aporte 4: Implementación de referencia de la plataforma diseñada

A partir de la especificación del sistema, se obtiene una implantación de referencia, lo que significa un conjunto de piezas de código libre, bajo una licencia *GPL2*[4], para garantizar que este producto y sus posibles derivados mantengan las mismas características libres.

Además de los fuentes y binarios de una versión específica, se distribuye una guía de compilación y de instalación. También se entregan algunos operadores evolutivos de referencia para poder utilizarlos.

Dado que la solución apunta a automatizar los procesos de integración y entrega continua, se implementan mecanismos para resolver esto, utilizando tecnologías concretas como parte de la solución de la plataforma.

Aporte 5: Documentación sobre el desarrollo e integración de componentes reutilizables

El diseño de la plataforma contempla el uso de componentes de *software* reutilizables que forman parte de la implementación de los algoritmos evolutivos.

En este sentido, se elaboró documentación que guía al usuario en el proceso de integración de su desarrollo. En ella se detallan las directivas necesarias para que un componente sea considerado elegible como parte de un algoritmo evolutivo, estandarizando algunos aspectos de su construcción y distribución.

Esta documentación se agrega al código fuente de la implementación de referencia.

Aporte 6: Resultados de la evaluación de la plataforma

La evaluación no es una tarea sencilla, pero existen algunas técnicas para poder realizarla, distinguiéndose entre ellas de acuerdo a costo, profundidad o complejidad; por ello, de acuerdo al contexto, es importante optar por una técnica adecuada [3].

En este caso, se opta por la técnica de *Skeleton System*, que consiste en la creación de una primera versión del sistema, donde se implementan las estructuras arquitectónicas principales y se ofrece un conjunto específico de funcionalidades completas ("de punta a punta"). A diferencia de una prueba de concepto o un prototipo, el producto no es descartado [3].

[4] `https://www.gnu.org/licenses/old-licenses/gpl-2.0.html`

A pesar de que es considerado como uno de los mecanismos de evaluación más costosos, ya que a diferencia de un prototipo, es necesario aplicar un proceso de desarrollo calificado para un producto destinado a producción, en este contexto puede llevarse a cabo. Como puntos fuertes, esta técnica provee uno de los análisis más exhaustivos y convincentes al diseño que convergen en un producto tangible que puede ser utilizado más allá de la actividad de evaluación [3].

Como parte de la evaluación del sistema, se realizan diferentes pruebas para conocer su rendimiento; al mismo tiempo, se realizan comparaciones con otras soluciones del mercado que permiten ejecutar algoritmos evolutivos.

Este aporte es clave para visualizar el valor agregado de la solución y así determinar cuáles son las fortalezas que la distinguen, de modo de poder explotarlas a futuro.

De igual forma, las pruebas de rendimiento son útiles para establecer puntos de mejora y posterior análisis de cambios.

1.5. Organización del documento

El resto del documento se organiza de la siguiente forma.

El Capítulo 1 brinda una visión general de las metas, motivaciones y objetivos del proyecto.

Luego, el Capítulo 2 y el Capítulo 3 brindan un marco teórico de las temáticas fundamentales a tratar en este trabajo: algoritmos evolutivos y arquitecturas de microservicios. Se provee un panorama general sobre el surgimiento de los algoritmos evolutivos, su utilidad y cuál es la esencia de su diseño y funcionamiento como un tipo de metaheurística de optimización. En cuanto a la arquitectura de microservicios, se compara esta arquitectura con otras que son de relevancia en este proyecto y se relaciona con las soluciones *cloud native*.

A partir de la presentación de un marco teórico sobre los temas más técnicos a tratar, se documenta en el Capítulo 4 el trabajo de análisis de soluciones que operan con algoritmos evolutivos y se sintetizan las ideas fundamentales obtenidas en diferentes publicaciones que tratan diversos puntos de vista del diseño de *frameworks* para algoritmos evolutivos. En esta parte se nutre el diseño de la solución gracias al aporte de trabajos de otros autores y la comprensión de la forma en que se desarrollan y utilizan las soluciones disponibles en el dominio público.

En el Capítulo 5 se describe el diseño de la solución propuesta mediante diferentes diagramas, casos de uso y atributos de calidad.

En el Capítulo 6 y el Capítulo 7 se detallan las formas de llevar adelante la implementación, indicando decisiones tecnológicas y mecanismos concretos para desarrollar una determinada parte de la solución o la creación de los ambientes. Como parte del proceso de desarrollo, se valida el resultado.

Por último, en el Capítulo 8, se da cierre al documento y se presentan un breve resumen de lo realizado y las conclusiones. Además, se agrega una sección que describe las diferentes actividades que pueden formar parte del trabajo a futuro, abarcando temas tales como la deuda técnica, mejoras tecnológicas, nuevas funcionalidades y visión a futuro de cómo evolucionar la plataforma.

CAPÍTULO 2

ALGORITMOS EVOLUTIVOS

Contenido

En este capítulo se presenta el marco conceptual referente a los algoritmos evolutivos, debido a que es un tema fundamental para alcanzar el entendimiento de la plataforma a desarrollar. Se presentan sus orígenes, su rol en la optimización de problemas, sus principales componentes y los tipos de problemas en donde se aplican.

2.1. Teorías evolutivas

Para comprender la esencia de los algoritmos evolutivos, es conveniente repasar el paso histórico de las teorías evolutivas, para tener claros los orígenes e influencias en estos métodos. Se toma como referencia para esta sección [4] y [5].

La primera doctrina que explicaba el origen de las especies fue la creacionista, la cual carecía de fundamento científico, ya que no había evidencia que la respaldara.

La teoría de la evolución contemporánea tiene soporte en los hechos y evidencias, no pudiéndose refutar hasta el momento por algún argumento o evidencia.

Algunos pioneros de esta teoría son George Louis Leclerc y Pierre Antoine de Monet. El primero es responsable de recopilar evidencia empírica para refutar el creacionismo, dando pie a la especulación de la evolución mediante un ancestro común entre los humanos y los simios, entre otras cosas. Pierre Antoine de Monet, por su parte, complementa las ideas de George Louis Leclerc proponiendo un mecanismo evolutivo de las especies basado en la adaptación de los individuos al ambiente, en conjunto con la herencia de las características adquiridas. Sin embargo, esta propuesta no es la aceptada hoy en día.

Tanto Charles Darwin como Alfred Russel Wallace continúan con los aportes a la teoría y plantean el concepto de la evolución de las especies mediante mecanismos de selección natural. Darwin obtiene información valiosa en sus viajes por el mundo y se pueden destacar los hallazgos relacionados con su pasaje por las islas Galápagos, donde encontró un ecosistema cerrado para analizar sus especies.

Los que continúan con la teoría fueron Gregor Mendel y August Weismann, entre otros, obteniendo avances en el plano de la genética y su transmisión de información genética entre los descendientes.

A partir de los conceptos descritos y su combinación, se logra la teoría neodarwinista. Esta teoría se puede explicar sistemáticamente en torno a los conceptos de: competencia entre los individuos, selección natural, transmisión hereditaria, reproducción y mutación.

2.2. Computación evolutiva

La computación evolutiva es un área de la ciencia de la computación que se encarga de resolver problemas mediante el uso de ideas de la evolución biológica, siendo una rama de la inteligencia artificial [6]. En [7] se presentan varias definiciones de inteligencia artificial, entre ellas, la de Kurzweil describe qué es el arte de crear máquinas capaces de realizar tareas que requieren inteligencia cuando son llevadas a cabo por personas.

Está claro que no existe un método que aplique eficientemente ante cualquier tipo de problema. Por ello, es necesario aclarar en qué escenarios son útiles las técnicas de computación evolutiva y en cuáles otros no lo son.

Estas técnicas son adecuadas para problemas complejos en los cuales no se conocen algoritmos eficientes, así como para problemas que tienen múltiples objetivos. En general, son admisibles cuando una solución aproximada es aceptable. También son útiles para problemas que varían de acuerdo a algún factor, como el tiempo, o en problemas donde el espacio de soluciones no es continuo [9].

Por otra parte, si se requiere una solución óptima, esta familia de métodos ya no garantiza su obtención y, por tanto, no son de utilidad. Además, si el problema puede resolverse utilizando técnicas específicas, como programación lineal o programación dinámica, entre otras, no tiene sentido aplicar estas aproximaciones, a no ser que se trate de una prueba académica o de concepto [9].

Como se mencionó, es preferible usar métodos exactos para resolver problemas, pero cuando se aplican a problemas difíciles, su resolución no es eficiente. La complejidad de los algoritmos aumenta de forma superpolinomial de acuerdo al tamaño de la entrada del problema y el tiempo o uso de recursos computacionales excede lo previsto [10].

Si se habla de eficiencia computacional, se puede describir como la cuantificación de la cantidad de recursos requeridos para resolver una tarea determinada. Esto se puede llevar a cabo mediante el conteo de operaciones básicas y su escala de acuerdo al tamaño de los datos ingresados al algoritmo. En la teoría de la complejidad computacional, los algoritmos se clasifican en clases de acuerdo a su eficiencia. En particular, cuando se habla de problemas complejos o difíciles, se alude a los de la categoría NP (*Nondeterministic Polynomial time*), donde la solución puede verificarse eficientemente. En contraposición, en la clase P (*Polynomial time*) se pueden resolver problemas de manera eficiente [11].

2.2.1. Metaheurísticas

Las metaheurísticas son una rama de la optimización estocástica, siendo esta una familia de algoritmos y técnicas que utilizan en cierta medida la aleatoriedad para encontrar o aproximar óptimos en problemas difíciles [12]. Por lo dicho anteriormente, estos métodos no son deterministas, es decir, no siempre se obtiene la misma solución para un problema determinado, ya que, como se mencionó, el azar está presente como parte de su proceso.

Las metaheurísticas entonces definen un mecanismo genérico para guiar la búsqueda de soluciones, a diferencia de las heurísticas que se centran en algún problema particular.

2.2.2. Algoritmos evolutivos

Los algoritmos evolutivos son técnicas de optimización que se basan en las teorías de la evolución natural. La optimización se puede definir como una metodología en la

que se busca tomar la mejor decisión posible para maximizar o minimizar un objetivo determinado [13].

Un problema de optimización puede plantearse de acuerdo a la fórmula (2.1).

$$\begin{aligned} &\text{Encontrar } \mathbf{x} = \{x_1, x_2, \ldots, x_n\}, \text{ que} \\ &\text{Minimice/Maximice } f(\mathbf{x}) \\ &\text{Sujeto a} \\ &\quad g_j(\mathbf{x}) \leq 0, j = 1, 2, \ldots, m \\ &\quad h_j(\mathbf{x}) = 0, j = 1, 2, \ldots, p \end{aligned} \tag{2.1}$$

En (2.1) se distingue la función objetivo $f(\mathbf{x})$ y un conjunto de variables de decisión $x_1, x_2, \ldots, x_n$ que representan las elecciones tomadas sobre el problema. Se busca entonces hallar un máximo o mínimo en la función objetivo, mediante la buena elección de las variables de decisión. Por último, se agregan restricciones para modelar y limitar el espacio de soluciones [13].

En problemas reales de ingeniería, se presentan desafíos tales como la presencia de funciones discontinuas, no lineales, no convexas o multimodales, lo que dificulta su resolución de manera eficiente. Incluso son desafiantes aquellos problemas multidimensionales, o dicho de otra forma, de múltiples objetivos. Los problemas de optimización pueden categorizarse de acuerdo a la existencia de restricciones, la naturaleza y valores posibles de las variables de decisión, número de funciones objetivo, entre otros [13].

Entonces, se remarca la existencia de al menos dos enfoques de resolución, el uso de técnicas de programación matemática o el uso de técnicas metaheurísticas. Las técnicas de metaheurísticas solventan situaciones en las que la programación matemática presenta limitaciones, como por ejemplo la convergencia prematura en óptimos locales, la necesidad de formular funciones objetivo o de restricción de forma rigurosa, requieren la existencia de derivadas para la función objetivo y las funciones de restricción, y dificultades varias al manejar variables mixtas [13].

De todas las técnicas metaheurísticas, los algoritmos evolutivos son especialmente efectivos en la solución de problemas de optimización restringidos, multidimensionales, no lineales, no convexos, de alta complejidad, con variables mixtas y multiobjetivo, para los cuales un modelo matemático tradicional es difícil de construir [13].

El proceso de optimización de un algoritmo evolutivo involucra varios conceptos y componentes que se describen a continuación. Para comenzar, manejan un conjunto de posibles soluciones, representadas por algún tipo de codificación. Este conjunto se denomina población, y cada uno de sus individuos se evalúa mediante una función de aptitud para determinar qué tan buenas son las soluciones obtenidas hasta el momento y seleccionar los individuos en las siguientes etapas. Luego de realizada una selección de soluciones, se someten a operadores evolutivos, que pueden afectarlas de manera individual o en conjunto, para generar nuevas soluciones. Las estrategias y configuraciones utilizadas en este punto definen la metodología de búsqueda en el espacio de soluciones, pudiendo obtener una mayor exploración o explotación. Al final de este paso se obtiene un nuevo conjunto de soluciones, que sirven de insumo para obtener una nueva población. El proceso es cíclico, es decir, que al obtener esta población se inician las etapas anteriores de igual manera, con la salvedad de que se espera que esta y las sucesivas generaciones aumenten la valoración de las soluciones del problema [14].

De lo anterior surge una representación en forma de pseudocódigo, útil para acercar los conceptos anteriores a instrucciones. En el algoritmo 1 se muestra un pseudocódigo de referencia.

Algoritmo 1. Pseudocódigo de un algoritmo evolutivo

```
1:  P(0) ← inicializar();
2:  generación ← 0;
3:  while not criterioDeParada do
4:  |   evaluar(P(generación));
5:  |   padres ← seleccionar(P(generación));
6:  |   hijos ← operadoresEvolutivos(padres);
7:  |   nuevaPoblación ← reemplazar(hijos,P(generación));
8:  |   generación++;
9:  |   (P(generación)) ← nuevaPoblación;
10: end
    Result: mejorSolucionEncontrada
```

Resumiendo el funcionamiento de los algoritmos evolutivos, se parte de una población inicial formada por un grupo de individuos que representan las posibles soluciones del problema de optimización. A través de múltiples operaciones evolutivas que afectan a los individuos de forma individual o colectiva, se genera una nueva población a partir de la original, y se espera que los individuos mejor adaptados sobrevivan, guiando así la búsqueda hacia una mejor solución. En cada ciclo de formación de una nueva generación, la función de aptitud otorga un valor a cada individuo, y el algoritmo concluye al cumplir con los criterios de parada establecidos.

2.2.3. Algoritmos evolutivos paralelos

La computación paralela y distribuida ha permitido construir aplicaciones que explotan estas características para desempeñar sus tareas con un mejor rendimiento. Los algoritmos evolutivos no son la excepción y se ha avanzado en diferentes técnicas y modelos para implementarlos de manera paralela y distribuida.

Para problemas no triviales, la realización de un algoritmo evolutivo bajo su forma iterativa básica puede requerir una gran cantidad de recursos computacionales, ya que debe evaluar y realizar operaciones sobre cada individuo de la población. Esto ha hecho necesario el diseño de algoritmos evolutivos eficientes [15].

Para lograr el paralelismo, básicamente, se puede optar por un mapeo del algoritmo clásico a un hardware paralelo, es decir, no hay distinción entre el modelo y la implementación, salvo por la división de tareas de forma concurrente. En este caso, el algoritmo no está estructurado de manera consciente para la paralelización, sino que se busca aumentar la simultaneidad en aquellas instrucciones que así lo permitan. Por otra parte, si se diseña el algoritmo bajo una perspectiva distinta, es decir, estructurando su población, también se hace uso del paralelismo, pero en este caso se modifica la idea básica para incorporar variantes. Estas técnicas de computación paralela se mapean con los conceptos de descomposición funcional o de dominio. Es importante resaltar que sin un soporte durante la ejecución, estas propuestas no podrán separar sus tareas y, por ende, no se percibirá ninguna mejora.

Algunos de los modelos paralelos más conocidos son el modelo maestro-esclavo, el modelo de islas y el modelo celular. Estos modelos estructuran sus componentes bajo determinadas responsabilidades y definición de reglas, logrando taxonomías con dinámicas propias.

El modelo maestro-esclavo es caracterizado por un proceso maestro, que distribuye la carga de trabajo a otros denominados esclavos. El sistema realiza la sincronización de tareas a través del maestro, que es el que conoce el estado general y dirige la ejecución de la metaheurística, ordenando a los esclavos a realizar ciertas operaciones [16].

El modelo de islas introduce la creación y manipulación de múltiples poblaciones, ejecutando de forma simultánea y colaborando entre sí, compartiendo periódicamente soluciones entre ellas, denominando este proceso como migraciones. A su vez, esto habilita la creación de diferentes topologías entre las islas, ya que conceptualmente se pueden modelar como grafos, donde los nodos son las islas y los vértices la capacidad de realizar migraciones [16].

Por último, el modelo celular representa un caso particular del modelo de islas, donde las migraciones son limitadas y cada una de ellas es representada por un único individuo [16].

Dentro de las tecnologías que pueden ser utilizadas para llevar adelante la paralelización se destaca la utilización de *Sockets*, estableciendo una conexión full dúplex entre procesos, la librería *MPI* para el pasaje de mensajes y cooperación entre procesos, y la utilización de internet, haciendo referencia al uso de protocolos que están diseñados para este ecosistema, por ejemplo, *HTTP* [15].

Al momento de desarrollar un software que ejecute de forma paralela y distribuida, se plantean nuevos desafíos, ya sea para aprovechar el hardware paralelo, para comunicar procesos remotos, o incluso para conciliar y sincronizar la ejecución de los componentes, lo que requiere un cuidadoso diseño para garantizar una ejecución confiable.

Uno de los efectos de la paralelización es el aumento de rendimiento, el cual puede ser medido por el *speedup*, según Barr e Hickman en [17], la medida de rendimiento paralelo más común para arquitecturas *MIMD*. El término *MIMD* es propuesto por Flynn en [18] y significa *Multiple Instruction, Multiple Data*, definiendo arquitecturas que se caracterizan por poseer múltiples unidades de procesamiento interconectadas, permitiendo que diferentes programas, con múltiples datos, puedan operar en diferentes procesadores a la vez. En otras palabras, cada procesador puede ejecutar diferentes instrucciones sobre diferentes datos en un momento dado.

Hay diferentes criterios para plantear esta medida, y pueden definirse como *speedup*, *speedup* relativo y *speedup* absoluto, que corresponden a las expresiones (2.2), (2.3) y (2.4) respectivamente.

$$S(p) = \frac{\text{Tiempo para resolver un problema con el código serial más rápido en una computadora paralela específica}}{\text{Tiempo para resolver el mismo problema con el código paralelo usando } p \text{ procesadores en la misma computadora}} \tag{2.2}$$

$$RS(p) = \frac{\text{Tiempo para resolver un problema con el código paralelo en un procesador}}{\text{Tiempo para resolver el mismo problema con el código paralelo en } p \text{ procesadores}} \tag{2.3}$$

$$AS(p) = \frac{\text{Tiempo más rápido de ejecución en cualquier computadora serial}}{\text{Tiempo de ejecución del código paralelo en } p \text{ procesadores de una computadora paralela}} \tag{2.4}$$

Cada una de estas medidas proporciona un punto de vista sobre la mejora obtenida al aumentar la cantidad de unidades de procesamiento. La elección del tipo de *speedup* a utilizar depende del contexto y del propósito de la evaluación.

En cuanto a la mejora obtenida, se puede clasificar según su factor, pudiendo ser sublineal, lineal o superlineal. Una mejora sublineal implica que el tiempo total consumido en todas las máquinas es mayor que el tiempo total de cómputo en una sola

máquina, asumiendo que no hay tiempo ocioso en el algoritmo paralelo. En cambio, el lineal registra el mismo tiempo en su versión paralela y secuencial, lo cual es esperado, ya que implica que el costo de paralelización y sobrecarga no es significativo. Por último, la superlineal corresponde a cuando el tiempo de cómputo del algoritmo paralelo es menor que su versión secuencial, lo que en algunos casos podría llegar a ser posible. La situación ideal es lograr el *speedup* lineal.

Hay otras medidas tales como la eficiencia, que es una normalización del *speedup* como se indica en la fórmula (2.5).

$$e(p) = \frac{S(p)}{p} \tag{2.5}$$

En este caso, una eficiencia igual a uno equivale a una mejora lineal, mientras que valores menores indican mejoras sublineales.

En resumen, los algoritmos evolutivos paralelos ofrecen beneficios esperados, como una mayor eficiencia, la capacidad de cooperar con otros procedimientos de búsqueda y una mayor amplitud en la búsqueda de soluciones o alternativas, todo ello a costa de una mayor complejidad en su implementación, despliegue e infraestructura.

Esta sección se basa en las lecturas de [15], [19] y [16].

CAPÍTULO 3

ARQUITECTURA DE MICROSERVICIOS

Contenido

En este capítulo se presentan conceptos relacionados con la arquitectura de software y el rol del arquitecto. Asimismo, se da un vistazo a la transformación y tendencias de estilos arquitectónicos a lo largo de los años, contrastando sus características. Se dedica una sección final para profundizar en los conceptos y principios de la arquitectura de microservicios, así como a los desafíos que presenta este tipo de diseño y sus posibles soluciones.

3.1. Arquitectura de software

En esta sección se presenta una descripción general del área arquitectura de software y se describe el ciclo de vida de estas arquitecturas.

3.1.1. Descripción general

Para comenzar a describir y adentrarse en los conceptos de arquitectura de software, es conveniente dar alguna definición de ella. Según Bass, Clements y Kazman, "La arquitectura del *software* de un sistema es el conjunto de estructuras necesarias para razonar acerca del sistema, que comprenden elementos de *software*, relaciones entre ellos y propiedades de ambos" [20, p. 4, traducción propia]. Esta definición destaca a la arquitectura como un conjunto de estructuras con relaciones y propiedades específicas, lo que brinda un punto de vista de acuerdo a cómo se compone materialmente. La interacción de estas estructuras define el comportamiento de la arquitectura y, por ende, el del *software*.

Entonces, según esta definición, cada sistema posee una arquitectura y, por lo que es relevante darle importancia, ofreciendo un buen diseño que pueda impactar positivamente en la implementación. En general, no es conveniente que suceda el camino

inverso, en el que la implementación define a la arquitectura. No todas las arquitecturas son buenas, y teniendo en cuenta que pueden permitir o impedir alcanzar los objetivos del sistema, es de suma importancia asumir las decisiones adecuadas al construir el diseño con base en algún criterio, eliminando las elecciones arbitrarias [20]. En resumen, el diseño y evaluación de la arquitectura son fundamentales en el ciclo de vida de un producto.

En [20] se encuentran una serie de lineamientos para aclarar el concepto de qué hace buena a una arquitectura. Las arquitecturas no se catalogan inherentemente como buenas o malas, sino que se ajustan en mayor o menor medida a su propósito. En el libro se diferencian las recomendaciones de proceso y de producto.

En cuanto al proceso, se puede destacar la priorización de requisitos en conjunto con una correcta especificación, de manera que sea posible tomar decisiones adecuadas. En cuanto a la documentación, se recomienda utilizar un método que permita la utilización de diferentes vistas para lograr una comunicación efectiva hacia los diferentes interesados. La evaluación de la arquitectura debe estar necesariamente presente y ser temprana en el ciclo de vida. Por último, los cambios incrementales en la implementación son preferidos, partiendo de un producto mínimo, en donde se incorporan funcionalidades y se descubren las dificultades o desafíos asociados al incremento.

Por otra parte, los lineamientos relacionados con la estructura se refieren a la buena separación de responsabilidades y distribución de funcionalidades. Asimismo, el uso de patrones arquitectónicos habilita la reutilización del conocimiento previo en problemas conocidos, a menos que se traten de requisitos sin precedentes, lo que no es usual. Además, se plantea que el sistema no debe depender de un producto o herramienta comercial, siendo preferible estructurar el diseño para que un cambio no sea costoso. Es útil contar con una cantidad reducida de formas en que los componentes se comunican; al acotar este número de vías de interacción, se logra una mayor comprensión, modificabilidad y confiabilidad. Por último, el arquitecto debe establecer pautas para manejar un conjunto reducido de áreas de contención de recursos, lo que beneficia el cumplimiento de los atributos de calidad.

Luego de acercar los conceptos de diseñar buenas arquitecturas, es natural presentar al responsable de ello, es decir, al arquitecto. Un arquitecto no es un desarrollador experto ni un director de proyectos, sino el principal agente de cambio que articula sus habilidades para lograr resolver los desafíos, adaptarse y entender el negocio y sus necesidades [21].

No hay duda de que la arquitectura es importante para lograr cumplir los requisitos no funcionales del sistema. Por ello, el arquitecto debe conciliar la elección de las habilidades del sistema ("-ility"[1]), ya que pueden competir o afectarse entre sí. Para ello, debe determinar cuáles son más importantes para el producto.

El rol del arquitecto destaca por articular habilidades como el liderazgo, comunicación, negociación, conocimiento técnico y analítico. En conjunto, se convierte en un profesional capaz de tener una visión general del equipo, negocio y clientes, con la capacidad de guiar para poder cumplir los objetivos [22].

Los arquitectos tienen que lidiar con el manejo de características arquitectónicas que abarcan diferentes aspectos de los proyectos de *software*, tales como el rendimiento, elasticidad, escalabilidad, modularidad, entre otros [23].

Un mecanismo efectivo para definir y evaluar los requerimientos de una arquitectura de *software* es la creación de métricas. Una forma de llevarlo a cabo es mediante funciones de aptitud, tomando los mismos conceptos de la computación evolutiva, pero para construir una arquitectura evolutiva a partir de ella. El cambio de la arquitectura se guía por la evaluación de las características mediante las métricas [23].

[1] Término empleado por ser el sufijo de requisitos no funcionales. En español, podría ser "-idad".

3.1.2. Ciclo de vida de la arquitectura del software

El diseño de la arquitectura de *software* es una parte fundamental del ciclo de vida de la arquitectura. Es el lugar donde se traducen los requisitos a un diseño y este a una implementación. Los arquitectos deben preocuparse no solo en este punto, sino en los asuntos detallados en [22], que son descritos a continuación y se visualizan en la figura 3.1.

Requisitos arquitectónicos

Entre todos los requisitos del sistema, se destacan un conjunto de ellos que impactan de forma particular en el diseño, ya que guían la elección de estructuras y componentes para satisfacer estas necesidades acordadas. Estos requisitos no incluyen solamente las funcionalidades de mayor importancia del sistema, sino que comprenden aspectos como atributos de calidad, tales como rendimiento, alta disponibilidad, seguridad, entre otros. Estos requisitos son los que conducen el diseño de la arquitectura.

Diseño arquitectónico

El diseño se puede entender como la traducción desde el plano de las necesidades (requisitos), al plano de las soluciones, mediante la utilización de estructuras, componentes, marcos de trabajo o código.

Tomando en consideración lo expuesto por Sommerville [24], el diseño arquitectónico se posiciona como un proceso creativo, donde se da lugar a la solución de los requisitos funcionales y no funcionales. Por lo tanto, este proceso no es sistemático ni responde a un método formulado. De lo anterior se puede concluir que un buen diseño se considera como tal, si satisface los requisitos establecidos.

Documentación arquitectónica

La creación de un documento que describa las estructuras provenientes del diseño es el desafío de esta etapa. En proyectos de gran porte o complejidad, la existencia de una documentación aporta un valor agregado fundamental para comprender, mantener, evolucionar o transferir la solución. Desafortunadamente, en la práctica, a menudo se convierte en una tarea evitada, lo que no implica que deje de ser importante.

Otro aspecto fundamental a destacar como parte de la necesidad de la documentación, se corresponde a la correcta comunicación que se espera lograr entre desarrolladores e interesados. De esta forma se asegura un entendimiento de los conceptos clave a implementar. Las formas de llevar adelante esta labor pueden ser diversas, más o menos efectivas, pero está claro que a medida que la complejidad y porte del producto a desarrollar crece, la manera en que se lleva adelante la documentación y comunicación de ella se torna más difícil, por lo que existen herramientas útiles como C4 para realizarlo [25].

Modelo de diagramas C4

El modelo C4 es un conjunto de diagramas y abstracciones jerárquicas, que se presentan como una herramienta para ayudar a diagramar y comunicar la arquitectura de un *software* [26].

Tomando conceptos de UML, C4 provee no solo un vocabulario para definir sistemas, sino que también un método para descomponer la visión en cuatro niveles que describen diferentes ideas: contexto, contenedor, componente y código [25].

El nivel de contexto es el que da una visión más general de todo el sistema. En este se describe cómo el sistema interactúa con otros sistemas y con los usuarios.

El nivel de contenedor expone las aplicaciones y bases de datos que contienen los sistemas; esos contenedores son artefactos desplegables y ejecutables independientes. Para comprender cómo se organizan las aplicaciones se utiliza el nivel de componentes, donde se visualizan agrupaciones de funcionalidades encapsuladas, lo que se traduce en una colección de implementaciones de código. Por último, se presenta como una vista opcional el código, donde se muestran y describen interacciones entre clases y módulos [25].

Una comparación interesante para ejemplificar las vistas, se da con el cuerpo humano, donde diferentes especialistas se enfocan en determinadas partes del todo, pudiendo obtenerse distintas estructuras fisiológicas de acuerdo al interés de cada uno [20].

Los sistemas modernos suelen ser complejos y, por lo tanto, comprenderlos todo de una vez no es una tarea trivial; de aquí se enfatiza la importancia de poder estructurar la documentación en diferentes vistas, logrando restringir la atención en estructuras más pequeñas [20].

Un caso donde la inclusión de diagramas realizados mediante C4 fue una decisión acertada es *Spotify*. En esta organización cuentan con miles de servicios interconectados, teniendo cientos de equipos que los mantienen. Ellos necesitaban una herramienta para homogeneizar su documentación y diagramas de arquitectura para poder así comunicarla de mejor manera. Como parte de los beneficios alcanzados por esto, se destaca el entendimiento del ciclo de vida, correcta visualización de diagramas y relación entre componentes, mejora en la colaboración entre equipos y la obtención de una descripción general de alto nivel para analizar patrones de diseño e inspeccionar deuda técnica [27].

Evaluación arquitectónica

En este punto del ciclo de vida, se han tomado una serie de decisiones basadas en el análisis y diseño, por lo que es natural y necesario someter la arquitectura del sistema a alguna evaluación, para verificar su rendimiento y compararlo con el deseado. Al relacionar esto con la definición de arquitecturas evolutivas de Ford, Parsons y Kua: "Una arquitectura evolutiva admite un cambio guiado e incremental en múltiples dimensiones." [28, p. 6, traducción propia], se denota la importancia de la evaluación continua para lograr un cambio guiado. En [28] se presentan las funciones de aptitud para corroborar las características del sistema a lo largo de los cambios. Las arquitecturas evolutivas son importantes en la actualidad, dado que mantienen actualizado el propósito del sistema, permitiendo sobrevivir en entornos cambiantes y competitivos.

