

Facultad de Tecnología Informática
Universidad Abierta Interamericana

Departamento de **Centro de Altos Estudios en Tecnología Informática**

Programa de Doctorado en **Informática**

Descubrimiento y diseño de microservicios asistido por IA a partir de requisitos textuales

Daniel Narváez
ORCID: 0009-0006-0210-6871

Director: **Dr. Gustavo Rossi**
Co-director: **Dr. Alejandro Fernández**

Año académico: **2026**

Quiero dedicar este trabajo.

A mi mamá, mi más grande ejemplo de fortaleza y resiliencia. Ella me enseñó a seguir adelante ante cualquier adversidad, trabajando incansablemente tanto para sustentar mi vida como para ser el pilar fundamental y nexo de conexión entre toda la familia. Su fuerza y su entrega incondicional me dieron la educación y la base para todo lo que soy.

A mi papá, que siempre me inculcó la idea de que la educación era el mayor bien que podía heredarme. Él me inspiró a seguir adelante hasta alcanzar mi meta de ser ingeniero, logro que tuve la dicha de compartir junto a él. Aunque ya no estés físicamente conmigo, tu presencia me acompaña en cada paso; sin ti yo no estaría aquí.

A Helen Ampié, mi compañera de vida. Conocerte cambió el rumbo de mi existencia; tu apoyo incondicional y tu confianza en mí fueron la luz que necesitaba para redireccionar mi vida y llegar hasta aquí. Este logro es tan tuyo como mío.

¡Lo logramos, papá! O, como tú dirías: "Lo lograste, jincho".

Resumen

Esta tesis investiga el problema de descubrir y diseñar arquitecturas de microservicios a partir de requisitos textuales mediante un enfoque generar–verificar. Se propone ArchiGenMS, un pipeline que integra: (i) modelos generativos guiados por prompt engineering y búsqueda evolutiva para producir candidatos arquitectónicos; y (ii) verificación formal en Lean junto con métricas estructurales (cohesión, granularidad y acoplamiento) para evaluar y certificar propiedades de calidad a nivel de diseño.

La metodología se enmarca en Design Science Research (DSR), articulando un ciclo de relevancia (problema de diseño greenfield desde texto), un ciclo de rigor (síntesis sistemática del estado del arte y formalización de métricas) y un ciclo de diseño (construcción y evaluación del artefacto). El resultado es un proceso extremo a extremo que parte de historias de usuario y artefactos textuales, genera alternativas arquitectónicas consistentes y aplica garantías formales sobre restricciones estructurales y umbrales operacionales de calidad. La verificación formal no solo valida, sino que guía la búsqueda de soluciones hacia regiones del espacio arquitectónico con mayor cohesión, menor acoplamiento y granularidad adecuada, reduciendo diseños plausibles pero estructuralmente deficientes.

Se presentan experimentos sobre conjuntos de requisitos en dominios heterogéneos, reportando: (a) convergencia del fitness evolutivo; (b) mejoras en las distribuciones de las métricas estructurales; (c) coste controlado de verificación por individuo; y (d) robustez mediante amenazas a la validez (constructo, interna, externa y de conclusión), control de semillas, y paquete de replicación.

Las contribuciones son: (1) un artefacto verificable para el descubrimiento y diseño de microservicios desde texto; (2) una formalización reusable de métricas estructurales en Lean y su integración en una función de aptitud con penalizaciones suaves; (3) un protocolo experimental reproducible (código, prompts versionados y datos) que permite evaluar calidad arquitectónica en escenarios greenfield. Finalmente, se delinean líneas futuras en NFRs dinámicos, propiedades temporales y transferencia industrial.

Palabras clave: IA neuro-simbólica, microservicios, modelos generativos, búsqueda evolutiva, verificación formal, Lean 4, métricas arquitectónicas, DSR.

Lista de publicaciones derivadas de la tesis

Revistas (Journals)

- D. Narváez, N. Battaglia, A. Fernández y G. Rossi, "Designing microservices using AI: A systematic literature review", *Software*, vol. 4, no. 1, art. 6, 2025, MDPI.

Trabajos Premiados

- D. Narváez y A. Fernández, "Aplicación de Inteligencia Artificial Generativa y Verificación Formal en el Descubrimiento de Microservicios", publicado en *Revista Abierta de Informática Aplicada (RAIA – UAI)*. (Trabajo galardonado con el Primer Lugar, Categoría Posgrado, en el Congreso Internacional de Innovación Tecnológica CIITI-TE 2025).

Congresos (Conferences)

- D. Narváez, N. Battaglia, A. Fernández y G. Rossi, "Descubrimiento automático de microservicios mediante modelos generativos y verificación formal", en *XXXI Congreso Argentino de Ciencias de la Computación (CACIC)*, Viedma, Argentina, 2025.
- D. Narváez, "Explorando el Uso de Inteligencia Artificial en el Descubrimiento y Diseño de Microservicios", en *Congresso Ibero-Americano em Engenharia de Software (CIbSE)*, 2025, pp. 224–231, SBC. (Simposio Doctoral).
- D. Narváez, G. H. Rossi y N. Battaglia, "Aplicación de Inteligencia Artificial en el Diseño de Microservicios", en *XXX Congreso Argentino de Ciencias de la Computación (CACIC)*, La Plata, Argentina, 2024.

Trabajos en Proceso de Evaluación (Submitted)

- D. Narváez, N. Battaglia, G. Rossi y A. Fernández, "ArchiGenMS: A Neuro-Symbolic Evolutionary Framework for Greenfield Microservice Design", enviado al *XXIX Congresso Ibero-Americano em Engenharia de Software (CIbSE 2026)*, Santo Domingo, República Dominicana. (Estado: Bajo revisión).
- D. Narváez, N. Balich y G. Rossi, "MAPE-KV: Una Arquitectura Neuro-Simbólica Verificada para la Orquestación de Enjambres Robóticos", enviado al *XXIX Congresso Ibero-Americano em Engenharia de Software (CIbSE 2026)*, Santo Domingo, República Dominicana. (Estado: Bajo revisión).

Índice general

Lista de publicaciones derivadas de la tesis I

Introducción 1

1. Introducción 2
1.1. Objetivos de la tesis . . . 5
1.2. Contribuciones principales . . . 5
1.3. Estructura de la tesis . . . 7

2. Estado del Arte 8
2.1. Arquitectura de Microservicios: Principios y Retos . . . 9
2.1.1. AI4SE: panorama breve y posicionamiento . . . 12
2.2. Uso de IA en el Descubrimiento y Diseño de Microservicios . . . 12
2.2.1. Enfoques Basados en Migración (Sistemas *Brownfield*) . . . 13
2.2.2. Enfoques para el Diseño *Greenfield* (Basados en Texto) . . . 14
2.2.3. Tendencias Emergentes: IA Generativa y LLMs . . . 15
2.2.4. LLMs para arquitectura: trazabilidad, justificabilidad y verificación . . . 16
2.3. Greenfield vs. Migración: Panorama y Herramientas Industriales . . . 16
2.4. Métricas Arquitectónicas para Evaluación de Microservicios . . . 16
2.4.1. Formulación matemática de métricas estructurales . . . 17
2.5. Métodos Formales en Ingeniería de Software . . . 19
2.5.1. Model checking vs. asistentes de prueba para calidad estructural . . . 19
2.6. Reproducibilidad y Reporte Experimental en Enfoques IA+Formales . . . 20
2.7. Datasets Públicos y Variabilidad Lingüística de Requisitos . . . 20
2.8. Protocolo de Evaluación y Reproducibilidad . . . 21
2.9. Posicionamiento de esta Tesis respecto del Estado del Arte . . . 21
2.10. Discusión y Vacíos Identificados . . . 21

3. Fundamentos Teóricos y Matemáticos 23
3.1. Fundamentos de Arquitectura de Microservicios . . . 23
3.2. Representación Formal de Requisitos Textuales . . . 25
3.3. Modelo Matemático Mínimo de Arquitectura . . . 25
3.4. Preliminares matemáticos . . . 26
3.5. Microcasos ilustrativos de cómputo de métricas . . . 27
3.5.1. Cómputo Métrico-Formal: De Genotipo JSON a Aptitud (Fitness) . . . 27

3.6. Métricas Arquitectónicas Computadas en el Experimento . . . 31
3.7. Lean 4 como Validador Computable . . . 32
3.7.1. Panorama . . . 32
3.7.2. Rol concreto en el pipeline . . . 33
3.7.3. Fragmento ilustrativo . . . 33
3.8. Conexión con el Algoritmo Evolutivo . . . 34
3.8.1. Función de aptitud . . . 34
3.8.2. Generación, validación, selección . . . 34
3.8.3. Reproducibilidad y sensibilidad . . . 34
3.9. Discusión, Limitaciones y Extensiones . . . 34
3.10. Síntesis . . . 35

4. Trabajos Relacionados **36**
4.1. Microservicios: evolución, patrones y desafíos persistentes . . . 36
4.2. IA para el descubrimiento y diseño de microservicios . . . 37
4.3. Métodos formales y métricas arquitectónicas . . . 39
4.4. Delimitación: Calidad Estructural Estática vs. Optimización Dinámica en Runtime . 40
4.5. Reproducibilidad, trazabilidad y comparación de enfoques . . . 41
4.6. Vacíos identificados y posicionamiento de esta tesis . . . 43
4.7. Síntesis crítica . . . 43

5. Enfoque Propuesto: Descubrimiento de Microservicios usando IA **45**
5.1. Metodología de Investigación (Design Science Research) . . . 45
5.2. Visión General del Enfoque . . . 46
5.3. Extracción Evolutiva de Microservicios . . . 47
5.4. Ingeniería de Prompts y Modelos Generativos . . . 51
5.5. Evaluación con Métricas Arquitectónicas . . . 52
5.6. Validación Formal en Lean . . . 54
5.7. Integración del Pipeline Completo . . . 56
5.7.1. Ubicación de la propuesta dentro de AI4SE . . . 57
5.8. Síntesis del Enfoque Propuesto . . . 58

6. Arquitectura y Herramientas de Soporte **59**
6.1. Arquitectura General del Sistema . . . 59
6.2. Entorno de Ejecución y Replicabilidad . . . 62
6.3. Integración con Lean 4 y Verificación Formal . . . 64
6.4. Módulo de Métricas y Evaluación Automática . . . 65
6.4.1. Criterio de *fitness* y métricas . . . 65
6.5. Orquestación y Flujo de Ejecución . . . 67
6.6. Síntesis del Capítulo . . . 68

7. Evaluación y Resultados **70**
7.1. Diseño Experimental . . . 70
7.2. Datasets y Escenarios de Prueba . . . 72
7.3. Resultados Experimentales . . . 73
7.3.1. Convergencia del proceso evolutivo . . . 73
7.3.2. Mejor individuo por escenario: patrones y casos atípicos . . . 76

7.3.3. Relación entre granularidad y acoplamiento 79
7.3.4. Factibilidad computacional 80
7.4. Discusión y Análisis Crítico 80
7.5. Limitaciones y Amenazas a la Validez 82
7.6. Síntesis del Capítulo 83

Conclusiones y Líneas Futuras **85**
7.7. Síntesis general de la investigación 85
7.8. Aportaciones principales 86
7.9. Limitaciones del estudio 87
7.10. Líneas futuras de investigación 88
7.11. Cierre 89

Conclusions and Outlook **85**

Bibliography **90**

A. Notación y convenciones **96**

Notación y convenciones **96**

B. Definiciones operativas y función de evaluación **97**

Definiciones operativas y función de evaluación **97**

Agradecimientos **100**

Acknowledgements **100**

Índice de figuras

2.1. Diagrama conceptual de una Arquitectura de Microservicios, ilustrando el API Gateway, servicios independientes, bases de datos descentralizadas y comunicación síncrona/asíncrona. 10

3.1. Ilustración de los principios de diseño de Cohesión (Alta vs. Baja) y Acoplamiento (Bajo vs. Alto). 24
3.2. Visualización del modelo matemático mínimo de una arquitectura $\mathcal{A} = (S, E, O, P)$. 26

5.1. Visión general de ArchiGenMS. El pipeline integra generación evolutiva de microservicios mediante LLMs, verificación formal con Lean y retroalimentación basada en métricas arquitectónicas. 47
5.2. Extracción evolutiva de microservicios a partir de requisitos textuales. El proceso integra LLMs, ingeniería de *prompts*, validación en Lean y selección iterativa mediante métricas arquitectónicas. 49
5.3. Estructura del genotipo evaluado en ArchiGenMS. La representación intermedia incluye microservicios, operaciones, parámetros, dependencias y metadatos, lo que permite su validación formal y análisis métrico. 50
5.4. Esquema de ingeniería evolutiva de *prompts* en ArchiGenMS. Los requisitos textuales se transforman en un *prompt* inicial que genera un genotipo arquitectónico; los resultados métricos retroalimentan el proceso, guiando la creación de variantes y la selección de mejores candidatos. 51
5.5. Integración del pipeline ArchiGenMS. Los requisitos textuales se transforman en arquitecturas de microservicios mediante un ciclo evolutivo de generación, evaluación y verificación formal. 57

6.1. Arquitectura general de ArchiGenMS en alto nivel. 60
6.2. Arquitectura de ArchiGenMS organizada por capas y módulos funcionales. 61

7.1. Convergencia promedio de la función de aptitud $f(G)$ (menor es mejor) a lo largo de cinco generaciones. 73
7.2. Distribución de LCOM_{avg} por generación (individuos válidos). 74
7.3. Distribución de $\text{Coupling}_{\text{max}}$ por generación (individuos válidos). 75
7.4. Intervalos de confianza (95 %) por bootstrap para $f(G)$ promedio por generación. . . 75
7.5. Intervalos de confianza (95 %) por bootstrap para LCOM_{avg} promedio por generación. 76
7.6. LCOM_{avg} de los mejores individuos por escenario (orden ascendente). 78

7.7. Arquitectura descubierta automáticamente para el caso g24-unibath. El diagrama ilustra el flujo de dominio y el bajo acoplamiento (Coupling_{max}=1) del artefacto generado. 79
7.8. Relación SGM_{max}–$\text{Coupling}_{\text{max}}$ en mejores individuos por escenario (Spearman ϱ y recta guía). 80

Índice de cuadros

2.1. Panorama comparativo de retos y evidencias reportadas en la literatura 12
2.2. Comparativa sintética de líneas relacionadas y vacíos identificados 17
2.3. Matriz de brechas metodológicas y línea de solución adoptada 22

4.1. Comparación sintética de líneas de trabajo relevantes 42

6.1. Resumen de capas y módulos de ArchiGenMS 62
6.2. Configuración de hardware y software utilizada 63

7.1. Resumen de proyectos incluidos en el dataset de Dalpiaz (2018) 72
7.2. Resumen de genotipos óptimos por proyecto 77
7.3. Métricas de calidad para el caso g24-unibath (mejor generación). 79

B.1. Definiciones operativas utilizadas a lo largo de la tesis 97

Lista de abreviaturas y símbolos

AI4SE Artificial Intelligence for Software Engineering.

API Application Programming Interface.

DDD Domain-Driven Design.

DSR Design Science Research.

E2E End-to-End.

FO (Fan-Out) / FI (Fan-In) Dependencias salientes/entrantes entre servicios.

JSON JavaScript Object Notation.

LCOM Lack of Cohesion of Methods (métrica de cohesión).

LLM Large Language Model.

MSL Mapeo Sistemático de la Literatura.

NFRs Non-Functional Requirements.

NLP Natural Language Processing (Procesamiento de Lenguaje Natural).

OOAD Object-Oriented Analysis and Design.

RQ Research Question (pregunta de investigación).

SBSE Search-Based Software Engineering.

SGM / SGM-SD Métrica de granularidad y su desviación estándar.

SLR Systematic Literature Review.

Capítulo 1

Introducción

La ingeniería de software contemporánea enfrenta una tensión fundamental: la necesidad de desarrollar sistemas escalables y mantenibles a gran velocidad [1], frente a la exigencia de garantizar rigor arquitectónico y robustez estructural [2]. Las arquitecturas de microservicios se han consolidado como respuesta predominante a los requerimientos de escalabilidad y despliegue independiente [3], ofreciendo modularidad que favorece resiliencia y entrega continua [1]. Esta transición está impulsada por entornos *cloud* donde las organizaciones requieren iterar rápidamente [4].