Implementación arquitectónica

En la última etapa del ciclo de vida de la arquitectura de *software* se encuentra la implementación de la arquitectura. Esta actividad implica una retroalimentación del diseño y una evaluación para realizar mejoras, además de revisar los requisitos y su posible impacto en cambios a modo de evolución del sistema.

Es importante que la implementación sea fiel al diseño, ya que de lo contrario se podrían perder las cualidades que éste garantizaba. Esta verificación es crucial para asegurar que los requisitos que llevaron a modelar la arquitectura se respeten y estén presentes en la implementación.

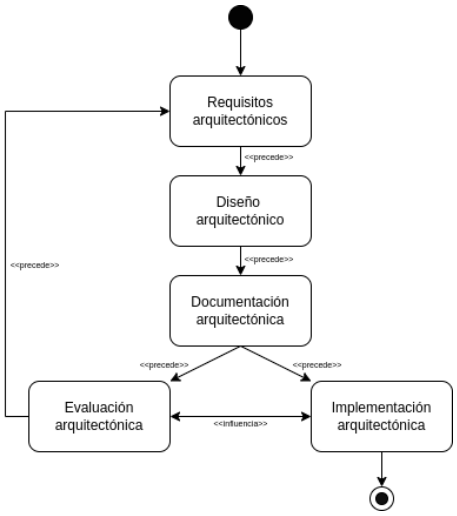

Figura 3.1: Actividades del ciclo de vida de la arquitectura de *software*.[2]

3.2. Patrones de arquitectura de software

Las arquitecturas obsoletas se transforman en un riesgo y pueden ocasionar una desventaja competitiva en las organizaciones. Por otra parte, una arquitectura moderna se convierte en un socio técnico, no solo por la tecnología en sí misma y sus beneficios, sino también por los aspectos organizativos de los equipos que son parte del proceso [29]. Se puede concluir que utilizando metodologías modernas y adecuadas a cada situación, en conjunto con una elección acertada del patrón de arquitectura, se puede marcar la diferencia y tener un valor agregado por sobre otras propuestas.

En esta sección se estudian diferentes patrones de arquitectura de *software* para entender sus características [30].

3.2.1. Arquitectura monolítica

El monolito es una unidad de despliegue en donde todas las funcionalidades se encuentran juntas [31]. A su vez, se distinguen diferentes tipos de sistemas monolíticos y algunos se describen a continuación.

Los monolitos de un solo proceso son aquellos sistemas en donde todo el código es desplegado y ejecutado en un único proceso. Todo el código es empaquetado como una unidad.

Como caso particular del anterior, se encuentra el monolito modular. Esta variación consiste en la separación lógica de componentes que potencialmente pueden trabajar y colaborar de manera independiente. El inconveniente aún radica en el despliegue, ya que permanecen combinados como un solo proceso. Aquí se vislumbra una ventaja o mejora respecto a la separación de responsabilidades e independencia, pero aún debe madurar para lograr un despliegue sin restricciones. Bajo este subconjunto se pueden diseñar o usar un sinfín de patrones para organizar los módulos, por ejemplo, el patrón de arquitectura más común es el de arquitectura en capas, donde cada componente

[2]Diagrama basado en el libro [22].

está organizado en una capa horizontal, desempeñando una tarea o rol específico en la aplicación [30].

A su vez, el acceso a los datos es un tema de interés y cuidado, ya que originalmente el monolito accede a una única base de datos, lo cual puede adaptarse en esta variante a un conjunto de instancias de bases de datos, asociadas a determinados módulos.

Por otra parte, se presenta al monolito distribuido, que se define como un sistema de múltiples servicios que requiere un despliegue en conjunto. Richards [30] manifiesta que este tipo de sistemas combina las desventajas de los sistemas distribuidos y de los monolitos de un único proceso.

Como principales desventajas y riesgos que presentan los monolitos, se ubican los problemas de acoplamiento tanto a nivel de código e implementación como de despliegue, situación que es agravada en donde hay equipos numerosos y con diferentes responsabilidades.

Por otra parte, este tipo de sistemas cuenta con diversas ventajas, muchos de ellos asociados a la simplicidad. Si un sistema es más simple, su entendimiento es más natural y tareas tales como el despliegue, corrección, revisión son llevadas adelante con menor esfuerzo. La reutilización de código en estos sistemas puede darse mediante el uso de bibliotecas comunes o la definición de código base, lo que nuevamente puede ser atractivo.

Entonces, no necesariamente un monolito es sinónimo de legado u obsoleto, tan solo se presenta como una decisión arquitectónica que puede o no ser la más adecuada para la situación. El arquitecto es el responsable de evaluar si las cualidades de este patrón se ajustan a las necesidades del sistema.

Característica	Valoración
Agilidad	Baja
Facilidad de despliegue	Baja
Testeabilidad	Alta
Performance	Baja
Escalabilidad	Baja
Facilidad de desarrollo	Alta

Tabla 3.1: Características de una arquitectura monolítica.[3]

3.2.2. Arquitectura de microservicios

Las arquitecturas de microservicios han ganado terreno en la industria como una alternativa viable a las arquitecturas monolíticas y las orientadas a servicios [30].

Para comprender este tipo de arquitectura, se introducen algunos conceptos. La noción de unidades desplegadas por separado es uno de los principios de ese patrón, donde cada componente es entregado por separado, permitiendo independencia, escalabilidad y desacoplamiento en las aplicaciones [30].

Otro de los conceptos fundamentales de este patrón corresponde a los componentes de servicios, cuyo tamaño o granularidad se define según las necesidades del proyecto o equipo de desarrollo. Este enfoque busca establecer de manera clara los límites de cada componente, no solo en términos de recursos, sino también de responsabilidades [32].

[3]Tabla basada en el libro [30]

Uno de los beneficios más visibles de este tipo de arquitectura es la división en módulos, lo que fomenta y facilita el cambio en el *software* [32].

El concepto de arquitectura distribuida también caracteriza a este patrón arquitectónico, significando que todos los componentes están desacoplados e interactúan mediante algún protocolo remoto (por ejemplo, *JMS*, *AMQP*, *REST*, *SOAP*). Esto otorga gran capacidad de escalado [30].

La colaboración de los procesos independientes no es un tema menor y típicamente se opta por dos tipos de patrones para solucionar esta problemática en este tipo de arquitectura, la coreografía y la orquestación. La coreografía se basa en el pasaje de mensajes de forma asíncrona entre todos los servicios y la orquestación establece una jerarquía entre los despliegues, donde alguno asume el rol de controlador del flujo y lo realiza sincrónicamente [33].

A su vez, el manejo de transacciones distribuidas también es un punto de diseño donde analizar, ya que en este tipo de sistemas hay varias soluciones. El patrón de arquitectura *Saga* permite manejar el flujo de operaciones entre microservicios y es ajustable a los dos patrones de colaboración nombrados anteriormente, coreografía y orquestación [34].

Por otra parte, los patrones *Sidecar* y *Service Discovery* son importantes para gestionar la interacción entre los servicios.

El patrón *Sidecar* consiste en desplegar componentes auxiliares junto al servicio principal dentro del mismo contenedor, pero manteniéndolos separados. Este enfoque facilita la incorporación de funcionalidades mediante un mecanismo de *proxy*, comunicándose con otros servicios sin invadir el código del servicio principal. De esta forma, el patrón ayuda a mantener la separación de responsabilidades y a implementar características comunes de forma consistente en múltiples servicios.

Por otro lado, el patrón *Service Discovery* aborda la problemática de localizar y conectarse con los servicios dentro de una arquitectura distribuida. Este patrón permite a los servicios encontrar dinámicamente la dirección de otros servicios, especialmente en entornos donde las instancias pueden escalar horizontalmente o ser reubicadas. Este patrón es especialmente útil para mejorar la resiliencia y la flexibilidad del sistema.

La complejidad del diseño y desarrollo de una solución basada en microservicios obliga a conocer y adoptar las buenas prácticas de este tipo de sistemas, para lograr un desarrollo más efectivo. Dentro de las recomendaciones más importantes se destaca el uso separado de almacenamiento o bases de datos por servicios, la independencia de la construcción y despliegue de cada microservicio, el correcto planteamiento y asignación de responsabilidades en donde tenga un único propósito por contenedor, el uso de contenedores, diseño de servicios sin estado, la adopción de un diseño orientado a eventos y el uso de un orquestador para manejar los despliegues [35].

Característica	Valoración
Agilidad	Alta
Facilidad de despliegue	Alta
Testeabilidad	Alta
Performance	Baja
Escalabilidad	Alta
Facilidad de desarrollo	Alta

Tabla 3.2: Características de una arquitectura de microservicios.[4]

[4]Tabla basada en el libro [30]

3.2.3. Arquitectura de microkernel

La arquitectura de *microkernel* o también conocida como el patrón de arquitectura de *plug-in* es un modelo que se ajusta a una aplicación basada en el producto. Una aplicación basada en el producto típicamente se refiere a un paquete de *software* que es obtenido y utilizado como un producto de terceros [30]. El patrón de *microkernel* aporta la capacidad de extender las funcionalidades de acuerdo a las necesidades específicas del negocio, desarrollando módulos propios.

En esta arquitectura se distinguen dos divisiones claras de tipos de componentes arquitectónicos, el *core system* y los *plug-in modules*. La lógica de la aplicación está dividida entre los componentes *plug-in* y el *core*, garantizando extensibilidad, flexibilidad y aislamiento de las funcionalidades y lógica personalizada [30]. El *core* del sistema contiene un conjunto mínimo de funcionalidades para mantener el sistema operacional; por otra parte, los módulos son componentes independientes que se especializan en proveer una funcionalidad específica.

Por último, este tipo de diseño necesita que el *core* del sistema conozca y pueda comunicarse con los *plug-in*, existiendo la necesidad de definir contratos entre ambos componentes, los cuales pueden ser estandarizados con alguna tecnología particular [30].

En la figura 3.2 se aprecia un diagrama que expresa la interacción entre los componentes de estas arquitecturas.

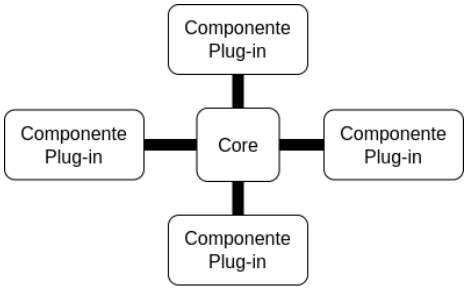

Figura 3.2: Diagrama de patrón de arquitectura Microkernel.[5]

Este tipo de arquitectura puede encontrarse embebida como parte de otra, como por ejemplo, dentro de una arquitectura de microservicios, de esta forma se obtiene sinergia entre ambos paradigmas. En el caso particular de las arquitecturas de microservicios, estas proveen un soporte para facilitar la evolución y desarrollo incremental.

Característica	Valoración
Agilidad	Alta
Facilidad de despliegue	Alta
Testeabilidad	Alta
Performance	Alta
Escalabilidad	Baja
Facilidad de desarrollo	Baja

Tabla 3.3: Características de una arquitectura de microkernel.[6]

[5] Diagrama basado en el libro [30].

3.2.4. Arquitectura conducida por eventos

Las arquitecturas conducidas por eventos son un patrón popular para los sistemas asíncronos distribuidos, permitiendo un alto grado de escalabilidad [30].

Este patrón está formado por dos grandes topologías, el mediador y el *broker*. El mediador es usado cuando se necesita orquestar múltiples pasos basados en eventos a través de un componente que sirve de mediador; en cambio, el *broker* sirve cuando los eventos son desencadenados y manejados sin la presencia de un componente central [30].

Como consideración a esta arquitectura, se destaca que es relativamente complejo de implementar por su naturaleza asíncrona. Además, hay que manejar diversos desafíos y dificultades en términos de implantación, para resolver la disponibilidad de procesos remotos, reintentos, fallas de comunicación y *timeout*, entre otras problemáticas. Los componentes son fuertemente desacoplados y puede traducirse en un desafío el hecho de modelar una unidad transaccional a través de ellos [30].

Por último, la mayor dificultad se centra en la creación, mantenimiento y gobernanza de los contratos de los eventos. Se sugiere la utilización de estándares de formato para aplicar a los datos entregados y recibidos como eventos, como por ejemplo, el formato JSON, en conjunto con una política de versionado de los mismos [30].

Característica	Valoración
Agilidad	Alta
Facilidad de despliegue	Alta
Testeabilidad	Baja
Performance	Alta
Escalabilidad	Alta
Facilidad de desarrollo	Baja

Tabla 3.4: Características de una arquitectura conducida por eventos.[7]

[6] Tabla basada en el libro [30]
[7] Tabla basada en el libro [30]

3.3. Microservicios como parte de una solución *cloud native*

Los elementos fundamentales de una arquitectura *cloud native* incluyen conceptos como: arquitectura de microservicios, arquitectura *serverless*, arquitectura conducida por eventos, computación en la nube, contenedores, desarrollo ágil y *DevOps* [36]. Entre estos elementos, la arquitectura de microservicios es frecuentemente reconocida como uno de los pilares fundamentales de *cloud native* [37].

Es importante destacar que los términos *cloud native* y *cloud computing* no deben confundirse. El uso de la nube no implica automáticamente la adopción de técnicas y herramientas que califiquen como nativas. Los principios de la computación nativa en la nube trascienden el simple uso de infraestructura en la nube; no obstante, dicha infraestructura puede ser un buen punto de partida para un cambio gradual, que permita obtener valor comercial en el proceso [38].

Adicionalmente, algunas definiciones de *cloud native* se centran en las tecnologías empleadas para empaquetar aplicaciones en contenedores, en la arquitectura de microservicios para manejar el despliegue, y en la implementación de infraestructura elástica y procesos *DevOps* para facilitar la entrega continua [39].

Por todo esto, resulta difícil pensar en una solución diseñada bajo principios *cloud native* sin que incorpore una arquitectura de microservicios como base. En la figura 3.3 se puede apreciar un diagrama con los elementos fundamentales de una arquitectura *cloud native*.

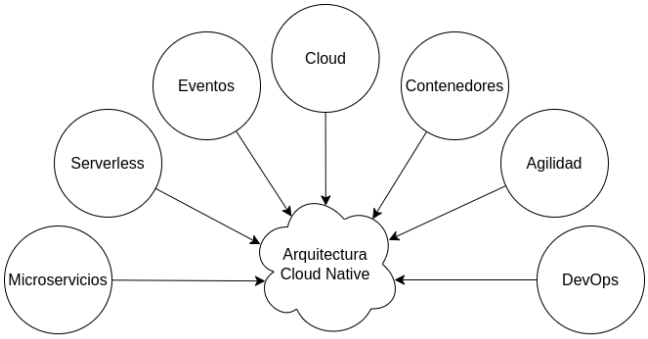

Figura 3.3: Elementos de una arquitectura *cloud native*.[8]

3.3.1. Cloud Native Computing Foundation

La *Cloud Native Computing Foundation* (CNCF) es una organización formada en el año 2015 en respuesta a la alta adopción de las tecnologías *cloud native*. Este proyecto nace por parte de la *Linux Foundation* y está formado por numerosos miembros [1].

El objetivo de la *CNCF* es fomentar y promover la adopción de tecnologías nativas en la nube, y al mismo tiempo garantizar que sean accesibles, disponibles y confiables [1].

La CNCF busca impulsar la adopción de este paradigma. Por medio del mantenimiento de un ecosistema de proyectos innovadores de código abierto, logra democratizar patrones modernos aptos para la nube [2]. Un ejemplo visible de ello es el

[8] Diagrama basado en el libro [36].

landscape[9] que agrupa las diferentes tecnologías de acuerdo a su utilidad dentro de la plataforma, ofreciendo visibilidad de las diferentes herramientas que pueden utilizarse, en conjunto con un respaldo de una comunidad a la hora de optar por ellas.

3.3.2. Beneficios del modelo *cloud native*

Existen claros beneficios a la hora de adoptar un desarrollo nativo para la nube.

Esta arquitectura ofrece independencia, permitiendo desplegar aplicaciones sin conflictos entre ellas. Además, una de las características más notables se relaciona con la resiliencia, que corresponde a la habilidad de permanecer disponible a pesar de la interrupción o eventos adversos que puedan ocurrir en términos de infraestructura.

El diseño también es beneficiado, ya que la utilización de contenedores ofrece un estándar de uso y distribución, lo que ayuda a la portabilidad e interoperabilidad. Como consecuencia de esto, las migraciones entre proveedores también se ven facilitadas.

La agilidad es crucial muchas veces en términos de negocio, ya que determina una ventaja competitiva y las aplicaciones *cloud native* colaboran ofreciendo despliegues flexibles a través de la red, lo que permite realizar iteraciones e incrementos con mayor asiduidad. La automatización no escapa a lo mencionado anteriormente, ya que ofrece mayor capacidad y funcionalidad para desarrollar y desplegar.

Por último, también se encuentra la capacidad de realizar actualizaciones sin tiempo de indisponibilidad, gracias a la orquestación de contenedores, que permite intercambiar los artefactos y dirigir el tráfico desde los clientes de manera segura [1].

[9] `https://landscape.cncf.io/`

CAPÍTULO 4

REVISIÓN DE ANTECEDENTES

Contenido

Es importante conocer soluciones que permitan desarrollar o configurar ejecuciones de algoritmos evolutivos, ya que esto permite conocer y analizar sus características principales y tomarlas en cuenta para el desarrollo de esta plataforma.

Además de considerar los productos disponibles para el público en general, también se analizan artículos que presentan propuestas para modelar algoritmos evolutivos bajo diferentes arquitecturas.

A través de este estudio, se incorporan diferentes aportes al diseño, a modo de enriquecer la propuesta de este proyecto.

4.1. Herramientas de optimización

En esta sección se analizan algunas herramientas diseñadas para asistir en la definición y ejecución de algoritmos evolutivos. Si bien estas soluciones no se basan en un diseño de microservicios, tienen una larga trayectoria en el campo y, a lo largo de sus diferentes versiones, han logrado mejorar su producto, comprendiendo sus problemáticas o puntos de mejora.

En este proyecto se aprovecha el conocimiento y experiencia del diseño de estas aplicaciones, reutilizándolo y adaptándolo a las necesidades específicas del proyecto, lo que permite enriquecer la solución final.

4.1.1. jMetal

jMetal es un *framework* de optimización multiobjetivo con metaheurísticas. Su código fuente se encuentra publicado en *GitHub*[1] y el mismo es *open source*. Esta

[1] `https://github.com/jMetal/jMetal`

herramienta, construida en Java, fue desarrollada por ingenieros para su uso en investigaciones. Fueron motivados por el afán de aplicar conocimiento en paralelismo a metaheurísticas multiobjetivo. El *software* disponible con el que contaban estaba escrito en el lenguaje *C* y carecía del paradigma de orientación a objetos, sumado a una curva de aprendizaje alta al momento de entender su código fuente. Estos ingenieros evaluaron sus necesidades y determinaron que necesitaban una herramienta fácil de usar, flexible, extensible y portable para usar en sus investigaciones, por lo que comenzaron el proyecto desde cero en el año 2006 [40].

El proyecto está estructurado en varios subproyectos que corresponden a las clases fundamentales, codificaciones de soluciones, implementaciones de algoritmos, problemas de referencia, experimentación y visualización, extensiones paralelas, diseño y configuración automática, así como a algoritmos basados en componentes.

Este *framework* ha estado bajo constantes cambios con el objetivo de obtener una mejor herramienta, realizando una revisión de su arquitectura y diseño. Además, ha sido objeto de estudio la experiencia del usuario, realizando esfuerzos por mejorarla.

Dentro de las funcionalidades que han sido trabajadas, se destaca el diseño de un *template* de algoritmo para lograr una alta configurabilidad y así otorgar versatilidad a la hora de crear una ejecución. Luego, la visualización de los resultados en gráficos y páginas web generadas también fue una funcionalidad que aporta gran valor a aquel que utilice el *framework*. La autoconfiguración de parámetros también fue prevista y, por último, un soporte eficiente para realizar tareas paralelas también fue analizado. El paralelismo puede darse en modos sincrónicos o asincrónicos y bajo un paradigma de ejecución en múltiples hilos [41].

El modelado del dominio del conjunto de entidades participantes en la definición de un problema de optimización en *jMetal* ayuda a comprender cómo conceptualizar cada uno de los componentes que intervienen en una ejecución. Si bien lo que se puede ver en la documentación y publicaciones es un diagrama de clases, que está enfocado a un paradigma de orientación a objetos, esto no deja de ser fuente de inspiración para poder diseñar una plataforma que se nutra de estos conceptos. Dentro de las constantes mejoras que ha tenido *jMetal* a lo largo del tiempo, el rediseño del modelo ha sido favorable, otorgando mayor simplicidad al conjunto de clases [42], [43].

4.1.2. HeuristicLab

HeuristicLab es un *framework* para trabajar con heurísticas y algoritmos evolutivos, desarrollado y mantenido desde 2002 por miembros del *HEAL* (*Heuristic and Evolutionary Algorithms Laboratory*) de la *University of Applied Sciences Upper Austria* [44]. Este proyecto de código abierto se encuentra disponible en *GitHub*[2].

Entre las funcionalidades destacadas de esta herramienta se encuentra su interfaz gráfica de usuario. Como se puede apreciar en la figura 4.1, cuenta con una gran cantidad de recursos tanto para definir experimentos como para interpretar los resultados. Esta interfaz se presenta en forma de ventana en el sistema operativo y requiere una instalación previa para utilizar el programa completo.

[2] `https://github.com/heal-research`

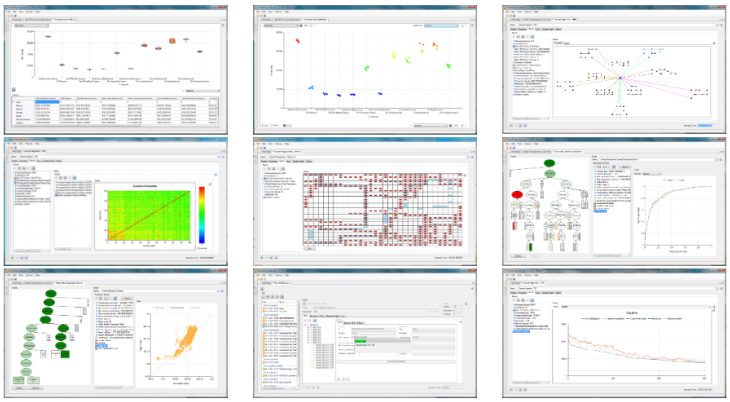

Figura 4.1: Capturas de la interfaz gráfica de HeuristicLab

La instalación de esta aplicación define una serie de requisitos del sistema. En particular, depende de *Microsoft .NET Framework 4.7.2*[3], lo que dificulta su instalación en entornos basados en UNIX. A pesar de que existen alternativas para estos sistemas operativos, como la utilización de *Mono*[4], el soporte es limitado en estas circunstancias.

Otra de las funcionalidades a destacar de la herramienta es su capacidad de prototipado de algoritmos en conjunto con una visualización interactiva de los experimentos y simulaciones.

En resumen, este proyecto destaca por brindar un ambiente integrado de trabajo, configurable y orientado a garantizar una buena experiencia de usuario.

4.1.3. ParadisEO

ParadisEO es un *framework* de optimización heurística de código abierto, cuyo código fuente se encuentra disponible en *GitHub*[5]. Esta herramienta tiene como principal objetivo ayudar a los usuarios a escribir sus propios algoritmos de optimización estocásticos de manera rápida y sencilla [45]. La madurez y trayectoria del proyecto se ven reflejadas en su evolución y trabajo desde el año 1999 hasta la actualidad.

Una de las características más destacables de *ParadisEO* es su enfoque en la eficiencia de implementación de sus algoritmos de resolución, lo cual se logra mediante un diseño modular, una amplia base de código de componentes, herramientas de diseño y selección de algoritmos automatizadas, y diversas opciones de paralelización para aumentar la velocidad de ejecución. Las motivaciones que tiene el proyecto de fondo se relacionan con la construcción de un *framework* donde la reutilización de código se destaque y, en conjunto con un profundo análisis del dominio, la solución permita al desarrollador escribir poco código. Características como la flexibilidad y adaptabilidad fueron tomadas en cuenta como atributos clave de la herramienta. La robustez y eficiencia también fueron tomadas en cuenta [46].

La arquitectura de este proyecto, combinada con la amplia variedad de algoritmos disponibles, permite un ensamblado a medida, con el objetivo de facilitar la exploración de diferentes algoritmos. Esto garantiza un esfuerzo mínimo al intercambiar componentes para experimentar y probar alternativas. En resumen, se trata de un diseño modular, multicapa y orientado a componentes. La comunicación en paralelo es realizada mediante librerías como *MPI* [46], [47].

[3] `https://dotnet.microsoft.com/download/dotnet-framework/net472`
[4] `https://www.mono-project.com/`
[5] `https://github.com/nojhan/paradiseo`

El diseño basado en componentes garantiza que la personalización sea tan sencilla como elegir los operadores evolutivos y conectarlos en la ejecución. En el caso de problemas clásicos con representaciones de soluciones estándar, basta con la cooperación de estos componentes con la definición de la función de aptitud. Sin embargo, para problemas que requieren una representación especial de las soluciones, es necesario no solo codificar la función de evaluación, sino también definir variantes de los operadores capaces de manejar la forma de las soluciones.

Debido a su rendimiento, el lenguaje de programación utilizado para su desarrollo es *C++*. Según Dreo, Liefooghe, Verel et al. [47], *ParadisEO* es uno de los pocos *frameworks* de optimización que hacen uso de este lenguaje. La elección de este lenguaje compilado y de bajo nivel les permite crear componentes de alto rendimiento y que hacen un mejor uso del *hardware* subyacente.

En relación con la experiencia del usuario, cabe destacar que este producto no cuenta con una interfaz de usuario, sino que la interacción se realiza mediante la definición de código. Además, se percibe que su instalación y uso pueden ser complejos, debido a las dependencias necesarias para compilar y al propio uso del *framework*, que también requiere conocimientos en *C++*, un lenguaje que puede presentar mayores desafíos respecto a otros.

4.2. Síntesis y análisis de artículos relevantes

A modo de conocer y relevar el conocimiento actual del tema y especialmente relacionado con las temáticas a abordar, se realiza una lectura de diferentes artículos que refieren a la implementación de algoritmos evolutivos bajo diferentes arquitecturas.

Algunas de las publicaciones de mayor relevancia se presentan en la tabla 4.1.

Título	Año	Referencia
«Specifying evolutionary algorithms in XML»	2003	[48]
«Service oriented evolutionary algorithms»	2013	[49]
«A Parallel Genetic Algorithm Framework for Cloud Computing Applications»	2014	[50]
«Towards a component-based software architecture for genetic algorithms»	2014	[51]
«Parallel Genetic Algorithms in the Cloud»	2017	[52]
«Introducing an Event-Based Architecture for Concurrent and Distributed Evolutionary Algorithms»	2018	[53]
«A modern, event-based architecture for distributed evolutionary algorithms»	2018	[54]
«Mapping evolutionary algorithms to a reactive, stateless architecture»	2018	[55]
«Scalable distributed evolutionary algorithm orchestration using Docker containers»	2020	[56]

Tabla 4.1: Tabla de publicaciones consultadas de mayor interés en la revisión de antecedentes

Los aspectos seleccionados para el análisis surgen de su aparición recurrente en la literatura consultada, así como de la constatación de sus beneficios para la solución propuesta.

La reutilización y la interoperabilidad se destacan como pilares fundamentales para la construcción de sistemas flexibles y modulares, capaces de adaptarse e integrarse, reduciendo esfuerzos en la etapa de desarrollo.

Por su parte, el uso de contenedores y herramientas de orquestación representa una solución moderna que garantiza despliegues eficientes y escalables. Asimismo, la comunicación y el paralelismo son elementos clave para mejorar significativamente el rendimiento en entornos distribuidos.

Finalmente, los paradigmas orientados a la nube abren nuevas posibilidades para integrar estas tecnologías en infraestructuras avanzadas, optimizando el aprovechamiento de sus recursos.

A continuación se procede con la presentación de la metodología utilizada en esta sección y el resumen de las ideas más importantes que se recolectan para hacer la solución de este proyecto.

4.2.1. Metodología de síntesis y análisis

Para llevar a cabo la síntesis y el análisis de artículos relevantes, se sigue un enfoque que incluye la búsqueda, selección y revisión de publicaciones sobre algoritmos evolutivos y propuestas arquitectónicas para este tipo de *software*.

Inicialmente, se realizan búsquedas en la plataforma *ResearchGate*[6], utilizando palabras clave como "algoritmos evolutivos", "arquitecturas de microservicios" y sus combinaciones. A partir de los artículos encontrados, se analizan las referencias citadas en estos trabajos, lo que permite identificar publicaciones adicionales de interés y construir una red de conocimiento más amplia sobre el tema.

Durante este proceso de revisión, se identifican ciertas temáticas recurrentes, entre las que destacan: la reutilización de componentes y la utilización de *frameworks*, la interoperabilidad e integración de sistemas, el uso de contenedores y herramientas de orquestación para un despliegue eficiente de aplicaciones, y los desafíos en la comunicación y paralelismo en entornos distribuidos.

Además, las publicaciones relacionadas con arquitecturas orientadas a servicios, orientadas a eventos y enfocadas a paradigmas en la nube, en el contexto de la ejecución de algoritmos evolutivos, resultan especialmente valiosas.

A partir de la lectura de cada publicación, se busca unificar conceptualmente los diferentes enfoques y perspectivas con el objetivo de construir un panorama general sobre las principales problemáticas y soluciones propuestas en la literatura. Este análisis permite identificar los aportes más relevantes para este proyecto y agruparlos en los aspectos que se desarrollan en las siguientes secciones.

4.2.2. Reutilización

La reutilización permite optimizar los tiempos de desarrollo al integrar componentes que resuelven problemáticas específicas, logrando una reducción de esfuerzos.

Como se detalló anteriormente, existen varias herramientas o *frameworks* que permiten resolver o modelar instancias de problemas de optimización. Esta abundancia de *software* de dominio público adolece de ciertos problemas que los investigadores enfrentan comúnmente, tal es el caso de la falta de flexibilidad y reutilización de las herramientas [57].

Estas soluciones ya han explotado el mecanismo de reutilización, a través de la definición de un *framework*. Un *framework* comúnmente es un conjunto de piezas de código que define un diseño abstracto para solucionar una familia de problemas

[6] `https://www.researchgate.net/`

relacionados. Además, brinda la capacidad de reutilizar el mismo código en diferentes implementaciones [58]; no obstante, la reutilización abarca mucho más que esto.