Esta transición, sin embargo, expone una contradicción metodológica central. La agilidad, como paradigma de gestión dominante, prioriza la respuesta al cambio y desalienta el diseño arquitectónico inicial exhaustivo (*Big Design Upfront* o BDUF). No obstante, la arquitectura de microservicios, para ser exitosa, exige decisiones de diseño estratégicas y correctas desde el inicio [1]. Como lo demuestra la evidencia empírica de la industria, la arquitectura compleja y los sistemas de mensajería que requieren los microservicios demandan un «esfuerzo de planificación que contradice la agilidad» [5]. De hecho, estudios de campo recientes confirman esta fricción, evidenciando que los equipos ágiles, en la práctica, continúan aplicando métodos de diseño y estimación tradicionales (como OOAD) que no están adaptados a la naturaleza de los microservicios [6].

La adopción de este paradigma, si bien beneficiosa, no es trivial. El desafío fundamental no reside en la implementación de un servicio individual, sino en el proceso de descomposición arquitectónica: decidir cómo particionar una aplicación compleja en un conjunto de servicios más pequeños e independientes [3, 7]. A diferencia de una arquitectura monolítica tradicional, donde todos los componentes (p. ej., la gestión de usuarios, el procesamiento de pagos y el inventario) coexisten en una única base de código, los microservicios exigen que estos componentes operen de forma autónoma. Este proceso de descubrimiento, especialmente en metodologías ágiles, sigue dependiendo de técnicas heurísticas y manuales como el *Domain-Driven Design* (DDD) o el *Event Storming* [8].

Este desafío de descomposición se magnifica en los escenarios *greenfield* (desarrollo desde cero), que son el foco de esta tesis. A diferencia de la migración *brownfield*, donde existen artefactos estructurales observables (código, bases de datos) [9, 10], el *greenfield* carece de ellos. El proceso de diseño ágil dicta que la única entrada disponible son los requisitos textuales, fragmentados e incrementales, tales como **historias de usuario** [11] o **casos de uso** [12].

El problema fundamental, por tanto, es el **descubrimiento**: cómo derivar una arquitectura robusta a partir de artefactos textuales ambiguos y sin un diseño inicial. La literatura industrial confirma que esta es la dificultad principal; el problema más reportado por los profesionales es

el de los **«cortes incorrectos» (*Wrong Cuts*)**, definido como la dificultad para encontrar una estrategia de descomposición efectiva y saber «dónde 'cortar' el sistema», algo que «aún no está claramente definido» [5].

Esta decisión de diseño es crítica. Una partición incorrecta puede generar una complejidad accidental significativamente mayor que la del monolito original. Si los servicios definidos no son funcionalmente independientes, pueden requerir una comunicación excesiva entre ellos para completar tareas básicas, resultando en un "monolito distribuido": un sistema que sufre tanto la rigidez del acoplamiento monolítico como la latencia y fragilidad de un sistema distribuido [4]. Esta complejidad accidental, cuando se vuelve inmanejable, ha llevado a una tendencia emergente en la industria de reversión al monolito, donde organizaciones optan por abandonar la arquitectura de microservicios para mitigar los altos costos operativos [13].

Fallas tempranas en esta etapa de definición de límites derivan en *olores arquitectónicos* (*architectural smells*): síntomas de un diseño deficiente que impactan directamente la mantenibilidad. Una revisión multivocal de la literatura ha identificado y catalogado estos olores, entre los que se encuentran la complejidad comunicacional excesiva, la aparición de *god services* (servicios monolíticos encubiertos que centralizan lógica disímil) o una granularidad subóptima (servicios demasiado grandes que retienen problemas de acoplamiento, o demasiado pequeños —*nano-services*— que disparan la sobrecarga de red) [14]. Estos errores anulan los beneficios de agilidad y escalabilidad que originalmente motivaron la adopción de la arquitectura [4, 15].

Por tanto, el diseño efectivo no se reduce a una simple fragmentación de código; exige una redefinición rigurosa de los límites funcionales y de datos del dominio, conocidos como *bounded contexts* [16]. El objetivo principal de un buen diseño es, por un lado, maximizar la *cohesión interna* —es decir, agrupar funcionalidades que están conceptualmente relacionadas y que tienden a cambiar juntas— y, por otro, minimizar el *acoplamiento externo* —reduciendo las dependencias entre servicios para asegurar su autonomía—.

Sin embargo, proponer los *bounded contexts* [16] como la solución simplemente desplaza el problema. **La identificación de estos límites no es un proceso algorítmico o trivial**. Por el contrario, la evidencia empírica [5] demuestra que es una actividad manual que la industria considera una de las principales desventajas del paradigma: es **«difícil de aprender»** y **«requiere desarrolladores altamente calificados (*senior developers*)»**, precisamente porque las «decisiones arquitectónicas son complejas» [5]. Investigaciones empíricas más recientes sobre la adopción de *Domain-Driven Design* (DDD) en la industria confirman que, si bien es el enfoque preferido, su aplicación sigue siendo abstracta y depende críticamente de la pericia y experiencia del equipo de arquitectos [17]. Esta dependencia de la pericia de un «arquitecto héroe» para interpretar el dominio crea un cuello de botella subjetivo, no repetible, y fundamentalmente anti-ágil, que justifica la necesidad de asistencia automatizada.

Para asistir esta actividad crítica, los avances recientes en inteligencia artificial (IA), especialmente los modelos de lenguaje grandes (LLMs), muestran un potencial disruptivo [18, 19]. El problema de descubrimiento de microservicios, no obstante, presenta dos vertientes radicalmente distintas. En *brownfield*, la migración de sistemas monolíticos existentes se apalanca en artefactos estructurados y observables (código fuente, trazas de ejecución, esquemas de bases de datos) para proponer particiones [9, 10].

Este desafío *greenfield* toca el núcleo de la promesa de esta arquitectura. En términos históricos, la promesa de los microservicios no es únicamente *escalar* despliegues, sino *escalar* la evolución del software: habilitar ciclos de cambio pequeños y frecuentes con límites bien definidos y equipos autónomos [1, 3].

Dichos límites no emergen espontáneamente; son el resultado de decisiones arquitectónicas que respetan el dominio y su semántica [16]. El salto desde el lenguaje natural (en *greenfield*) a las estructuras arquitectónicas es inherentemente *no determinista*: múltiples particiones pueden ser compatibles con el mismo conjunto de historias de usuario, y ninguna evidencia sintáctica garantiza por sí sola la cohesión funcional. Como se mencionó, en *greenfield* la señal estructural está ausente; la semántica del dominio se expresa de forma dispersa, incompleta y, a veces, contradictoria [20, 21].

Las soluciones *greenfield* que han intentado automatizar este salto han recurrido predominantemente a PLN y aprendizaje automático. Propuestas como SEMGROMI [11] o GreenMicro [12] agrupan historias o casos de uso basándose en *similitud semántica*. Sin embargo, esta heurística es fundamentalmente defectuosa: *similitud semántica* no implica *cohesión funcional*. Operaciones como "Crear Usuario" y "Eliminar Usuario" son semánticamente distantes (antónimos), pero funcionalmente co-dependen y deben residir en el mismo servicio cohesivo de "Gestión de Usuarios". Por ello, la sola heurística lingüística no ofrece garantías sobre la calidad estructural de las particiones resultantes.

La irrupción de LLMs añade un riesgo adicional y más sutil: la generación de artefactos "plausibles" [22, 23]. Los LLMs pueden producir archivos JSON, YAML o código que *aparentan* corrección, pero que pueden ocultar inconsistencias estructurales, *alucinaciones* de dependencias o violaciones de los principios de alta cohesión y bajo acoplamiento [19, 24, 25].

Aquí se sitúa el vacío metodológico que motiva este trabajo. La diferencia entre *plausibilidad textual* (lo que un LLM genera) y *corrección estructural* (lo que la arquitectura requiere) es crítica. Esta brecha entre lo que un modelo puede *expresar* y lo que un sistema puede *sostener* operativamente [2], sumada a los hallazgos de nuestro mapeo sistemático (MSL) [20] y la revisión sistemática (SLR) [18], evidencia la ausencia de un *pipeline* híbrido que combine el poder generativo de la IA con la **validación cuantitativa y formal** de la calidad arquitectónica. Sin un mecanismo de verificación integrado en el bucle de diseño, el *greenfield* asistido por IA corre el riesgo de automatizar y perpetuar arquitecturas estructuralmente deficientes.

Para ilustrar este desfase, considérese un dominio de *comercio electrónico* con requisitos textuales que incluyen: "Crear usuario", "Eliminar usuario", "Calcular impuestos", "Registrar pedido" y "Emitir factura". Un agrupamiento puramente semántico podría separar "Crear" de "Eliminar" por antónimos y juntar "Calcular impuestos" con "Emitir factura" por contigüedad léxica. No obstante, desde el diseño, las dos primeras operaciones pertenecen al mismo servicio cohesivo de *Gestión de Usuarios*, mientras que "Calcular impuestos" involucra invariantes fiscales independientes que no deben acoplarse rígidamente a *Facturación*. Este desfase entre semántica superficial y cohesión funcional es el tipo de error que ArchiGenMS busca prevenir mediante verificación métrico–formal.

Para abordar este vacío, la investigación adopta el marco de **Design Science Research (DSR)** [26, 27], orientado a crear y evaluar artefactos relevantes. El artefacto central es **ArchiGenMS** [28]. La hipótesis es que la "ilusión de corrección" de los LLMs y la insuficiencia de heurísticas de PLN deben contrarrestarse con garantías formales explícitas. Se propone un *pipeline* de *generar–verificar* donde IA generativa y búsqueda evolutiva [29] exploran el espacio de diseños, mientras un verificador formal en **Lean 4** [30] actúa como oráculo estructural. El oráculo evalúa cada candidato con métricas formalizadas de cohesión (LCOM), granularidad (SGM/SGM-SD) y acoplamiento (*Fan-Out*) [31, 32]. Así, la verificación formal no opera como validación tardía, sino como motor que guía la evolución, descartando arquitecturas "plausibles pero deficientes" y forzando diseños robustos y verificables.

1.1. Objetivos de la tesis

El objetivo central es proponer un marco conceptual y práctico que, a partir de requisitos textuales, genere propuestas de microservicios y garantice su corrección estructural mediante verificación formal en Lean. La metodología articula cinco fases: extracción semántica, construcción de grafos de dependencias, diseño asistido por LLMs, formalización de métricas de cohesión, granularidad y acoplamiento, y verificación de invariantes arquitectónicos. Para ello se desarrollan guías, *scripts* y módulos en Lean que operacionalizan las recomendaciones metodológicas.

- Diseñar y validar un *pipeline* de ingeniería de *prompts* que extraiga conceptos, entidades y relaciones a partir de artefactos textuales (historias de usuario y documentos de requisitos), usando modelos basados en *transformers* y *embeddings*.
- Definir una representación intermedia basada en grafos de dependencias y apoyar con LLMs el agrupamiento de funcionalidades en microservicios, maximizando cohesión interna y minimizando acoplamiento entre servicios.
- Formalizar métricas de calidad arquitectónica (p. ej., LCOM, SGM/SGM-SD, *Fan-Out*) como definiciones y lemas en Lean; establecer invariantes (ausencia de ciclos de dependencia) y validarlas automáticamente sobre cada propuesta generada.
- Implementar un prototipo que integre extracción semántica, agrupamiento asistido por LLMs y verificación formal, permitiendo medir consistencia interna y comparar frente a enfoques manuales o herramientas existentes.
- Realizar una evaluación empírica con *datasets* públicos de requisitos (comercio electrónico, reservas de vuelo, facturación hospitalaria) [33], comparando LLMs+verificación formal frente a enfoques manuales o heurísticos.

Alineación con DSR y trazabilidad metodológica Los objetivos operativizan el *ciclo de diseño* de DSR con artefactos ejecutables y criterios de evaluación observables [26, 27]. La extracción semántica y la representación en grafos materializan el *entendimiento del problema*; la generación asistida por LLMs y la búsqueda evolutiva concretan el *diseño del artefacto*; y la verificación en Lean instrumenta el *ciclo de rigor*, cerrando el lazo con métricas cuantitativas (LCOM, SGM/SGM-SD y *fan-out*) [31, 32]. Esta trazabilidad permite que cada decisión de diseño se justifique en términos de propiedades verificables, reduciendo la distancia entre especificaciones textuales y garantías estructurales.

1.2. Contribuciones principales

Este trabajo se ubica en la intersección entre ingeniería de software, IA y verificación formal para el diseño y descubrimiento automático de microservicios desde requisitos textuales.

En primer lugar, se consolidó una base teórica mediante un Mapeo Sistemático (MSL) presentado en CACIC 2024 [20] y una Revisión Sistemática (SLR) publicada en *Software* (MDPI) [18]. Ambos estudios identifican tendencias y desafíos en el uso de IA para diseño de microservicios, destacando el vacío metodológico en *greenfield*, especialmente en validación estructural y en la integración PLN–métricas [31, 32].

En segundo lugar, se formalizó el problema doctoral bajo DSR [27], definiendo preguntas de investigación, objetivos específicos y una estrategia metodológica centrada en ingeniería de *prompts* y verificación estructural mediante métricas computables. Este planteo se discutió en el Doctoral Symposium del CIbSE 2025 [21].

Como contribución central, se presenta **ArchiGenMS**, un *pipeline* evolutivo que integra LLMs, ingeniería de *prompts* y verificación formal en Lean. ArchiGenMS genera y refina arquitecturas evaluadas con métricas formalizadas (LCOM, SGM/SGM-SD, *Fan-Out*) [31, 32]. A diferencia de enfoques puramente heurísticos o de ML [22], introduce verificación formal que garantiza consistencia estructural durante el descubrimiento.

La validación de este artefacto se ha desarrollado en dos publicaciones complementarias: (1) una **validación cuantitativa** global, que analiza la convergencia del *pipeline* sobre 22 *datasets* públicos [33], aceptada para el XXXI Congreso Argentino de Ciencias de la Computación (CACIC) 2025 [28]; y (2) un **análisis cualitativo profundo** de un caso de estudio (g24-unibath), presentado en el Congreso Internacional de Innovación Tecnológica (CIITI-TE) 2025. Este segundo trabajo obtuvo el **Primer Lugar en la Categoría Posgrado** y, gracias a dicho reconocimiento, fue seleccionado para su publicación extendida en la *Revista Abierta de Informática Aplicada (RAIA – UAI)* [34].

Originalidad y alcance La originalidad radica en articular un *pipeline* de generar–verificar donde la verificación formal no es una fase terminal, sino un componente *en bucle* que guía la búsqueda, filtrando diseños plausibles pero inviables. En contraste con enfoques basados sólo en similitud o aprendizaje supervisado [11, 12], ArchiGenMS integra un oráculo formal (Lean) *dentro* del proceso generativo, y lo combina con métricas arquitectónicas de uso extendido [31, 32]. El resultado es un marco transferible a dominios *greenfield* donde la evidencia estructural es escasa y la fuente primaria es textual.

Finalmente, se realizó una evaluación empírica con *datasets* públicos de requisitos [33]. Los resultados muestran una reducción del 44 % en la función de aptitud tras cinco generaciones, con cohesión media (0.23) y acoplamiento saliente ($\leq 1{,}15$) alineados con recomendaciones de la literatura [29, 31, 32]. Estos hallazgos confirman la viabilidad del enfoque y sientan bases para aplicaciones industriales.

Delimitación y supuestos

Este trabajo se circunscribe a escenarios *greenfield* con requisitos textuales en lenguaje natural, asumiendo disponibilidad de historias de usuario y/o descripciones narrativas. No se emplean trazas de ejecución ni análisis estático de código. La evaluación se centra en propiedades estructurales —cohesión, granularidad y acoplamiento medido como *fan-out*— y en la verificación de invariantes arquitectónicos mediante Lean [30]. Aspectos operacionales (latencia, *throughput*) se dejan como trabajo futuro integrable con observabilidad en tiempo de ejecución.

Enunciado de tesis. Un *pipeline* de generar–verificar que integra LLMs con verificación formal en Lean puede guiar la búsqueda hacia diseños con mayor cohesión, granularidad adecuada y menor acoplamiento, mitigando artefactos plausibles pero estructuralmente deficientes [18, 19, 30].

Supuestos de trabajo. (S1) Los requisitos textuales capturan objetivos y restricciones funcionales relevantes. (S2) LCOM, SGM/SGM-SD y *fan-out* son indicadores útiles en diseño inicial [31, 32]. (S3) La especificación en Lean modela adecuadamente las invariantes consideradas. (S4) La guía métrica no degrada la validez semántica de los servicios resultantes. Estos supuestos delimitan la

validez de constructo (S2), interna (S3) y externa (S1), y se retoman en Metodología (DSR) y en Amenazas a la validez.

Marco disciplinar. La disciplina de la **Inteligencia Artificial para la Ingeniería de Software (AI4SE)** aplica métodos de IA —tales como la optimización metaheurística y el aprendizaje automático— para asistir y automatizar actividades del ciclo de vida del software (análisis de requisitos, diseño arquitectónico, pruebas, mantenimiento y evolución) [35]. Nuestros propios mapeos sistemáticos de la literatura han analizado la aplicación de estas técnicas específicamente al diseño de microservicios [18, 20]. En adelante utilizaremos la sigla *AI4SE*. Esta tesis se ubica específicamente en AI4SE para *diseño arquitectónico* (*architecture/design automation*), con foco en escenarios *greenfield*.

1.3. Estructura de la tesis

La tesis se organiza en seis capítulos, además de esta introducción. El **Capítulo 2** (Estado del Arte) presenta el estado del arte sobre diseño de microservicios, uso de PLN y LLMs en ingeniería de software, métricas arquitectónicas y métodos formales. El **Capítulo 3** (Fundamentos Teóricos y Matemáticos) expone fundamentos teóricos y matemáticos: teoría de grafos, espacios métricos y métricas arquitectónicas, nociones de lógica y teoría de tipos dependientes, y la formalización en Lean. El **Capítulo 4** (Trabajos Relacionados) analiza trabajos relacionados: generación automática, *prompt engineering*, evaluación arquitectónica, verificadores y revisiones sistemáticas, incluyendo publicaciones propias. El **Capítulo 5** (Enfoque Propuesto: Descubrimiento de Microservicios usando IA) describe el enfoque propuesto para descubrimiento asistido por IA: visión del *pipeline*, objetivos, estrategias de *prompting*, formalización y métricas de éxito. El **Capítulo 6** (Arquitectura y Herramientas de Soporte) detalla la arquitectura e implementación de **ArchiGenMS**: componentes, integración con APIs de LLMs, módulo de métricas, gestión de versiones y *prompts*, y el *pipeline* de verificación en Lean, así como los *datasets*. El **Capítulo 7** (Evaluación y Resultados) reporta evaluación y resultados: escenarios, comparación manual vs. asistido, métricas, cobertura de verificación, rendimiento y costes, y discusión crítica con limitaciones y extensiones. Finalmente, **Conclusiones y Perspectivas** sintetiza aportes, lecciones y líneas futuras.

Capítulo 2

Estado del Arte

El diseño de arquitecturas de microservicios se ha consolidado como un paradigma dominante en el desarrollo de sistemas distribuidos modernos. La transición desde arquitecturas monolíticas hacia arquitecturas basadas en microservicios responde a la necesidad de mayor escalabilidad, resiliencia y adaptabilidad frente a entornos de negocio cambiantes. Sin embargo, este paradigma no está exento de desafíos: la correcta identificación de los límites de cada servicio, el mantenimiento de la cohesión interna, la reducción del acoplamiento entre componentes y la alineación con los *bounded contexts* de la organización constituyen problemas abiertos que impactan directamente en la calidad del software.

Contexto industrial y evidencia empírica La adopción de microservicios ha estado guiada tanto por principios arquitectónicos como por restricciones operacionales reales: equipos autónomos, despliegue continuo, separación de responsabilidad y límites de datos que reflejen *bounded contexts* [16, 1]. Revisiones y estudios empíricos reportan barreras estables en el tiempo: definición de límites de servicio, complejidad de datos particionados y brecha de habilidades DevOps/observabilidad [3, 4, 36, 7]. La síntesis sugiere que el costo total de propiedad mejora sólo si la descomposición inicial preserva alta cohesión y bajo acoplamiento; de lo contrario, la deuda estructural se manifiesta como latencias inter-servicio, proliferación de *god services* y fragilidad ante cambios de requisitos [37, 15].

En paralelo, los avances recientes en inteligencia artificial, particularmente en procesamiento de lenguaje natural (PLN) y modelos generativos de gran escala, han abierto nuevas posibilidades para asistir a ingenieros y arquitectos de software en tareas tradicionalmente complejas, como la extracción de requisitos, la generación de artefactos de diseño y la documentación de arquitecturas. Estos enfoques, aunque prometedores, aún presentan limitaciones importantes relacionadas con la verificabilidad de los resultados, la reproducibilidad de los experimentos y la integración con métricas de calidad arquitectónica.

Asimismo, la literatura especializada ha puesto de relieve la relevancia de las métricas para evaluar la calidad de arquitecturas de microservicios, así como el papel de los métodos formales para garantizar propiedades estructurales en sistemas distribuidos. No obstante, la combinación de estas líneas de investigación con el uso de inteligencia artificial continúa siendo incipiente y carece de propuestas integradoras que permitan automatizar y validar el diseño de microservicios desde requisitos textuales.

El presente capítulo aborda el estado del arte en torno a cuatro ejes principales: (i) los principios

y retos de las arquitecturas de microservicios; (ii) el uso de técnicas de inteligencia artificial en el descubrimiento y diseño de microservicios; (iii) las métricas arquitectónicas utilizadas para evaluar la calidad estructural; y (iv) la aplicación de métodos formales en la validación de arquitecturas software. Cada uno de estos ejes permite contextualizar los avances recientes y, al mismo tiempo, resaltar los vacíos existentes que justifican la propuesta desarrollada en esta tesis.

2.1. Arquitectura de Microservicios: Principios y Retos

Las arquitecturas de microservicios se caracterizan por la descomposición de sistemas complejos en unidades independientes y desplegables de forma autónoma, denominadas microservicios. Cada microservicio encapsula una funcionalidad específica del dominio, puede ser desarrollado con tecnologías heterogéneas y se comunica con otros a través de interfaces ligeras, usualmente mediante APIs REST o mecanismos de mensajería asincrónica. Este paradigma favorece la escalabilidad horizontal, la resiliencia ante fallos y la capacidad de evolución incremental de los sistemas [3].

Estos servicios no son componentes aislados; interactúan siguiendo principalmente dos patrones de comunicación:

- **Comunicación Síncrona (Petición/Respuesta):** Un servicio invoca a otro directamente (usualmente mediante APIs REST o gRPC) y espera activamente una respuesta.
- **Comunicación Asíncrona (Basada en Eventos):** Un servicio publica un evento en un *message broker* (o "Bus de Eventos"), sin conocer a sus consumidores. Otros servicios se suscriben a dichos eventos y reaccionan de manera desacoplada.

Si bien la elección del patrón de comunicación y la gestión de la consistencia de datos distribuida (p. ej., mediante el patrón *Saga*) son desafíos críticos en la *implementación* y *operación* de microservicios, esta tesis se enfoca en el problema previo y fundamental: el diseño *estructural* estático.

Por lo tanto, el análisis de los protocolos de comunicación en tiempo de ejecución, así como la optimización de métricas operacionales (latencia, tolerancia a fallos, *throughput*), se delimitan explícitamente fuera del alcance de esta investigación. El foco de este trabajo reside en la validación de la calidad arquitectónica estática (cohesión, granularidad y acoplamiento) derivada de los requisitos textuales. Este paradigma, en sus distintas modalidades de comunicación, favorece la escalabilidad horizontal, la resiliencia ante fallos y la capacidad de evolución incremental de los sistemas [3].

En la **Figura 2.1** se ilustra un diagrama conceptual de este estilo arquitectónico, destacando sus componentes canónicos.

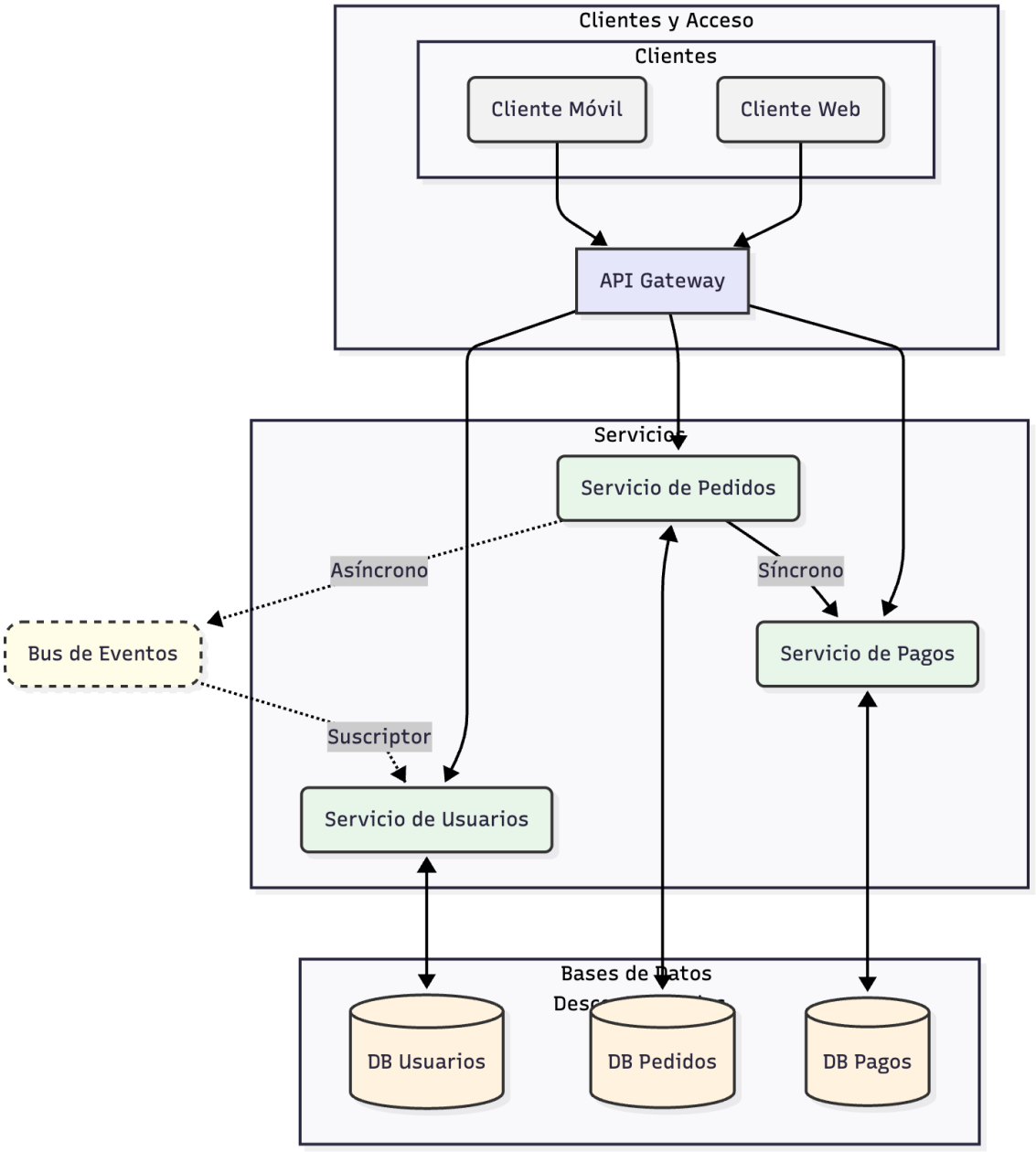


Figura 2.1: Diagrama conceptual de una Arquitectura de Microservicios, ilustrando el API Gateway, servicios independientes, bases de datos descentralizadas y comunicación síncrona/asíncrona.

Tal como se observa en la Figura 2.1, entre los principios fundamentales de este estilo arquitectónico destacan la alta cohesión dentro de cada servicio, el bajo acoplamiento entre com-

ponentes, la independencia en el despliegue y la alineación con los *bounded contexts* definidos por el dominio de negocio. Dichos principios han permitido a las organizaciones reducir el tiempo de entrega de nuevas funcionalidades y mejorar la mantenibilidad del software.

La adopción de este paradigma ha sido impulsada por factores industriales clave, incluyendo la necesidad de escalabilidad y la independencia de los equipos de desarrollo. Sin embargo, estudios empíricos sobre la adopción industrial confirman que la migración y el diseño *greenfield* introducen barreras significativas. Encuestas y estudios de caso en la industria señalan que la definición de los límites del servicio, la complejidad de la gestión de datos y la falta de habilidades especializadas son los obstáculos más frecuentemente reportados por las organizaciones [36, 4]. De hecho, la definición de límites es el reto central. Investigaciones empíricas confirman que el *Domain-Driven Design* (DDD) es el enfoque metodológico predominante para esta tarea, aunque su aplicación en la práctica sigue siendo abstracta y altamente dependiente de la pericia del arquitecto [17]. Esta dificultad se intensifica en los marcos de desarrollo ágiles, donde el descubrimiento de los límites del servicio debe competir con la naturaleza iterativa del proceso, obligando a los equipos a depender de técnicas heurísticas [8].

De hecho, la literatura ha sido consolidada a través de múltiples estudios sistemáticos que mapean el panorama de desafíos [37, 15]. Estas revisiones confirman que, si bien los beneficios de escalabilidad y agilidad son claros, los retos de diseño y operacionales son profundos y recurrentes. Los desafíos más críticos identificados en la literatura pueden agruparse en tres categorías principales:

Uno de los desafíos más críticos radica en la determinación del tamaño y alcance adecuados de cada microservicio, problema conocido como **granularidad**.

1. Desafíos de Descomposición y Granularidad. Un servicio demasiado amplio (*monolito encubierto*) retiene los problemas de acoplamiento del monolito, mientras que un exceso de fragmentación (servicios "nano") conduce a una explosión de la complejidad comunicacional y altos costes de orquestación. Asimismo, una descomposición pobre lleva a la aparición de *olores arquitectónicos* (*architectural smells*) bien documentados, como los *god services* (servicios que concentran múltiples responsabilidades) y a un alto acoplamiento aferente, erosionando los principios de independencia y cohesión [12, 37, 14]. De hecho, la consecuencia extrema de una descomposición fallida —que genera una complejidad accidental y un coste operativo insostenibles— es la tendencia emergente de la "reversión al monolito", donde organizaciones (incluyendo casos de alto perfil) optan por abandonar la arquitectura de microservicios y retornar a un diseño monolítico [13].

2. Complejidad del Cómputo Distribuido. Adicionalmente, la transición a microservicios transforma llamadas a funciones locales (baratas y seguras) en llamadas de red (costosas y falibles). Esto introduce complejidades inherentes a los sistemas distribuidos, destacando dos problemas principales: la **gestión de transacciones distribuidas** y la **consistencia de datos** [15]. La necesidad de mantener la consistencia a través de múltiples bases de datos descentralizadas (una por servicio) obliga a abandonar las transacciones ACID tradicionales en favor de patrones como Sagas y *eventual consistency*, cuya implementación es notoriamente compleja [4].

3. Desafíos Operacionales y Prácticos. Finalmente, la literatura identifica un conjunto de desafíos prácticos y operacionales [7]. Estos incluyen la necesidad de una **observabilidad** robusta (monitoreo, logging y tracing distribuidos) —identificadas empíricamente como las áreas de diseño más críticas por los expertos— [38] para depurar un sistema donde una sola petición puede

atravesar múltiples servicios. Además, se incrementa la complejidad en la **gestión de APIs** y la **seguridad** (autenticación y autorización en cada punto de entrada). Se reporta también una **brecha de habilidades** significativa, ya que el paradigma exige una madurez en prácticas de DevOps, automatización y orquestación de contenedores (ej. Kubernetes) que muchas organizaciones aún no poseen [7, 3]. Estudios de campo confirman esta brecha, reportando que, en la práctica, las organizaciones que adoptan microservicios en un contexto ágil continúan utilizando las mismas notaciones de diseño OOAD y métodos de estimación subjetivos que empleaban en arquitecturas monolíticas, fallando en adaptar sus prácticas de ingeniería a los nuevos retos del paradigma [6].

Estas problemáticas han motivado la investigación de enfoques automáticos y asistidos por inteligencia artificial para guiar el diseño de microservicios desde etapas tempranas, con el fin de mitigar errores de diseño que repercuten en el ciclo de vida completo del software.