La diversidad de soluciones provoca aumentos de costos a la hora de querer reutilizar o integrar código, debido a incompatibilidades o bien porque es muy difícil extraer ciertas implementaciones utilizadas dentro de un determinado *framework* [49], [59].

Hay que tener en cuenta que, muchas veces, los *frameworks* disponibles superponen capacidades y ofrecen características similares, lo que ante la falta de coordinación y estandarización se traduce en esfuerzo duplicado [60].

En la tabla 4.2, se presenta una comparación entre varios *frameworks* de optimización.

Nombre	Diseño*	Lenguaje	Comunicación	Licencia
ECJ	OO	Java	Sockets	AFL
MALLBA	OO	C++	MPI	Freeware
jMetal	OO	Java	N/A	GNU/LGPL
DREAM	OO	Java	DRM	GNU/GPL
ParadisEO	OO	C++	MPI	CeCILL
HeuristicLab	OO/OP	.NET	Web-Services	GNU/GPL
METCO	OO	C++	MPI	N/A
JCLEC	OO	Java	N/A	GNU/GPL
Algorithm::Evol.	OO	Perl	N/A	GNU/GPL
gridUFO	OS	Java	Web-services	N/A
OSGiLiath	OS	Java	Web-services	GNU/LGLP

Tabla 4.2: Comparativa de diferentes *frameworks* para ejecutar algoritmos evolutivos.[7]
* (OO=orientado a objetos, OP=orientado a plug-in, OS=orientado a servicios.)

De la comparación, se evidencia que hay una marcada tendencia a un diseño orientado a objetos y lenguaje Java. Por otra parte, los métodos de comunicación son diversos, pero tienden a utilizar interfaces para definir sus interacciones. Respecto a esto último, existe una propuesta de un modelado genérico para describir algoritmos evolutivos. Esta forma de representación está basada en *XML* y busca independencia tecnológica [48]. Esta idea es interesante, ya que se busca modelar el dominio de forma genérica, desacoplando las implementaciones concretas y estableciendo un contrato a la hora de compartir información.

Los *frameworks* también cuidan aspectos como la extensibilidad y flexibilidad, siendo características claves para la mantenibilidad. El nivel de extensibilidad refleja lo fácil que es ampliar o mejorar el *software*. La flexibilidad, por otra parte, se centra en la capacidad de que pueda ser adaptado o usarse de una manera diferente a la que fue diseñado originalmente [61].

Por tanto, cuando se habla de reutilización, hay que enfocarse en cómo se reutilizan las piezas de código, lo que se enfoca a la realización de un *framework* o bien a las características de extensibilidad y flexibilidad, que se atacan con un diseño que prevea estos atributos de calidad.

Las funcionalidades provistas por estos marcos de trabajo, además de agregar capacidad de monitoreo, análisis, paralelismo y muchas otras características satelitales a la creación y ejecución estricta de algoritmos evolutivos, pueden ser valiosas particularmente para contextos tales como la industria o la academia [60].

[7] Tabla basada en publicaciones [49], [59].

En general, los *frameworks* utilizados para este tipo de problemas componen la solución interactuando de acuerdo a cómo se presenta en la figura 4.2. Allí se ve cómo una aplicación basa su codificación en la integración al *framework* para resolver un problema. El *framework* provee un conjunto de metaheurísticas y codificaciones que son utilizadas para resolver de manera genérica cualquier problema.

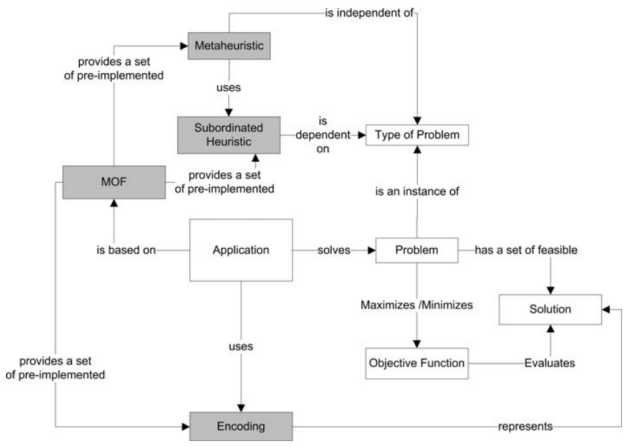

Figura 4.2: Mapa conceptual del uso de un framework de optimización.[8]

Las ventajas de utilizar un *framework* de este tipo, primero que nada, están en la eficiencia y reducción de tiempos de implementación, agregando funcionalidades o características fácilmente. Además, pueden presentar herramientas adicionales como las de monitoreo, estadística, reporte, soporte para ejecución en paralelo, entre otras, que ayudan a agregar funcionalidades interesantes con mínimo esfuerzo. Al utilizar soluciones codificadas dentro de las librerías, las mismas fueron probadas por más usuarios y esto reduce la presencia de errores de programación. Por último, los usuarios con menos experiencia en el campo pueden hacer uso del *software* para aplicar estas metodologías [60].

Otra ventaja tiene que ver con que ellos proveen un conjunto correcto, totalmente funcional y versionado de técnicas de metaheurísticas y sus variantes, dando facilidades para la implementación y definiciones de problemas con sus correspondientes soluciones [60].

Como contraparte negativa, la curva de aprendizaje es un factor a tener en cuenta, ya que trasciende al conocimiento de metaheurísticas, debido a que hay que aprender a utilizar la herramienta de manera adecuada para lograr definir problemas exitosamente. En otras oportunidades hay inflexibilidad para adaptar variantes de las metaheurísticas. También puede introducirse una complejidad a la hora de probar el *software* y sus dependencias. La elección del *framework* adecuado es crucial, ya que puede haber un costo alto al migrar de uno a otro [60].

En resumen, la reutilización acelera el desarrollo de aplicaciones destinadas a la ejecución de algoritmos evolutivos, al reducir esfuerzos y mejorar la modularidad del sistema. El uso de *frameworks* proporciona una forma eficaz de distribuir soluciones listas para implementar con un mínimo esfuerzo de desarrollo, aunque requiere familiarizarse con su correcta utilización.

[8] Diagrama basado en el libro [60].

4.2.3. Interoperabilidad

Se entiende por interoperabilidad a la capacidad que tiene un sistema de poder interactuar con otros, realizando una comunicación e intercambio de información efectivamente [61].

Hay algunas consideraciones a tener en cuenta al momento de diseñar una solución que posea esta característica. Algunas de ellas se detallan a continuación.

El diseño debe contemplar componentes bajamente acoplados, pensados para ser reutilizables e intercambiables [51]. Una forma de lograr esto es mediante el uso de interfaces de entrada y de salida para los componentes. Estas interfaces deben clasificarse de acuerdo a su utilización, como internas o de negocio y externas o de usuario [51].

Una idea que complementa a lo anterior tiene que ver con el diseño de sistemas modulares, basados en una arquitectura de *microkernel*. Con este tipo de arquitectura se pueden extender fácilmente las funcionalidades de un *software*. Según Wagner, Winkler, Pitzer et al. [62], la arquitectura *microkernel* es el siguiente paso en la definición de sistemas de optimización heurística.

Para lograr un sistema modular, no basta con separar un proyecto en módulos, sino que ellos deben ser componentes autónomos ejecutables. La complejidad de modularizar sistemas grandes puede darse a la hora de articular cada módulo como una unidad ejecutable. Otra complejidad está dada por la correcta separación de responsabilidades, las cuales deben ser únicas y bien definidas [62]. La definición de interfaces ayuda al ensamblado entre módulos y el sistema central [62].

Por último, un desarrollo *open source* diseñado para ser extensible, es favorecido en casos donde la comunidad puede utilizar el producto o enriquecerlo [62].

Entonces, la interoperabilidad garantiza que los sistemas puedan integrarse y comunicarse de manera efectiva en entornos heterogéneos. Esta capacidad es fundamental para construir soluciones escalables y flexibles.

En resumen, el diseño de una arquitectura modular, con componentes desacoplados que se comuniquen a través de interfaces estandarizadas y que cuenten con una buena separación de responsabilidades, contribuye a la capacidad de interoperabilidad del sistema.

4.2.4. Uso de contenedores y orquestación

El uso de contenedores otorga flexibilidad en términos de despliegue de las aplicaciones. Además, su uso ofrece garantía de portabilidad, característica que describe cuán eficientemente y efectivamente un *software* puede adaptarse o ser transferido de un ambiente a otro [61].

En la figura 4.3 se aprecian las diferentes formas de disponibilizar una aplicación.

En la primera forma, denominada *bare metal*, las aplicaciones son desplegadas directamente en una máquina dedicada y el sistema operativo debe brindar un ambiente apropiado para su ejecución. Bajo esta modalidad puede existir incompatibilidad de dependencias en el sistema.

Luego, la siguiente forma de definir un ambiente es mediante máquinas virtuales. Al esquema anterior se agrega un hipervisor, que crea una capa de abstracción sobre el *hardware* y atiende a múltiples sistemas operativos.

Por otra parte, el uso de contenedores no implica la puesta en marcha de un sistema operativo completo, por ello es más ligero que ejecutar una máquina virtual. El contenedor ejecuta como un proceso sobre el sistema operativo, aislándose entre sí y ejecutando dentro de su propio espacio de nombres. Una máquina virtual ejecuta su propio sistema operativo, que puede ser diferente al del anfitrión, lo que no sucede con

los contenedores [63]. Así como las máquinas virtuales necesitan de un hipervisor, los contenedores requieren un *container engine*.

Por último, los enfoques anteriores pueden ser combinados y poseer contenedores dentro de máquinas virtuales. Esto puede ser útil cuando se desean explotar determinados beneficios de cada paradigma.

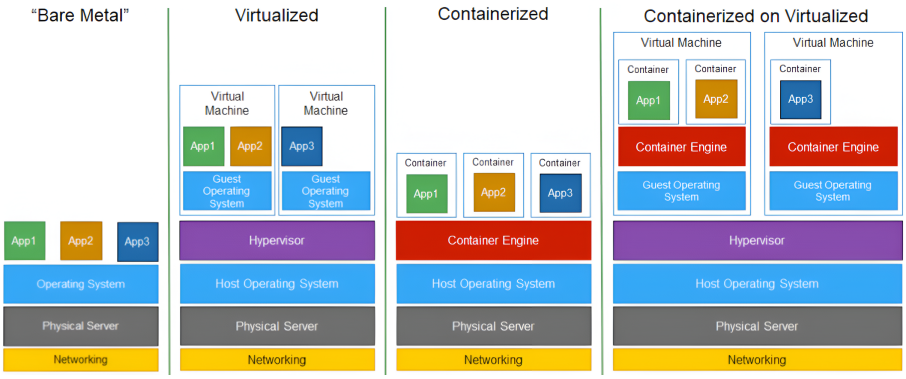

Figura 4.3: Comparación de despliegue de aplicaciones en ambientes con y sin virtualización, con uso de contenedores y ambos esquemas a la vez.[9]

Introducida la solución de contenedores, resta entender si es natural construir un sistema que opere con algoritmos evolutivos bajo esta forma.

Existen propuestas para implementar algoritmos evolutivos distribuidos, haciendo uso de contenedores y acompañados de herramientas de orquestación. En particular, hay una solución que trabaja sobre problemas reales, como la planificación manufacturera, utilizando estas tecnologías [56]. Además, es importante mencionar que Dziurzanski, Zhao, Przewozniczek et al. [56] reconocen que un despliegue sobre *Kubernetes* provee mayores facilidades para un desarrollo en la nube.

Por otra parte, la concurrencia es un hecho al momento de construir algoritmos evolutivos paralelos, lo que implica la definición de un mecanismo de colaboración. En el ámbito de los contenedores, hay pruebas de que la eficiencia en la comunicación puede verse impactada, dado que se realiza a través de la red y puede ser más lenta que otras alternativas [65].

Por último, los paradigmas de programación, en conjunto con la infraestructura que se posee, definen la forma de llevar adelante la implementación de algoritmos evolutivos paralelos. Estos factores impactan directamente en el diseño e implementación del sistema [55].

Dentro de las opciones analizadas, la utilización de microservicios, contenedores y una comunicación bajo un paradigma de *publish-subscribe*, con tecnologías *REST*, ha logrado *speed-up* superlineal en experimentos realizados en una plataforma *Kubernetes* [66]. De igual manera, en otros estudios también se propone el uso de microservicios con el mismo tipo de comunicación, asegurando que los servicios queden desacoplados y así poder ensayar diferentes topologías [67].

En resumen, el uso de contenedores y herramientas de orquestación contribuye a la mejora de la interoperabilidad y a la reutilización. Estas tecnologías proporcionan un entorno estandarizado que facilita la integración de componentes y su portabilidad entre diferentes plataformas y permiten la creación de arquitecturas modulares.

[9]Diagrama basado en publicación [64].

Sin embargo, el rendimiento puede verse penalizado, por tratarse de una solución de virtualización.

4.2.5. Comunicación y paralelismo

La comunicación y el paralelismo son aspectos fundamentales en el diseño de sistemas distribuidos. Una adecuada selección de los mecanismos de implementación puede generar mejoras significativas en el rendimiento y la escalabilidad de los algoritmos evolutivos.

El hecho de paralelizar tareas implica la definición de los mecanismos de comunicación y colaboración entre procesos. La topología y arquitectura de la solución, acompañada de la elección tecnológica, definen el camino de cómo implementar estos mecanismos.

Existen diferentes propuestas para resolver la comunicación entre los componentes de un algoritmo evolutivo. A continuación se presentan algunas de ellas.

La utilización de bases de datos para alojar el modelo de manera global es una alternativa. Esto establece un mecanismo de coordinación entre los procesos, garantizando la consistencia de los datos mediante el uso de transacciones [68].

Por otra parte, la comunicación mediante un esquema *P2P* ha sido puesta a prueba con resultados exitosos en términos de escalabilidad [69]. Esto implica un diseño adecuado de los componentes para soportar este protocolo.

También existen casos donde se resuelve el paralelismo mediante *grid-computing*, interfaces *MPI* y *gRPC* [70]. En estos casos es usual el uso de bibliotecas que implementan estas soluciones [49], [59].

La utilización de técnicas *map-reduce* también es comúnmente utilizada para paralelizar este tipo de algoritmos [50].

Por último, se pueden encontrar arquitecturas que basan la comunicación de sus componentes a través de colas de mensajes. En estos casos, la propia población es la que se modela en dichos mensajes [54].

En resumen, en esta sección se revisan los distintos enfoques para implementar mecanismos de comunicación y paralelismo. Las alternativas discutidas proporcionan una base para diseñar soluciones que optimicen tanto la colaboración entre procesos como la ejecución eficiente de tareas paralelas.

4.2.6. Paradigmas orientados a la nube

Para llevar adelante tareas de desarrollo, despliegue y ejecución de algoritmos genéticos paralelos en la nube, no basta con la adopción de un conjunto de tecnologías concretas, tales como *Docker*, *CoreOS* y *RabitMQ*, sino que es necesario acompañar las actividades de los usuarios o desarrolladores por medio de un flujo de trabajo. Este flujo debe modelar las actividades necesarias relativas a la creación y utilización del sistema [52], [71].

La utilización de este paradigma no ofrece solamente beneficios, también establece algunas restricciones con las que lidiar, entre ellas, los tiempos de carga al desplegar los componentes. Una vez contemplado y manejado este proceso, la solución responde eficientemente [72].

Otro desafío que existe al migrar a un paradigma nativo en la nube es el aumento de dificultad, al trabajar con sistemas distribuidos. Esto requiere una adopción de herramientas y procesos para poder mitigarla.

En cuanto al paralelismo en los algoritmos evolutivos, se suele usar MPI, pero en los proveedores de la nube no se garantiza su correcto funcionamiento y no logra escalar entre múltiples *clusters* [72], [73]. Una alternativa para coordinar tareas en un

ambiente en la nube es *map-reduce*. Esta técnica es soportada por los proveedores en la nube, pero puede requerir un ajuste en el diseño de las soluciones [72].

Por otra parte, una arquitectura con un enfoque orientado a eventos, puede conducir a una solución nativa para la nube, que incluso puede converger a una solución *serverless*. La dificultad asociada a esto es la adaptación de las funciones a un modelo sin estado [53].

Por otra parte, referente a cuestiones operativas, es necesario monitorizar los costos asociados a la infraestructura. En casos donde se utilice un servicio de pago es aún más prioritario, en comparación con una nube privada *on-premise*. El monitoreo sirve para administrar conscientemente los recursos, pero requiere establecer procesos o herramientas para llevar adelante la tarea.

Las diferentes restricciones generan resistencia al cambio cultural asociado a prácticas y procesos [37], no obstante, conociendo estas dificultades se pueden explotar los beneficios del paradigma.

En resumen, se analizan las oportunidades y desafíos que ofrecen los paradigmas nativos de la nube. Este análisis resalta cómo las arquitecturas basadas en eventos y servicios pueden mejorar la flexibilidad y escalabilidad, pero requieren una adaptación de las funciones utilizadas por los algoritmos evolutivos a este modelo.

4.3. Resumen de aportes para este proyecto

En este capítulo se analizaron diferentes herramientas de uso público, que sirven para ejecutar algoritmos evolutivos, además de descubrir diferentes estudios y propuestas que se enfocan en diversos diseños y tecnologías para definir nuevas arquitecturas en este tipo de sistemas.

Los diseños conceptuales de *jMetal* [42] y los *frameworks* de optimización [60] son importantes para comprender y nutrir el modelo y diseño de este proyecto.

Las problemáticas relacionadas con la flexibilidad y con la portabilidad definen un camino orientado a la definición de servicios independientes, con sus respectivas interfaces [49].

Los mecanismos de comunicación propuestos son varios, pero uno enfocado a eventos puede ser interesante para garantizar un mayor grado de desacoplamiento. Una arquitectura de *microkernel* también es apropiada para ofrecer una lógica central y permitir la definición de componentes propios.

El uso de contenedores es beneficioso en términos de interoperabilidad y portabilidad, y más en un contexto de microservicios. La complejidad de administración y despliegue de este tipo de solución orientada a la *cloud*, requiere automatización y definición de un flujo de trabajo para los diferentes actores que operen con la plataforma.

CAPÍTULO 5

ANÁLISIS Y DISEÑO DE LA SOLUCIÓN

Contenido

En general, el proceso de diseño y la definición de arquitectura de *software* no son actividades diferentes, ya que ambos involucran la toma de decisiones y operan con las habilidades y materiales disponibles, para satisfacer los requerimientos y restricciones del producto [22].

Conceptos tales como el propósito, requisitos, restricciones y preocupaciones arquitectónicas son los que conducen y guían al proyecto [22]. Por lo tanto, en esta sección se establecen y analizan los requisitos del sistema, tanto a nivel de casos de uso como de atributos de calidad, lo que resulta en un diseño arquitectónico.

5.1. Análisis de requisitos

En esta sección se detallan los requisitos del sistema, definiendo el modelo y comportamiento del mismo. La forma de presentar estos conceptos está dada por la definición de las funcionalidades, atributos de calidad y restricciones.

Los requisitos funcionales se expresan mediante casos de uso y estos describen qué puede realizar el sistema. Los requisitos no funcionales son presentados mediante escenarios donde se cuantifica el comportamiento del sistema y las restricciones fijan o limitan decisiones en el diseño.

5.1.1. Modelo de dominio

Luego de reconocer los diferentes conceptos esenciales que participan en la definición de un algoritmo evolutivo, producto del análisis del modelado y diseño de las herramientas utilizadas para este fin, se construye el modelo de dominio presentado en la figura 5.1.

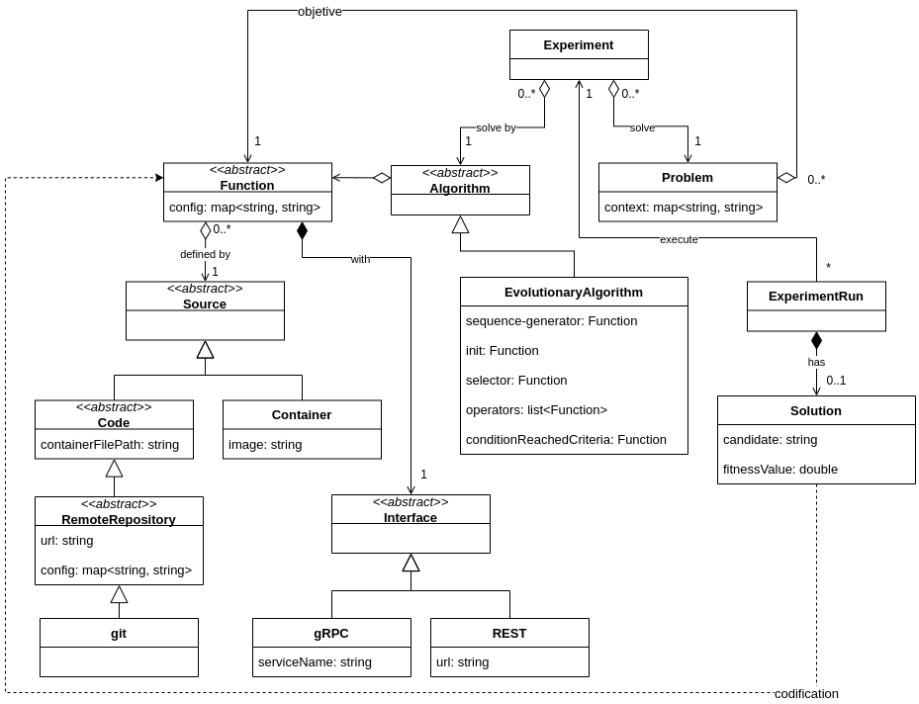

Figura 5.1: Diagrama de modelo de dominio

Cada elemento de esta realidad puede ser definido por separado o embebido dentro de otro, respetando la taxonomía modelada. De esta forma, un *Experiment* es el componente más completo, que describe la representación de lo que será la ejecución de un *Algorithm* en el contexto de resolución de un *Problem*.

El *Algorithm* es una representación abstracta del mecanismo de resolución del problema, por lo cual, requiere una implementación concreta que otorgue un comportamiento adecuado. En este proyecto, la única especialización de este elemento es la de *EvolutionaryAlgorithm*, formada por un conjunto de *Function* con roles específicos dentro del esquema de modelado de este tipo de metaheurística.

Por su parte, un *Problem* también es compuesto por una *Function*, cuyo rol corresponde a la función objetivo del problema, a la que se desea hallar su óptimo. Un *Problem* agrega a la definición de una *Function*, agregando directrices de configuración definidas como *context*. Estos parámetros son enviados a la función objetivo y, para aquellos casos donde el componente soporte su recepción, se logra una adaptación sin necesidad de modificar código fuente o tener que compilar.

Respecto al elemento *Function*, está definido por un *Source* y una *Interface*, ambos abstractos. El *Function* cuenta con dos variantes en esta primera entrega, el *Code* y el *Container*. A su vez, el *Code* está implementado por un repositorio remoto llamado *RemoteRepository*, el cual prevé diferentes tecnologías, como por ejemplo es el caso de *git*.

Todos los elementos anteriores especifican y describen a los diferentes recursos, que componen un experimento. Mientras que un *Experiment* corresponde a un descriptor, la instancia de ejecución de él está representada por medio de un *ExperimentRun*. Como resultado de correr un *ExperimentRun*, se obtiene una *Solution*, que ofrece la solución candidata al problema definido, obtenida en el contexto de la aplicación del

algoritmo evolutivo. Respecto a la representación de la solución candidata, se genera un vínculo entre ella y la *Function* que ofició de objetivo, ya que mediante su contrato establece la codificación del candidato.

Para homogeneizar el tratamiento de estas entidades dentro de la plataforma, se agregan los conceptos de *Resource* y *Specification*. Estos abarcan a todos los elementos anteriores, exceptuando *ExperimentRun* y *Solution*.

Un *Resource* define la capacidad de creación, identificación, versionado, modificación y eliminación en cada instancia dentro de la plataforma. Mientras que la *Specification* representa la estructura de cada entidad.

En la figura 5.2 se presenta la relación entre todos estos conceptos, ejemplificado solamente para las entidades *Experiment* y *Problem*.

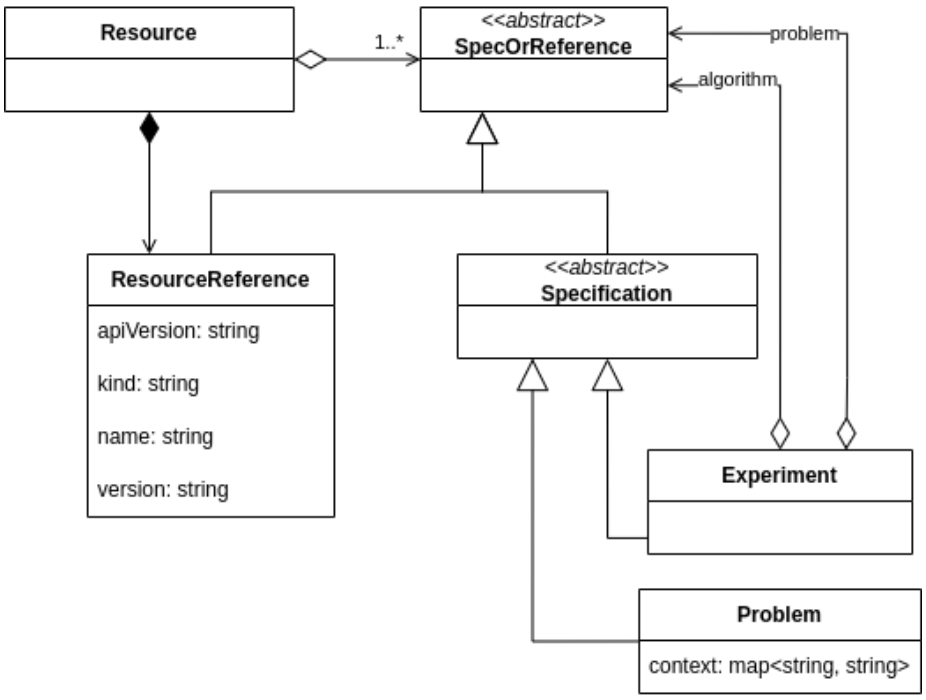

Figura 5.2: Concepto de *Resource* y *Specification*

Bajo esta solución, la plataforma administra objetos de tipo *Resource*, donde cada uno es identificable mediante el conjunto de datos definidos en *ResourceReference*. A su vez, se introduce el patrón de diseño *Composite*, ya que *Resource* está conformado o bien por referencias a otros elementos creados anteriormente o por nuevas definiciones presentadas a través de su especificación.

En el ejemplo de la figura 5.2, un *Experiment* es administrado por la plataforma mediante un *Resource* de *kind experiment* y está conformado por referencias o definiciones a *Problem* y *Algorithm*, dado que este elemento está compuesto por ellos, como se presentó en el diagrama de la figura 5.1.

A su vez, los elementos que heredan de *Specification* definen sus atributos específicos, además de su relación con los demás elementos de la plataforma.

5.1.2. Modelo de casos de uso

En la figura 5.3 se presentan los casos de uso más relevantes del sistema.

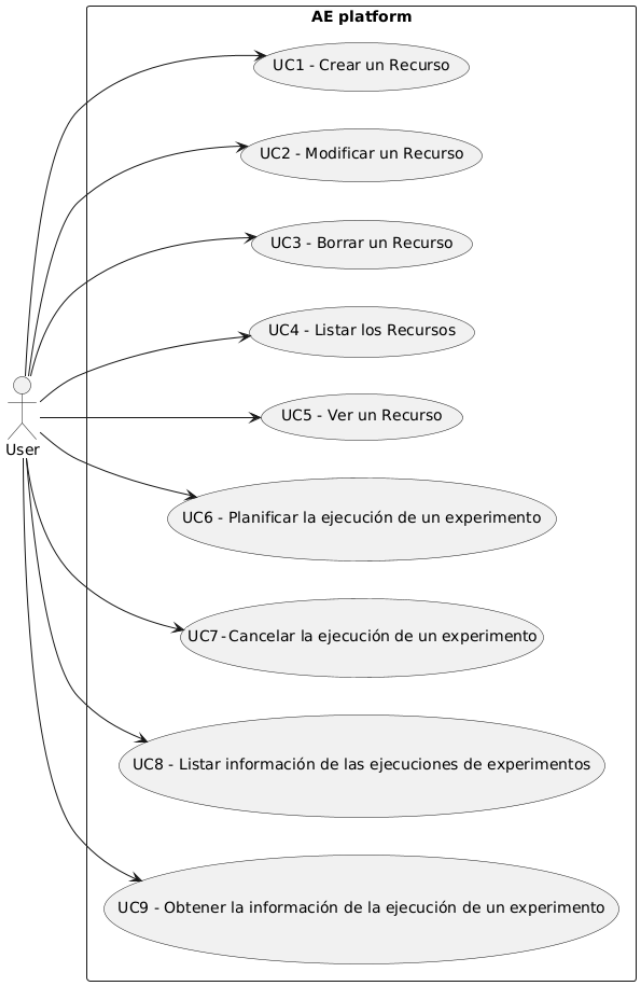

Figura 5.3: Diagrama de casos de uso

A continuación se presenta una descripción de cada uno de ellos.

UC-1: Crear un Recurso

Un usuario que cuenta con la representación de un Recurso envía esta información al sistema para su creación. Cada recurso posee un tipo y, de acuerdo a este, se define la estructura de la información.

Cada recurso es identificable dentro del sistema y es persistido para su posterior utilización. En la definición de un nuevo Recurso se puede referenciar a otro que ya haya sido creado.

Dentro de los datos que son genéricos a todos los recursos, se encuentra la versión del mismo, que no debe coincidir con ninguna otra versión del mismo tipo de recurso

en el sistema.

UC-2: Modificar un Recurso

Un usuario que cuenta con la representación de un Recurso envía esta información al sistema para su modificación. Se sustituye el Recurso anterior por la nueva representación enviada.

Esta operación es admitida siempre y cuando el recurso involucrado no haya sido utilizado como parte de una ejecución previa.

UC-3: Borrar un Recurso

Un usuario que cuenta con el identificador de un Recurso creado en el sistema, solicita la eliminación de este dato. Una vez realizado el borrado, no se podrá utilizar como parte de otro Recurso.

En caso de que el Recurso que se está intentando eliminar sea referenciado por otro, la operación no es admitida por la plataforma; por tanto, si se desea continuar con el proceso de eliminación, primero se deben eliminar los Recursos que contengan referencias a este.

Si el recurso a eliminar fue utilizado como parte de la definición de la ejecución de un experimento, el recurso es conservado en la plataforma, pero de forma separada a los demás recursos de su mismo tipo.

UC-4: Listar los Recursos

Un usuario lista los recursos existentes en el sistema. Al momento de solicitar el listado se pueden indicar los filtros que se deseen, como por ejemplo el tipo del Recurso.

Una vez resuelta la petición, se despliega una lista con la información de los recursos que cumplan con las condiciones de búsqueda establecidas en los filtros.