Cuadro 2.1: Panorama comparativo de retos y evidencias reportadas en la literatura

Eje	**Manifestación típica**	**Evidencia representativa**
Granularidad y límites	Riesgo de *god services*, sobredescomposición y difusión de responsabilidad	[12, 37, 14, 13, 8]
Datos y transacciones	Consistencia eventual, Sagas y orquestación compleja	[15, 4]
Operación y gobernanza	Observabilidad distribuida, seguridad por perímetro múltiple, gestión de APIs	[7, 3, 38]
Adopción organizacional	Brecha de habilidades, alineación con *bounded contexts*	[36, 16, 1, 17, 6]

2.1.1. AI4SE: panorama breve y posicionamiento

El campo de *AI4SE* ha consolidado líneas clásicas como la reparación automática de programas (*program repair*) mediante análisis semántico [39], la predicción de defectos con aprendizaje profundo [40], y la revisión crítica de supuestos históricos en pruebas—por ejemplo, que la *cobertura* por sí sola no predice efectividad del conjunto de pruebas [41]. En paralelo, la evaluación de modelos de lenguaje entrenados en código ha abierto una ruta práctica para asistencia en tareas de ingeniería [42].

Este trabajo se sitúa en la intersección entre AI4SE y arquitectura de software: utiliza modelos generativos para *proponer* candidatos de diseño (*architecture/design automation*) y un componente formal/medible para *verificar* calidad estructural. A diferencia de líneas como reparación de programas o pruebas, aquí el objetivo no es corregir implementaciones existentes sino guiar una descomposición *greenfield* con garantías estructurales cuantificables.

2.2. Uso de IA en el Descubrimiento y Diseño de Microservicios

El uso de técnicas de inteligencia artificial para asistir el diseño de microservicios ha ganado relevancia en los últimos años, en particular a partir de los avances en procesamiento de lenguaje natural (PLN) y en modelos de aprendizaje profundo. La motivación principal radica en la dificultad que supone derivar arquitecturas de microservicios directamente desde artefactos textuales, como

historias de usuario o documentos de requisitos, donde la ambigüedad y la falta de estandarización dificultan la definición de límites de servicio y dependencias.

Esta aplicación de la IA se enmarca en gran medida dentro de la disciplina de la **Ingeniería de Software Basada en Búsqueda (SBSE)** [35]. La SBSE reformula los problemas de la ingeniería de software —como el diseño arquitectónico y la modularización— como problemas de optimización [43]. Como se analizará en detalle en el Capítulo 4, el problema de encontrar la partición arquitectónica óptima es computacionalmente intratable (NP-hard), lo que justifica el uso de metaheurísticas (como los algoritmos de *clustering* o los algoritmos evolutivos que se exploran en esta tesis) para navegar el vasto espacio de soluciones de diseño.

El análisis de la literatura, consolidado en nuestras revisiones sistemáticas [20, 18], revela que la aplicación de IA en esta área se ha bifurcado en dos líneas de investigación principales, con diferentes puntos de partida y objetivos: (1) enfoques de migración desde sistemas existentes (*brownfield*) y (2) enfoques de diseño desde requisitos (*greenfield*).

2.2.1. Enfoques Basados en Migración (Sistemas *Brownfield*)

La mayoría de las soluciones de IA documentadas, y una gran parte de los estudios identificados en nuestro SLR [18], se centran en el problema de la migración: cómo descomponer un sistema monolítico existente en microservicios. Estos enfoques *brownfield* tienen la ventaja de poder analizar artefactos concretos, como el código fuente, los registros (logs) de ejecución o las estructuras de bases de datos.

Una de las herramientas más prominentes en esta línea es **Mono2Micro** [9]. La metodología de Mono2Micro es un ejemplo claro de análisis dinámico. Primero, instrumenta el código del monolito para generar trazas de ejecución detalladas (logs) mientras el sistema es operado, idealmente cubriendo los principales casos de uso. Segundo, aplica un novedoso algoritmo de *clustering* témporo-espacial. El componente "espacial" agrupa entidades de negocio (ej. clases, objetos de datos) que son accedidas conjuntamente en un mismo flujo de ejecución. El componente "temporal" analiza la secuencia de llamadas para identificar interacciones fuertemente acopladas en el tiempo. El resultado es un conjunto de particiones candidatas que agrupan funcionalidades cohesivas. La principal limitación de este enfoque es su total dependencia de la calidad y completitud de los registros de ejecución; si los casos de uso ejecutados para generar los logs no son representativos, las particiones resultantes serán subóptimas.

De forma similar, **CARGO** [10] propone un análisis de dependencia guiado por IA, pero se enfoca principalmente en el análisis estático. CARGO construye un grafo de dependencia detallado a partir del código fuente, modelando llamadas a funciones, acceso a datos y relaciones estructurales. Posteriormente, aplica algoritmos de particionamiento de grafos para identificar "costuras" (*seams*) en la aplicación, buscando minimizar el "corte" (es decir, el número de dependencias que cruzarían los límites de los nuevos servicios). El objetivo es generar particiones que sean estructuralmente independientes, reduciendo la necesidad de costosas transacciones distribuidas. Su limitación es la inherente a todo análisis estático: puede ser confundido por características dinámicas del lenguaje (como la reflexión o el polimorfismo) y puede no capturar la cohesión a nivel de *negocio* o dominio, agrupando código que es estructuralmente similar pero funcionalmente dispar.

Otros enfoques, como el propuesto por Trabelsi et al. [44], también aplican *machine learning* en contextos *brownfield*, pero se centran en el análisis de tipos de datos y la estructura del código fuente para identificar candidatos.

Si bien estos enfoques son valiosos para la modernización de sistemas legados, su aplicabilidad

es nula en escenarios *greenfield*, que es el foco de esta tesis. En un proyecto *greenfield* no existe código fuente que analizar, ni registros de ejecución que agrupar, ni esquemas de bases de datos que particionar. El único artefacto de entrada es la descripción textual, ambigua y a menudo incompleta, de los requisitos.

2.2.2. Enfoques para el Diseño *Greenfield* (Basados en Texto)

Una segunda línea de investigación, más alineada con esta tesis, aborda el diseño *greenfield*. Estos enfoques sí parten de artefactos textuales (historias de usuario o casos de uso) y aplican técnicas de PLN y aprendizaje automático para proponer una arquitectura.

El desafío del diseño *greenfield* comienza con la naturaleza de su artefacto de entrada: el texto en lenguaje natural. Antes de que el diseño pueda ser propuesto, los requisitos deben ser comprendidos. Por ello, una línea de investigación paralela (y prerrequisito para el diseño) se ha centrado en el uso de IA para la *Ingeniería de Requisitos* (RE). Esto incluye trabajos sobre la calidad de las propias historias de usuario, como el *Quality User Story (QUS) framework* [45, 46], y enfoques de IA (PLN y ML) para transformar automáticamente historias de usuario en modelos de diseño o extraer entidades relevantes [47]. Si bien esta tesis no se centra en la extracción de requisitos, reconoce que la calidad del texto de entrada (el *dataset* de Dalpiaz [33]) es la base sobre la cual operan los enfoques de diseño *greenfield* como SEMGROMI, GTMicro y el propio ArchiGenMS.

SEMGROMI [11] es un ejemplo representativo de esta familia. Su metodología se basa en la agrupación semántica. Primero, procesa las historias de usuario y extrae relaciones lingüísticas. Segundo, utiliza modelos de *word embeddings* (como Word2Vec o GloVe) para calcular la similitud semántica entre las historias. Finalmente, aplica un algoritmo de *clustering* (ej. k-means) sobre estos vectores de similitud, donde cada *cluster* resultante se propone como un microservicio candidato. La debilidad fundamental de este enfoque es que la similitud semántica no garantiza la cohesión funcional. Por ejemplo, "Crear Usuario" y "Eliminar Usuario" son semánticamente distantes (antónimos), pero funcionalmente deben pertenecer al mismo "Servicio de Usuario". SEMGROMI podría tender a separarlos, creando servicios poco cohesivos.

Esta debilidad fundamental no es meramente conceptual, sino que responde a un defecto técnico documentado en la literatura de PLN. Investigaciones recientes como la de Pérez et al. [48] han demostrado empíricamente por qué los enfoques basados en *embeddings* (como los que utiliza SEMGROMI) son propensos a errores. El estudio identifica el problema fundamental de la **anisotropía** ("cono estrecho"), un fenómeno donde las representaciones vectoriales se agrupan en el espacio dimensional. La consecuencia directa de esta distorsión es que frases semánticamente distintas pueden recibir una puntuación de similitud coseno artificialmente alta, generando una alta tasa de **falsos positivos**. El trabajo de Pérez et al. evidencia esto con ejemplos claros donde pares de frases sin relación semántica obtienen altas puntuaciones de similitud, comprobando técnicamente por qué la heurística "similitud semántica = cohesión funcional" es fundamentalmente defectuosa y no fiable para la identificación robusta de microservicios.

Limitación técnica de la similitud semántica como proxy de cohesión Sea $\mathbf{u}_i, \mathbf{u}_j \in \mathbb{R}^d$ la representación de dos historias de usuario. La similitud coseno

$$\mathrm{sim}(\mathbf{u}_i, \mathbf{u}_j) = \frac{\mathbf{u}_i \cdot \mathbf{u}_j}{\|\mathbf{u}_i\| \, \|\mathbf{u}_j\|}$$

no codifica causalidad funcional ni co-uso operativo. En espacios de *embeddings* reales se observa *anisotropía*, lo que sesga sim hacia valores altos incluso para pares no relacionados, incrementando

falsos positivos de *clustering* [48]. Por ello, la hipótesis "alta similitud ⇒ alta cohesión" es, en general, inválida, y puede separar operaciones antónimas pero co-dependientes o agrupar por contigüidad léxica entidades que deben permanecer desacopladas por invariantes del dominio [18, 11, 49].

De manera análoga, **GreenMicro** [12] se enfoca en identificar microservicios a partir de casos de uso y entidades de bases de datos descritas textualmente, utilizando un enfoque de *clustering* para agrupar casos de uso relacionados. Esta línea de trabajo evolucionó recientemente hacia **GTMicro** [49], una propuesta tecnológicamente más avanzada que reemplaza los *embeddings* tradicionales por modelos *transformers* contextuales (como BERT). GTMicro logra así una comprensión semántica más profunda de los requisitos textuales, mejorando la precisión del agrupamiento lingüístico. Sin embargo, aunque estos enfoques resuelven el punto de partida textual y representan una mejora en la sofisticación del PLN aplicado, siguen basándose en la misma heurística: la *similitud semántica* como indicador suficiente de *cohesión funcional*. Como se identificó en nuestra SLR [18] y se discutió arriba, esta heurística es defectuosa y propensa a falsos positivos. En consecuencia, no existe garantía de que los *clusters* generados —independientemente de la sofisticación del modelo— cumplan con umbrales aceptables de cohesión (LCOM) o acoplamiento (*Fan-Out*).

2.2.3. Tendencias Emergentes: IA Generativa y LLMs

Con la irrupción de los modelos generativos (LLMs), ha surgido una tercera línea de investigación que explora su uso directo en el diseño arquitectónico. Estudios exploratorios, como los de Stojanovic et al. [23] y Adams et al. [22], han examinado la capacidad de ChatGPT para identificar microservicios a partir de descripciones o incluso para generar código de microservicios.

Estos trabajos iniciales son cruciales porque resaltan una dualidad: por un lado, los LLMs demuestran una capacidad sin precedentes para extraer entidades y generar artefactos plausibles a partir de texto libre. No obstante, estudios de vanguardia que emplean modelos de razonamiento avanzado, como el de Pereira et al. [50], han revelado que el desempeño varía significativamente según la estrategia de *prompting*. En su evaluación de GPT-4 o3, los autores hallaron que, si bien el modelo identifica con éxito la mayoría de los servicios necesarios (alta *recall*), las arquitecturas generadas mediante *zero-shot* presentan un acoplamiento bidireccional excesivo, recibiendo una calificación de plausibilidad de apenas $2/5$ por expertos humanos. Este fenómeno refuerza la conclusión de que los resultados carecen de consistencia estructural y son altamente sensibles a la formulación del *prompt*, evidenciando una brecha crítica en la **verificabilidad** intrínseca de los diseños generados.

Buscando mayor estructura, otros trabajos han propuesto marcos para guiar el diseño. **PF4MD** [51], por ejemplo, combina *Problem Frames* (un método de análisis de requisitos) con métricas de complejidad para guiar la descomposición, aunque sin la flexibilidad generativa de los LLMs. Más recientemente, asistentes como el propuesto por Díaz-Pace et al. [52] utilizan LLMs para guiar a arquitectos novatos en la toma de decisiones, actuando como un "copiloto" que sugiere patrones de diseño.

Un panorama general de la IA Generativa en arquitectura de software, presentado por Esposito et al. [19], confirma esta tendencia, pero identifica la reproducibilidad, la confianza y, sobre todo, la **verificabilidad** como los principales desafíos pendientes para que estas herramientas pasen de ser asistentes exploratorios a herramientas de ingeniería robustas.

2.2.4. LLMs para arquitectura: trazabilidad, justificabilidad y verificación

Los estudios recientes sobre LLMs en ingeniería/arquitectura de software reportan avances en ideación, documentación y apoyo a decisiones, pero también limitaciones en consistencia, trazabilidad y justificabilidad de las salidas [22, 23, 25]. En respuesta, algunas propuestas combinan *prompt engineering* con mecanismos de evaluación o guías de decisiones para capturar conocimiento arquitectónico y reducir la deriva [51, 52]. Revisiones de alcance amplio identifican como temas críticos la verificabilidad *ex post*, la reproducibilidad y la confianza, especialmente cuando la generación se integra en procesos de diseño [19].

En este punto, la literatura de métodos formales ofrece un *ancla* para cerrar el ciclo: las salidas generadas deben ser sometidas a verificaciones y a cómputo de métricas sobre representaciones intermedias bien definidas [2, 53, 54, 55, 30]. Esta línea guía el posicionamiento de esta tesis en la intersección IA generativa–métricas–verificación formal.

2.3. Greenfield vs. Migración: Panorama y Herramientas Industriales

Dos líneas complementarias aparecen en la literatura: (i) *greenfield*, donde se diseña desde requisitos textuales o modelos conceptuales; y (ii) *migración*, donde se parte de sistemas monolíticos existentes. En el frente *greenfield*, se han propuesto métodos que agrupan historias de usuario o escenarios para derivar candidatos a microservicios [11, 12, 49, 47]. Por su parte, el frente de *migración* ha cristalizado en herramientas industriales que analizan dependencias y *hotspots* de cambio para sugerir cortes de servicio, como Mono2Micro [9] o CARGO [10].

Revisiones y mapeos previos muestran que, aunque la motivación y el proceso difieren entre greenfield y migración, los *criterios de calidad estructural* (cohesión, acoplamiento, granularidad) son transversales [15, 37, 7]. Esto sugiere que las métricas computables y la verificación formal pueden actuar como "lenguaje común" para comparar propuestas en ambos contextos.

2.4. Métricas Arquitectónicas para Evaluación de Microservicios

El desafío de la descomposición en microservicios es, en esencia, la aplicación de principios de diseño de software establecidos, como los propuestos en el *Domain-Driven Design (DDD)* [16]. Principios como la definición de *Bounded Contexts* buscan explícitamente maximizar la cohesión interna y minimizar el acoplamiento externo [1]. Sin embargo, estos principios son cualitativos. Para que un proceso de diseño, y especialmente uno automatizado, pueda ser evaluado y optimizado, se requiere una traducción de estos principios a indicadores cuantitativos. Es en este contexto donde las métricas arquitectónicas se vuelven fundamentales, ofreciendo un lenguaje formal para medir lo que el DDD describe conceptualmente.

La evaluación cuantitativa de arquitecturas de microservicios es esencial para garantizar principios fundamentales como cohesión interna, bajo acoplamiento e independencia funcional. Estos atributos inciden directamente en la mantenibilidad, escalabilidad y resiliencia de los sistemas, de

Cuadro 2.2: Comparativa sintética de líneas relacionadas y vacíos identificados

Línea	**Aportes representativos**	**Limitaciones recurrentes**
Microservicios (fundamentos, adopción)	Estilo, principios, patrones y retos de granularidad/cohesión/acoplamiento [56, 57, 1, 4, 3]	Criterios prácticos heterogéneos; falta de evaluación sistemática previa a implementación
IA para descubrimiento (greenfield)	Agrupamiento semántico y modelos desde historias/escenarios [11, 12, 49, 47]	Dependencia de similitud; sensibilidad a idioma/estilo [48]
LLMs en arquitectura	Soporte a decisiones, generación guiada y evaluación de modelos de razonamiento (o3) [22, 23, 51, 52, 19, 50]	Trazabilidad débil; el *zero-shot* produce alto acoplamiento; dependencia de validación experta humana
Migración (monolito→microservicios)	Herramientas y evidencia industrial (dependencias, *hotspots*) [9, 10]	Dependencia del código legado; menos aplicable en greenfield
Métricas y métodos formales	Marcos de métricas y verificación automatizada [31, 32, 2, 53, 54, 55, 30]	Integración fragmentaria con generación automática

modo que contar con métricas bien definidas permite no solo comparar diseños alternativos, sino también validar propuestas generadas automáticamente.

El concepto de métricas para evaluar la calidad estructural del software precede a los microservicios. Enfoques clásicos, como las métricas orientadas a objetos de Chidamber y Kemerer, ya proponían cuantificar la cohesión (LCOM) y el acoplamiento (CBO) a nivel de clases. En el paradigma de microservicios, el desafío se traslada de las clases a los servicios distribuidos. Como se relevó en nuestra revisión sistemática [18], la literatura sobre métricas de microservicios se divide en dos grandes familias: (1) las que miden la **calidad estructural estática** (calidad del diseño) y (2) las que miden el **comportamiento dinámico y no funcional** (calidad del *runtime*).

En el contexto de la **calidad estructural estática**, que es el foco de esta tesis, se han desarrollado marcos de referencia ampliamente reconocidos. Al-Debagy y Martinek [31] proponen un conjunto de métricas para evaluar granularidad, acoplamiento y reutilización, con el objetivo de guiar la toma de decisiones arquitectónicas en escenarios tanto de migración como de diseño inicial. Por su parte, Taibi y Systä [32] introducen un marco de descomposición y evaluación métrica que combina principios de diseño con indicadores de calidad, ofreciendo lineamientos para detectar problemas estructurales como los denominados *god services* o el exceso de fragmentación.

Entre las métricas estáticas más utilizadas destacan la *Lack of Cohesion of Methods* (LCOM), que estima la cohesión interna de un microservicio; el acoplamiento de salida (*fan-out*), que mide sus dependencias hacia otros servicios; y la *Service Granularity Metric* (SGM), que evalúa la cantidad y distribución de operaciones para determinar si un servicio presenta un tamaño apropiado.

2.4.1. Formulación matemática de métricas estructurales

Sea un diseño candidato $\mathcal{S} = \{S_1, \ldots, S_k\}$, donde cada S_a es un servicio con conjunto de operaciones $\mathcal{O}_a$ y entidades internas $\mathcal{E}_a$.

Cohesión (LCOM, versión basada en co-uso). Defínase $U(o) \subseteq \mathcal{E}_a$ como el conjunto de entidades usadas por la operación $o \in \mathcal{O}_a$. Una forma operativa de LCOM para servicios es:

$$\mathrm{LCOM}(S_a) \;=\; 1 \;-\; \frac{1}{|\mathcal{O}_a|(|\mathcal{O}_a|-1)} \sum_{\substack{o_i, o_j \in \mathcal{O}_a \\ i<j}} \frac{|U(o_i) \cap U(o_j)|}{|U(o_i) \cup U(o_j)|}.$$

Valores menores indican mayor co-uso interno y, por tanto, mayor cohesión [31].

Granularidad (SGM/SGM-SD). Sea $g(S_a) = |\mathcal{O}_a|$ la cardinalidad operativa del servicio y $\mu = \frac{1}{k}\sum_{a=1}^{k} g(S_a)$ la media. Una métrica de dispersión de granularidad puede expresarse como:

$$\text{SGM-SD} \;=\; \sqrt{\frac{1}{k}\sum_{a=1}^{k} \big(g(S_a) - \mu\big)^2},$$

donde valores altos sugieren heterogeneidad problemática (macroservicios junto a *nanoservicios*). Umbrales prácticos se reportan en marcos de evaluación [32].

Acoplamiento saliente (*fan-out*). Sea $D(S_a)$ el conjunto de dependencias salientes de S_a hacia otros servicios. Defínase

$$\mathrm{FO}(S_a) \;=\; |D(S_a)|, \qquad \mathrm{FO_avg} \;=\; \frac{1}{k}\sum_{a=1}^{k} \mathrm{FO}(S_a).$$

Valores menores reducen la complejidad de coordinación y el riesgo de cascadas de fallos [31].

Propiedades deseables. Para diseños sin ciclos de dependencia, el grafo dirigido $(\mathcal{S}, \mathcal{D})$ es acíclico, con $\sum_a \mathrm{FO}(S_a) = |\mathcal{D}|$. En presencia de ciclos, FO subestima la complejidad efectiva; de ahí la conveniencia de verificar en Lean la invariancia de aciclicidad como precondición del cálculo métrico [30]. La combinación ($\downarrow$ LCOM, $\downarrow$ FO, $\downarrow$ SGM-SD) define una región estructuralmente saludable para candidatos [31, 32].

Paralelamente, la segunda familia de métricas (el **comportamiento dinámico**) ha recibido una atención considerable en la literatura, aunque no forma parte del alcance de este trabajo. Estos estudios proponen métricas y modelos de IA para la optimización del *runtime*, tales como el autoescalado (*autoscaling*) predictivo basado en aprendizaje automático [58, 59], la adaptación dinámica del sistema usando aprendizaje por refuerzo profundo [60], la implementación de tolerancia a fallos proactiva [61] y la depuración de cuellos de botella en el rendimiento usando IA [62]. Estas métricas son cruciales para la operación, pero presuponen una arquitectura estructuralmente sólida, que es precisamente lo que las métricas estáticas (LCOM, SGM) buscan asegurar en la fase de diseño.

Aunque estas métricas *estáticas* (LCOM, SGM, etc.) han sido aplicadas en estudios experimentales y en procesos de migración desde sistemas monolíticos, su uso en escenarios *greenfield* aún es limitado. Además, como se detalla en el párrafo anterior, la mayoría de trabajos las aplican de forma empírica, sin un respaldo formal que asegure su correcta interpretación o que permita integrarlas en *pipelines* automáticos. Este vacío motiva la integración de mecanismos de verificación formal, aspecto que se desarrolla en el Capítulo 3, donde se presentan las definiciones matemáticas de cada métrica y su formalización en Lean [30].

Alcance de las fórmulas en este capítulo. Las expresiones presentadas para LCOM, *fan-out* y SGM se incluyen aquí con un propósito estrictamente expositivo —como *operacionalizaciones representativas* en la literatura— para ilustrar cómo los principios de cohesión, acoplamiento y granularidad se han cuantificado en trabajos previos [31, 32]. **No constituyen las definiciones normativas adoptadas por esta tesis**. Las definiciones formales utilizadas en el resto del manuscrito, así como su verificación en Lean 4, se establecen rigurosamente en el Capítulo 3 y se emplean de forma consistente en los Capítulos 5 y 7. Este deslinde garantiza compatibilidad con nuestras publicaciones previas y evita ambigüedades terminológicas.

> **Nota metodológica.** Cuando se cite LCOM, *fan-out* o SGM fuera de este capítulo, se entenderán, salvo indicación contraria, según las definiciones formales fijadas en el Capítulo 3, que son las empleadas por el verificador en Lean 4 [30]. Las fórmulas aquí resumidas cumplen un rol de estado del arte y pueden diferir en notación o variante de la métrica respecto de las adoptadas por esta tesis.

2.5. Métodos Formales en Ingeniería de Software

Los métodos formales constituyen un enfoque riguroso para el análisis y la validación de sistemas de software, al basarse en especificaciones matemáticas que permiten demostrar propiedades sobre su comportamiento. Su aplicación en el ámbito de sistemas distribuidos y arquitecturas complejas busca reducir la ambigüedad inherente a las descripciones informales y proveer garantías sobre aspectos críticos como seguridad, consistencia y ausencia de errores lógicos. La lógica, de hecho, provee el marco fundamental para razonar sobre los programas y sus propiedades [63].

Clarke y Wing [2] presentan una revisión pionera que sintetiza el estado del arte de los métodos formales en ingeniería de software, resaltando tanto sus beneficios como las barreras para su adopción a gran escala. A partir de allí, el campo se ha diversificado en varias familias de herramientas, siendo las más prominentes el *model checking* y los asistentes de pruebas.

El *model checking*, consolidado por Clarke, Grumberg y Peled [53], es una técnica automática que explora exhaustivamente el espacio de estados de un sistema para verificar el cumplimiento de propiedades, usualmente expresadas en lógica temporal. Herramientas como **TLA+**, propuesta por Lamport [55], se han convertido en un estándar industrial (usado por Amazon, por ejemplo) para especificar y verificar algoritmos y sistemas concurrentes y distribuidos, asegurando que el *comportamiento* del sistema (ej. un protocolo de consistencia) sea correcto.

De manera complementaria, **Alloy**, introducido por Jackson [54], es un lenguaje de modelado ligero que emplea lógica relacional y técnicas de *bounded model checking* (análisis en un alcance finito). Alloy es particularmente potente para verificar propiedades *estructurales* de modelos de software, encontrando contraejemplos de invariantes (ej. “un administrador siempre debe tener acceso” o “no deben existir ciclos en esta jerarquía”).

2.5.1. Model checking vs. asistentes de prueba para calidad estructural

Herramientas de *model checking* como TLA+ verifican propiedades temporales en espacios de estados (p. ej., ausencia de *deadlocks*) [55, 53]; Alloy explora invariantes relacionales en dominios acotados [54]. En contraste, asistentes de prueba como Lean permiten definir métricas como objetos matemáticos y demostrar lemas de corrección/monotonía, conectando directamente con los indicadores LCOM/SGM/*fan-out* [30]. Para un proceso de diseño *greenfield* centrado en estructura

estática, esta segunda familia resulta más natural: formaliza el oráculo métrico, enmarca invariantes (aciclicidad) y habilita una verificación *en bucle* dentro del proceso generativo.

En síntesis, los métodos formales proporcionan un marco robusto para garantizar la corrección de arquitecturas software. No obstante, su adopción en el contexto específico del diseño de microservicios aún es incipiente. La mayoría de los trabajos se centra en el análisis de protocolos (TLA+) o modelos generales (Alloy), pero no aborda la validación de métricas arquitectónicas como cohesión, acoplamiento o granularidad en escenarios *greenfield*. Esta tesis busca llenar ese vacío mediante la integración de verificación formal en Lean con técnicas de inteligencia artificial generativa, garantizando la consistencia estructural de arquitecturas descubiertas automáticamente.

2.6. Reproducibilidad y Reporte Experimental en Enfoques IA+Formales

La integración de IA con métodos formales exige prácticas de reporte que favorezcan la *replicabilidad* y la *auditabilidad* de resultados. En particular, cuando el verificador es un asistente de pruebas o compilador formal, su versión y modo de ejecución forman parte del diseño experimental tanto como el código del pipeline. Herramientas como Lean 4 proveen un ecosistema estable de construcción (vía *Lake*), lo que habilita describir el "circuito reproductible": versión del verificador, comandos de compilación/ejecución y artefactos de salida estructurados [30]. En esta línea, disponer de un paquete público con *código-fuente*, *datasets*, *scripts* y *resultados* estandarizados (por ejemplo, JSON/JSONL) es un requisito práctico para contrastar hipótesis y comparar corridas [64].

Desde el ángulo metodológico, la literatura clásica de verificación y especificación (model checking, Alloy, TLA+) subraya que la claridad en los supuestos, la explicitación de convenciones y la precisión de las propiedades verificadas condicionan la validez externa de los experimentos [2, 53, 54, 55]. En términos de diseño científico, el *artefacto* también es resultado de investigación y debe documentarse con el mismo estándar que el protocolo experimental [26]. En este contexto, el *reporte mínimo* recomendado incluye: (i) versión del verificador y comando de ejecución; (ii) versión del intérprete/compilador del orquestador; (iii) *seeds* y parámetros del generador; (iv) estructura de los archivos de entrada/salida; y (v) *hashes* o identificadores de corridas [30, 64].

2.7. Datasets Públicos y Variabilidad Lingüística de Requisitos

Los enfoques basados en requisitos textuales requieren colecciones abiertas y curadas para evaluar la generalidad de sus resultados. El conjunto de historias de usuario de Dalpiaz es una referencia ampliamente utilizada, con estructura canónica y anotaciones útiles para tareas de PLN [33]. En paralelo, trabajos sobre *calidad de historias de usuario* (QUS) ponen énfasis en criterios lingüísticos y estructurales que impactan directamente en la extracción automática de información [46, 45].

No obstante, la dependencia de datasets textuales introduce un riesgo de sesgo derivado de la redacción. Mientras que Pereira et al. [50] mitigan esto mediante la curación manual de ejemplares (*few-shot*), la literatura en español reporta una alta variabilidad léxica y semántica que dificulta la detección de similitudes funcionales [48]. ArchiGenMS aborda esta problemática mediante la desacoplación del proceso generativo de la validación estructural; de este modo, la calidad del diseño

final no queda supeditada exclusivamente a la precisión léxica del requisito o a la sofisticación de los *embeddings*, sino a la convergencia métrica certificada por el oráculo en Lean 4.

2.8. Protocolo de Evaluación y Reproducibilidad

Datasets. Se emplean requisitos textuales públicos en dominios representativos (comercio electrónico, reservas de vuelo, facturación hospitalaria), siguiendo el repositorio de historias de usuario de referencia [33].
Variables de resultado. (i) Cohesión media (LCOM, menor es mejor); (ii) acoplamiento medio (FO_avg); (iii) granularidad y dispersión (SGM, SGM-SD); (iv) validez estructural (aciclicidad verificada).
Diseño comparativo. Se comparan: (a) particiones base por *clustering* semántico (familia *greenfield*–PLN) [11, 49]; (b) particiones asistidas por LLM sin verificación [22, 23]; (c) *pipeline* generar–verificar con oráculo métrico–formal (propuesta).
Réplica y control. Se fijan semillas y plantillas de *prompting*; las salidas se validan contra esquemas tipados. Se publican artefactos (scripts, especificaciones y pruebas Lean) para permitir réplica y auditoría [18, 30].
Amenazas a la validez. *Interna:* sensibilidad a *prompting* (mitigada con plantillas y control de temperatura); sesgos de representación textual (mitigado con dominios múltiples). *Externa:* extrapolación a contextos industriales con artefactos adicionales (logs, esquemas); este trabajo se acota a *greenfield* textual. *Constructo:* adecuación de LCOM/SGM/*fan-out* como proxies de calidad; se apoya en marcos reconocidos [31, 32]. *Conclusión:* tamaño del efecto reportado con intervalos y pruebas no paramétricas cuando aplique.

2.9. Posicionamiento de esta Tesis respecto del Estado del Arte

La propuesta doctoral se ubica en la convergencia de tres corrientes: (i) generación asistida por IA/LLMs desde requisitos textuales, (ii) métricas arquitectónicas computables y (iii) verificación formal operativa. A diferencia de trabajos centrados exclusivamente en similitud semántica o en *prompting* sin validación, el pipeline que proponemos utiliza LLMs para *proponer*, Lean 4 para *validar y calcular* señales estructurales, y una función de aptitud simple para *seleccionar* alternativas con mejor cohesión, granularidad y menor acoplamiento. Esta posición responde explícitamente a las lagunas señaladas por la literatura de IA en arquitectura (verificabilidad, reproducibilidad y confianza) [22, 23, 51, 52, 19] y retoma el énfasis clásico de los métodos formales en especificación y verificación precisas [2, 53, 54, 55, 30].

En suma, la contribución se alinea con los hallazgos de nuestro propio mapeo y revisión sistemática [20, 18] y cristaliza en un artefacto reproducible publicado para facilitar comparación y evaluación independiente [64].

2.10. Discusión y Vacíos Identificados

El recorrido presentado en este capítulo evidencia que el diseño de arquitecturas de microservicios se encuentra en un punto de confluencia entre prácticas consolidadas y tendencias emergentes. Se ha

demostrado que el paradigma presenta desafíos profundos que van más allá del diseño, abarcando la complejidad del cómputo distribuido y la operación práctica (Sección 2) [37, 15, 7].

Para abordar estos retos, la investigación en IA se ha bifurcado. Los enfoques *brownfield* (ej. [9, 10]) han madurado, pero son inaplicables a escenarios *greenfield*. Los enfoques *greenfield* (ej. [11, 49]) resuelven el punto de partida textual, pero lo hacen mediante heurísticas de similitud semántica, sin garantías de calidad. Finalmente, los emergentes LLMs [22, 19] ofrecen un poder generativo sin precedentes, pero introducen serios problemas de consistencia y verificabilidad.