Esta información es retornada de forma paginada para poder escalar sin dificultades ante grandes cantidades de definiciones de recursos.

UC-5: Ver un Recurso

Un usuario obtiene la información detallada de un recurso indicando los valores identificativos de él, que son el nombre, versión y tipo.

UC-6: Planificar la ejecución de un experimento

Un usuario que cuenta con la representación de un experimento envía esta información al sistema para planificar su ejecución. Esta información involucra datos como el tipo de algoritmo evolutivo a utilizar, los parámetros de configuración, la función objetivo a optimizar, los criterios de terminación, la fecha y hora deseada de inicio, mecanismos de notificación y un identificador del experimento, entre otras cosas.

UC-7: Cancelar la ejecución de un experimento

Un usuario cancela la planificación o ejecución de un experimento. Si el mismo se encuentra planificado, pero no ha comenzado, simplemente se despacha. En cambio, si se encuentra en ejecución, se determina si su detención es realizada de inmediato o se solicita que culmine lo antes posible, como, por ejemplo, en la siguiente iteración del experimento.

UC-8: Listar información de las ejecuciones de experimentos

Un usuario lista los experimentos existentes en el sistema de forma paginada y ordenados de acuerdo a su fecha de creación.

Se despliega la información fija de los experimentos y el estado en el momento de la consulta.

UC-9: Obtener la información de la ejecución de un experimento

Un usuario recibe la información detallada de un experimento dado. Junto a la configuración y estado del mismo, se obtienen los eventos asociados al experimento.

5.1.3. Atributos de calidad

Los atributos de calidad califican el desempeño de los requisitos funcionales o del producto en general. Su obtención se alcanza mediante la correcta elaboración de estructuras y comportamientos de la arquitectura [20].

Una forma de describir este tipo de atributos es mediante escenarios. Este tipo de definición se compone de seis partes: estímulo, fuente del estímulo, respuesta, medida de la respuesta, ambiente y artefacto, tal como se puede apreciar en la figura 5.4.

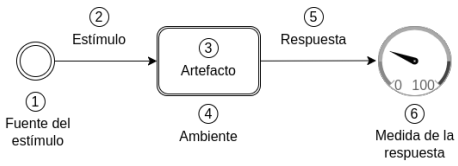

Figura 5.4: Partes del escenario de los atributos de calidad.[1]

El uso de un escenario para describir los atributos de calidad de un sistema define el flujo de un evento que afecta al sistema. Comenzando por la fuente del estímulo, se presenta la entidad que genera una condición que altera el sistema, en particular, algún artefacto que lo compone. El ambiente es el conjunto de condiciones en donde el estímulo ocurre a la vez de su respuesta. Por último, la métrica evalúa el comportamiento del accionar del sistema en relación con la generación de la respuesta [74].

A continuación se presentan diferentes atributos de calidad expresados bajo esta forma.

[1] Diagrama basado en el libro [74].

QA-1: Escalabilidad

El sistema debe ser capaz de ajustar su capacidad de procesamiento ante experimentos que requieran mayor nivel de cómputo paralelo. Ante situaciones de baja demanda, el sistema debe disminuir las instancias de procesamiento para evitar malgastar el uso de recursos.

Estímulo:	Generación de información a procesar
Fuente del estímulo:	La información generada en las diferentes etapas y funciones del algoritmo evolutivo
Ambiente:	El sistema se encuentra desplegado y con recursos disponibles para crear nuevos microservicios
Artefacto:	El sistema
Respuesta:	Se ajusta la cantidad de réplicas de los operadores del experimento
Métrica de la respuesta:	La cantidad de datos sin procesar para un tipo de operador es proporcional al número de réplicas de los operadores y no debe superar las 100 unidades

Tabla 5.1: Definición del escenario 1 para escalabilidad.

QA-2: Evolucionabilidad

Con esta característica, el sistema está preparado para la adición de nuevas funcionalidades sin afectar negativamente a las demás y, a su vez, sin incurrir en grandes desafíos o dificultades al momento de modificar el código existente.

Estímulo:	Necesidad de agregar una nueva metaheurística de optimización
Fuente del estímulo:	Un usuario que colabora con una nueva funcionalidad en el *core* del sistema
Ambiente:	Etapa de diseño de una nueva versión del sistema
Artefacto:	El código fuente del sistema
Respuesta:	La modificación del código y empaquetado de la nueva versión, sin efectos secundarios en las demás funcionalidades existentes
Métrica de la respuesta:	La cantidad de líneas de código modificadas en el código fuente del sistema no debe superar el orden de los cientos (sin contar las nuevas clases introducidas)

Tabla 5.2: Definición del escenario 1 para evolucionabilidad.

QA-3: Resiliencia

La plataforma es capaz de recuperarse ante eventos adversos e inesperados.

Al contar con resiliencia, el sistema no solamente puede manejar situaciones de falla, sino que al suceder, puede seguir operando y así lograr una buena disponibilidad.

Estímulo:	Una falla inesperada en un componente del sistema
Fuente del estímulo:	Error de programación
Ambiente:	El sistema se encuentra operando
Artefacto:	Un operador
Respuesta:	Reinicio de un operador
Métrica de la respuesta:	El operador debe poder volver a iniciar y procesar información en un período de tiempo no mayor al orden de las decenas de segundos

Tabla 5.3: Definición del escenario 1 para resiliencia.

QA-4: Interoperabilidad

La interoperabilidad en el contexto de este desarrollo refiere a la capacidad de integrar diversos componentes de forma sencilla y práctica, por medio de una interfaz definida y no importando la tecnología de fondo de cada componente agregado.

Estímulo:	La ejecución de un componente
Fuente del estímulo:	Adición de un operador dentro de un experimento
Ambiente:	El sistema desplegado normalmente
Artefacto:	El sistema
Respuesta:	El operador se comunica con el resto del sistema
Métrica de la respuesta:	El operador desarrollado de forma externa interactúa con el resto del sistema a través de sus interfaces

Tabla 5.4: Definición del escenario 1 para interoperabilidad.

QA-5: Modificabilidad

La capacidad de modificar el comportamiento de la ejecución de un experimento es medida por medio de este atributo de calidad.

El sistema debe proveer un mecanismo sencillo para modificar la implementación de los diferentes algoritmos, por medio de la definición de operadores.

Estímulo:	Necesidad de agregar un nuevo operador para un algoritmo evolutivo
Fuente del estímulo:	Un usuario que utiliza el sistema
Ambiente:	El sistema se encuentra desplegado
Artefacto:	El sistema
Respuesta:	Se crea un recurso en el sistema que representa el nuevo operador evolutivo
Métrica de la respuesta:	La cantidad de líneas modificadas en el manifiesto que declara el algoritmo debe ser del orden de las decenas

Tabla 5.5: Definición del escenario 1 para modificabilidad.

5.1.4. Restricciones

Aquí se detallan las restricciones del sistema. Las restricciones son decisiones de diseño que se han tomado de antemano y no es posible negociarlas [22].

CON-1: Arquitectura de microservicios

El objetivo del proyecto es evaluar el uso de una arquitectura basada en microservicios en la definición de una plataforma capaz de modelar y ejecutar algoritmos evolutivos. Por lo tanto, se establece como requisito obligatorio el diseño de la plataforma bajo estas condiciones arquitectónicas.

CON-2: Uso de tecnologías libres

El producto final estará compuesto por un conjunto de tecnologías libres, lo que permite que sean modificadas y distribuidas sin restricciones comerciales.

Por otra parte, al construir un producto a partir de este tipo de tecnologías, el resultado también preservará estas cualidades. Esto es valioso, ya que el uso de este estará bajo los lineamientos de *GPL2*.

5.2. Caso de estudio

Para visualizar un escenario común que sirva como caso de estudio, se utiliza el problema *OneMax*.

El problema consiste en maximizar la suma de un conjunto $x = \{x_1, x_2, \cdots, x_n\}$, dónde $x_i \in \{0, 1\}$.

Entonces, el problema se traduce en maximizar la función $f(x) = \sum_{i=1}^{n} x_i$.

Este es un problema de optimización trivial, ya que su solución es conocida de antemano. Por esta razón, se utiliza para realizar pruebas de funcionamiento y rendimiento sobre algún método de optimización.

El problema *OneMax* podría ser utilizado para resolver un caso específico en el que se pueda establecer una correspondencia con el modelo. Un ejemplo sería un escenario donde se desea maximizar el precio de un carrito de compras con N lugares y artículos de valor constante V, teniendo en *stock* al menos N artículos para agregar al carrito. En este caso, la solución sería simplemente llenar el carrito.

Cuando se realiza la correspondencia entre el modelo conceptual y la representación de la tira binaria, lo que se lleva a cabo es lo que se conoce como codificación. En el ejemplo, cada posición de la tira binaria representa un lugar en el carrito y la aparición de un 0 o un 1 indica si tiene o no un artículo en esa posición.

Es importante enfatizar que este tipo de problema no es útil para modelar problemas de optimización complejos, pero debido a su simplicidad, es elegido para introducir y analizar su resolución mediante algoritmos evolutivos.

La técnica de búsqueda de soluciones que utilizan los algoritmos evolutivos es estocástica y es utilizada para problemas complejos de optimización, en donde es difícil aplicar otros métodos.

Retomando el caso de estudio, el mismo puede verse como un *Experiment*, ya que establece un *Problem*, que está definido por la *Function OneMax* y su mecanismo de resolución *EvolutionaryAlgorihm*.

Para entender cómo el *EvolutionaryAlgorihm* se ejecuta, se puede analizar bajo este caso de estudio. Primero debe generar una población con diversos individuos, tal como aparece representado en la figura 5.5.

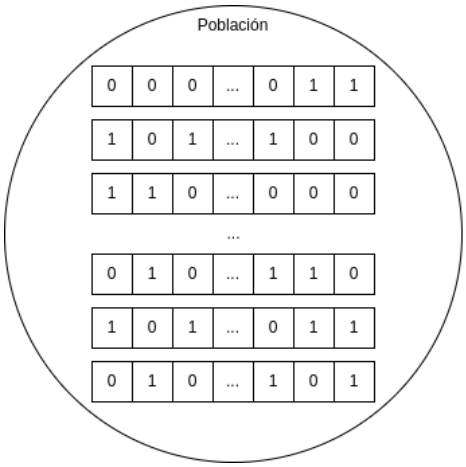

Figura 5.5: Representación de población de un AE

Para lograr esta población, el *EvolutionaryAlgorihm* genera los individuos de forma aleatoria por medio de la *Function init*.

Una vez obtenida la población inicial, se realizan operaciones sobre estos individuos, lo que da como resultado una nueva generación. Este modelo generacional define que una generación es el resultado de una iteración del algoritmo evolutivo, es decir, el algoritmo trabaja sistemáticamente sobre la población actual N para obtener una población $N + 1$, donde se espera que los individuos hayan cambiado de manera que guíen la búsqueda hacia el óptimo.

En la figura 5.6 se aprecia cómo la población evoluciona entre generaciones.

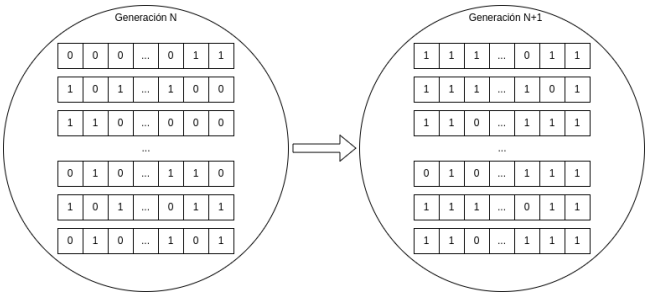

Figura 5.6: Representación de generaciones en un AE

Lo que resta por presentar son los operadores que llevan adelante el cambio de los individuos, los cuales son modelados por *Function*. Para eso, en la figura 5.7 se pueden ver los operadores de cruzamiento y mutación realizando cambios en los individuos.

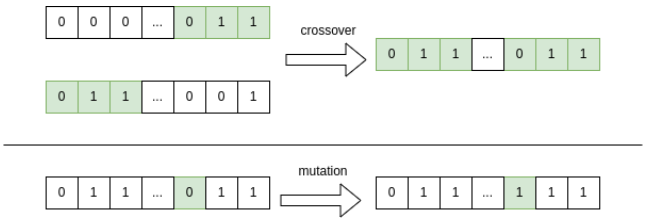

Figura 5.7: Representación de resultado de operadores de un AE

Los operadores de cruzamiento o *crossover* toman varios individuos y generan otro a partir de ellos, pudiéndose ver progenitores y descendientes. Por su parte, el operador de mutación o *mutation* realiza variaciones individuales.

Combinando todo lo anterior se logra que la población cambie de alguna forma a través del tiempo, de acuerdo a cómo se implemente el cambio, es decir, a qué tipo de operadores se utilizan y cómo se configuran, el cambio puede o no conducir la búsqueda hacia el óptimo.

En todo momento la población es medida por la función de aptitud o *fitness*, para conocer cuál es el óptimo conseguido hasta el momento.

A modo de resumen, para el caso de estudio se comienza con una población de carritos generados aleatoriamente. Por medio de distintos operadores, se modifican los carritos, por ejemplo, tomando dos de ellos y creando uno nuevo con partes de cada uno. Al cabo de repetir este proceso, la población de carritos debería haber cambiado y, de acuerdo con la elección de operadores y parámetros de configuración en ellos, se espera conseguir un individuo que maximice el problema.

5.3. Diseño de la solución

La forma de comunicar el diseño de la solución se realiza mediante diagramas que otorgan diferentes vistas para el ecosistema planteado. Esto se logra por medio de un conjunto de abstracciones jerárquicas, organizadas en diagramas con niveles de detalle adecuados al tipo de audiencia.

Las vistas utilizadas en los diagramas son: contexto de sistema, contenedores y componentes. Estas vistas corresponden al modelado C4 [26].

En el contexto se encuentra la visualización más general de la solución, donde se aprecian todos los sistemas involucrados, sus relaciones y los usuarios que participan con un determinado rol.

Luego, las sucesivas vistas otorgan mayor detalle, centrando un elemento e inspeccionando más a fondo su composición. En el caso de los diagramas de contenedores, se encuentra la representación de las aplicaciones o almacenes de datos. Estas son unidades desplegables de forma independiente.

Por último, los componentes son parte de los contenedores y los mismos no pueden ser separados, ya que son considerados parte del mismo proceso ejecutable. Estos definen responsabilidades específicas bajo interfaces bien definidas.

Entonces, la representación de los diferentes sistemas que colaboran se trata del contexto del sistema, las aplicaciones y servicios desplegados de forma independiente conforman los contenedores y la organización de módulos internos de estos artefactos son los componentes.

Los diferentes niveles de detalle ayudan a comunicar de forma efectiva aquello que es de interés a cada tipo de audiencia.

5.3.1. Vista de contexto del sistema

Esta vista se centra en los sistemas de *software*. Un sistema de *software* ofrece el más alto nivel de abstracción, en cuanto a agrupación de componentes. Además, estos se caracterizan por ser los sistemas que ofrecen valor a los clientes.

La vista de contexto presenta la perspectiva más general de todas, permitiendo visualizar sistemas propios y externos, en conjunción a sus actores. Se logra la mayor abstracción de todos los conceptos y es ideal para dar una impresión de las ideas de la plataforma.

En la figura 5.8 se aprecia un diagrama de contexto de la solución.

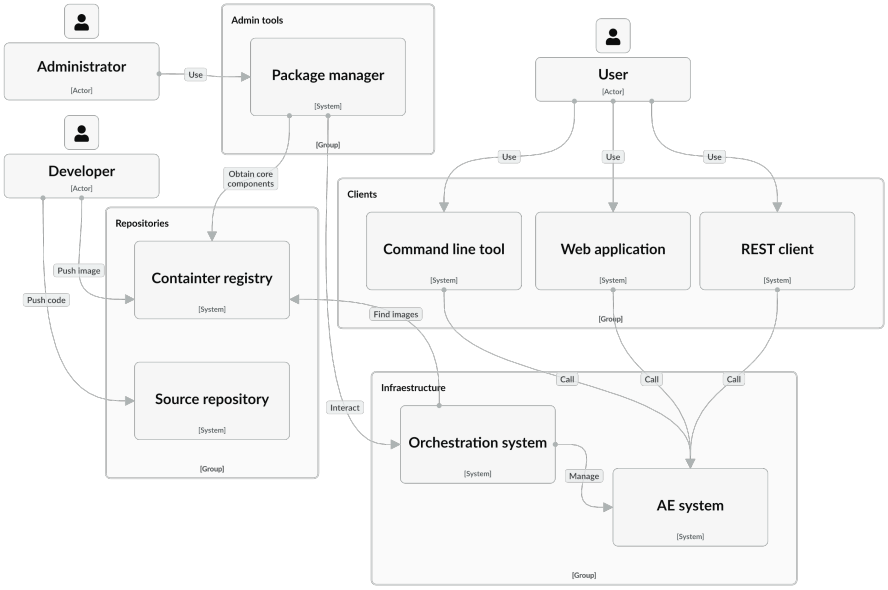

Figura 5.8: Diagrama de contexto de la solución

La plataforma es utilizada directa o indirectamente por tres tipos de actores: usuarios, administradores y desarrolladores.

El usuario desarrollador es aquel que mantiene el *Core* de la solución, alojando los fuentes en un repositorio y construyendo imágenes para su posterior uso y despliegue.

El administrador es el encargado de instalar el núcleo del sistema, utilizando el manejador de paquetes que se adecúe al ambiente en donde se despliega la solución. Para ejemplificar esto, si *Kubernetes* fuese el sistema orquestador, el manejador podría ser *Helm*. El despliegue del sistema involucra las imágenes generadas por el desarrollador con anterioridad.

De igual modo que el usuario desarrollador, el usuario común, es decir, aquel que quiere hacer uso de la plataforma para ejecutar algún algoritmo evolutivo, se vale de los mismos mecanismos de control de versiones y construcción de imágenes, pero en este caso de sus propios operadores, que utilizará en sus experimentos o que compartirá con otros usuarios.

El usuario puede interactuar con la plataforma por medio de una API REST, lo que define varias posibilidades en cuanto a clientes. Conociendo la definición de la interfaz, es posible realizar llamadas mediante alguna herramienta, como por ejemplo *Postman* o *Insomnia*. La definición de esta interfaz permite construir una capa de abstracción por encima de esto y ofrecer una mejor experiencia de usuario, creando aplicaciones con interfaz web o algún cliente de línea de comandos.

5.3.2. Vista de contenedores del sistema

En la figura 5.9 se aprecia un diagrama de contenedores del sistema *AE system*.

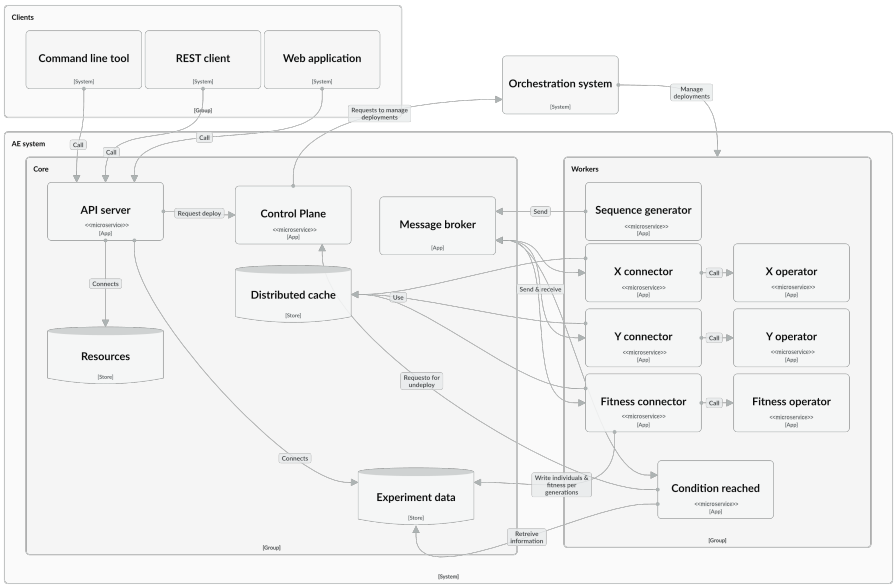

Figura 5.9: Diagrama de contenedores del sistema *AE system*

Aquí se aprecia una vista con centro en la plataforma a construir, logrando una visión más clara de los elementos que intervienen dentro de la plataforma y los que ésta necesita para operar.

Como grandes grupos se distingue al *Core* y *Workers*. El *Core* está conformado por los elementos esenciales que operan la lógica de todo el sistema, siguiendo el patrón arquitectónico de *microkernel*.

Las peticiones externas son recibidas mediante el *API server* y son almacenadas en la base de datos de recursos. Para los casos en los que la solicitud se refiere a la ejecución de un experimento, se genera una interacción con el *Control Plane*.

El *Control Plane* es el encargado de atender los pedidos de despliegues de componentes, realizando la traducción necesaria para poder comunicarse con el sistema de orquestación, para delegar su acción.

Por último, se encuentran los *Workers* con sus respectivas bases de datos y el sistema de mensajería. Esta agrupación contiene los operadores necesarios que ejecutan un experimento basado en un algoritmo evolutivo.

Para abstraer la complejidad de integración al sistema de mensajería y ofrecer una integración natural de los operadores al ecosistema de la plataforma, se agregan contenedores denominados conectores, que ofrecen de nexo entre la *API* del operador y el resto del sistema.

El caché distribuido permite ahorrar llamadas a los operadores realizadas con anterioridad, mientras que la base de datos de experimentos persiste el estado de los individuos por generación. El *broker* de mensajería es el mecanismo por el cual se realiza la comunicación entre microservicios.

5.3.3. Vista de componentes del sistema

En la figura 5.10 se aprecia un diagrama de los componentes del contenedor *API server*.

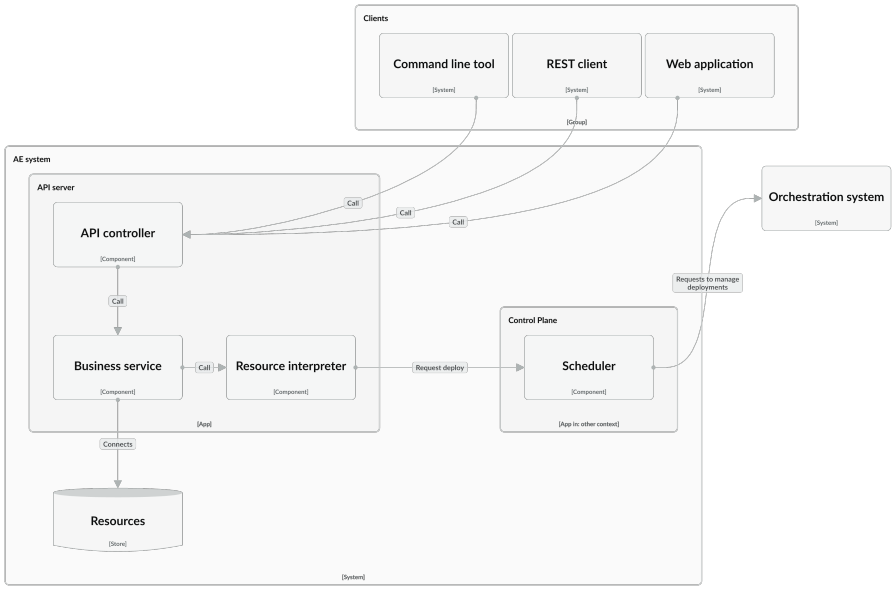

Figura 5.10: Diagrama de componentes del contenedor *API server*

Las llamadas realizadas desde los diferentes clientes son recibidas por medio del *API controller*, componente que define los *endpoints* de la plataforma. Esta pieza puede alojar en un futuro capacidades de seguridad y manejo de usuarios. En este primer desarrollo no se ha agregado estas capacidades, pero tomando en cuenta que el uso de este sistema está enfocado a una modalidad *Software as a Service*, se plantea un escenario compartido en donde se ve interesante esta adición.

Una vez realizado el manejo de las solicitudes a la *API*, son manejadas por el *Business service*, realizando tareas de validación y guardado.

Dado que la reutilización de definiciones es importante en la plataforma, se destina una base de datos para guardar los recursos que modelan los diferentes elementos de los algoritmos evolutivos.

Para el caso de las solicitudes de ejecución, es necesario interactuar con el *Control Plane*. Este componente se encarga de realizar las solicitudes correspondientes al sistema de orquestación para que los servicios sean desplegados tal cual fueron modelados en la representación del experimento.

Como la representación de los recursos es un modelo propio del sistema, es necesario realizar una traducción a los manifiestos de despliegue que comprenda el sistema de orquestación. El *Resorce interpreter* es el componente con el rol asignado a esto, permitiendo tener un mapeo al estilo de infraestructura como código.

5.3.4. Vista dinámica del sistema

En esta sección se presentan diagramas de secuencia para ilustrar el flujo de interacción entre los diferentes componentes del sistema. Estos diagramas son útiles para entender cómo se comunican las partes involucradas y en qué orden se producen las operaciones.

Los casos de uso UC-1 y UC-6 son elegidos para ejemplificar el comportamiento del sistema a través de estos diagramas. Esto es así ya que los mismos refieren a la creación de recursos en el sistema y a la planificación de un experimento, a partir de recursos creados anteriormente.

En la figura 5.11 se aprecia cómo se lleva a cabo la creación de un recurso.

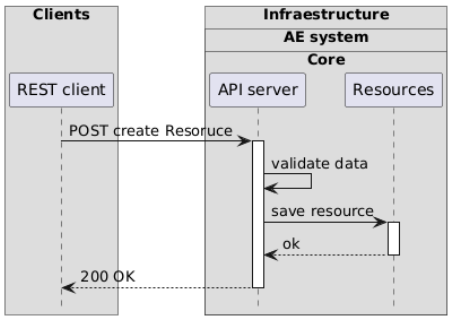

Figura 5.11: Diagrama de secuencia del caso de uso UC-1: Crear un Recurso.

En este caso, la solicitud comienza con una llamada *REST*, la cual es procesada por el *API server*. Después de validada la información, se guarda el modelo del recurso en la base de datos correspondiente.

Estos recursos serán referenciados al momento de crear una ejecución de un algoritmo evolutivo. Un ejemplo de esto puede ser la definición de la función de *fitness*.

Luego, en la figura 5.12 se presenta un caso más complejo, ya que la ejecución de un experimento conlleva la creación e interacción de muchos componentes.

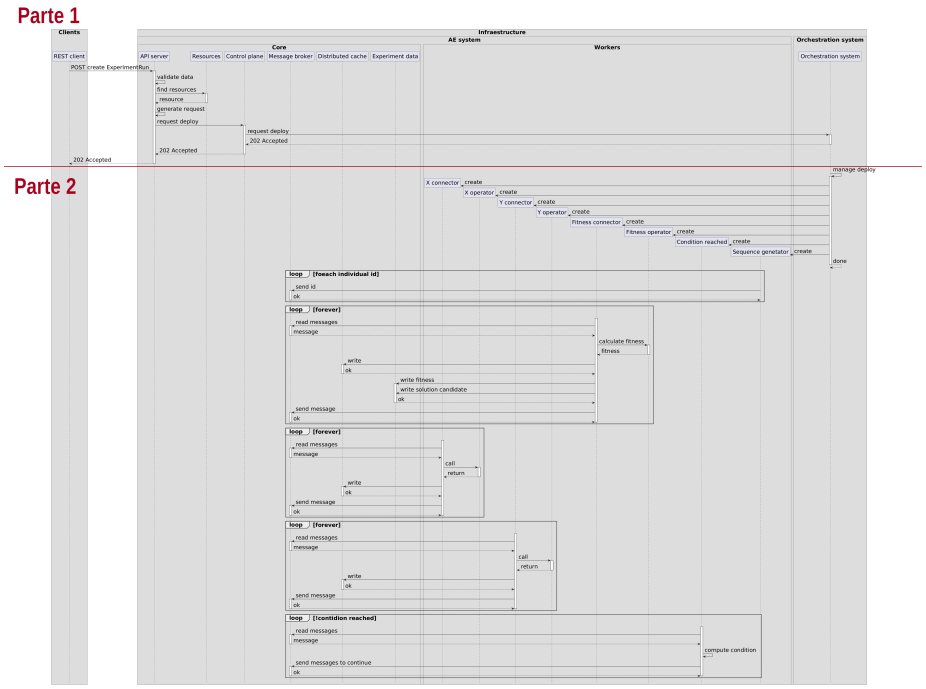

Figura 5.12: Diagrama de secuencia del caso de uso UC-6: Planificar la ejecución de un experimento.

Para dar mejor legibilidad a la figura 5.12, se presenta separada en dos partes, correspondientes a las figuras 5.13 y 5.14.

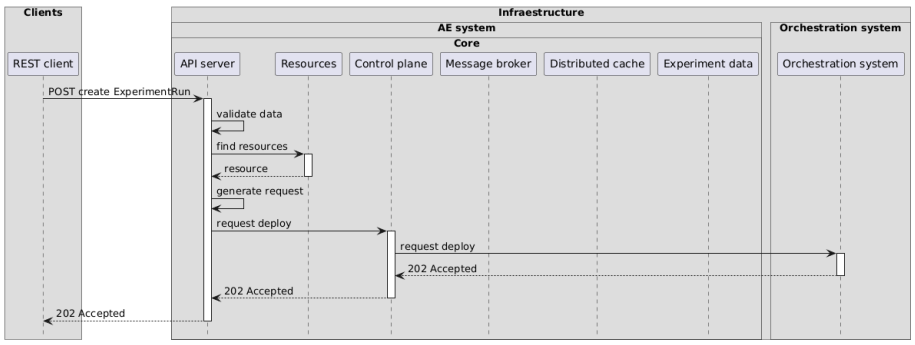

Figura 5.13: Parte 1 del diagrama de secuencia del caso de uso UC-6: Planificar la ejecución de un experimento.