Asimismo, el análisis de la literatura sobre calidad diferenció las métricas de *calidad estructural estática* (ej. [31, 32]), que evalúan el diseño, de las métricas de *comportamiento dinámico* (ej. [58, 60]), que evalúan el *runtime*. Paralelamente, la revisión de métodos formales distinguió entre *model checkers* como TLA+ [55] (para verificar comportamiento) y *theorem provers* como Lean [30] (para verificar corrección matemática).

Síntesis integradora y brechas El triángulo *greenfield*–PLN, LLMs generativos y verificación arroja un patrón consistente: generación sin garantías produce artefactos plausibles pero estructuralmente frágiles [18, 19]. Las métricas estructurales ofrecen un lenguaje operativo, aunque rara vez se formalizan como oráculos [31, 32]. La pieza faltante es un lazo *generar–verificar* con un asistente de prueba [30].

Cuadro 2.3: Matriz de brechas metodológicas y línea de solución adoptada

Brecha	**Manifestación en la literatura**	**Línea de solución**
Generación sin verificación	Clustering semántico y LLMs sin garantías [11, 49, 50, 22]	Oráculo métrico–formal en bucle (*generar–verificar*)
Métricas no formalizadas	LCOM/SGM usados empíricamente [31, 32]	Definiciones y lemas en Lean [30]
Reproducibilidad limitada	Sensibilidad a *prompts*, falta de artefactos públicos [19]	Plantillas, esquemas tipados y repositorio de evaluación [33, 18]

De este análisis se desprenden tres vacíos principales que esta tesis aborda directamente:

1. **El vacío de la Verificación en la Generación.** Los enfoques *greenfield* [11] y los basados en LLMs [22] generan arquitecturas "plausibles" pero no "verificadas". No existe un marco que combine la generación automática de arquitecturas de microservicios con su validación formal.

2. **El vacío de la Formalidad en las Métricas.** La escasa integración de métricas arquitectónicas cuantificables (LCOM, SGM) dentro de los procesos de descubrimiento y diseño asistidos por IA. Como se vio, estas métricas se aplican de forma empírica y carecen de un soporte formal que asegure la correcta implementación de su definición matemática.

3. **El vacío del Foco Metodológico.** La falta de evidencia empírica en escenarios *greenfield*, donde se diseñan arquitecturas desde requisitos textuales. La mayoría de las herramientas de IA robustas (Mono2Micro, CARGO) se centran en la migración *brownfield*.

Estos vacíos motivan la propuesta desarrollada en esta tesis, orientada a integrar técnicas de inteligencia artificial generativa, métricas arquitectónicas y verificación formal en un *pipeline* unificado para el descubrimiento automático de microservicios.

Capítulo 3

Fundamentos Teóricos y Matemáticos

El diseño de arquitecturas de microservicios requiere no solo lineamientos conceptuales, sino también bases teóricas y matemáticas que permitan describir con precisión sus propiedades y garantizar la corrección de las decisiones adoptadas. Mientras que los principios arquitectónicos ofrecen guías prácticas para alcanzar cohesión, bajo acoplamiento y alineación con el dominio [16, 1], su formalización *computable* constituye un paso necesario para reducir ambigüedad, explicitar convenciones y habilitar procesos de evaluación reproducibles en el tiempo.

Los métodos formales han demostrado ser herramientas eficaces para especificar, razonar y comprobar propiedades de sistemas complejos [2]. Hitos como el *model checking* [53], TLA+ [55] y Alloy [54] se consolidaron tanto en la academia como en la industria. Más recientemente, Lean 4[1] combina un asistente de pruebas basado en teoría de tipos con un *runtime* eficiente, lo que habilita desde demostraciones interactivas hasta cálculos deterministas sobre estructuras de datos [30].

En paralelo, la investigación reciente en inteligencia artificial ha mostrado el potencial de los modelos generativos para asistir el diseño de arquitecturas software [20, 18, 19], incluyendo el descubrimiento de microservicios a partir de requisitos textuales. Sin embargo, sin una definición clara de métricas y un *pipeline* de evaluación consistente, la verificación de la calidad de dichas propuestas se debilita.

Este capítulo establece los fundamentos teórico–matemáticos que sustentan esta tesis. En particular, se cubre: (i) los principios esenciales de microservicios y su vínculo con calidad estructural; (ii) la representación de requisitos textuales y los límites de enfoques basados en similitud; (iii) un modelo matemático mínimo para representar arquitecturas como grafos rotulados; (iv) las métricas empleadas *tal como se calcularon en el experimento*, con propiedades básicas; (v) el rol operativo de Lean 4 como validador computable; y (vi) la conexión con el algoritmo evolutivo que minimiza una función de aptitud escalar simple. Todo ello *respeta estrictamente* lo implementado y ejecutado.

3.1. Fundamentos de Arquitectura de Microservicios

El paradigma de microservicios surge como evolución frente a las limitaciones de arquitecturas monolíticas y SOA, enfatizando despliegue independiente, acoplamiento débil e interfaces ligeras [56]. El enfoque de *Domain-Driven Design* (DDD) [16] aporta el concepto de *bounded contexts* para

[1] Versión usada en los experimentos: **v4.22.0-rc3** (Lake). Véase §6.2 para el entorno completo de hardware y software.

delimitar servicios con responsabilidades claras y alta cohesión; Newman compendia patrones de diseño y operación ampliamente adoptados [1]. Desde la academia, la síntesis de Dragoni et al. traza la evolución del estilo, su consolidación y su relación con la nube, la contenerización y el desarrollo ágil [57].

En la práctica, los retos se concentran en: (a) descomposición y granularidad; (b) complejidad del cómputo distribuido (transacciones, consistencia, resiliencia, idempotencia); y (c) operación y gobernanza (observabilidad, seguridad, versionado, automatización) [3, 12]. Estos ejes motivan introducir *métricas computables* que permitan evaluar, comparar y seleccionar alternativas durante el diseño, antes de incurrir en costos de implementación.

Calidad estructural: intuición operativa

Cohesión alta y acoplamiento bajo siguen siendo los principios rectores. Operativamente: (i) mayor cohesión implica que las operaciones de un servicio comparten datos/propósitos; (ii) menor acoplamiento implica menos dependencias salientes críticas; (iii) granularidad balanceada evita tanto *macro-servicios* (grandes y heterogéneos) como *nano-servicios* (programación distribuida innecesaria).

La Figura 3.1 ilustra estos dos principios de diseño.

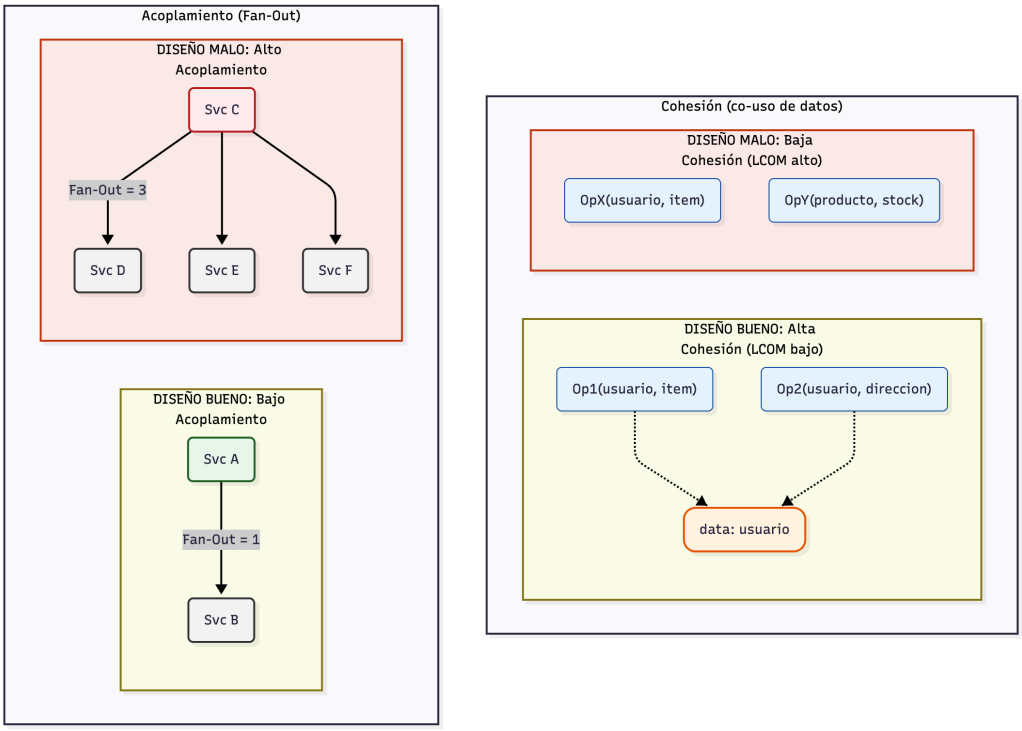


Figura 3.1: Ilustración de los principios de diseño de Cohesión (Alta vs. Baja) y Acoplamiento (Bajo vs. Alto).

AI4SE en esta tesis: definición operativa

En el sentido de esta tesis, *AI4SE* se entiende de forma operativa: un lazo *generar–verificar* donde (i) un modelo generativo propone arquitecturas candidatas desde requisitos textuales y (ii) un validador computable en Lean 4 calcula métricas estructurales y verifica invariantes mínimos. Esta lectura contrasta con otras ramas de AI4SE (p. ej., reparación automática [39], predicción de defectos [40] o evaluación de modelos para tareas de programación [42]) y fija el foco en *diseño arquitectónico greenfield* con evaluación cuantitativa.

3.2. Representación Formal de Requisitos Textuales

Los requisitos textuales se capturan con frecuencia como **escenarios** [48] o **historias de usuario**, privilegiando comunicación clara con *stakeholders* pero introduciendo ambigüedad, omisiones y variabilidad. El marco QUS de Lucassen et al. formaliza criterios de calidad y reporta evidencias empíricas de mejora cuando se aplican guías disciplinadas [45, 46]. La disponibilidad de *datasets* abiertos, como el de Dalpiaz, ha impulsado aproximaciones reproducibles al procesamiento automático de requisitos [33].

En la línea IR–IA, se han explorado transformaciones de texto a modelos de diseño [47] y esquemas de agrupamiento semántico (e.g., SEMGROMI) para proponer candidatos a microservicios [11]. Sin embargo, la *similitud semántica* no garantiza *cohesión funcional*. La anisotropía de *embeddings* puede elevar artificialmente la similitud coseno entre frases no relacionadas, aumentando falsos positivos [48]. Por ello, en esta tesis la IA *propone*, pero un validador computa métricas y un algoritmo evolutivo *selecciona* en función de señales de calidad estructural.

3.3. Modelo Matemático Mínimo de Arquitectura

Para razonar sin imponer una notación pesada, modelamos una arquitectura candidata como un grafo dirigido rotulado:

$$\mathcal{A} = (S, E, O, P),$$

donde S es un conjunto finito de servicios; $E \subseteq S \times S$ son llamadas dirigidas (*fan-out*); $O : S \to \mathcal{P}(Op)$ asigna a cada servicio su conjunto finito de operaciones; $P : Op \to \mathcal{P}(Param)$ asigna a cada operación su conjunto finito de parámetros. Convención operativa: se *rechazan* lazos $s \to s$ y referencias a servicios inexistentes.

Notación. $O_s := O(s)$, $P(o)$ parámetros de $o \in O_s$, y $N^+(s) := \{t \in S : (s,t) \in E\}$. La Figura 3.2 traduce este modelo formal a un diagrama conceptual.

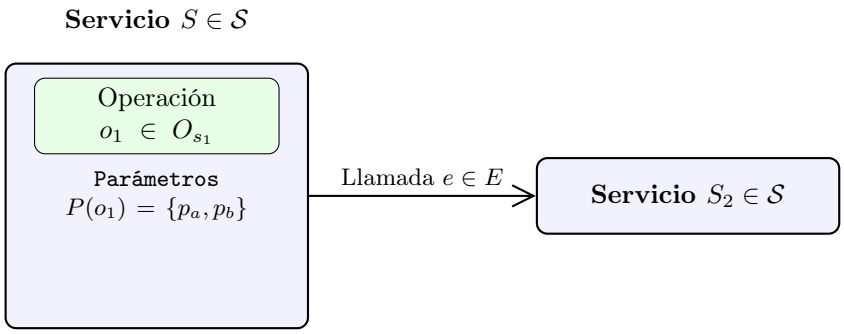


Figura 3.2: Visualización del modelo matemático mínimo de una arquitectura $\mathcal{A} = (S, E, O, P)$.

Hipótesis y alcance

Las métricas son **estructurales y estáticas**. No se modelan métricas de *runtime* (latencia, caudal, SLA) ni propiedades temporales. Tampoco se imponen umbrales duros: las señales se combinan escalarmente en la función de aptitud.

3.4. Preliminares matemáticos

Este apartado fija una notación mínima para el resto del capítulo, sin introducir resultados avanzados.

Conjuntos finitos. Denotamos por $\mathcal{S}$ al conjunto finito de microservicios y por O_s al conjunto (finito) de operaciones del microservicio $s \in \mathcal{S}$. Usaremos $|X|$ para la cardinalidad de un conjunto finito X y notación de comprensión estándar. Estos básicos provienen de la teoría elemental de conjuntos [65].

Grafo de llamadas. El sistema induce un grafo dirigido y simple $G = (V, E)$, donde $V = \{\, s \mid s \in \mathcal{S} \,\}$ y

$$E \subseteq \{\, (s_i, s_j) \in V \times V \mid s_i \neq s_j \,\}.$$

Un arco $(s_i, s_j) \in E$ significa que s_i invoca a s_j. El *fan-out* de s es su grado de salida $d^+(s) = |\{\, t \in V \mid (s,t) \in E \,\}|$. Este encuadre es el estándar de teoría de grafos aplicada [66] y es el que utilizamos para cuantificar acoplamiento saliente.

Agregación sobre servicios. Dada una función métrica $\mu : \mathcal{S} \to \mathbb{R}_{\geq 0}$, usaremos los agregados elementales:

$$\operatorname{avg}(\mu) = \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} \mu(s), \qquad \max(\mu) = \max_{s \in \mathcal{S}} \mu(s), \qquad \operatorname{sum}(\mu) = \sum_{s \in \mathcal{S}} \mu(s).$$

Cuando apliquemos una *desviación estándar poblacional* a una lista de valores $x_1, \ldots, x_n$, usaremos

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2}, \quad \bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i,$$

coincidiendo con la implementación computable del experimento (divisor n).

Convención sobre similitud semántica. Aunque esta tesis *no* usa la similitud coseno como criterio de agrupamiento, advertimos que su geometría puede ser engañosa en embeddings contextuales (fenómeno de *anisotropía*), motivo por el cual priorizamos métricas estructurales y verificación formal [67].

Con esta notación, las métricas de la Sección 3.6 y la función de aptitud empleada en el experimento quedan formalizadas de manera precisa y operativa.

3.5. Microcasos ilustrativos de cómputo de métricas

Presentamos ejemplos pequeños que reproducen exactamente la semántica computable usada en el experimento.

3.5.1. Cómputo Métrico-Formal: De Genotipo JSON a Aptitud (Fitness)

Para materializar la integración entre la generación de artefactos y la verificación formal, esta sección presenta un ejemplo constructivo unificado. El propósito es ilustrar, paso a paso, cómo el pipeline de ArchiGenMS traduce el artefacto JSON—el genotipo $\mathcal{G}$ generado por el LLM—en las métricas estructurales computables y, finalmente, en el valor escalar de la función de aptitud ($\mathcal{F}$).

Este proceso es determinista y ejecutable. El orquestador de Python invoca al verificador compilado en Lean 4 [30], el cual actúa como un oráculo que deserializa el JSON, aplica las definiciones formales de las métricas (Sección 3.6) y verifica los invariantes estructurales (Anexo B).

Paso 1: El Genotipo JSON (Salida del LLM)

Considérese el siguiente genotipo candidato $\mathcal{G}$, generado por el LLM y representado en formato JSON. Este artefacto es la entrada directa al validador Lean.

```
{
  "microservices": [
    {
      "name": "UserService",
      "ops": [
        {"name": "createUser", "params": ["user", "pass"]},
        {"name": "updateUser", "params": ["user", "data"]},
        {"name": "getInfo", "params": ["ticket"]}
      ]
    },
    {
      "name": "AuthService",
      "ops": [
        {"name": "login", "params": ["user", "pass"]},
        {"name": "resetPass", "params": ["user"]}
      ]
    },
    {
```

```
      "name": "DataService",
      "ops": [
        {"name": "getData", "params": ["dataId"]}
      ]
    }
  ],
  "calls": [
    {"caller": "UserService", "callee": "AuthService"},
    {"caller": "UserService", "callee": "DataService"},
    {"caller": "AuthService", "callee": "DataService"}
  ]
}
```

Listado 3.1: Genotipo JSON candidato ($\mathcal{G}$) de entrada al verificador.

Paso 2: Verificación de Invariantes Estructurales (Lean)

Antes de calcular las métricas, el validador Lean [30] deserializa el JSON (Listado 3.1) y comprueba los invariantes estructurales definidos (ver Anexo B):

- **I1 (Integridad de Referencias):** Se comprueba que `caller` y `callee` en todas las llamadas (`UserService`, `AuthService`, `DataService`) existan en la lista `microservices`. (Resultado: OK).
- **I2 (No Self-Call):** Se comprueba que no existan llamadas `(s, s)`. (Resultado: OK).

Dado que Valid($\mathcal{G}$) es verdadero, el validador procede al cómputo de métricas.

Paso 3: Cómputo de Métricas por Servicio (Lean)

El verificador itera sobre cada servicio en la lista `"microservices"` y analiza la lista `çalls"` para calcular las métricas individuales.

Servicio "UserService" (S_1).

- **SGM (Granularidad):** Lean cuenta los elementos en la lista `ops`.

$$\text{SGM}(S_1) = |ops| = 3.$$

- **LCOM (Cohesión):** Se computan los pares de operaciones ($\binom{3}{2} = 3$ pares):
  - `(createUser, updateUser)`: Comparten `"user"`. $\rightarrow P_s$.
  - `(createUser, getInfo)`: No comparten parámetros. $\rightarrow P_{ns}$.
  - `(updateUser, getInfo)`: No comparten parámetros. $\rightarrow P_{ns}$.

  Resultados: $|P_s| = 1$; $|P_{ns}| = 2$.

$$\text{LCOM}(S_1) = \frac{|P_{ns}|}{|P_{ns}| + |P_s|} = \frac{2}{2+1} = 0{,}667.$$

- **SGM_SD (Dispersión Granular):** Se mapean las operaciones a la cardinalidad de sus `params`: $[|\{\texttt{"user"}, \texttt{"pass"}\}|, |\{\texttt{"user"}, \texttt{"data"}\}|, |\{\texttt{"ticket"}\}|] = [2, 2, 1]$. Media: $\bar{x}_1 = (2 + 2{+}1)/3 = 1{,}667$. Varianza (poblacional): $\sigma_1^2 = \frac{(2-1{,}667)^2+(2-1{,}667)^2+(1-1{,}667)^2}{3} = \frac{0{,}111+0{,}111+0{,}444}{3} = 0{,}222$.

$$\text{SGM_SD}(S_1) = \sqrt{0{,}222} = 0{,}471.$$

Servicio "AuthService" (S_2).

- **SGM:** $\text{SGM}(S_2) = |ops| = 2$.
- **LCOM:** Se computa el único par ($\binom{2}{1} = 1$ par):
  - `(login, resetPass)`: Comparten `"user"`. $\rightarrow P_s$.

  Resultados: $|P_s| = 1$; $|P_{ns}| = 0$. Por convención computable (Sección 3.6), si $|P_s| = 0$ o $|P_{ns}| = 0$:

$$\text{LCOM}(S_2) = 0{,}0.$$

- **SGM_SD:** Mapeo de cardinalidad: $[|\{\texttt{"user"}, \texttt{"pass"}\}|, |\{\texttt{"user"}\}|] = [2, 1]$. Media: $\bar{x}_2 = (2 + 1)/2 = 1{,}5$. Varianza: $\sigma_2^2 = \frac{(2-1{,}5)^2+(1-1{,}5)^2}{2} = \frac{0{,}25+0{,}25}{2} = 0{,}25$.

$$\text{SGM_SD}(S_2) = \sqrt{0{,}25} = 0{,}500.$$

Servicio "DataService"(S_3).

- **SGM:** $\text{SGM}(S_3) = |ops| = 1$.
- **LCOM:** Por convención (Sección 3.6), si $|ops| < 2$, no hay pares que computar: $\text{LCOM}(S_3) = 0{,}0$.
- **SGM_SD:** Por convención (Sección 3.6), si $|ops| = 1$, la desviación es nula: $\text{SGM_SD}(S_3) = 0{,}0$.

Grafo de Llamadas (Acoplamiento). El verificador analiza la lista `çalls"` del **mismo genotipo JSON** (Listado 3.1) para computar el Fan-Out (grado de salida d^+) de cada servicio [66]:

- **Coupling (S_1):** Filtra `calls` por `caller == ÜserService"`. Encuentra `callee: .ªuthService"` y `callee: "DataService"`.

$$\text{FO}(S_1) = d^+(S_1) = 2.$$

- **Coupling (S_2):** Filtra `calls` por `caller == .ªuthService"`. Encuentra `callee: "DataService"`.

$$\text{FO}(S_2) = d^+(S_2) = 1.$$

- **Coupling (S_3):** Filtra `calls` por `caller == "DataService"`. No encuentra llamadas.

$$\text{FO}(S_3) = d^+(S_3) = 0.$$

Paso 4: Agregación de Métricas y Salida Lean

Lean agrega las métricas individuales en los cuatro indicadores globales que definen la función de aptitud:

- $LCOM_{avg}$: $\frac{\text{LCOM}(S_1)+\text{LCOM}(S_2)+\text{LCOM}(S_3)}{3} = \frac{0{,}667+0{,}0+0{,}0}{3} = 0{,}222$.
- SGM_{max}: $\max\{\text{SGM}(S_1), \text{SGM}(S_2), \text{SGM}(S_3)\} = \max\{3, 2, 1\} = 3{,}0$.
- SGM_SD_{sum}: $\text{SGM_SD}(S_1)+\text{SGM_SD}(S_2)+\text{SGM_SD}(S_3) = 0{,}471+0{,}500+0{,}0 = 0{,}971$.
- $Coupling_{max}$: $\max\{\text{FO}(S_1), \text{FO}(S_2), \text{FO}(S_3)\} = \max\{2, 1, 0\} = 2{,}0$.

El verificador Lean 4 serializa estos resultados en un JSON de salida, que es retornado al orquestador de Python:

```
{
  "status": "OK",
  "message": "",
  "metrics": {
    "lcom_avg": 0.222,
    "sgm_max": 3.0,
    "sgm_sd_sum": 0.971,
    "coupling_max": 2.0
  }
}
```

Listado 3.2: Reporte JSON de salida del verificador Lean para el genotipo $\mathcal{G}$.

Paso 5: Cómputo de la Función de Aptitud Agregada ($\mathcal{F}$)

Finalmente, el orquestador de Python recibe este JSON de salida. Al verificar que `status == "OK"`, procede a calcular la aptitud del genotipo $\mathcal{G}$ sumando linealmente los valores del objeto `metrics`, tal como se define en el Anexo B:

$$\mathcal{F} = \text{avg(LCOM)} + \max(\text{SGM}) + \text{sum(SGM_SD)} + \text{Coupling}_{\text{máx}}$$

Sustituyendo los valores del Listado 3.2:

$$\mathcal{F} = \underbrace{0{,}222}_{\text{avg(LCOM)}} + \underbrace{3{,}0}_{\max(\text{SGM})} + \underbrace{0{,}971}_{\text{sum(SGM_SD)}} + \underbrace{2{,}0}_{\text{Coupling}_{\text{máx}}}$$

$$\boxed{\mathcal{F} = 6{,}193.}$$

Este valor escalar $\mathcal{F}$ representa la ïnaptitud"del genotipo $\mathcal{G}$, que el algoritmo evolutivo de ArchiGenMS utilizará para comparar, seleccionar y descartar individuos en su búsqueda de arquitecturas estructuralmente superiores.

Discusión y Trazabilidad. Este ejemplo constructivo demuestra que el pipeline ArchiGenMS no opera como una çaja negra". Cada métrica estructural—cohesión, granularidad y acoplamiento—se deriva de manera determinista y computable a partir de la estructura explícita del genotipo JSON.

La integración del verificador Lean 4 [30] no solo aporta rigor al cálculo, sino que impone invariantes estructurales que filtran *alucinaciones* o diseños inconsistentes generados por el LLM.

Este mecanismo de "generar-verificar.[es], por tanto, el núcleo metodológico que permite a ArchiGenMS optimizar la calidad arquitectónica de forma automática y reproducible [28]. El uso de métricas estructurales (LCOM, SGM, Fan-Out) [31, 32] se fundamenta en la necesidad de obtener propiedades verificables y objetivas del diseño, en contraste con métricas basadas en la proximidad semántica de embeddings (p.ej., [67]), las cuales pueden ser anisotrópicas y no garantizar la coherencia estructural [48]. El modelo de grafo y acoplamiento sigue la literatura estándar de teoría de grafos [66].

3.6. Métricas Arquitectónicas Computadas en el Experimento

Las métricas se calculan mediante un ejecutable en Lean 4 (modo operativo) que procesa el genotipo JSON propuesto por IA y retorna:

$$\{\, LCOM_avg,\ SGM_max,\ SGM_SD_sum,\ Coupling_max \,\}.$$

Si la estructura es inválida (e.g., *self-calls*, referencias inexistentes), el validador retorna `ERROR` y el evolutivo asigna una penalización alta.

Cohesión (LCOM, versión computable)

Para $s \in S$ y O_s:

$$P_s := \big\{\{o_1, o_2\} \subseteq O_s : P(o_1) \cap P(o_2) \neq \emptyset\big\}, \quad P_{ns} := \big\{\{o_1, o_2\} \subseteq O_s : P(o_1) \cap P(o_2) = \emptyset\big\}.$$

Definición computada:

$$LCOM(s) = \begin{cases} 0{,}0, & \text{si } |O_s| < 2 \text{ o } |P_s| = 0 \text{ o } |P_{ns}| = 0, \\ \dfrac{|P_{ns}|}{|P_{ns}| + |P_s|}, & \text{en otro caso.} \end{cases}$$

Agregado reportado: $LCOM_avg := \frac{1}{|S|} \sum_{s \in S} LCOM(s)$.

Granularidad (SGM)

$$SGM(s) := |O_s|, \qquad SGM_max := \max_{s \in S} SGM(s).$$

Intuitivamente, SGM_max penaliza *macro-servicios* y empuja a distribuciones más balanceadas.

Variación de granularidad (SGM-SD)

Sea $X_s := \{|P(o)| : o \in O_s\}$. Definimos:

$$SGM_SD(s) := \sqrt{\frac{1}{|O_s|} \sum_{o \in O_s} \big(|P(o)| - \overline{|P|}\big)^2}, \quad SGM_SD_sum := \sum_{s \in S} SGM_SD(s).$$

Caso degenerado: si $|O_s| = 0$, por convención computable $SGM_SD(s) = 0{,}0$.

Acoplamiento saliente (Fan-Out)

$$Coupling(s) := |N^+(s)|, \qquad Coupling_max := \max_{s \in S} Coupling(s).$$

Interpretación: reducir el máximo de dependencias externas disminuye fragilidad ante cambios.

Propiedades básicas (bajo las convenciones del experimento)

Rango y no-negatividad. $LCOM(s) \in [0, 1]$; $SGM(s), SGM_SD(s), Coupling(s) \geq 0$.
Invariancia por renombre. Las métricas no cambian por renombrado de servicios/operaciones.
Monotonicidad simple. SGM_max y $Coupling_max$ son no-decrecientes ante inserciones de operaciones y aristas, respectivamente; $LCOM_avg$ y SGM_SD_sum dependen de co-ocurrencias y uniformidad resultante.

Ejemplos numéricos

LCOM. Sea $O_s = \{o_1, o_2, o_3\}$ con $P(o_1) = \{a, b\}$, $P(o_2) = \{b, c\}$, $P(o_3) = \{x\}$. Pares: (o_1, o_2) comparten b; (o_1, o_3) y (o_2, o_3) no comparten. Así $|P_s| = 1$, $|P_{ns}| = 2$ y $LCOM(s) = 2/(2 + 1) =$ 0,667. Si o_3 cambia a $P(o_3) = \{a\}$, entonces $|P_s| = 2$, $|P_{ns}| = 1$ y $LCOM(s) =$ 0,333 (mejor cohesión).

SGM y SGM_SD. Si $|O_s| = 4$ y $|P(o)| = \{2, 2, 2, 2\}$, entonces $SGM(s) = 4$ y $SGM_SD(s) = 0$ (uniformidad perfecta). Si cambiamos una firma a 6 parámetros, $SGM_SD(s)$ aumenta, penalizando heterogeneidad.

$Coupling_max$. Si un servicio concentra 6 dependencias salientes y el resto ≤ 2, el máximo domina el agregado y la aptitud empeora; redistribuir llamadas reduce el máximo y mejora la señal.

3.7. Lean 4 como Validador Computable

La validación de calidad en esta tesis se centra en la *estructura estática* de la arquitectura (cohesión, acoplamiento, granularidad) y no en el comportamiento dinámico. Bajo este foco, adoptamos Lean 4 como un componente *operativo* del *pipeline*: un ejecutable que formaliza y calcula métricas, verifica invariantes estructurales mínimas y produce evidencias reproducibles de corrección.[2]

Esta elección metodológica contrasta con herramientas orientadas al análisis temporal de estados, y alinea la verificación con el lenguaje matemático de las métricas empleadas en el capítulo.

3.7.1. Panorama

Model checking y lenguajes de especificación (TLA+, Alloy) son idóneos para propiedades temporales, de concurrencia y consistencia [53, 55, 54]. En este trabajo el foco es *calidad estructural estática* del diseño, por lo que se adopta Lean 4 en modo **operativo**: ejecutable que calcula métricas y verifica estructura mínima [30] (véase § 6.2).

[2] Detalles del entorno en § 6.2.

3.7.2. Rol concreto en el pipeline

El ejecutable Lean: (i) parsea el genotipo JSON; (ii) rechaza *self-calls* y referencias a servicios inexistentes; (iii) computa $LCOM_avg$, SGM_max, SGM_SD_sum y $Coupling_max$ usando `Float`; (iv) retorna un JSON con `status`, `message` y `metrics`. No se incluyen pruebas de invariantes ni umbrales duros: esas extensiones se registran como trabajo futuro razonable.

3.7.3. Fragmento ilustrativo

El siguiente listado ilustra el núcleo del **validador**. En lugar de mostrar solo el cálculo de métricas (cuya implementación completa es extensa), este fragmento demuestra cómo Lean 4 impone *invariantes estructurales* antes de cualquier cómputo.

El código define las estructuras de datos mínimas e implementa la función `validateInvariants`, que comprueba la ausencia de auto-llamadas (*self-calls*) y la integridad de las referencias (que todos los `caller` y `callee` existan), tal como se describe en la Sección 3.7.2.

```
-- Definici n de tipos de datos (simplificada)
structure Service where
  name : String
  deriving Repr, BEq

structure Call where
  caller : String
  callee : String
  deriving Repr, BEq

structure Genotype where
  microservices : List Service
  calls : List Call
  deriving Repr

/--
 - Valida invariantes estructurales basicos.
 - I1: No hay auto-llamadas (self-calls).
 - I2: Todas las llamadas refieren a servicios existentes.
 - Retorna (Except String Unit): 'Unit' en exito o un 'String' de error.
-/
def validateInvariants (gt : Genotype) : Except String Unit := do
  -- 1. Construir un HashSet con los nombres de servicios validos
  let serviceNames := (gt.microservices.map (fun s => s.name)).toHashSet

  -- 2. Iterar sobre cada llamada y verificar invariantes
  for c in gt.calls do
    -- I1: Verificar que no sea una auto-llamada
    if c.caller == c.callee then
      throw s!"Error [I1]: Auto-llamada (self-call) en '{c.caller}'"

    -- I2: Verificar integridad de referencias
    if ¬ (serviceNames.contains c.caller) then
      throw s!"Error [I2]: 'caller' no existe: '{c.caller}'"
```

```
35
36      if ¬ (serviceNames.contains c.callee) then
37        throw s!"Error [I2]: 'callee' no existe: '{c.callee}'"
38
39    -- Si todas las validaciones pasan
40    pure () -- Retorna OK (Unit)
```

Listado 3.3: Implementación de la validación de invariantes en Lean 4.