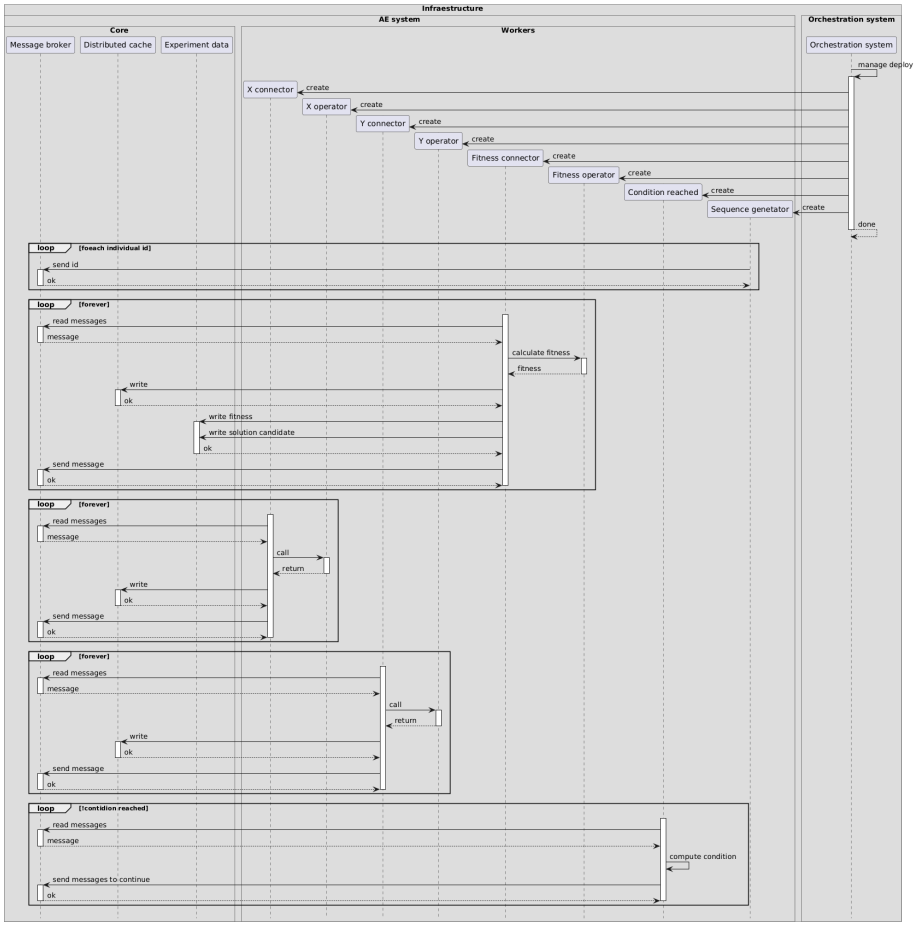

Figura 5.14: Parte 2 del diagrama de secuencia del caso de uso UC-6: Planificar la ejecución de un experimento.

Al igual que el caso de uso anterior, se comienza por una solicitud *REST*. En ella se envía la información necesaria y los recursos involucrados, los cuales deben ser buscados en la base de datos de recursos.

Una vez encontrados, se debe generar un *request* a partir de ellos, transformando esta información a un *request* adecuado para el *Control plane*. El *Control plane* es el encargado de interactuar con el sistema orquestador.

En este punto ya se comienzan a ver varias respuestas 202, indicando la realización asíncrona de estas tareas.

En cuanto a los operadores, se puede apreciar que se dispone del componente llamado *connector*, que intermedia entre los operadores y el sistema de mensajería. Para el caso del *fitness connector*, se ven operaciones adicionales referentes al guardado del valor de aptitud de los individuos y la determinación del candidato a solución en un momento dado.

Por último, una vez creados todos los microservicios pertenecientes al grupo de *Workers*, los mismos pueden comenzar a interactuar mediante el *message broker*. Esta interacción es generada por parte de un componente llamado *sequence generator*, el cual coloca los mensajes iniciales que generarán las iteraciones entre generaciones.

A su vez, el microservicio llamado *condition reached* es el encargado de mantener en funcionamiento el flujo de mensajes de acuerdo al cumplimiento de la condición estipulada como de parada.

CAPÍTULO 6

CONSTRUCCIÓN DE LA SOLUCIÓN

Contenido

En esta sección se describe el mecanismo de creación y configuración de la infraestructura que aloja la solución.

Además, se presentan las decisiones tomadas en la etapa de implementación del sistema, abarcando temas referentes al proceso de desarrollo, estándares adoptados, tecnologías y herramientas.

6.1. Etapas de construcción de la solución

Para la realización del sistema fue necesario separar el trabajo en tres grandes partes: instalación de infraestructura para ejecutar las aplicaciones, elección y configuración de productos de terceros como parte del ecosistema y desarrollo de componentes propios.

Dado que el diseño de la arquitectura está basado en una solución de microservicios, es de suma utilidad contar con un sistema que pueda contenerlos y administrarlos. En ese sentido, se opta por *Kubernetes* como sistema orquestador de servicios. Esta decisión conlleva la instalación de infraestructura acorde.

Luego de contar con un ambiente dónde desplegar los microservicios, resta la elaboración de los componentes del sistema a construir. Tomando un enfoque orientado a la reutilización, se eligen determinados productos para configurar en la plataforma.

Para los componentes a medida de este sistema se debe elegir cómo implementar concretamente algunas funcionalidades y qué tecnologías lo llevarán adelante.

En las siguientes secciones se detalla cómo se da lugar a cada línea de trabajo para componer la totalidad del desarrollo.

6.2. Instalación de infraestructura

La construcción de aplicaciones basadas en microservicios presenta un desafío al desarrollador: la ejecución coordinada de los diferentes servicios de forma local. Por esto, es útil contar con un ambiente similar al de despliegue, para así obtener mayores facilidades al desarrollar y mejores garantías en cuanto al resultado.

Si bien, es posible ejecutar cada componente directamente en el equipo de desarrollo, la dificultad de coordinar cada uno de ellos es proporcional a la cantidad de proyectos. Estrategias que automaticen la configuración, inicio y fin de estos servicios promueven una forma de desarrollo más ágil y eficiente.

En casos donde se necesita levantar una serie de microservicios y conectarlos en una misma red, *Docker Compose*[1] es suficiente.

Sin embargo, el sistema necesita un ambiente final para ejecutar, donde la administración de los servicios sea una característica del producto. Para este fin se utiliza *Kubernetes*, introduciendo el desafío de instalación y configuración de esta herramienta.

El objetivo es realizar la instalación de un *cluster Kubernetes* en un servidor alojado en *Fing*. Para ello, como primeros pasos se utiliza *Vagrant*[2], ya que es una herramienta que permite automatizar la creación de ambientes para desarrollo. Su gran fortaleza radica en la forma en que se definen los ambientes, siendo declarativa, lo que garantiza la creación de entornos replicables de forma sencilla y automática.

Luego de ensayar la creación de *clusters* en entornos virtualizados, en un equipo local, fue necesario empezar a realizar pruebas en el servidor, por lo cual se comienza considerando una instalación sin virtualización.

Esta instalación tiene como característica que es un único nodo que toma la totalidad de la capacidad del servidor. Esto permitió tener una instalación directa sobre el sistema operativo base, sin el *overhead* de virtualización, a costo de tener un único nodo en el *cluster Kuberentes*.

Luego de realizadas pruebas, se divisa como inconveniente que la cantidad de *Pods* recomendados por nodo. Un *Pod* es una unidad básica de ejecución dentro de *Kubernetes*, representando uno o más contenedores, con la característica de compartir recursos y red.

Las buenas prácticas recomiendan no superar los 110 *Pods* por nodo, lo que establece una limitante en términos de escalabilidad. Esto plantea una decisión: respetar las recomendaciones de dimensionamiento, lo que puede resultar en el desaprovechamiento de recursos o en una limitación del escalado, o bien no cumplir con las restricciones, a fin de permitir que un nodo soporte una mayor cantidad de *Pods*.

Ambas opciones presentan desventajas. Limitar la cantidad de instancias a 110 puede desaprovechar la capacidad de cómputo disponible en escenarios que requieran más contenedores, mientras que ignorar las buenas prácticas de dimensionamiento podría poner en riesgo el funcionamiento del sistema.

Como solución a lo anterior, se opta por incrementar la cantidad de nodos del *cluster*, asegurando así un equilibrio entre escalabilidad y estabilidad.

En este caso, las consideraciones de dimensionamiento de un *cluster* de gran tamaño establecen como valores máximos 5000 nodos con 150000 *Pods* en total y no más de 300000 contenedores [75]. Estos valores superan ampliamente las restricciones de un solo nodo.

Para lograr aumentar la cantidad de nodos de la infraestructura, y dado que solo se cuenta con un servidor, el planteo es la creación de máquinas virtuales para alojar todos los nodos.

[1] `https://docs.docker.com/compose/intro/features-uses/`
[2] `https://www.vagrantup.com/`

Se crean cinco máquinas virtuales, donde una de ellas cumple un rol de *master* y las demás de *workers*. El nodo *master* cumple tareas de gestión y configuración, mientras que los *workers* son los encargados de ejecutar los demás contenedores. El sistema operativo *guest* es *FlatCar* una distribución comunitaria de Linux diseñada para cargas de trabajo en contenedores, con alta seguridad y bajo mantenimiento, que nace como un *fork* de *CoreOS*.

La virtualización se realiza por medio de KVM[3]. KVM es una solución de virtualización para Linux, que convierte al sistema operativo en un hipervisor de tipo 1 [76]. Además, la gestión de las máquinas virtuales se puede llevar a cabo fácilmente a través de la API de *libvirt*[4].

La forma de crear y aprovisionar estas máquinas virtuales se realiza mediante la herramienta *Terraform*, logrando versionado de los ambientes y pudiendo replicar las tareas de forma sencilla. Luego, las instalaciones de los nodos de *Kubernetes* se realizan mediante un conjunto de definiciones para *Ansible* que toman nombre de *Kubespray*[5], permitiendo crear un ambiente productivo [77].

Combinando entonces los mecanismos de despliegue y configuración de ambientes, se logra obtener infraestructura como código, conocido como IaC (del inglés *Infrastructure as Code*). Esto permite describir el estado deseado de los sistemas y los pasos para llegar a ellos.

En resumen, se logra la instalación de un *cluster Kubernetes* virtualizado, mediante mecanismos automatizados y replicables, permitiendo así una ventaja en términos de administración.

[3] `https://linux-kvm.org/page/Main_Page`
[4] `https://libvirt.org/`
[5] `https://github.com/kubernetes-sigs/kubespray`

6.3. Elección y configuración de productos de terceros

Algunos de los componentes que forman parte del sistema no fueron implementados, sino que fueron configurados. Aquí se detallan cuáles productos fueron agregados y las consideraciones específicas para su uso en este ecosistema.

6.3.1. Principios para la selección de productos

Para seleccionar entre la oferta de productos disponibles en el mercado, se valoran diversos criterios.

Como primera aproximación a la elección, se realizan búsquedas sobre reseñas de las diferentes herramientas capaces de solucionar una determinada funcionalidad. Por ejemplo, se consulta el *landscape*[6] de la *CNCF*, ya que agrupa diversos proyectos clasificados por funcionalidades y permite una rápida comparación de licencias, documentación, lenguajes, nivel de madurez, participación de la comunidad, entre otros aspectos. Esta información es recopilada de diversas fuentes y presentada de manera centralizada.

En la figura 6.1 se muestra un ejemplo del despliegue de información de una herramienta en el *landscape*.

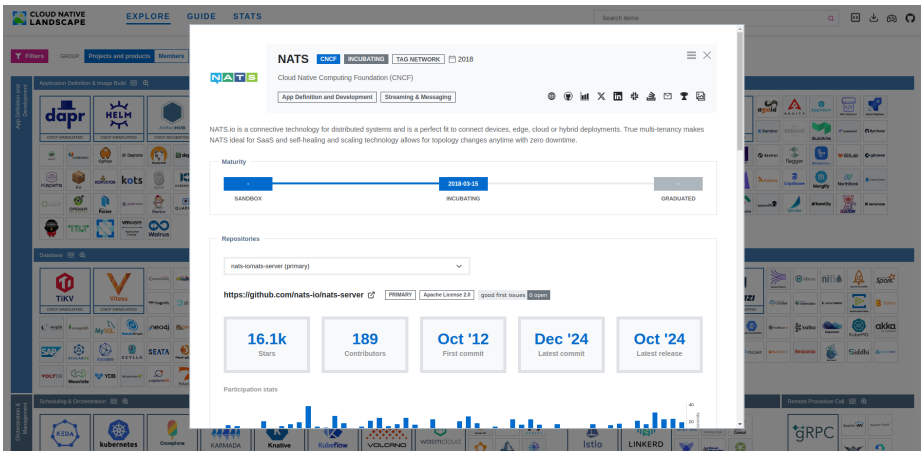

Figura 6.1: Captura de pantalla del *landscape* de la *CNCF*

Otro aspecto considerado en la elección de productos fue el conocimiento previo de alguno de ellos. En caso de no contar con dicho conocimiento, se llevó a cabo la realización de prototipos para validar su uso.

6.3.2. Productos utilizados

Aquí se describen los productos configurados en la solución propuesta, justificando su selección y explicando su utilidad.

Para detallar cada uno, es importante ubicarlos dentro del diseño realizado. Por esta razón, en la figura 6.2 se presenta una modificación del diagrama de contenedo-

[6] `https://landscape.cncf.io/`

res del sistema *AE system*, mostrado anteriormente, para visualizar los componentes involucrados y las tecnologías utilizadas para implementarlos.

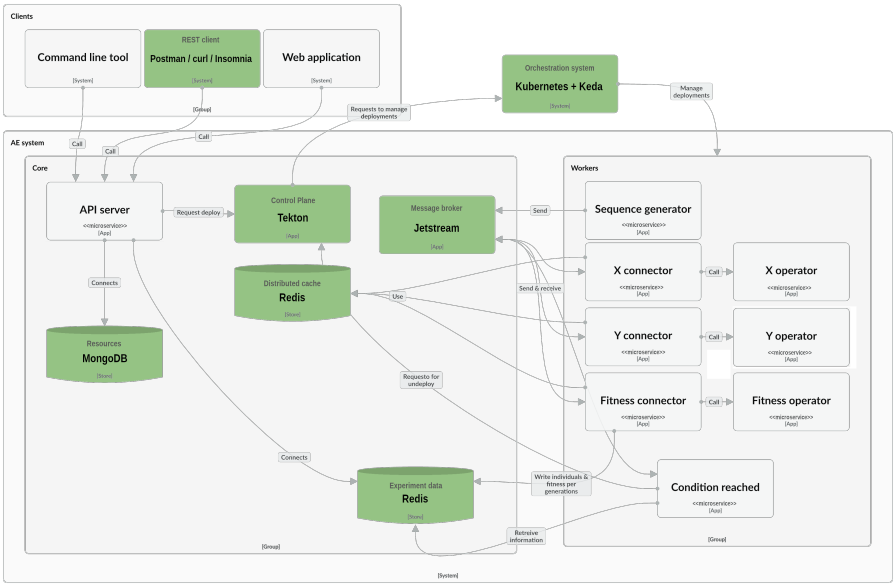

Figura 6.2: Diagrama de contenedores del sistema *AE system* instanciando productos utilizados

Sistema de mensajería

Como herramienta de mensajería entre los operadores, se escoge *Jetstream*, que es una tecnología que ejecuta sobre *Nats*[7], agregando funcionalidades y mayores calidades al servicio que ofrece *Nats*, convirtiéndose en un sistema de persistencia distribuido.

Como alternativa a este componente, se analizó *Apache Kafka*[8], desarrollando un prototipo para evaluar su uso. Sin embargo, no se logró alcanzar el funcionamiento esperado debido a problemas en su configuración. Esto no implica que la herramienta no sea adecuada para este sistema, sino que su integración no se realizó correctamente.

Por su parte, al realizar pruebas con *JetStream*, esta solución demostró ser más adecuada gracias a su documentación clara y guiada, la cual incluye ejemplos detallados de configuración del *broker*, así como código para productores y consumidores en diversos lenguajes de programación.

Además, el respaldo de la *CNCF* como proyecto incubado[9] (una categoría que agrupa proyectos considerados estables y listos para uso productivo) refuerza la elección. Por estas razones, *JetStream* se consolidó como parte de la plataforma de este proyecto.

JetStream organiza sus datos en *streams*, que son secuencias de mensajes duraderas, organizados por temas, que admiten políticas de entrega y retención configurables. Las políticas de entrega determinan cómo se envían los mensajes a los consumidores. Por otro lado, las políticas de retención establecen cuánto tiempo o cuántos mensajes se retienen en el *stream*. Esto proporciona flexibilidad para adaptarse a diversos casos de uso, desde la entrega garantizada hasta la retención a largo plazo de datos.

[7] `https://nats.io/`
[8] `https://kafka.apache.org/`
[9] `https://www.cncf.io/projects/`

Para la entrega de mensajes a los consumidores admite tanto el modelo *push* como *pull*. En el modelo *push* los mensajes son enviados activamente a los consumidores tan pronto como están disponibles en el *stream*, proporcionando una entrega inmediata. Por otro lado, el modelo *pull* permite que los consumidores soliciten mensajes cuando estén listos para procesarlos, brindando un mayor control sobre la velocidad de consumo. En la plataforma se opta por la utilización del modelo *pull*.

Uno de los desafíos a solucionar utilizando *Jetstream* tiene que ver con la *deduplication*. Este concepto abarca la política de entrega de un mensaje a la herramienta exactamente una vez, es decir, que no se duplique la entrega y que las operaciones puedan ser idempotentes. La forma de llevar adelante esta característica es mediante la definición de límites por *subject* a un solo mensaje, sumado a la correcta creación de *subjects*, utilizando un identificador del mensaje como parte del nombre del *subject* [78].

En el marco de la arquitectura propuesta, los servicios pueden ser reiniciados y sustituidos por otras instancias, por lo que la forma en que se manipulan los datos debe ser rigurosa en términos de su diseño. Una solución clave para evitar la pérdida de información entre etapas del algoritmo evolutivo (por ejemplo, entre un operador y otro) es la de evitar el borrado de los mensajes hasta asegurar su entrega en la siguiente fase. Esta idea combinada con la idempotencia en el envío de un mismo mensaje garantiza que no haya pérdida de información; no obstante, en casos puntuales puede generar el efecto de procesar más de una vez la misma tarea, lo que puede ser minimizado con estrategias de *caché* distribuido, ya que cada microservicio no puede mantener un estado consigo al ser considerados descartables o reemplazables.

Una solución ideal para esto comprende el uso de transacciones a nivel del código de los microservicios. Luego de implementado el sistema sobre *Jetstream*, se encuentra en *Kafka* una solución más integral debido a que, mediante sus clientes, es posible definir transacciones, lo que permite además englobar operaciones entre sistemas y lograr, por ejemplo, una escritura en una base de datos y en el sistema de mensajería. Con *Jetstream* sigue siendo posible, aunque es necesario implementar dichos mecanismos para lograr un comportamiento similar [79], [80].

Bases de datos y caché distribuido

La base de datos destinada para alojar la representación de los recursos que conforman los elementos de un algoritmo evolutivo es *MongoDB*. La elección de esta tecnología se ve respaldada por tratarse de una base de datos no relacional y de código abierto. Las diferentes representaciones pueden ser persistidas como documentos sin preocuparse por los esquemas asociados, otorgando versatilidad a la hora de agregar nuevos tipos de recursos.

Luego, en el contexto de utilización de microservicios, se puede afirmar que el diseño de ellos debe carecer de estado y los reinicios pueden acontecer en cualquier momento. Bajo este escenario, algunas interacciones entre componentes pueden repetirse debido a alguna falla durante algún flujo que no se logra completar.

Un caso de esto puede darse cuando un conector toma un mensaje del sistema de mensajería, para luego comunicarse con un operador. Si el conector sufre un reinicio luego de completada la comunicación con el operador, pero no antes de confirmar el procesamiento del mensaje, este será consumido con posterioridad por algún otro servicio. Para evitar que el operador realice lo hecho anteriormente de forma exitosa, el resultado es guardado en la *caché* por un tiempo, lo que ofrece mejores tiempos de respuesta ante una falla, además de ahorrar tiempo de cómputo.

La tecnología seleccionada para este propósito es *Redis*[10], debido a su amplia tra-

[10] `https://redis.io/`

yectoria, popularidad dentro de la comunidad, amplia documentación y buena reputación. Estos factores la convierten en una opción confiable como punto de partida para el proyecto.

Escalador de aplicaciones

Para lograr una plataforma elástica, es decir, que pueda ajustar la utilización de recursos de acuerdo a las necesidades en un tiempo específico, se debe implementar alguna estrategia para este fin.

El mecanismo asociado a esta técnica consta en la utilización de métricas, para que mediante la consulta periódica de ellas, se tomen las decisiones correspondientes de aumentar la cantidad de instancias de aplicaciones.

Por defecto, en un *cluster Kubernetes* se cuentan con métricas de utilización de CPU y memoria por parte de los *Pods*, lo que a priori y de acuerdo al tipo de aplicaciones desplegadas, puede no significar una correspondencia directa a la necesidad de aumentar las réplicas de un *Deployment*.

Los mecanismos nativos para el escalado automático no son útiles para este escenario, por lo que se opta por la introducción de *Keda*.

Keda[11], es una herramienta que ofrece el manejo de métricas de los diferentes servicios, manipulando y ajustando mediante eventos la cantidad de instancias de un despliegue. Al tratarse de una herramienta versátil y adecuada a las necesidades del proyecto, se opta por su utilización.

Antes de incluir a *Keda* como parte de la solución, se realizaron pruebas para generar métricas propias en las aplicaciones desarrolladas. Esto aumentaba la complejidad de las aplicaciones debido a la necesidad de incorporar lógica adicional y nuevas dependencias. Además, era necesario contar con un sistema que gestionara las métricas para que *Kubernetes* pudiera tomar decisiones de escalado basadas en ellas.

Se llevó a cabo una prueba de concepto utilizando *Prometheus*[12] en combinación con la generación de métricas *custom*. Aunque *Prometheus* es una herramienta ampliamente reconocida y suele ser una de las primeras opciones para este propósito, esta alternativa fue descartada tempranamente debido a la necesidad de modificar cada despliegue. En cambio, *Keda* gestiona este proceso de forma transparente, eliminando la necesidad de desarrollos a medida.

Despliegues automáticos

La integración continua (*CI*) refiere a la inclusión segura de los cambios hacia la rama principal del proyecto, mientras que el despliegue continuo (*CD*) permite disponibilizar automáticamente los artefactos o construcciones al ambiente productivo. Entonces, la serie de acciones automatizadas que conducen al código desde la estación de trabajo del desarrollador a producción, como parte del proceso de despliegue continuo, es conocido como *pipeline* [38].

La herramienta elegida para instanciar estas acciones es *Tekton*. *Tekton* es un *framework* de código abierto que permite la creación de *pipelines* CI/CD que puede ser utilizado en un entorno en la nube.

Esta herramienta conceptualiza los elementos en tres categorías: *step*, *task* y *pipeline* y se puede apreciar un ejemplo de cómo se componen en la figura 6.3. Los *step* definen una operación, se podría catalogar como la unidad mínima de trabajo dentro de estos componentes. Luego los *tasks* agrupan los *steps* bajo un cierto orden especí-

[11] `https://keda.sh/docs/2.15/concepts/`
[12] `https://prometheus.io/`

fico. Por último, un pipeline organiza la jerarquía de ejecución entre diferentes *steps* [81].

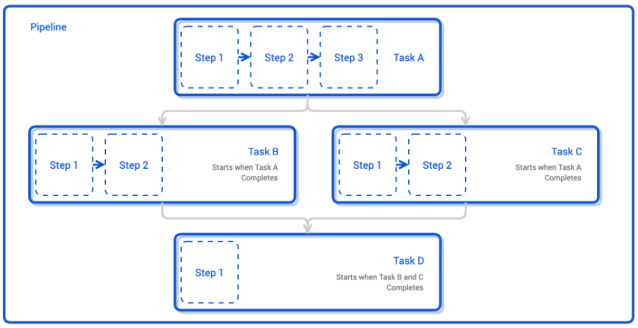

Figura 6.3: Diagrama de ejemplo de la disposición de elementos de un *pipeline* de *Tekton*.[13]

Uno de los mecanismos para disparar la ejecución de un *pipeline* tiene que ver con la intervención de un usuario, solicitando una instancia de ejecución. Dentro de la plataforma, un recurso de tipo *Pipeline* sirve de plantilla, mientras que uno de tipo *PipelineRun*, al ser creado, logra como efecto desencadenar la creación de las tareas que ejecutan los pasos del *pipeline*.

Para lograr automatismo dentro del sistema y a su vez ofrecer una capa de abstracción que oculte la existencia de esta tecnología concreta para realizar el despliegue de los componentes que ejecutan el algoritmo evolutivo, se opta por dejar de lado la utilización del recurso *PipelineRun* y en su lugar definir *EventListener* y *Triggers*. En el diagrama de la figura 6.4 se aprecian algunos de los componentes que son importantes para esta solución, en particular el *EventListener* que escucha eventos en un determinado puerto dentro del *cluster* para así, mediante un *TiggerTemplate*, que es la plantilla que modela de forma genérica las diferentes instancias a ejecutar de un pipeline [82].

La plataforma recibe una petición de parte de un usuario para ejecutar un experimento y esto concluye en la comunicación desde el componente que recibe el pedido al *EventListener* para desplegar todos los operadores y piezas que componen el algoritmo evolutivo dentro de esta realidad de microservicios.

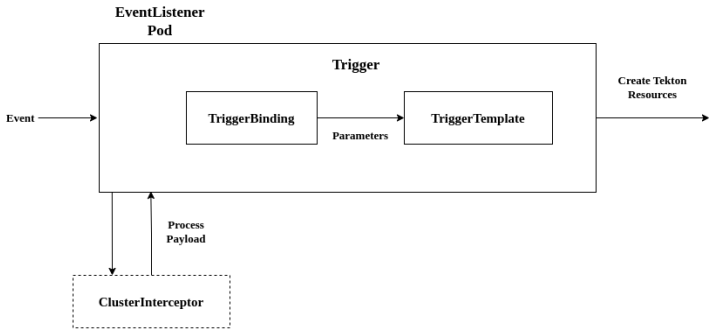

Figura 6.4: Elementos que componen la solución de *triggers* en *Tekton*.[14]

[13]Diagrama obtenido de la documentación [81].

6.3.3. Resumen de productos

A modo de resumen, se presenta la tabla 6.1 que posee la selección de productos para la solución propuesta.

Producto	Funcionalidad	Motivo
Jetstream	Sistema de mensajería	Licencia libre Buena documentación Alto rendimiento
Keda	Escalador de aplicaciones	Licencia libre Buena documentación Facilidad de uso
Redis	Caché distribuido	Licencia libre Conocimiento previo
MongoDB	Base de datos	Licencia libre Reputación
Tekton	*Framework CI/CD*	Licencia libre Reputación

Tabla 6.1: Productos seleccionados acompañado del motivo.

[14]Diagrama obtenido de la documentación [82].

6.4. Desarrollo de componentes propios

En esta sección se describen los componentes creados para dar lugar a la solución y la justificación de cada decisión.

En la figura 6.5 se presenta una modificación del diagrama de contenedores del sistema *AE system*, mostrado anteriormente, para visualizar los desarrollos realizados y las tecnologías utilizadas para implementarlos.

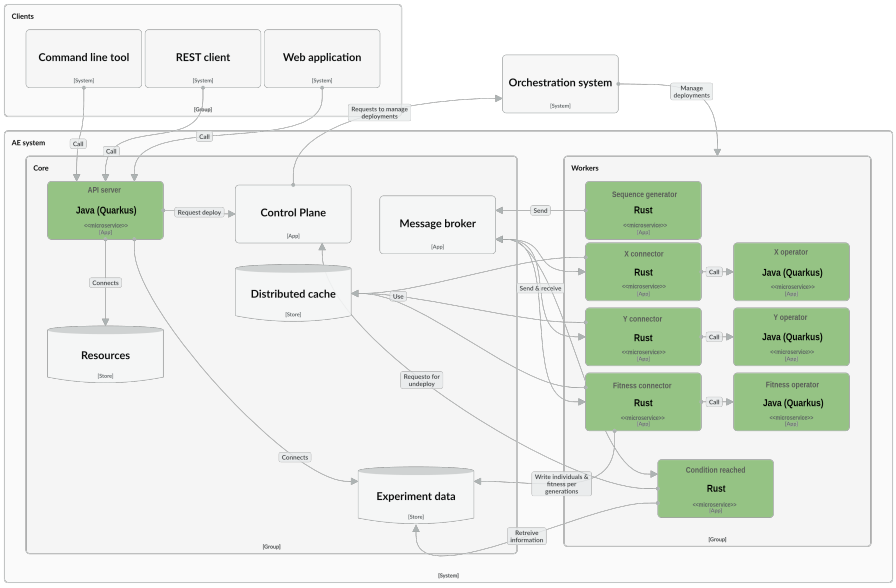

Figura 6.5: Diagrama de contenedores del sistema *AE system* instanciando tecnologías utilizadas en desarrollos propios

Los componentes desarrollados en esta parte se agrupan en *API server* y *workers*.

El *API server* posee la responsabilidad de ser la interfaz entre el usuario y el sistema. En esta sección se describen los mecanismos implementados para llevar a cabo esta interacción.

Por su parte, los *workers* son los contenedores relacionados a los operadores del algoritmo evolutivo. Los *workers* se dividen en operadores y conectores por motivos que se detallan en su sección correspondiente.

6.4.1. API server

El mecanismo de comunicación entre el usuario y la plataforma está definido mediante una *API REST*, perteneciente al componente *API controller* en la aplicación *API server*.

Una *API REST* es una interfaz que permite realizar integraciones entre aplicaciones de forma remota a través de una red [83].

El término *REST* no define un protocolo ni un estándar, sino que establece los límites de arquitectura, ya que las implementaciones que pueden surgir a partir de su utilización pueden ser diversas.

En general, la adopción de una solución tecnológica conlleva el estudio de la forma más efectiva de llevar adelante su instrumentación, en general se logra mediante la experiencia de la comunidad y siguiendo guías o buenas prácticas. En este caso, para encontrar el mayor aprovechamiento de un modelado *REST*, se opta por la adopción del modelo de madurez desarrollado por Leonard Richardson [84].

Para obtener el mayor beneficio de la utilización de *REST*, Richardson define la existencia de cuatro niveles de madurez, donde cada uno establece diferentes niveles de madurez. Elementos tales como la introducción de recursos, verbos HTTP y controles conducen a la obtención de un mejor y más sólido diseño.

El primer nivel tan solo hace uso del protocolo HTTP, no organiza ni define cómo se utilizan los *endpoints*. El segundo nivel establece la organización de las rutas bajo una semántica de recursos, lo que estandariza los *paths*, brindando a su vez una mayor capacidad de expresión. Luego, el tercer nivel sigue la idea del anterior, explotando la utilización de los verbos HTTP en favor de diferenciar las diferentes operaciones sobre los recursos. Por último, el nivel más alto agrega a las entidades la capacidad de referenciar a otras por medio de hipervínculos.

En este proyecto se alcanza el tercer nivel de madurez.

Por otra parte, este conjunto de *APIs* está destinado a que el usuario pueda interactuar con la plataforma, por lo tanto, es necesario compartir una documentación apropiada para que pueda entender de qué forma actuar con el sistema. *OpenAPI* ofrece esto y más, ya que no solamente describe las interfaces, sino que también asiste al desarrollador o usuario tanto en la comprensión como en la ejecución.

Lo más importante es que se establece un medio estandarizado donde se definen las operaciones y el modelo, lo que permite la implementación de clientes en diferentes tecnologías, logrando alta compatibilidad [85].

Mecanismo de interacción

Como se describió anteriormente, la API expuesta establece la forma en que el usuario realiza experimentos sobre la plataforma. La API presenta operaciones estándar para la manipulación de entidades, del estilo de alta, baja y modificación. Las mismas están descritas a través de la especificación *OpenAPI*.

Para explotar el uso declarativo de los componentes, se decide que el mecanismo de definición del algoritmo evolutivo y su ejecución se realice mediante un manifiesto en formato *YAML*. La elección de este formato radica en que es mucho más legible que otros como *XML*, el cual es fuente de inspiración de este.

En la figura 6.6 se aprecia un cuerpo de mensaje para una creación de una ejecución de un experimento.

```yaml
apiVersion: core/v1
kind: experimentRun
spec:
  experiment:
    experimentSpec:
      problem:
        ref:
          apiVersion: core/v1
          kind: function
          name: OneMax 50
          version: v2.1
      algorithm:
        evolutionaryAlgorithmSpec:
          init:
            containerSpec:
              image: lcostela/init-operator-onemax
              interfaceLocation:
                grpc:
                  serviceName: uy.edu.fing.proygrado.service/Init
          crossover:
            ref:
              apiVersion: core/v1
              kind: function
              name: crossover
              version: v1
          mutation:
            ref:
              apiVersion: core/v1
              kind: function
              name: mutation
              version: v1
      context:
        populationSize: 2000
        generations: 5

```

Figura 6.6: Ejemplo de representación de un *ExperimentRun* en formato *YAML*.

En el ejemplo de la figura 6.6 se aprecia que algunos componentes son declarados para un solo uso, como por ejemplo, la especificación del contenedor *init*. Esto es así, ya que no presenta un campo de referencia *ref*.

Sin embargo, es posible reutilizar definiciones anteriores, como es el caso del operador *crossover*, dado que fue creado anteriormente y en este punto solamente bastó con indicar los atributos que lo identifican.

De esta forma es posible manipular el modelo diseñado de forma eficaz, con la capacidad de reutilizar definiciones que se utilicen de forma recurrente. Esto es especialmente útil para casos en donde se pueden desarrollar operadores genéricos sobre alguna codificación especial, o bien, para un problema particular, intercambiar implementaciones para lograr un mejor desempeño de la acción de búsqueda del óptimo.

Cuando se crean recursos, solamente se persisten para su posterior utilización, pero al momento de crear una ejecución de un experimento, eso desencadena una serie de acciones adicionales. En este caso, además de realizar las validaciones del modelo, se lleva a cabo una traducción de la especificación del experimento a manifiestos *Kubernetes*. Esto es realizado por el *Resource interpreter*, en colaboración con *Tekton* para su posterior despliegue.

Para desarrollar fácilmente el *API server*, se opta por la utilización del lenguaje Java[15].

Para ejecutar una aplicación desarrollada con este lenguaje es necesario contar con una máquina virtual llamada JVM (*Java Virtual Machine*), que es el componente encargado de interpretar el código compilado y ejecutarlo. Esto trae consigo un beneficio importante que se refiere a la portabilidad, es decir, una misma aplicación puede ser

[15] `https://www.java.com/en/download/help/whatis_java.html`

ejecutada en ambientes máquinas diferentes, incluso con sistemas operativos distintos, ya que es la máquina virtual la que se encarga de abstraer las interacciones con el sistema. Por otra parte, el uso de este tipo de solución provoca un mayor consumo de memoria y arranques lentos en las aplicaciones, lo que va en contra de los principios de una aplicación enfocada en la nube, por lo que se opta por la utilización de *Quarkus*[16].

Quarkus logra construir aplicaciones con un *stack* nativo de Kubernetes, adaptando el producto a los lineamientos deseados para la nube. En conjunto de *GraalVM*[17], se logran construcciones aún más ligeras.

6.4.2. Workers

A modo conceptual, en la figura 6.7 se puede visualizar cómo un algoritmo evolutivo es modelado en la plataforma.

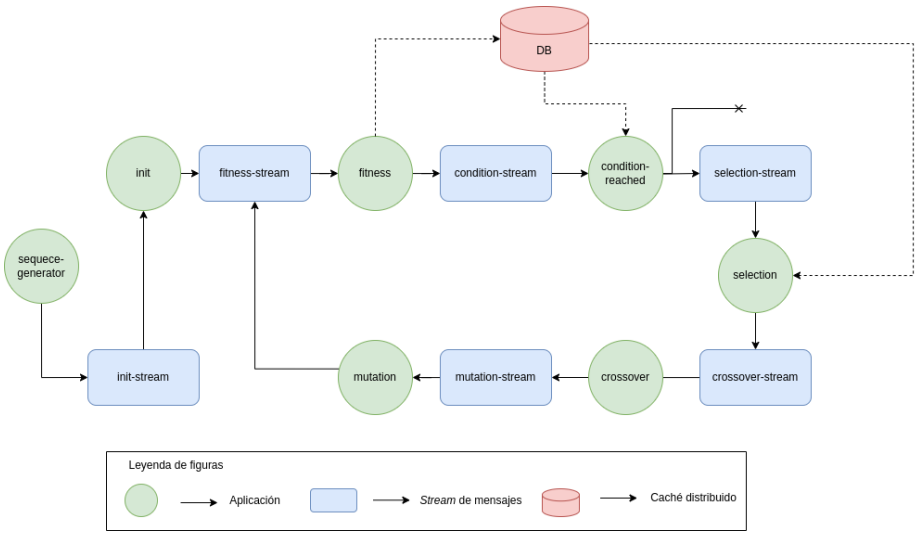

Figura 6.7: Diagrama de visualización de componentes de ejecución de un algoritmo evolutivo

En la figura 6.7 se presenta un escenario en donde las comunicaciones entre los servicios que componen al algoritmo evolutivo se comunican mediante el pasaje de mensajes. El comportamiento generacional del algoritmo es modelado mediante un ciclo en la cadena de mensajería, el cual es interrumpido al alcanzar una condición de parada.

Conceptualmente, cada operador está compuesto por el conector y el operador en sí mismo. Esto se visualiza en un ejemplo particular en la figura 6.8.

[16] `https://quarkus.io/`
[17] `https://www.graalvm.org/`

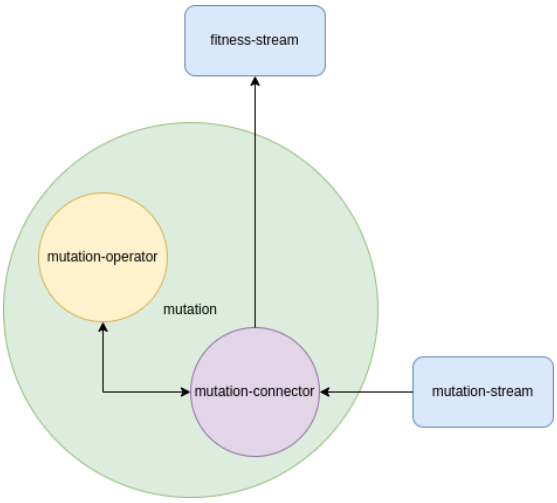

Figura 6.8: Diagrama de relación entre operador y conector

Esto no implica una adhesión al patrón *sidecar*, sino que es una forma de ejemplificar cómo se relacionan los diferentes servicios.

La razón de esta separación radica en la abstracción que el conector ofrece sobre el ecosistema de la plataforma, por ejemplo, ocultando la tecnología de mensajería concreta e independizando al operador de la solución de fondo.

Conectores

Para realizar una integración sin dependencias entre el desarrollo asociado al operador y la plataforma, se crean diferentes conectores para ser intermediarios en las tareas de comunicación.

Dado que es de suma importancia que estos conectores sean sumamente eficientes en términos de uso de recursos, porque es deseado que puedan replicarse masivamente, se escoge como lenguaje a *Rust*.

El lenguaje de programación *Rust*[18], es ideal para crear aplicaciones de alto rendimiento, que logran un uso eficiente de la memoria al prescindir de un *garbage collector*. Gracias a su rendimiento predecible y bajo consumo de recursos, este lenguaje es ideal para desarrollar servicios en la red.

Existen dos opciones para desplegar estos componentes conectores, una es ejecutando en el mismo *Pod* bajo el patrón *Sidecar*, como se aprecia en la figura 6.9. Esta elección está motivada por la necesidad de aplicar este patrón para desacoplar la lógica de comunicación a los servicios de la plataforma, tales como bases de datos y sistema de mensajería, sin recargar el uso de memoria de todo el sistema, ya que al usar este diseño, cada microservicio debe contar con dos contenedores en lugar de uno.

[18] `https://www.rust-lang.org`

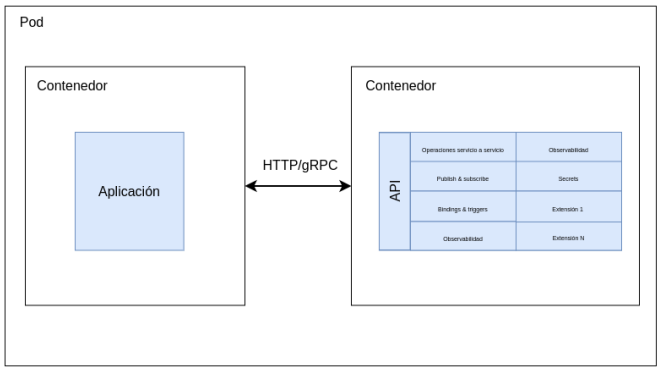

Figura 6.9: Esquema del patrón de diseño Sidecar.

Esto es posible, ya que los contenedores dentro de un *Pod* pueden comunicarse mediante la dirección de *localhost* [86], lo que garantiza un mecanismo sencillo de implementación para el patrón y además asegura que cada espacio de direcciones dentro de los *Pods* está aislado de los demás.

El inconveniente de este tipo de despliegue radica en la necesidad de escalar de forma uniforme operadores y conectores, lo cual puede no ser necesario siempre e incluso puede ser problemático en desarrollos de operadores que consuman grandes cantidades de recursos.

La alternativa al despliegue con el patrón *Sidecar* es la utilización de *Server-Side Service discovery* [87].

Bajo este esquema, la generación de servicios de ruteo, con nomenclatura diseñada de forma estratégica, permite que la creación dinámica de operadores sea fácilmente resuelta mediante un nombre de dominio interno en el *cluster*.

Conector genérico

Se desarrolla un conector genérico, para poder ser utilizado como nexo entre los operadores desarrollados por el usuario y la plataforma.

La capacidad generalista de este conector surge principalmente de dos decisiones, la externalización de configuración y el manejo de datos mediante bytes.

La configuración del componente es pasada como parámetros al contenedor, lo que permite reutilizar el binario sin necesidad de construirlo para cada ocasión. Datos como el nombre del servicio a invocar, la dirección o nombres de los *streams* de la cola de mensajes, son los que se pasan como parámetros.

Hay que recordar que el ciclo operativo de un conector involucra recibir mensajes desde el *broker* de mensajería, para luego entregarlos al operador asociado y, una vez resuelta la llamada, entregar el mensaje en el próximo *stream*. Todo esto de forma cíclica.

Una de las dificultades para llevar este objetivo adelante era el manejo de los datos entre los diferentes componentes. Para ejemplificar esto, si un operador recibe un tipo de dato y retorna eventualmente otro distinto. Para que el sistema opere, es necesario que haya consistencia entre lo que retorna un operador y lo que recibe el siguiente.

Un chequeo a nivel de contrato puede garantizar ello, pero a nivel de implementación era requerido compilar el conector para ofrecer la capacidad de manejo de ese tipo de dato específico. La implementación de *gRPC* para *Rust* dicta que el modelo se construya al momento de compilar la aplicación.

Una alternativa consta en el uso dinámico de descubrimiento y manipulación de modelo, conocido como *reflection*, pero en las pruebas realizadas se constataron diferentes complejidades que descartaron este camino.

Por lo cual, se elige la manipulación de datos a nivel de bytes, siendo el segundo punto detallado como decisión importante para este componente.

Un cliente *gRPC* antes de realizar un llamado o luego de recibir una respuesta, realiza transformaciones del formato transmitido por la red al modelo de dato. En este caso, se modifica el código autogenerado para el cliente, para realizar uno que manipule los bytes recibidos y no los convierta al modelo. Esto permite mayor rendimiento, dado que no se realizan transformaciones que no serán utilizadas y además permite la utilización del conector de forma genérica.

6.4.3. Operadores

Los operadores son los componentes intercambiables por parte del usuario, lo que permite flexibilizar la generación de algoritmos evolutivos.

Los lenguajes de programación que admite la plataforma no implican una restricción, ya que su arquitectura de microservicios permite que convivan diferentes tipos de tecnologías y colaboren entre ellos.

Los operadores tienen aún más libertad en cuanto a su elección, ya que depende de las necesidades o conocimientos del usuario.

Para lograr una colaboración efectiva entre componentes, además de definir la tecnología de comunicación remota, es necesario establecer un contrato o interfaz que permita abstraer la implementación concreta. También es de utilidad la especificación del modelo de datos por medio de una herramienta que pueda ser utilizada por diferentes tipos de lenguajes.

Para los operadores, esta interfaz de comunicación remota es gRPC.

gRPC

La tecnología *gRPC* es considerada como un moderno *framework*, desarrollado por *Google* y destinado a la ejecución remota de procesos, enfocado en la alta eficiencia.

Esta solución utiliza y aprovecha HTTP/2 como protocolo de transporte de la información, al mismo tiempo que emplea en capa de aplicación una codificación de datos conocida como *Protocol Buffers* (ProtoBuf). Esta interfaz define un mecanismo extensible, independiente del lenguaje y brinda la capacidad de codificar datos estructurados, pudiéndose comparar con el formato XML, pero más pequeño, rápido y simple. La utilización de este tipo de definición de datos se basa en la definición de una estructura abstracta (como se aprecia en el ejemplo de la figura 6.10), para que luego se genere el código fuente en el proyecto, logrando aumentar la capacidad de interoperar bajo diferentes lenguajes de programación y aceptando un mismo flujo de datos [88].

```
syntax = "proto3";

message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 results_per_page = 3;
}
```

Figura 6.10: Ejemplo de definición de un mensaje en un archivo ProtoBuf.[19]

Por defecto, *gRPC* utiliza *ProtoBuf* como IDL (*Interface Definition Language*, es decir, lenguaje de definición de la interfaz); no obstante, puede ser intercambiado por otro formato como *JSON* [89].

Dentro de la definición del *ProtoBuf* se encuentra la capacidad de expresar operaciones remotas, como se puede apreciar en la figura 6.11.

```
service SearchService {
  rpc Search(SearchRequest) returns (SearchResponse);
}
```

Figura 6.11: Ejemplo de definición de una operación remota en un *ProtoBuf*.[20]

En conjunto, la definición de una función remota con su correspondiente modelo de datos para su entrada y salida, resulta en la posibilidad de autogenerar cliente y servicio a partir de esto. Las librerías provistas para los lenguajes soportados por *gRPC* facilitan la creación de todos los componentes, como es presentado en la figura 6.12.

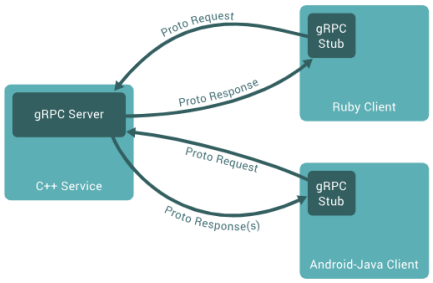

Figura 6.12: Esquema de funcionamiento de *gRPC*.[21]

Esta solución otorga ventajas al ser utilizada dentro de una arquitectura de microservicios en términos de rendimiento, por sobre las APIs HTTP/JSON tradicionales.

La razón de esta productividad está asociada a que la codificación y decodificación de los datos al ser transportados es menos costosa, dado que no está enfocada a ser legible para el usuario, sino que optimiza la conversión y transporte por la red para lograr mejores tiempos.

Por ejemplo, la representación de un *JSON* simple como {"id":42} ocupa 9 *bytes* bajo *UTF-8*, empeorando el escenario para su contraparte en *XML* <id>42</id>, que

[21] Captura obtenida en la documentación [89].

ocupa 11 *bytes*, sin agregar espacios o saltos de línea que usualmente son incluidos para lograr una mayor legibilidad. En un *ProtoBuf* estos datos son transportados como 0x08 0x2a ocupando 2 *bytes*, que es varias veces menor a las otras codificaciones presentadas. Por otra parte, los algoritmos para analizar tanto *JSON* como *XML* son costosos, ya que deben reconocer cada término del mensaje y encontrar una correspondencia en el objeto dentro de la aplicación [90].

La utilización de *gRPC* es la mejor alternativa para lograr el paralelismo entre procesos que pueden darse mediante el uso de *MPI*, pero en los proveedores de la nube no se garantiza su correcto funcionamiento e incluso no logra escalar entre múltiples *clusters* [72], [73].

Operadores específicos

Para simplificar la complejidad de diseño de la plataforma, algunos operadores son fijos, es decir, no pueden ser intercambiados sin conocer la arquitectura interna de la plataforma.

Los operadores fijos son los de selección, reemplazo y condición de parada. Estos fueron los que presentaban dificultades a la hora de abstraer los conceptos de búsqueda, sin tener relación con el medio de almacenamiento específico.

6.4.4. Resumen de tecnologías

A modo de resumen, se presenta la tabla tab:resumenprod2 que posee la elección de las tecnologías nombradas en esta sección.

Producto	Motivo
Rust	Licencia libre Alto rendimiento Binarios nativos
Java (Quarkus)	Licencia libre Conocimiento previo Alto rendimiento Binarios nativos
gRPC	Licencia libre Alto rendimiento
OpenAPI	Estándar de facto

Tabla 6.2: Productos elegidos acompañado del motivo.

6.5. Consideraciones adicionales

El *software* es mejorado a lo largo del tiempo y esta evolución es contenida dentro de diferentes versiones. Cada versión puede introducir mejoras, nuevas funcionalidades o solucionar problemas. Usualmente, los números de versiones reflejan este tipo de cambios.

A su vez, el ciclo de vida de una liberación presenta varias fases, en donde el artefacto es consolidado para su lanzamiento. Comúnmente se manejan las etapas *pre-alpha*, *alpha*, *beta*, *release candidate* y *general release*, donde cada una determina un grado de madurez del producto respecto a las pruebas y validaciones realizadas [91].

6.5. Consideraciones adicionales

Dada la cantidad y tipos de pruebas realizadas sobre el proyecto, se establece que la versión obtenida corresponde a una etapa *alpha*, donde la aplicación se prueba por primera vez, se encuentra en etapas tempranas de desarrollo, presenta sensibles oportunidades de mejora.

En la fase *alpha*, el foco principal se centra en identificar posibles errores en la aplicación, con miras a solucionarlos en pos de lograr la estabilidad y el funcionamiento correcto. El objetivo es lograr que el programa opere de acuerdo a sus especificaciones previas.

Como siguientes pasos, la formalización de técnicas de *testing* más completas y rigurosas puede ayudar a lograr una versión de mayor calidad y apta para un uso seguro y confiable en producción.

CAPÍTULO 7

EVALUACIÓN DEL SISTEMA CONSTRUIDO

Contenido

Una de las principales actividades llevadas adelante en el proyecto está estrechamente relacionada con la construcción de un sistema. El producto obtenido es conducido por una serie de requisitos funcionales y atributos de calidad que deben ser evaluados para conocer si cumplen o no con su especificación; para ello, es necesario someter a prueba el *software* para determinar su estado.

La construcción de una arquitectura evolucionable implica no solamente construir, sino que realizar construcciones validables [28]. Los aspectos a tener en cuenta en este punto se dividen en dos grandes grupos: la forma en que el artefacto final, en conjunto a su diseño de fondo, es útil para realizar las tareas por lo que fue diseñado, en este caso la ejecución de algoritmos evolutivos y, por otra parte, el modo en que se desempeñan esas acciones, relacionándolo fuertemente con los atributos de calidad definidos.

En esta sección se presentan pruebas realizadas sobre la plataforma construida, haciendo referencia a los atributos de calidad planteados en este proyecto. Para algunos atributos de calidad, se ofrece una descripción de cómo se visualiza el cumplimiento de los mismos.

Luego, se describen las pruebas construidas para *jMetal*, con el fin de realizar una comparación entre ambas soluciones.

Las pruebas fueron realizadas en diferentes ambientes de prueba que se detallan a continuación.

El equipo local cuenta con un procesador Intel(R) Core(TM) i7-8650U de arquitectura x86_64, frecuencia de 1.90 GHz y un total de 8 núcleos. En cuanto a la memoria RAM, posee un total de 16 GB en una unidad de tipo DDR4 con una velocidad de 2400 MT/s.

El servidor cuenta con un procesador AMD Opteron(tm) Processor 6172 de arquitectura x86_64, frecuencia de 2.10 GHz y un total de 24 núcleos. En cuanto a la

memoria RAM, posee un total de 24 GB en 6 unidades de 4 GB de tipo DDR3 con una velocidad de 1300 MT/s.

En ambos casos, los equipos cuentan con sistema operativo *Linux*, siendo *Ubuntu*[1] la versión *24.04* del equipo local y *Rocky Linux*[2] la versión *8.10 (Green Obsidian)* del servidor. Los datos empíricos que se presentan en las diferentes gráficas corresponden a ejecuciones sobre el servidor.

También se realiza una comparación funcional en términos de los resultados obtenidos por la ejecución del algoritmo evolutivo, con el objetivo de validar el correcto funcionamiento del sistema construido.

Para llevar a cabo las evaluaciones de los sistemas, se utiliza nuevamente el caso de estudio planteado: el problema de *OneMax*. Este problema consiste en maximizar la cantidad de unos en un arreglo y se adapta a una realidad sencilla, como maximizar el valor de un carrito de compras con una capacidad fija, llenándolo con artículos de un valor constante.

La elección del problema *OneMax* es estratégica, ya que se utiliza como *benchmark* (prueba de rendimiento). Además, al conocerse su solución óptima, resulta sencillo determinar si el algoritmo funciona de manera adecuada.

7.1. Definición del problema de acuerdo al caso de estudio

Todas las pruebas se basan en el problema *OneMax*.

El problema consiste en maximizar la suma de un conjunto $x = \{x_1, x_2, \cdots, x_n\}$, dónde $x_i \in \{0, 1\}$.

Entonces, el problema se traduce en maximizar la función $f(x) = \sum_{i=1}^{n} x_i$.

Habiendo recordado esto, resta definir cuál es el valor n para ejecutar las pruebas. Se elige que la cantidad de variables sea 50, ya que este valor ofrece un equilibrio entre la simplicidad del problema y una complejidad suficiente para evaluar el desempeño del algoritmo evolutivo.

Por ello, el problema se define como: $f(x) = \sum_{i=1}^{50} x_i$, dónde $x_i \in \{0, 1\}$. La solución es la que asigna el valor 1 a todas las variables, obteniendo un valor óptimo de 50.

7.2. Evaluación de la solución propuesta

Como se adelantó, la evaluación se separa en dos secciones: la funcional y atributos de calidad.

A continuación se presenta cada una de ellas.

7.2.1. Funcionalidad

Se realizan 10 ejecuciones de algoritmos evolutivos para intentar resolver el problema de *OneMax* para una tira de tamaño 50. Para ello, se define una prueba con una población inicial de 4000 individuos y un total de 5 generaciones como criterio de parada.

Como resultado se obtiene un candidato a solución que posee un valor de 39 como *fitness*. Esta solución candidata fue la mejor reportada entre todas las ejecuciones realizadas.

[1] `https://ubuntu.com/blog/tag/ubuntu-24-04-lts`
[2] `https://rockylinux.org/`

7.2.2. Atributos de calidad

La evaluación es guiada por la revisión de los atributos de calidad planteados en el proyecto.

Atributos como la escalabilidad y la resiliencia cuentan con pruebas concretas realizadas sobre el desarrollo. Por otro lado, los demás atributos se describen como una aproximación inicial hacia su justificación, quedando pendientes validaciones concretas que podrán realizarse en el futuro.

Escalabilidad

La escalabilidad de la solución se implementa mediante la multiplicación de procesos que atienden a los pasos del algoritmo evolutivo.

Esta escalabilidad puede llevarse adelante de manera estática, es decir, con intervención humana al configurar los valores deseados, o bien realizando una automatización.

Primero se realiza una prueba para conocer cuál será la topología que mejor rendimiento ofrezca. La topología a la que se hace referencia es a la de los operadores y conectores, teniendo como alternativas al patrón *Sidecar* o *Service discovery*.

La prueba consistió en la comparación de 10 ejecuciones del problema de referencia, variando la cantidad de réplicas asociadas a los microservicios. Esto resulta en una comparación de ambas formas de despliegue, obteniendo una serie de tiempos que se pueden visualizar en la figura 7.1.

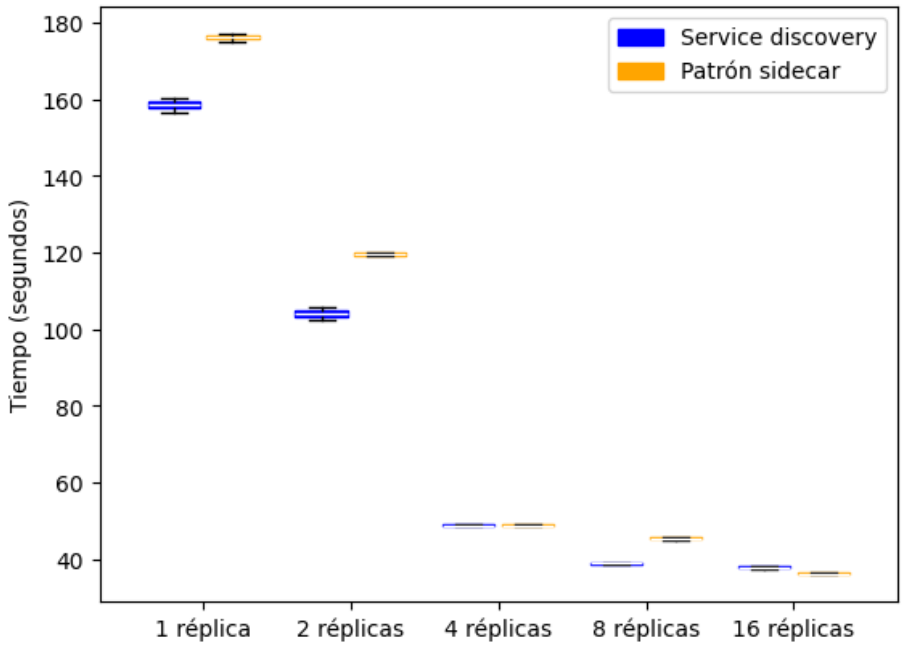

Figura 7.1: Comparativa de tiempos bajo diferentes topologías, ejecutando *OneMax*, n=50 y población de 2000

En este caso, no hay beneficios notorios para aislar los contenedores de operador y conector en un mismo *Pod*, por lo cual, el uso separado de estos componentes es la

elección adecuada. Además, al disponer de los contenedores de forma independiente, es posible escalar de manera más flexible. Hay que tomar en cuenta que los operadores son componentes que pueden ser desarrollados por los usuarios y puede que no se realicen bajo tecnologías que consuman pocos recursos, por lo que tener igual cantidad de réplicas de conectores termina siendo un desaprovechamiento de recursos.

Por todo esto, el patrón *Service discovery* es el que mejor se adapta a las necesidades del proyecto y será utilizado en las sucesivas pruebas.

Por otra parte, se analizó la utilización de técnicas de escalado automático contra la ausencia de ellas. Como se mencionó anteriormente, para automatizar el escalado, se debe explotar fuertemente la recolección de métricas, para su posterior análisis a través de un operador específico. El componente que realiza esta labor es *Keda*, utilizando como métrica la medida de cantidad de mensajes en un *stream* dentro de *Jetstream*. Esto manipula el *HPA* de los diferentes servicios, alterando en tiempo de ejecución la cantidad de instancias.

HPA refiere a *Horizontal Pod Autoscaler* y es el objeto de *Kubernetes* que permite aumentar o disminuir la cantidad de réplicas de una aplicación automáticamente, para ajustarse idealmente a un valor de referencia en un determinado período de tiempo [92], [93].

En la figura 7.2 se visualiza el resultado de tiempos de ejecución utilizando un escalado manual contra uno automático.

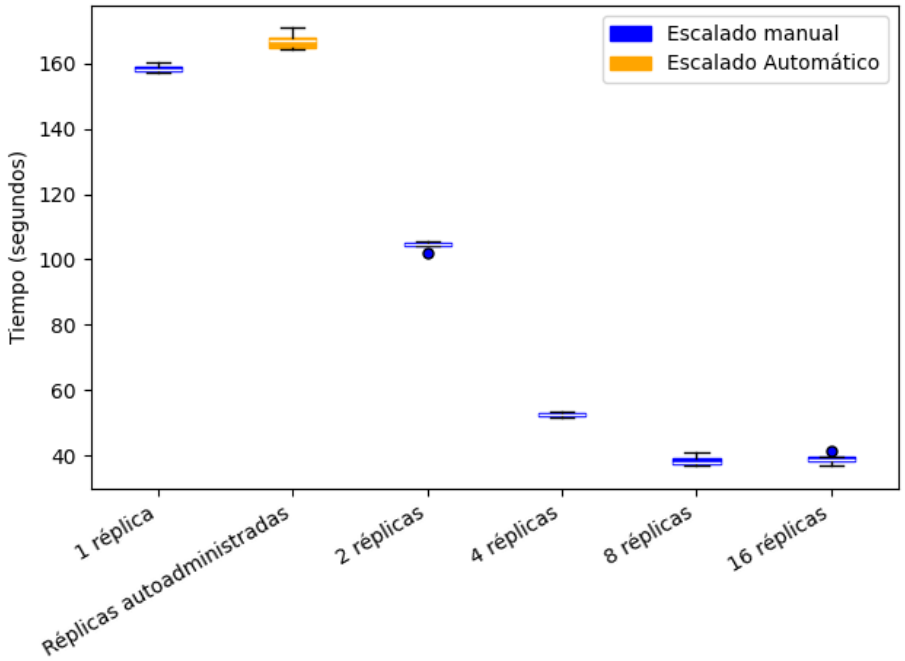

Figura 7.2: Comparativa de tiempos bajo diferentes técnicas de escalado, ejecutando *OneMax*, n=50 y población de 2000

Es esperable que el rendimiento del sistema sea menor al instrumentar un escalado mediante eventos, ya que se generan brechas temporales entre la lectura de la métrica, evaluación y decisión de escalado. Si bien en términos absolutos el inicio de un contenedor es veloz, tardando en el orden de unidades de segundos, en el contexto

del problema es mucho tiempo. Esos segundos de demora son críticos al momento de procesar mensajes, ya que es tiempo desperdiciado.

En experimentos largos, el ajuste inicial de cantidad de instancias se realizaría al inicio y luego el sistema se estabiliza en términos de cantidad de réplicas.

Luego, se decide evaluar el comportamiento del sistema respecto a la cantidad de población, fijando la cantidad de réplicas de microservicios en 16. De esto surge como resultado la gráfica de la figura 7.3. De esto se aprecia que el tiempo aumenta de forma lineal.

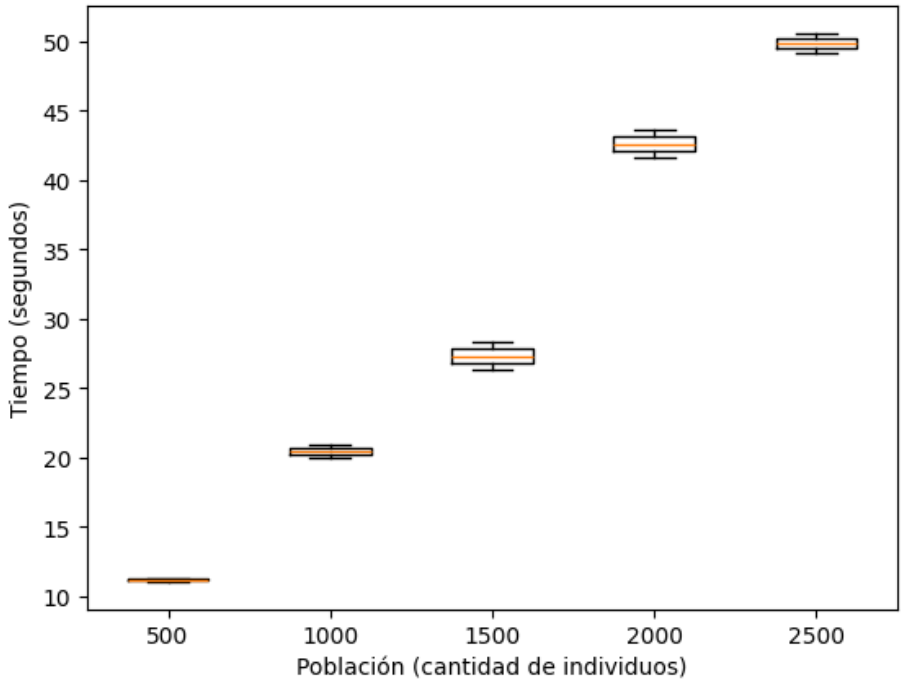

Figura 7.3: Tiempo de ejecución para OneMax, n=50 y población variable bajo 16 réplicas

En resumen, tomando el caso despliegue bajo *Service discovery*, se obtiene una medida de eficiencia reportada utilizando la mediana de tiempos. En la tabla 7.1 se puede visualizar esta información.

Cantidad de réplicas	*Speedup* relativo	Eficiencia
2	1.50	0.75 (Sublineal)
4	3.21	0.80 (Sublineal)
8	4.02	0.50 (Sublineal)
16	4.13	0.26 (Sublineal)

Tabla 7.1: Tabla de *speedup* y eficiencia.

El comportamiento del rendimiento, si bien mejora al aumentar la cantidad de réplicas, lo hace a tasas menores cada vez, y esto puede explicarse por diferentes fac-

tores, como cuellos de botella en las comunicaciones, ancho de banda, sincronizaciones o falta de capacidad de procesamiento en el *broker* de mensajería o caché distribuido.

En particular, un punto de mejora sin resolver, que tiene que ver con el pasaje de mensajes entre generaciones. Esta problemática se da porque, bajo la implementación realizada, es necesario terminar de procesar de forma completa una generación para comenzar con la siguiente.

Si bien, podría comenzarse a realizar algunas operaciones de la siguiente generación sin haber completado las anteriores, el sistema corre peligro de bloquearse de forma indefinida a la hora de ejecutar el operador de selección, ya que consumiría mensajes que indican la realización de selección para una generación determinada, la cual no tiene individuos. La elección de culminar una generación para comenzar la siguiente genera tiempos muertos en algunos componentes, pero garantiza que el algoritmo se completará. Una forma de resolver esto para aumentar la tasa de procesamiento estaría dada por la reestructura del *stream* de mensajes al conector de selección y una lógica modificada para decidir adelantar operaciones de manera segura.

En cuanto al escalado automático, se debe explorar aún más para poder encontrar una configuración que otorgue un balance razonable entre utilización de recursos y rendimiento.

Resiliencia

Se realizan pruebas manuales, las cuales constan de baja de servicios. Esto genera que *Kubernetes* detecte la terminación del contenedor y planifique en ese instante una nueva creación de esa instancia.

Al hacer esto durante un experimento, no se registraron interrupciones en el servicio ni demoras considerables.

El mecanismo ideal para llevar adelante la validación de esta característica está relacionado con la instrumentación de alguna herramienta que provoque fallas en los diferentes servicios desplegados, de forma automática y configurable, para poder medir el rendimiento de la plataforma bajo diferentes escenarios y evaluar cuán rápido se recupera de ello. Una herramienta que lleva a delante este concepto es *Chaos Monkey*[3].

La inclusión de este tipo de prueba automática queda planificada a futuro.

Evolucionabilidad

El diseño de este sistema está enfocado en torno a la definición de algoritmos evolutivos. Un siguiente paso sería aumentar el abanico de posibles tipos de algoritmos.

En un principio, la forma de validar esto sería mediante la implementación de otro tipo de algoritmo, lo cual es costoso en términos de tiempo de desarrollo y estudio, por lo que no es tenido en cuenta para esta instancia.

De todas formas, serían necesarios ajustes en la implementación y diseño para garantizar mayor flexibilidad en la definición de nuevos algoritmos. Así mismo, sería interesante plantear mecanismos declarativos para definir la infraestructura subyacente y los métodos de comunicación de los servicios, para tener un sistema más generalista y no tan atado a un tipo de solución que es la de algoritmos evolutivos.

Interoperabilidad

Al desarrollar y agregar un nuevo operador evolutivo en un experimento, se verifica la sencillez de este proceso dentro de la plataforma.

[3] `https://netflix.github.io/chaosmonkey/`

La cadena de operadores define un flujo de transformación de datos que debe garantizar la compatibilidad de interfaces en términos de entradas y salidas. Esto ayuda a lograr interoperabilidad entre los servicios, independientemente de la tecnología subyacente.

Modificabilidad

Al momento de realizar pruebas y desarrollar un operador, ya se está comprobando que este atributo de calidad está presente en el sistema.

El comportamiento de la ejecución es modificado fácilmente, por medio de la declaración descriptiva de un experimento, que cuenta con un conjunto de operadores. Al intercambiar la implementación de estos operadores, incluso sustituyendo operadores que cumplan un determinado rol por otro del mismo, pero con una codificación distinta, se logra la modificación del comportamiento fácilmente, tan solo cambiando una sola línea en el manifiesto que define el experimento.

7.3. Evaluación de jMetal

Para tener una referencia contra un producto utilizado anteriormente, se escoge a *jMetal*. Si bien no son tecnologías equivalentes y su comparación se centra en las capacidades de cada herramienta, es decir, cómo pueden resolver un mismo problema, evaluando su rendimiento y características.

7.3.1. Funcionalidad

Al igual que para el sistema desarrollado, se realizan 10 ejecuciones para intentar resolver el problema de *OneMax*, con un tamaño de 50 variables. También se utiliza una población inicial de 4000 individuos y un total de 5 generaciones como criterio de parada.

Como resultado se obtiene un candidato a solución que posee un valor de 39 como *fitness*.

7.3.2. Atributos de calidad

Las pruebas realizadas sobre este *framework* están enfocadas en la escalabilidad y resiliencia, tal como se realizó para la solución propuesta.

Escalabilidad

El *framework* permite dos tipos de paralelismo *Out of the box*, es decir, listo para usar sin tener que realizar modificaciones en el código. Los tipos de paralelismo que pueden ser utilizados corresponden a un escenario sincrónico y asincrónico [94].

El paralelismo sincrónico es un tipo de paralelización que se basa fuertemente en el algoritmo secuencial, pero en el momento de realizar las evaluaciones, las mismas se realizan de forma paralela mediante la creación de hilos.

Se realizan 20 ejecuciones para obtener los tiempos de respuesta, para el problema de *OneMax*, con una población de 2000 individuos y un criterio de parada de evaluación de 5 generaciones, que equivale a unas 10000 evaluaciones. En la figura 7.4 se ven los resultados.

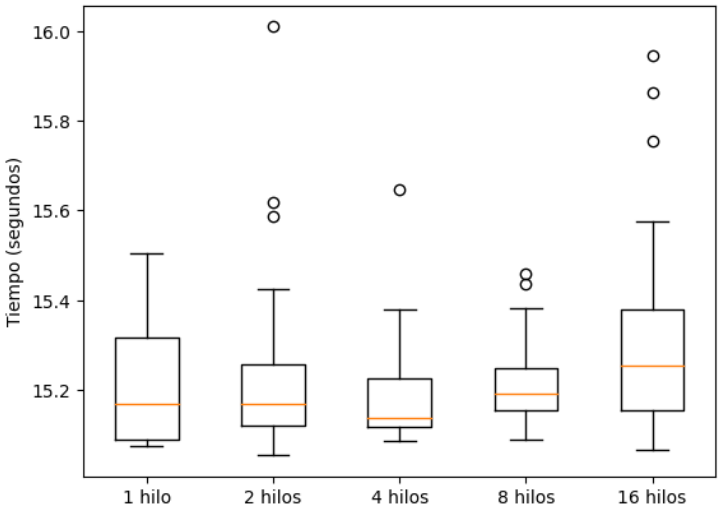

Figura 7.4: Tiempo de ejecución para OneMax, n=50 y población=2000 en jMetal sincrónico

Los resultados de esta prueba, demuestran que la mejora no es significativa, dado que el paralelismo solamente se aplica a la evaluación y en este caso no es una tarea que insuma demasiado tiempo. Incluso al aumentar demasiado la cantidad de hilos, el sistema termina degradándose.

Luego se ensaya una prueba en donde se cambia el tamaño de la población, fijando una cantidad de hilos igual a 4. De esto se obtienen los siguientes tiempos que se aprecian en la figura 7.5.

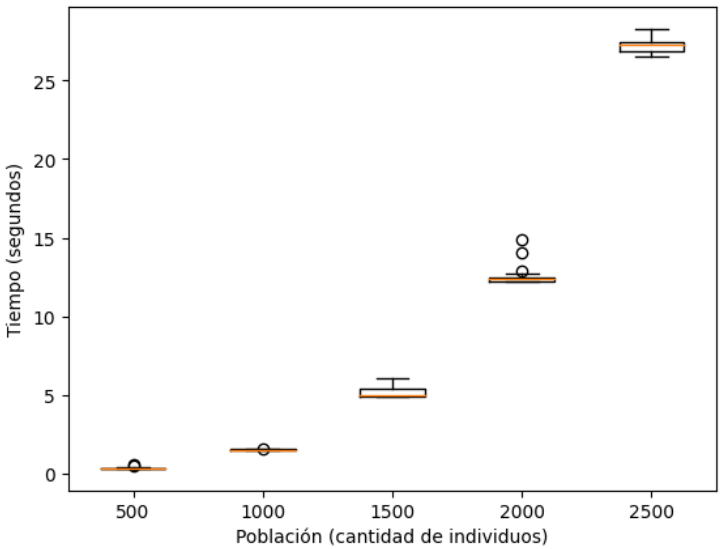

Figura 7.5: Tiempo de ejecución para OneMax, n=50 y población variable en jMetal sincrónico y 4 hilos

En este caso, el aumento vertiginoso de los tiempos demuestra que no logra escalar linealmente este tipo de paralelismo en cuanto al tamaño de la población.

En cuanto al paralelismo asincrónico, se puede mencionar que según los autores del *framework*, este mecanismo se basa en la evaluación temprana de los individuos, evitando esperas [94]. Para esto, la implementación ofrecida se basa en una topología de Maestro-Esclavo, llevada a cabo por varios hilos.

Se repiten las mismas pruebas anteriores y se obtienen los resultados de las figuras 7.6 y 7.7.

En la figura 7.6 se ve la disposición de tiempos insumidos en la culminación del algoritmo, respecto a la cantidad de hilos configurados y en la figura 7.7 cómo se comporta la herramienta para distintos tamaños de población, bajo una cantidad de hilos fija igual a 4.

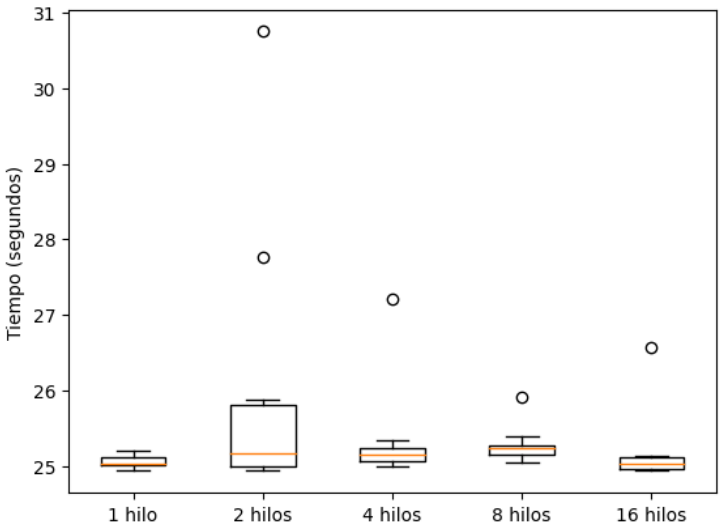

Figura 7.6: Tiempo de ejecución para OneMax, n=50 y población=2000 en jMetal asincrónico

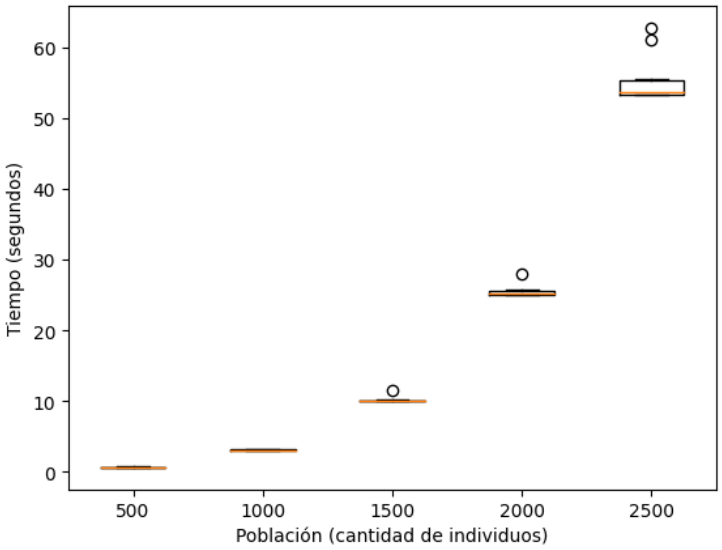

Figura 7.7: Tiempo de ejecución para OneMax, n=50 y población variable en jMetal asincrónico y 4 hilos

Se ve que no hay mejora significativa al aumentar la cantidad de hilos y además presenta las mismas dificultades al aumento del tamaño de la población.

Por su parte, la coordinación y creación de estructuras concurrentes parecen tener un impacto negativo, ya que en comparación con su contraparte sincrónica, presenta peores tiempos de respuesta.

En cuanto a la eficiencia, utilizando la mediana de tiempos para la solución sincrónica, se obtienen los resultados de la tabla 7.2 se puede visualizar esta información.

Cantidad de hilos	*Speedup* relativo	Eficiencia
2	0.99	0.50 (Sublineal)
4	1.00	0.25 (Sublineal)
8	1.00	0.13 (Sublineal)
16	0.99	0.06 (Sublineal)

Tabla 7.2: Tabla de *speedup* y eficiencia.

Resiliencia

En esta sección se quiere mencionar que *jMetal* no ofrece en sí mismo mecanismos para recuperarse frente a fallas.

Esto es responsabilidad del programador, codificando esta característica por su cuenta. Esto puede ser un problema para usuarios que no dominen el lenguaje *Java*, ya que tendrán que implementar estas funcionalidades o bien configurar alguna biblioteca para este fin.

7.4. Comparación entre *jMetal* y la solución propuesta

En esta sección se presenta un breve resumen de la comparativa de ambas herramientas.

7.4.1. Funcionalidad

En ambos casos se obtuvo el mismo resultado y no fue el óptimo del problema.

Hay que remarcar que, si bien el valor de *fitness* no fue el óptimo conocido, esto se debe a que se debió configurar adecuadamente el algoritmo evolutivo en cuestión. Esto es responsabilidad del usuario, ya que implica la elección de operadores evolutivos y su configuración específica, siendo una problemática separada.

En términos funcionales, la plataforma opera adecuadamente al ejecutar un algoritmo evolutivo, esto se refiere a la coordinación de microservicios, comunicación y colaboración para componer cada parte del algoritmo evolutivo.

7.4.2. Atributos de calidad

Aquí se brinda una comparativa en términos de los atributos de calidad más destacados en ambos casos.

Escalabilidad

La escalabilidad fue uno de los puntos más analizados. En esta parte se resumen los resultados obtenidos en cada parte a modo de comparación.

En cuanto a las medidas de *speedup* y eficiencia, la solución propuesta tiene mejor puntuación, obteniendo mejor rendimiento respecto a la cantidad de procesos paralelos agregados. Sin embargo, reporta peores tiempos en general, como se puede ver en la figura 7.8

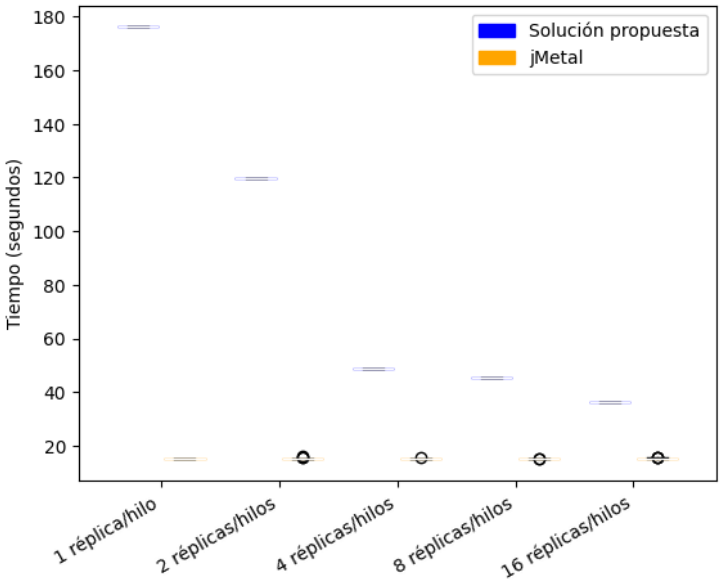

Figura 7.8: Comparación de tiempos entre soluciones de población 2000

Por otra parte, el sistema construido soporta un aumento de la población de individuos respondiendo con aumentos de tiempos lineales. Se aprecia que para poblaciones muy pequeñas *jMetal* supera ampliamente al desarrollo realizado, debiéndose seguramente al *overhead* que el sistema agrega. Sin embargo, el pasa a ser cada vez más despreciable para pruebas de mayor porte respecto al tamaño de la población.

En la figura 7.9 se ven los tiempos de ambas soluciones variando la población. Para la plataforma desarrollada se fijan 16 réplicas y para *jMetal* 4 hilos.

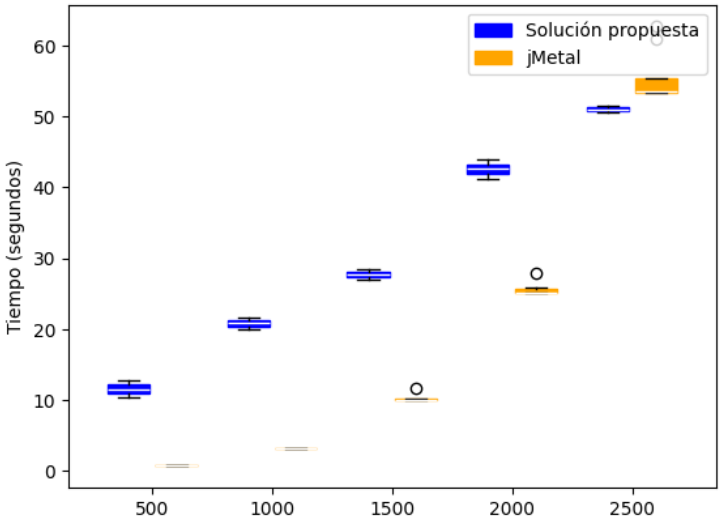

Figura 7.9: Comparación de tiempos entre jMetal con 4 hilos y la plataforma con 16 réplicas

Se puede visualizar que para una población de tamaño 2500, la plataforma logra mejorar los tiempos de *jMetal* por primera vez.

Resiliencia

Aquí simplemente hay que mencionar que la solución propuesta tiene ventaja sobre *jMetal*, ya que se ofrecen mecanismos de recuperación ante fallos de manera transparente.

Como se mencionó en la sección de *jMetal*, para poder agregar esta característica a un desarrollo basado en este *framework* es necesario tener conocimiento específico del lenguaje de programación en el que está construido.

En la solución propuesta, el desarrollador no debe preocuparse por estos aspectos, ya que son gestionados como parte de la plataforma. Esto, además, resulta especialmente útil cuando se realizan experimentos de larga duración, donde una falla no determinará el colapso del sistema ni el término abrupto del algoritmo evolutivo. Todo esto evita el desperdicio de horas de cómputo y la necesidad de volver a comenzar el experimento.

7.4.3. Uso de recursos

El uso de recursos impacta a la escalabilidad, pero en esta sección se desea brindar otra perspectiva de esta medida, ya que dicho atributo de calidad ya fue estudiado de forma separada.

Existen aspectos importantes a destacar respecto a la utilización de recursos que se presentan aquí.

Primero, hay que mencionar que la solución desarrollada requiere infraestructura y componentes extras a la aplicación que ejecuta algoritmos evolutivos. Con esto se quiere decir que hay recursos comprometidos a atender estos procesos, que se especializan en tareas de orquestación y administración de microservicios. Por ejemplo, se asigna a los componentes críticos de *Kubernetes* alrededor de 4 CPU y 6 GB de RAM para asegurar un correcto funcionamiento del *cluster*.

Además, hay que tomar en cuenta que la asignación de recursos a los despliegues asociados al *broker* de mensajería y caché distribuido también es considerable para que su rendimiento sea apropiado.

Entonces, el uso de la plataforma presume que, dada la capacidad total de cómputo de un equipo, la misma no será aprovechada en su totalidad para la ejecución de algoritmos evolutivos.

Los conectores desarrollados son un componente introducido para este tipo de arquitectura, por lo que su consumo de recursos debe ser pequeño. Esto se logra mediante una compilación nativa y empaquetado en una imagen vacía, conocida como *Scratch*, para que únicamente contenga el binario. Los resultados obtenidos son de microservicios, utilizan 1 MB de memoria.

Otro aspecto importante, a destacar, se posiciona desde el punto de vista de *jMetal*. Para las pruebas se decide no modificar el código fuente, es decir, utilizar las funcionalidades y clases provistas por el *framework*.

Al tomar este camino, se nota que no hay mejoras significativas al aumentar el paralelismo. Quizás sea por la implementación tomada por los autores o bien por alguna particularidad en la versión del entorno de ejecución, es decir, la JVM de Java.

Al monitorizar el uso de recursos del proceso Java, el mismo no logra utilizar más de 2 CPU, lo que seguramente lo limita para ser más efectivo en sus resoluciones. En este caso, esta solución no agrega tanto *overhead* como la contraparte de *Kubernetes*, no logra utilizar la capacidad disponible, como sí lo hace la plataforma desarrollada.

Es interesante la solución adaptada en la nube para escenarios donde se pueda utilizar el *software* como servicio.

7.5. Consideraciones adicionales

La elaboración de este proyecto no surge como respuesta a la necesidad de alguien, sino que su finalidad comprende, entre otras cosas, la validación de una determinada metodología y arquitectura para la familia de sistemas que implementan heurísticas de optimización. Si bien la motivación no es promovida entonces por una necesidad, este sistema tiene como fin el uso por parte de algún interesado, por lo que el conjunto de casos de uso y la experiencia de uso que se desprende de ellos debe ser puesta a prueba para conocer si el potencial de esta herramienta puede equiparar a las demás que se pueden utilizar hoy en día para modelar y ejecutar algoritmos evolutivos.

Una forma de obtener retroalimentación y comentarios acerca del producto desarrollado es mediante la realización de *Focus Groups*.

La técnica de *Focus Groups* establece la realización de discusiones planificadas con un grupo de interesados de entre 3 y 12 participantes. Como resultado se obtienen las percepciones del grupo [95].

Algunas de las etapas que comprenden la realización de esta técnica tienen que ver con la definición de qué aspectos van a ser estudiados en la sesión, para luego planificar el encuentro, que usualmente es comprendido entre 2 o 3 horas. La selección de los participantes puede traducirse en una de las tareas más difíciles, por el hecho de que deben ser representativos para el tipo de comunidad al que se intenta dirigir las consultas, en este caso corresponde a conocedores de algoritmos evolutivos o metaheurísticas de optimización. Una vez realizado lo anterior, culmina con la realización del encuentro y su posterior análisis de datos [95].

La realización de *Focus Groups* es planteada para este proyecto, pero no es concretada por las dificultades a la hora de encontrar un grupo para el mismo. Es una actividad que puede ser realizada en fases posteriores con miras a obtener una versión de alcance general y productiva para la comunidad.

La obtención de retroalimentación y comentarios enriquece el producto, logrando un ajuste en los casos de uso, apuntando a una mejor experiencia de usuario, ampliando la mirada y horizontes a lo que la comunidad necesita.

CAPÍTULO 8

CONCLUSIONES Y TRABAJO FUTURO

Contenido

8.1. Resumen del proyecto

La realización de este proyecto ha brindado diversas oportunidades de aprendizaje y mejora para crecer como profesional.

Se abarcaron dos grandes temas de forma conjunta, que son los algoritmos evolutivos y la arquitectura de microservicios. En particular, se ahondó en la adaptación de este tipo de sistemas a un ambiente enfocando en la utilización de la nube, con miras a lograr una implementación *cloud native*.

Se logran satisfactoriamente las actividades de análisis y diseño y se comienzan las etapas de implementación y distribución, pero con trabajo por delante. La versión actual del sistema se puede catalogar como *Alpha*, ya que su nivel de madurez debe ser mejorado y trabajado.

Se identifican además un conjunto de actividades de mejora y de nuevas funcionalidades.

8.2. Conclusiones

Las conclusiones obtenidas, luego de culminadas las actividades de investigación, diseño e implementación, se organizan en torno a dos grupos: las que tienen que ver con la forma en que se realizó el proyecto y las que tratan de los resultados obtenidos.

En cuanto al planteo de objetivos, se abarcan aspectos fundamentales para guiar a la meta de la creación de una metodología para adaptar las etapas de desarrollo, despliegue y administración de heurísticas de optimización a un modelo *cloud native*. El objetivo final es ambicioso; sin embargo, en esta primera instancia se identifican pequeños objetivos y variados para comenzar con la tarea.

En cuanto a la distribución del tiempo, se invirtió un tiempo considerable al análisis y lectura de publicaciones, lo que otorgó mayor seguridad y confianza en el diseño propuesto, pero influyó en una implementación más reducida a la planificada o esperada. El trabajo en solitario ayuda a administrar las tareas y decisiones más fácilmente porque no hay que negociar o discutir, pero resta riqueza en la solución y además recaen todas las responsabilidades en una sola persona, dilatando los tiempos.

Una vez terminado el desarrollo, se reconoce que el prototipo obtenido necesita mayor intervención para ser mejorado; sin embargo, el mayor valor que puede alcanzar la plataforma estaría dado por medio de instancias de refinamiento de los requisitos del sistema, por ejemplo, por medio de *Focus Group* y con esto ajustar la implementación, pruebas y validaciones en pos de liberar una versión productiva. En cuanto a las pruebas y validaciones, se reconoce que es un punto a mejorar, realizando tareas más formales, tanto puntuales como recurrentes dentro del flujo de construcción.

Por otra parte, las conclusiones que se obtienen respecto a la solución realizada son satisfactorias.

Una de las preguntas que dieron inicio a la realización de este proyecto tuvo respuesta y afirma que la arquitectura de microservicios es adecuada para este tipo de sistemas. Existen algunas condiciones para esta afirmación, ya que no siempre será el mejor diseño, pero en un entorno compartido y para un tipo de producto de *software* como servicio es beneficioso en términos de ahorro de recursos, abstracción y simplicidad en el desarrollo, escondiendo dificultades de infraestructura y mayor capacidad de compartir y reutilizar piezas de código entre miembros que utilizan la plataforma.

En cuanto a los objetivos planteados, se logran abarcar con diferentes niveles de profundidad.

Los objetivos 1, 2 y 3 que tratan del análisis e investigación fueron tratados en los capítulos iniciales, donde se relacionan las características de las diferentes arquitecturas y se releva información de las herramientas existentes y publicaciones relacionadas.

En cuanto al diseño, se logra una elaboración con base en las actividades de análisis, tomando inspiración de los diferentes aportes obtenidos de la lectura de casos similares y propuestas de diseño. Por esto, los objetivos 4 y 5 también son alcanzados.

Luego, el objetivo 6 abarca las tareas de implementación y validación de la propuesta. Aunque se logra desarrollar la plataforma, su nivel de madurez podría no ser suficiente para su uso en producción. Las pruebas realizadas cumplen con lo esperado, mostrando resultados interesantes en cuanto a los atributos de escalabilidad y resiliencia. Respecto a su evaluación funcional, se concluye que es satisfactoria.

Por último, y relacionado con el producto obtenido, se plantea un objetivo donde se debe establecer el trabajo a futuro, ya que, como se ha mencionado, es vital concientizar que hay grandes oportunidades de mejora y, si se aspira a lograr un producto funcional y de alcance general, es necesario iterar e incrementar el desarrollo y diseño de la solución.

Se han podido estudiar y poner en práctica diversos temas actuales referentes al modelo *cloud native*, lo que ha sido un desafío interesante a abordar durante la realización del proyecto.

8.3. Trabajo a futuro

El alcance definido y el resultado de la implementación han dejado un gran margen de mejoras y trabajo a abordar en el futuro.

Dentro de las actividades a realizar se encuentran mejoras asociadas a la deuda técnica en pos de lograr la primera liberación para su uso en producción, sustitución de implementaciones tecnológicas sin alterar el diseño logrado y, luego, la planificación de nuevas funcionalidades o mejoras.

8.3.1. Deuda técnica y versión productiva

La deuda técnica siempre está presente en los desarrollos y esta no es la excepción.

Para lograr cumplir con el desarrollo del prototipo, fue necesario prescindir de ciertas actividades durante la implementación.

No se han agregado pruebas automáticas en las piezas de código del *core* del sistema. Tanto la adición de *Unit Test*, como el análisis estático de código, serían fundamentales para establecer un producto seguro.

Lo anterior refiere a omisiones conscientes, pero resta por establecer mecanismos los cuales ayuden a encontrar defectos, ya sea por medio de una herramienta o revisiones de código.

La versión obtenida se puede clasificar como una versión *Alpha*, lo que evidencia que necesita mayor nivel de pruebas para lograr su contraparte estable.

8.3.2. Mejoras tecnológicas

Se plantearon algunas herramientas interesantes y útiles para llevar adelante diferentes aspectos del desarrollo, pero no pudieron ser agregadas en la versión actual.

Dentro de las herramientas a destacar está la prueba de *Kafka* como sustitución de *Nats* y el utilitario de *Chaos Monkey* para evaluar la resiliencia del sistema.

8.3.3. Nuevas funcionalidades

El producto apunta a pertenecer al grupo de SaaS, es decir, *software* como servicio, y para llegar a ese grado de servicio es requerido no solo mejorar el prototipo, sino que agregar nuevas funcionalidades.

Agentes de monitoreo y trazabilidad serían ideales para brindar transparencia en la ejecución, un *stack* tecnológico que se alinea a estas características tiene que brindar capacidad de centralizado de *logs*, métricas y trazas. A su vez, la adopción de herramientas de visualización dinámica aporta mucho valor al producto final.

El diseño contempló la existencia de usuarios que sean capaces de compartir los recursos del *cluster*, pero no llegó a ser implementado. Un manejo interno de usuarios y hasta mecanismos de autenticación externos basados en otras plataformas pueden convertirse en opciones viables.

La ejecución de experimentos puede convertirse en una tarea que es planificada y ejecutada durante períodos de tiempo largos en el orden de horas o días, por lo que el manejo de notificaciones por mail también fue previsto e incluso probado en etapas tempranas, pero no ha sido agregado en el prototipo.

La interfaz de usuario se define mediante una especificación de *OpenAPI*, lo que permite una documentación clara y efectiva de los distintos *endpoints REST*. No obstante, la distribución de un cliente para líneas de comando es una práctica común en muchos sistemas, ya que abstrae aún más al usuario de la interfaz y le permite interactuar directamente a través de comandos en una terminal. Además, la incorporación de una interfaz web podría resultar útil al garantizar una experiencia de usuario mejorada. Lo más relevante de estos puntos es que el núcleo del sistema está basado en las operaciones *REST*, lo que permite desarrollar estas interfaces de manera independiente en el futuro, sin necesidad de modificar el *core*.

Es necesario generar mecanismos genéricos de creación de nuevos algoritmos, ya sea con la creación de un lenguaje de dominio específico, explotando la riqueza y simplicidad de una solución declarativa.

Dado que los algoritmos evolutivos son métodos estocásticos, resulta importante ofrecer la posibilidad de intercambiar y usar diferentes métodos de generación de números pseudoaleatorios de forma transparente. Esto permite al usuario seleccionar entre distintas implementaciones o incluso servicios externos basados en fenómenos físicos verdaderamente aleatorios [96].

Por otra parte, la definición de mecanismos de búsqueda y selección no ha sido resuelta de forma general; los operadores relacionados con estas actividades son fijos

e implementan técnicas aleatorias, pero para brindar mayor flexibilidad para crear nuevos algoritmos, es necesario definir una interfaz de consulta de datos general y agnóstica de lenguaje.

En cuanto a funcionalidades que agreguen valor a la experiencia del usuario, se puede nombrar a la adición de capacidad de análisis de los resultados obtenidos en los experimentos, por ejemplo, con la obtención de gráficas o estadísticas a partir de los resultados o comparativas entre diferentes experimentos.

La clasificación de operadores agregando etiquetas y lógica de fondo puede permitir modelar variantes de los algoritmos, como por ejemplo las relaciones entre islas.

En la línea de mejoras, se encuentra la definición de infraestructura como código. En este caso, más que código, sería por medio de los manifiestos, evitando que sea imperativo. Hoy en día, los servicios que intervienen en un experimento son fijos y no se pueden modificar, por lo que sería una gran mejora otorgar la capacidad de que para que un mismo algoritmo se pueda tener la posibilidad de optar por diferentes herramientas, es decir, por ejemplo, escoger qué tipo de *broker* de mensajería desplegar. Asimismo, podría ser interesante definir diferentes tipos de topologías e incluso un lenguaje de dominio específico para declarar la infraestructura y las relaciones entre los servicios mediante algún esquema.

La concurrencia puede ser ampliamente mejorada aplicando un modelado del pseudocódigo de las metaheurísticas y agregando capacidad de expresión para tareas paralelas y dependencias.

Por último, la adición de un *pipeline* de construcción de imágenes a partir de los fuentes fue prevista en el diseño, pero no se agregó. Con esta característica, un usuario solamente debe codificar un operador, subirlo a un repositorio de código y, por medio de la referencia a él, la plataforma se encarga de construir la imagen correspondiente.

BIBLIOGRAFÍA

[1] *What is cloud native?* en-US. dirección: https://www.oracle.com/cloud/cloud-native/what-is-cloud-native/ (visitado 25-09-2023).

[2] *Who We Are*, en-US. dirección: https://www.cncf.io/about/who-we-are/ (visitado 25-09-2023).

[3] N. Rozanski y E. Woods, *Software systems architecture: working with stakeholders using viewpoints and perspectives*, 2nd ed. Upper Saddle River, NJ: Addison-Wesley, 2012, OCLC: ocn755213567, ISBN: 9780321718334.

[4] *OpenFING. Clase 2 de Algoritmos Evolutivos*. dirección: https://open.fing.edu.uy/courses/ae/2 (visitado 23-03-2023).

[5] *Philosophica: Enciclopedia filosófica on line — Voz: Teoría de la Evolución*, es. dirección: https://www.philosophica.info/voces/evolucion/Evolucion.html (visitado 20-03-2023).

[6] T. Back y H.-P. Schwefel, «Evolutionary computation: an overview,» en *Proceedings of IEEE International Conference on Evolutionary Computation*, Nagoya, Japan: IEEE, 1996, págs. 20-29, ISBN: 9780780329027. DOI: 10.1109/ICEC.1996.542329. dirección: http://ieeexplore.ieee.org/document/542329/ (visitado 04-05-2022).

[7] S. J. Russell, P. Norvig y E. Davis, *Artificial intelligence: a modern approach* (Prentice Hall series in artificial intelligence), 3rd ed. Upper Saddle River: Prentice Hall, 2010, ISBN: 9780136042594.

[8] R. Kurzweil, *The age of intelligent machines*, eng, 3. print. Cambridge, Mass: MIT Press, 1999, ISBN: 9780262610797.

[9] *Computación Evolutiva: The Next Big Thing - T3chFest 2019*, es. dirección: https://t3chfest.es/2019/programa/computacion-evolutiva-the-next-big-thing/ (visitado 24-03-2023).

[10] S. Nesmachnow, «Algoritmos genéticos paralelos y su aplicación al diseño de redes de comunicaciones confiables,» es, 2004. dirección: https://www.colibri.udelar.edu.uy/jspui/handle/20.500.12008/2932 (visitado 24-03-2023).

[11] S. Arora y B. Barak, *Computational complexity: a modern approach*, eng. Cambridge: Cambridge University Press, 2009, OCLC: 443221176, ISBN: 9780511532900.

[12] S. Luke, *Essentials of metaheuristics: a set of undergraduate lecture notes*, eng. Washington, DC: The author, 2010, OCLC: 1058914910, ISBN: 9780557148592.

[13] S. Tamilselvi, «Introduction to Evolutionary Algorithms,» en, en *Genetic Algorithms*, S. Ventura, J. María Luna y J. María Moyano, eds., IntechOpen, oct. de 2022, ISBN: 9781803551777. DOI: 10.5772/intechopen.104198. dirección: https://www.intechopen.com/chapters/81369 (visitado 24-03-2023).

[14] *Evolutionary Computation - Part 1*, en-US, abr. de 2016. dirección: https://www.alanzucconi.com/2016/04/06/evolutionary-coputation-1/ (visitado 27-03-2023).

[15] E. Alba y M. Tomassini, «Parallelism and evolutionary algorithms,» en, *IEEE Transactions on Evolutionary Computation*, vol. 6, n.º 5, págs. 443-462, oct. de 2002, ISSN: 1089-778X. DOI: 10.1109/TEVC.2002.800880. dirección: http://ieeexplore.ieee.org/document/1041554/ (visitado 31-01-2023).

[16] D. Sudholt, «Parallel Evolutionary Algorithms,» en, en *Springer Handbook of Computational Intelligence*, J. Kacprzyk y W. Pedrycz, eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, págs. 929-959, ISBN: 9783662435052. DOI: 10.1007/978-3-662-43505-2_46. dirección: http://link.springer.com/10.1007/978-3-662-43505-2_46 (visitado 03-05-2022).

[17] R. S. Barr y B. L. Hickman, «Feature Article—Reporting Computational Experiments with Parallel Algorithms: Issues, Measures, and Experts' Opinions,» en, *ORSA Journal on Computing*, vol. 5, n.º 1, págs. 2-18, feb. de 1993, ISSN: 0899-1499, 2326-3245. DOI: 10.1287/ijoc.5.1.2. dirección: http://pubsonline.informs.org/doi/10.1287/ijoc.5.1.2 (visitado 27-03-2023).

[18] M. Flynn, «Very high-speed computing systems,» *Proceedings of the IEEE*, vol. 54, n.º 12, págs. 1901-1909, 1966, ISSN: 0018-9219. DOI: 10.1109/PROC.1966.5273. dirección: http://ieeexplore.ieee.org/document/1447203/ (visitado 27-03-2023).

[19] E. Cantú-Paz, *Efficient and accurate parallel genetic algorithms* (Genetic algorithms and evolutionary computation 1). Boston, Mass: Kluwer Academic Publishers, 2000, ISBN: 9780792372219.

[20] L. Bass, P. Clements y R. Kazman, *Software architecture in practice*, eng, Third edition. Upper Saddle River, NJ: Addison-Wesley, 2013, OCLC: 825819423, ISBN: 9780132942775.

[21] G. Hohpe, *The software architect elevator: redefining the architect's role in the digital enterprise*, First edition. Beijing [China] ; Sebastopol, California: O'Reilly Media, 2020, ISBN: 9781492077541.

[22] H. Cervantes y R. Kazman, *Designing software architectures: a practical approach* (The SEI series in software engineering). Boston: Addison-Wesley, 2016, ISBN: 9780134390789.

[23] M. Richards y N. Ford, *Fundamentals of software architecture: an engineering approach*, First edition. Sebastopol, CA: O'Reilly Media, Inc, 2020, OCLC: on1089438191, ISBN: 9781492043454.

[24] I. Sommerville, *Software engineering* (Always learning), eng, Tenth edition, global edition. Boston Columbus Indianapolis New York San Francisco Hoboken Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo: Pearson, 2016, ISBN: 9781292096131.

[25] *Building Infrastructure Platforms*. dirección: https://martinfowler.com/articles/building-infrastructure-platform.html (visitado 11-10-2023).

[26] *The C4 model for visualising software architecture*. dirección: https://c4model.com/ (visitado 11-10-2023).

[27] S. Engineering, *Software Visualization — Challenge, Accepted*, en-US, jul. de 2022. dirección: https://engineering.atspotify.com/2022/07/software-visualization-challenge-accepted/ (visitado 11-10-2023).

[28] N. Ford, R. Parsons y P. Kua, *Building evolutionary architectures: support constant change*, First edition. Beijing: O'Reilly, 2017, OCLC: on1011099941, ISBN: 9781491986363.

[29] *1 What is architecture modernization? · Architecture Modernization MEAP V01*, en-US. dirección: https://livebook.manning.com/architecture-modernization/chapter-1/v-1 (visitado 21-03-2023).

[30] M. Richards, *Software architecture patterns*, eng. Place of publication not identified: O'Reilly Media, 2015, OCLC: 910936563, ISBN: 9781491924242.

[31] S. Newman, *Monolith to microservices: evolutionary patterns to transform your monolith*, eng, First edition. Beijing: O'Reilly, 2020, OCLC: 1303303317, ISBN: 9781492047810.

[32] E. Wolff, *Microservices: flexible software architecture*. Boston: Addison-Wesley, 2017, OCLC: ocn965730846, ISBN: 9780134602417.

[33] A. Megargel, C. M. Poskitt y V. Shankararaman, «Microservices Orchestration vs. Choreography: A Decision Framework,» en *2021 IEEE 25th International Enterprise Distributed Object Computing Conference (EDOC)*, Gold Coast, Australia: IEEE, oct. de 2021, págs. 134-141, ISBN: 9781665435796. DOI: 10.1109/EDOC52215.2021.00024. dirección: https://ieeexplore.ieee.org/document/9626189/ (visitado 24-10-2023).

[34] baeldung, *Saga Pattern in Microservices*. dirección: https://www.baeldung.com/cs/saga-pattern-microservices (visitado 23-10-2023).

[35] ByteByteGo, *EP67: Top 9 Microservice Best Practices*, en. dirección: https://blog.bytebytego.com/p/ep67-top-9-microservice-best-practices (visitado 05-10-2023).

[36] S. R. Goniwada, *Cloud native architecture and design: a handbook for modern day architecture and design with enterprise-grade examples*, eng. Berkeley: Apress, 2022, ISBN: 9781484272268.

[37] *¿Qué es la nube nativa?* es. dirección: https://cloud.google.com/learn/what-is-cloud-native?hl=es (visitado 25-09-2023).

[38] J. Domingus y J. Arundel, *Cloud native DevOps with Kubernetes: building, deploying, and scaling modern applications in the Cloud*, Second edition. Sebastopol, CA: O'Reilly Media, Inc, 2022, OCLC: on1282004009, ISBN: 9781098116828.

[39] J. MSV, *10 Key Attributes of Cloud Native Applications*, en-US, jul. de 2018. dirección: https://thenewstack.io/cloud-native/10-key-attributes-of-cloud-native-applications/ (visitado 25-09-2023).

[40] *jMetal*. dirección: https://jmetal.readthedocs.io/en/latest/index.html (visitado 28-03-2023).

[41] A. J. Nebro, J. Pérez-Abad, J. F. Aldana-Martin y J. García-Nieto, «Evolving a Multi-objective Optimization Framework,» en, en *Applied Optimization and Swarm Intelligence*, E. Osaba y X.-S. Yang, eds., Singapore: Springer Singapore, 2021, págs. 175-198, ISBN: 9789811606618. DOI: 10.1007/978-981-16-0662-5_9. dirección: https://link.springer.com/10.1007/978-981-16-0662-5_9 (visitado 29-03-2023).

[42] A. J. Nebro, J. J. Durillo y M. Vergne, «Redesigning the jMetal Multi-Objective Optimization Framework,» en, en *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, Madrid Spain: ACM, jul. de 2015, págs. 1093-1100, ISBN: 9781450334884. DOI: 10.1145/2739482.2768462. dirección: https://dl.acm.org/doi/10.1145/2739482.2768462 (visitado 24-03-2023).

[43] J. J. Durillo y A. J. Nebro, «jMetal: A Java framework for multi-objective optimization,» en, *Advances in Engineering Software*, vol. 42, n.º 10, págs. 760-771, oct. de 2011, ISSN: 09659978. DOI: 10.1016/j.advengsoft.2011.05.014. dirección: https://linkinghub.elsevier.com/retrieve/pii/S0965997811001219 (visitado 29-03-2023).

[44] *HeuristicLab*. dirección: https://dev.heuristiclab.com/trac.fcgi/ (visitado 28-03-2023).

[45] *Paradiseo*. dirección: https://nojhan.github.io/paradiseo/ (visitado 28-03-2023).

[46] S. Cahon, N. Melab y E.-G. Talbi, «ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics,» en, *Journal of Heuristics*, vol. 10, n.º 3, págs. 357-380, mayo de 2004, ISSN: 1381-1231. DOI: 10.1023/B:HEUR.0000026900.92269.ec. dirección: http://link.springer.com/10.1023/B:HEUR.0000026900.92269.ec (visitado 28-03-2023).

[47] J. Dreo, A. Liefooghe, S. Verel et al., «Paradiseo: from a modular framework for evolutionary computation to the automated design of metaheuristics: 22 years of Paradiseo,» en, en *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, Lille France: ACM, jul. de 2021, págs. 1522-1530, ISBN: 9781450383516. DOI: 10.1145/3449726.3463276. dirección: https://dl.acm.org/doi/10.1145/3449726.3463276 (visitado 28-03-2023).

[48] J. J. M. Guervós, P. Á. C. Valdivieso, G. R. López y M. G. Arenas, «Specifying evolutionary algorithms in XML,» en *Computational Methods in Neural Modeling*, G. Goos, J. Hartmanis, J. van Leeuwen, J. Mira y J. R. Álvarez, eds., vol. 2686, Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, págs. 502-509, ISBN: 9783540448686. DOI: 10.1007/3-540-44868-3_64. dirección: http://link.springer.com/10.1007/3-540-44868-3_64 (visitado 18-06-2022).

[49] P. García-Sánchez, J. González, P. A. Castillo, M. G. Arenas y J. J. Merelo-Guervós, «Service oriented evolutionary algorithms,» en, *Soft Computing*, vol. 17, n.º 6, págs. 1059-1075, jun. de 2013, ISSN: 1432-7643, 1433-7479. DOI: 10.1007/s00500-013-0999-5. dirección: http://link.springer.com/10.1007/s00500-013-0999-5 (visitado 04-05-2022).

[50] E. Apostol, I. Bălu̧tă, A. Gorgoi y V. Cristea, «A Parallel Genetic Algorithm Framework for Cloud Computing Applications,» en, en *Adaptive Resource Management and Scheduling for Cloud Computing*, F. Pop y M. Potop-Butucaru, eds., ép. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2014, págs. 113-127, ISBN: 9783319134642. DOI: 10.1007/978-3-319-13464-2_9.

[51] L. G. Rodriguez, H. A. Diosa y S. Rojas-Galeano, «Towards a component-based software architecture for genetic algorithms,» en *2014 9th Computing Colombian Conference (9CCC)*, Pereira, Colombia: IEEE, sep. de 2014, págs. 1-6, ISBN: 9781479967179. DOI: 10.1109/ColumbianCC.2014.6955331. dirección: http://ieeexplore.ieee.org/document/6955331/ (visitado 04-05-2022).

[52] P. Salza, «Parallel Genetic Algorithms in the Cloud,» Tesis doct., abr. de 2017. dirección: https://www.researchgate.net/publication/317625039 (visitado 21-09-2023).

[53] J. J. Merelo Guervós y J. M. García-Valdez, «Introducing an Event-Based Architecture for Concurrent and Distributed Evolutionary Algorithms,» en *Parallel Problem Solving from Nature – PPSN XV*, A. Auger, C. M. Fonseca, N. Lourenço, P. Machado, L. Paquete y D. Whitley, eds., vol. 11101, Cham: Springer International Publishing, 2018, págs. 399-410, ISBN: 9783319992532. DOI: 10.1007/978-3-319-99253-2_32. dirección: http://link.springer.com/10.1007/978-3-319-99253-2_32 (visitado 03-05-2022).

[54] J.-M. García-Valdez y J.-J. Merelo-Guervós, «A modern, event-based architecture for distributed evolutionary algorithms,» en *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ép. GECCO '18, New York, NY, USA: Association for Computing Machinery, jul. de 2018, págs. 233-234, ISBN: 9781450357647. DOI: 10.1145/3205651.3205719. dirección: https://doi.org/10.1145/3205651.3205719 (visitado 25-04-2022).

[55] J. J. Merelo y J.-M. García-Valdez, «Mapping evolutionary algorithms to a reactive, stateless architecture: using a modern concurrent language,» en, en *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, Kyoto Japan: ACM, jul. de 2018, págs. 1870-1877, ISBN: 9781450357647. DOI: 10.1145/3205651.3208317. dirección: https://dl.acm.org/doi/10.1145/3205651.3208317 (visitado 03-05-2022).

[56] P. Dziurzanski, S. Zhao, M. Przewozniczek, M. Komarnicki y L. S. Indrusiak, «Scalable distributed evolutionary algorithm orchestration using Docker containers,» en, *Journal of Computational Science*, vol. 40, pág. 101 069, feb. de 2020, ISSN: 18777503. DOI: 10.1016/j.jocs.2019.101069. direc-

ción: https://linkinghub.elsevier.com/retrieve/pii/S1877750319303333 (visitado 03-05-2022).

[57] T. Lenaerts y B. Manderick, «Building a genetic programming framework: The added-value of design patterns,» en *Genetic Programming*, G. Goos, J. Hartmanis, J. van Leeuwen et al., eds., vol. 1391, Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, págs. 196-208, ISBN: 9783540697589. DOI: 10.1007/BFb0055939. dirección: http://link.springer.com/10.1007/BFb0055939 (visitado 03-05-2022).

[58] B. Johnson Ralph and Foote, «Designing Reusable Classes,» *Journal of Object-Oriented Programming*, vol. 1, págs. 22-35, jun. de 1988.

[59] P. García-Sánchez, «Service Oriented Architecture for Adaptive Evolutionary Algorithms: Implementation and Applications,» Tesis doct., 2014. dirección: https://figshare.com/articles/thesis/Service_Oriented_Architecture_for_Adaptive_Evolutionary_Algorithms_Implementation_and_Applications/1166547/1 (visitado 21-09-2023).

[60] J. A. Parejo, A. Ruiz-Cortés, S. Lozano y P. Fernandez, «Metaheuristic optimization frameworks: a survey and benchmarking,» en, *Soft Computing*, vol. 16, n.º 3, págs. 527-561, mar. de 2012, ISSN: 1432-7643, 1433-7479. DOI: 10.1007/s00500-011-0754-8. dirección: http://link.springer.com/10.1007/s00500-011-0754-8 (visitado 26-09-2023).

[61] J. Ingeno, *Software architect's handbook: become a successful software architect by implementing effective architecture concepts*, eng. Birmingham Mumbai: Packt, 2018, ISBN: 9781788624060.

[62] S. Wagner, S. Winkler, E. Pitzer et al., «Benefits of Plugin-Based Heuristic Optimization Software Systems,» en, en *Computer Aided Systems Theory – EUROCAST 2007*, R. Moreno Díaz, F. Pichler y A. Quesada Arencibia, eds., vol. 4739, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, págs. 747-754, ISBN: 9783540758662. DOI: 10.1007/978-3-540-75867-9_94. dirección: http://link.springer.com/10.1007/978-3-540-75867-9_94 (visitado 26-09-2023).

[63] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu y W. Zhou, «A Comparative Study of Containers and Virtual Machines in Big Data Environment,» en *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, San Francisco, CA, USA: IEEE, jul. de 2018, págs. 178-185, ISBN: 9781538672358. DOI: 10.1109/CLOUD.2018.00030. dirección: https://ieeexplore.ieee.org/document/8457798/ (visitado 18-10-2023).

[64] A. Xu. «What are the differences between Virtualization (VMware) and Containerization (Docker)?» en. (), dirección: https://blog.bytebytego.com/p/what-are-the-differences-between (visitado 12-03-2024).

[65] N. Kratzke, «About Microservices, Containers and their Underestimated Impact on Network Performance,» en, 2015. DOI: 10.13140/RG.2.1.2039.3046. dirección: http://rgdoi.net/10.13140/RG.2.1.2039.3046 (visitado 16-05-2022).

[66] H. Khalloof, P. Ostheimer, W. Jakob, S. Shahoud, C. Duepmeier y V. Hagenmeyer, «Superlinear Speedup of Parallel Population-Based Metaheuristics: A Microservices and Container Virtualization Approach,» en, en *Intelligent Data Engineering and Automated Learning – IDEAL 2019*, H. Yin, D. Camacho, P. Tino, A. J. Tallón-Ballesteros, R. Menezes y R. Allmendinger, eds., vol. 11871, Cham: Springer International Publishing, 2019, págs. 386-393, ISBN: 9783030336073. DOI: 10.1007/978-3-030-33607-3_42. dirección: http://link.springer.com/10.1007/978-3-030-33607-3_42 (visitado 04-05-2022).

[67] H. Khalloof, P. Ostheimer, W. Jakob, S. Shahoud, C. Duepmeier y V. Hagenmeyer, «A Distributed Modular Scalable and Generic Framework for Parallelizing Population-Based Metaheuristics,» en, en *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, E. Deelman, J. Dongarra y K. Karczewski, eds., vol. 12043, Cham: Springer International Publishing, 2020, págs. 432-444, ISBN: 9783030432287. DOI: 10.1007/978-3-030-43229-4_37. dirección: http://link.springer.com/10.1007/978-3-030-43229-4_37 (visitado 06-10-2023).

[68] A. Bollini y M. Piastra, «Distributed and Persistent Evolutionary Algorithms: A Design Pattern,» en *Genetic Programming*, R. Poli, P. Nordin, W. B. Langdon y T. C. Fogarty, eds., vol. 1598, Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, págs. 173-183, ISBN: 9783540488859. DOI: 10.1007/3-540-48885-5_14. dirección: http://link.springer.com/10.1007/3-540-48885-5_14 (visitado 17-05-2022).

[69] J. L. J. Laredo, A. E. Eiben, M. Van Steen y J. J. Merelo, «EvAg: a scalable peer-to-peer evolutionary algorithm,» en, *Genetic Programming and Evolvable Machines*, vol. 11, n.º 2, págs. 227-246, jun. de 2010, ISSN: 1389-2576, 1573-7632. DOI: 10.1007/s10710-009-9096-z. dirección: http://link.springer.com/10.1007/s10710-009-9096-z (visitado 28-09-2023).

[70] A. Munawar, M. Wahib, M. Munetomo y K. Akama, «The design, usage, and performance of GridUFO: A Grid based Unified Framework for Optimization,» en, *Future Generation Computer Systems*, vol. 26, n.º 4, págs. 633-644, abr. de 2010, ISSN: 0167739X. DOI: 10.1016/j.future.2009.12.001. dirección: https://linkinghub.elsevier.com/retrieve/pii/S0167739X09001800 (visitado 04-05-2022).

[71] P. Salza, F. Ferrucci y F. Sarro, «Develop, Deploy and Execute Parallel Genetic Algorithms in the Cloud,» en, en *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, Denver Colorado USA: ACM, jul. de 2016, págs. 121-122, ISBN: 9781450343237. DOI: 10.1145/2908961.2909024. dirección: https://dl.acm.org/doi/10.1145/2908961.2909024 (visitado 03-05-2022).

[72] S. Di Martino, F. Ferrucci, V. Maggio et al., *Towards Migrating Genetic Algorithms for Test Data Generation to the Cloud*, English, cap., ene. de 2001. dirección: https://www.igi-global.com/gateway/chapter/www.igi-global.com/gateway/chapter/72229 (visitado 18-06-2022).

[73] A. Verma, X. Llorà, D. E. Goldberg y R. H. Campbell, «Scaling Genetic Algorithms Using MapReduce,» en *2009 Ninth International Conference on Intelligent Systems Design and Applications*, ISSN: 2164-7151, nov. de 2009, págs. 13-18. DOI: 10.1109/ISDA.2009.181.

[74] *Software Architecture in Practice, 4th Edition*, eng, 1st edition. [Erscheinungsort nicht ermittelbar] Addison-Wesley Professional 2021, 2021, OCLC: 1247847006, ISBN: 9780136885979.

[75] *Considerations for large clusters*, en. dirección: https://kubernetes.io/docs/setup/best-practices/cluster-large/ (visitado 20-11-2023).

[76] *¿Qué es KVM?* Dirección: https://www.redhat.com/es/topics/virtualization/what-is-KVM (visitado 09-10-2024).

[77] *Installing Kubernetes with deployment tools*, en. dirección: https://kubernetes.io/docs/setup/production-environment/tools/ (visitado 20-11-2023).

[78] *Infinite message deduplication in JetStream*, en-us, nov. de 2022. dirección: https://nats.io/blog/new-per-subject-discard-policy/ (visitado 19-11-2023).

[79] *NATS by Example - Change Data Capture via Debezium (CLI)*. dirección: https://natsbyexample.com/examples/integrations/debezium/cli (visitado 19-11-2023).

[80] *Microservices Pattern: Transactional outbox*. dirección: http://microservices.io/patterns/data/transactional-outbox.html (visitado 19-11-2023).

[81] *Concept model*, en. dirección: https://tekton.dev/docs/concepts/concept-model/ (visitado 23-11-2023).

[82] *Triggers and EventListeners*, en. dirección: https://tekton.dev/docs/triggers/ (visitado 23-11-2023).

[83] *REST API*, es. dirección: https://www.redhat.com/es/topics/api/what-is-a-rest-api (visitado 19-11-2023).

[84] M. Fowler, *Richardson Maturity Model*. dirección: https://martinfowler.com/articles/richardsonMaturityModel.html (visitado 21-03-2023).

[85] *What is OpenAPI?* en-US. dirección: https://www.openapis.org/what-is-openapi (visitado 14-12-2023).

[86] *Pod networking*, en. dirección: https://kubernetes.io/docs/concepts/workloads/pods/ (visitado 01-12-2023).

[87] *Microservices Pattern: Pattern: Server-side service discovery*, microservices.io. dirección: http://microservices.io/patterns/server-side-discovery.html (visitado 28-10-2024).

[88] *Protocol Buffers*, en. dirección: https://protobuf.dev/ (visitado 11-12-2023).

[89] *Introduction to gRPC*, en. dirección: https://grpc.io/docs/what-is-grpc/introduction/ (visitado 11-12-2023).

[90] M. Gravell, *Answer to "How are protocol-buffers faster than XML and JSON?"* Sep. de 2018. dirección: https://stackoverflow.com/a/52148242 (visitado 11-12-2023).

[91] *5 Stages of the Software Release Life Cycle*. dirección: https://themanifest.com/software-development/blog/software-release-life-cycle (visitado 22-11-2023).

[92] V. Kanumuri, *Kubernetes HPA using Custom Metrics*, en, sep. de 2019. dirección: https://medium.com/@avkanumuri/kubernetes-hpa-using-custom-metrics-b14d70e50818 (visitado 30-11-2023).

[93] *Horizontal Pod Autoscaling*, en. dirección: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (visitado 29-11-2023).

[94] *Sub-module jmetal-parallel*. dirección: https://jmetal.readthedocs.io/en/latest/parallel.html (visitado 28-10-2024).

[95] J. Kontio, L. Lehtola y J. Bragge, «Using the focus group method in software engineering: obtaining practitioner and user experiences,» en *Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE '04.*, ago. de 2004, págs. 271-280. DOI: 10.1109/ISESE.2004.1334914. dirección: https://ieeexplore.ieee.org/document/1334914/ (visitado 22-11-2023).

[96] *What's this fuss about true randomness?* Dirección: https://www.random.org (visitado 12-12-2024).

Esta es la última página.
Compilado el viernes 27 diciembre, 2024.