3.8. Conexión con el Algoritmo Evolutivo

El algoritmo evolutivo es el motor que guía la búsqueda de arquitecturas óptimas. Su operación se basa en tres componentes clave que se detallan a continuación: la función de aptitud que cuantifica la calidad de cada genotipo, el ciclo de generación y selección que refina la población, y la sensibilidad del proceso que asegura la reproducibilidad.

3.8.1. Función de aptitud

La búsqueda minimiza la suma directa:

$$\mathcal{F} = LCOM_avg + SGM_max + SGM_SD_sum + Coupling_max.$$

Si el genotipo es inválido estructuralmente, se asigna una penalización grande (fitness alto). Esta definición evita *tuning* de pesos y facilita el análisis de sensibilidad.

3.8.2. Generación, validación, selección

La IA *propone* arquitecturas a partir de historias de usuario (prompt base e inspirado); Lean *valida y computa* métricas; el evolutivo *selecciona* por $\mathcal{F}$. El sistema persiste por individuo: genotipo, métricas, $\mathcal{F}$, validación y tiempos, además del mejor por archivo y un CSV global para auditoría y replicación.

3.8.3. Reproducibilidad y sensibilidad

El uso de una función agregada simple permite comparar ejecuciones entre datasets y corridas, y auditar trayectorias evolutivas. Cambios locales (p.ej., reducción de una dependencia o homogeneización de firmas) impactan predeciblemente en $\mathcal{F}$, favoreciendo interpretabilidad.

3.9. Discusión, Limitaciones y Extensiones

Estatuto teórico. Las métricas empleadas se anclan en literatura de evaluación de diseños [31, 32] y aquí se concretan como funciones computables sobre un grafo rotulado. Esta concreción no minimiza su fundamento: por el contrario, explicitar supuestos y convenciones es condición de comparabilidad inter-experimentos.

Límites actuales. No se modelan propiedades dinámicas (rendimiento, latencia, SLA) ni se imponen umbrales duros. Tampoco se detectan ciclos inter-servicios como condición de rechazo.

Incluir pesos, normalizaciones o criterios multiobjetivo alteraría el comportamiento del evolutivo y se reserva como trabajo futuro.

Posibles extensiones. (i) Normalizar señales para mitigar dominancias; (ii) tratamiento explícito de ciclos (p.ej., penalización suave); (iii) exploración multiobjetivo (frentes de Pareto cohesión–acoplamiento–granularidad); (iv) integración posterior de propiedades de *runtime* (con *proxies* estáticos o simulaciones).

3.10. Síntesis

Este capítulo ha establecido los fundamentos teóricos y matemáticos que sustentan el artefacto ArchiGenMS, articulando el **Ciclo de Rigor** de la metodología DSR [26, 27]. Se ha transitado desde los principios arquitectónicos cualitativos (Sección 3.1) y la justificación de por qué las heurísticas de similitud semántica (PLN) son insuficientes (Sección 3.2), hasta un modelo matemático mínimo ($\mathcal{A}$) que define una arquitectura como un grafo rotulado (Sección 3.3).

El núcleo de la contribución de este capítulo reside en la **formalización computable** de las métricas de calidad estructural (LCOM, SGM, SGM-SD y Coupling/Fan-Out). A diferencia de las definiciones teóricas, la Sección 3.6 ha establecido la semántica precisa y las convenciones para casos degenerados que utiliza el ejecutable del experimento, garantizando la reproducibilidad.

Se ha demostrado la **trazabilidad completa** del pipeline mediante un ejemplo constructivo (Sección 3.5.1), ilustrando inequívocamente cómo el validador en Lean 4 (Sección 3.7) traduce un artefacto JSON (Listado 3.1) en un reporte métrico determinista (Listado 3.2), tras verificar sus invariantes estructurales.

Finalmente, se ha consolidado la función de aptitud ($\mathcal{F}$) como la suma lineal de estos indicadores, proveyendo la señal cuantitativa, objetiva y formalmente verificada que guía la búsqueda del algoritmo evolutivo (Sección 3.8). Habiendo establecido esta base formal, los capítulos siguientes pueden proceder a describir la implementación del *pipeline* (Capítulo 6) y su evaluación empírica (Capítulo 7) sobre la base de estas métricas validadas.

Capítulo 4

Trabajos Relacionados

La investigación sobre microservicios ha evolucionado de manera sostenida en la última década, desde definiciones y patrones fundacionales hasta la incorporación de enfoques con inteligencia artificial (IA) y esfuerzos por dotar de verificabilidad y reproducibilidad a las propuestas generadas automáticamente. El estado del arte puede organizarse en tres líneas complementarias que convergen en esta tesis: (i) la evolución del paradigma y sus desafíos prácticos (granularidad, cohesión, acoplamiento, migraciones); (ii) el empleo de IA—desde NLP/ML hasta LLMs—para asistir el descubrimiento y el diseño; y (iii) la utilización de métodos formales y marcos de métricas para evaluar calidad estructural. Esta tesis se sitúa precisamente en la intersección: integra generación asistida por IA con métricas *computables* y verificación operativa en Lean 4, cerrando la brecha entre plausibilidad y garantía de corrección estructural.

4.1. Microservicios: evolución, patrones y desafíos persistentes

La definición canónica de microservicios como servicios pequeños, autónomos y desplegables de forma independiente, asociada a Lewis y Fowler, marcó un hito conceptual que orientó tanto la práctica industrial como la investigación [56]. Dragoni et al. posicionan este paradigma dentro del ecosistema de software contemporáneo (nube, contenerización, orquestación), destacando que no es sólo un catálogo de patrones sino una filosofía de diseño con implicaciones organizacionales y de operación [57]. Newman sistematiza principios y prácticas ya consolidados: modelar en torno al dominio, despliegue independiente, automatización de infraestructura, observabilidad y tolerancia a fallos [1].

Estudios empíricos y mapeos sistemáticos describen motivaciones (escalabilidad, autonomía de equipos) y dificultades recurrentes: delimitar servicios, gobernar datos distribuidos y sostener la comunicación inter-servicios sin agravar el acoplamiento [4, 15, 37, 7, 3, 36]. Estos trabajos convergen en un diagnóstico: la calidad estructural—en particular la combinación de *alta cohesión*, *bajo acoplamiento* y *granularidad balanceada*—es difícil de sostener sin criterios explícitos, datos de apoyo y herramientas de evaluación.

Estos trabajos convergen en un diagnóstico: la calidad estructural—en particular la combinación de *alta cohesión*, *bajo acoplamiento* y *granularidad balanceada*—es difícil de sostener sin criterios

explícitos, datos de apoyo y herramientas de evaluación. La dificultad no es meramente teórica; estudios empíricos con expertos han validado que áreas críticas como la gestión de datos, la observabilidad (monitoreo y logging) y la infraestructura son percibidas como los mayores desafíos en la práctica [38]. Revisiones multivocales han catalogado formalmente los *olores arquitectónicos* (*architectural smells*) —como los *God Services* o la *Persistencia Compartida*— que surgen de un diseño deficiente [14]. La consecuencia extrema de estos desafíos es la tendencia documentada de *reversión al monolito*, donde organizaciones optan por abandonar la arquitectura de microservicios para mitigar la complejidad accidental y los costos operativos insostenibles [13].

Migración desde monolitos. En escenarios *brownfield*, la descomposición guiada por análisis estático/dinámico de dependencias ha producido herramientas prácticas. Mono2Micro explota trazas y estructuras para suprimir dependencias y sugerir cortes [9]; CARGO incorpora IA para analizar dependencias y asistir la partición de componentes [10]. Otros trabajos introducen enfoques semánticos o basados en tipos para legados [44]. Aunque efectivos en migración, estos métodos dependen del código y de la ejecución; resultan menos adecuados cuando el punto de partida son *requisitos textuales* (contexto *greenfield*), foco de esta tesis.

Síntesis. La literatura consolida beneficios (escalabilidad, mantenibilidad) y evidencia retos persistentes: definir límites adecuados y evaluar calidad estructural antes de implementar. Esta tensión abre la puerta a enfoques que integren datos textuales, criterios cuantitativos y verificación.

4.2. IA para el descubrimiento y diseño de microservicios

Esta segunda línea de investigación, el empleo de Inteligencia Artificial para asistir el descubrimiento y diseño, es fundamental para el posicionamiento de esta tesis. El análisis de la literatura revela que este campo no es monolítico, sino que se articula en torno a tres enfoques complementarios que han evolucionado en paralelo: (1) los fundamentos teóricos de la optimización arquitectónica, basados en la disciplina de la **Ingeniería de Software Basada en Búsqueda (SBSE)**; (2) las técnicas de **NLP y Machine Learning (ML)** aplicadas al agrupamiento semántico de requisitos; y (3) las **tendencias generativas** emergentes que utilizan Modelos de Lenguaje Grandes (LLMs).

No obstante, antes de analizar los enfoques de automatización (IA), es imperativo establecer la línea de base de la práctica industrial.

Práctica Industrial Vigente: Enfoques Heurísticos y Manuales

La evidencia empírica más reciente confirma que la adopción de microservicios en la industria se sustenta, predominantemente, en métodos manuales, heurísticos y dependientes de la pericia del equipo. Investigaciones basadas en encuestas y entrevistas [17, 6] demuestran que, si bien el *Domain-Driven Design* (DDD) es reconocido como el principal enfoque metodológico para definir los límites del servicio [17], su aplicación sigue siendo abstracta, carece de un proceso unificado y depende críticamente de la experiencia de arquitectos *senior* [17].

En los contextos ágiles, esta dependencia de la heurística se confirma: el descubrimiento de microservicios se apoya en talleres colaborativos como el *Event Storming* o la aplicación manual de los patrones de DDD [8]. Esta brecha entre los principios del paradigma y las herramientas disponibles genera una notable inercia metodológica. Estudios estructurados con practicantes revelan que

las organizaciones continúan utilizando notaciones tradicionales de Análisis y Diseño Orientado a Objetos (OOAD) —como diagramas de clase y componentes— y métodos de estimación subjetivos que fueron diseñados para arquitecturas monolíticas, fallando en adaptar sus prácticas de ingeniería a la naturaleza distribuida y asíncrona de los microservicios [6].

Esta línea de base (manual, dependiente del experto y basada en la adaptación de métodos preexistentes) define el *status quo* que los enfoques de AI4SE, incluyendo ArchiGenMS, buscan superar mediante la automatización, la cuantificación y la verificación.

Fundamentos en la Ingeniería de Software Basada en Búsqueda (SBSE)

El uso de IA para la optimización del diseño arquitectónico no es una práctica aislada, sino que se fundamenta en la disciplina de la **Ingeniería de Software Basada en Búsqueda (SBSE)** [35]. La SBSE reformula los problemas de ingeniería de software como problemas de optimización, donde las técnicas metaheurísticas se emplean para navegar espacios de soluciones que son vastos o computacionalmente intratables [43].

El problema canónico en esta área es la **modularización de software**. Trabajos seminales demostraron que la optimización de la modularidad (es decir, agrupar entidades de código para maximizar la cohesión y minimizar el acoplamiento) es formalmente un problema de particionamiento de grafos [68]. Hallar la partición óptima en este contexto es un problema **NP-hard**, lo que invalida los enfoques de búsqueda exhaustiva y justifica plenamente el uso de metaheurísticas como los Algoritmos Evolutivos (AE).

Los primeros trabajos en esta línea, como la herramienta "Bunch", utilizaron funciones de aptitud (*fitness functions*) mono-objetivo, como la **Modularization Quality (MQ)**, que colapsa la cohesión y el acoplamiento en un único valor escalar a maximizar [68].

Sin embargo, para que un AE opere eficazmente sobre una arquitectura, tanto la representación (**genotipo**) como los operadores genéticos (**cruce** y **mutación**) deben estar adaptados al dominio. Se demostró que los operadores estándar son "semánticamente destructivos" en este contexto. Por ello, se propusieron representaciones y operadores de cruce específicos que preservan los "Bloques de Construcción" (módulos cohesivos) al crear nueva descendencia [69], validando la necesidad de un enfoque semántico para la variación.

Posteriormente, el campo maduró al reconocer que la cohesión y el acoplamiento son **objetivos fundamentalmente en conflicto** [70]. Un diseño con acoplamiento externo cero (un monolito) tiene una cohesión desastrosa, mientras que un diseño con cohesión perfecta (cada clase en su módulo) tiene un acoplamiento explosivo. Esta constatación impulsó la transición del estándar de SBSE desde la optimización mono-objetivo hacia la **Optimización Multi-Objetivo (MOO)**.

Los enfoques MOO, que emplean algoritmos como **NSGA-II** o **NSGA-III** [71], no buscan una única solución "perfecta". En su lugar, generan un **Frente de Pareto**: un conjunto de soluciones óptimas no-dominadas que exponen los diferentes *trade-offs* (intercambios) posibles entre cohesión y acoplamiento, permitiendo al arquitecto tomar una decisión de diseño informada.

Esta misma metodología MOO ha sido identificada como el enfoque de vanguardia para la descomposición de microservicios, que es una instancia moderna del problema de modularización. Trabajos recientes utilizan NSGA-II para extraer microservicios optimizando explícitamente la cohesión y el acoplamiento estructural [72]. La frontera actual de la investigación, representada por trabajos como MONO2REST, ya aplica NSGA-III para optimizar simultáneamente objetivos estructurales (dependencias de código) y objetivos **semánticos** (derivados del análisis PLN del código y comentarios) [73].

NLP/ML sobre requisitos textuales

Una línea de trabajo parte de la hipótesis de que las historias de usuario y casos de uso contienen suficiente señal para proponer límites de servicio. SEMGROMI agrupa historias por similitud semántica para derivar candidatos a microservicios [11]. GreenMicro enfoca desarrollos *greenfield* a partir de casos de uso textuales [12]. GTMicro incorpora *deep transformers* y *clustering* para identificar microservicios en etapas tempranas [49]. En paralelo, se han explorado transformaciones más directas de historias hacia artefactos de diseño con pipelines de ML/NLP que reducen intervención manual [47]. *Limitación central*: la similitud lingüística no es garantía de cohesión funcional; además, estos enfoques suelen carecer de un eslabón de rechazo automático de estructuras inconsistentes.

LLMs y generación guiada

La irrupción de LLMs ha motivado evaluar su utilidad en tareas arquitectónicas. Adams et al. [22] estudian ChatGPT en microservicios, reportando utilidad inicial pero con problemas críticos de consistencia; Stojanovic y Lazarević [23] analizan su capacidad para identificar microservicios, destacando ventajas en la extracción de entidades pero debilidades en la precisión del diseño. En esta misma línea, el estudio reciente de Pereira et al. [50] aporta evidencia empírica de vanguardia al evaluar el modelo de razonamiento GPT-4 o3. Sus resultados revelan una dicotomía técnica: mientras el modelo alcanza un alto recobro funcional en sistemas de referencia (Bookstore, PetClinic), las arquitecturas generadas mediante *zero-shot* presentan un acoplamiento bidireccional excesivo —calificado con apenas 2/5 en plausibilidad por expertos—, confirmando que la capacidad de razonamiento lingüístico no garantiza *per se* la calidad estructural. Para mitigar estas derivas, propuestas como PF4MD [51] combinan *prompt engineering* con guías de diseño, mientras que Díaz-Pace et al. [52] proponen asistentes para arquitectos novatos que incorporan conocimiento explícito. No obstante, revisiones exhaustivas [25, 19] señalan que la *verificabilidad* y la *confianza* siguen siendo problemas transversales. A pesar de la disponibilidad de datasets como el de Dalpiaz [33], persiste la carencia de un *benchmark* que evalúe la calidad estructural bajo métricas computables, vacío que ArchiGenMS busca cubrir mediante la integración de un oráculo formal.

Síntesis. La aplicación de IA al diseño de arquitecturas se sustenta en los fundamentos de la SBSE [35, 43], que formaliza el problema como una optimización metaheurística [68]. Los enfoques actuales divergen: (1) métodos de NLP/ML que agrupan requisitos [11, 49], (2) generación guiada por LLMs [19], y (3) optimización multi-objetivo (MOO) basada en AE para código existente [72, 73]. En todas las líneas persiste la distancia entre lo *plausible* (generado semánticamente) y lo *estructuralmente correcto*. La integración de criterios de evaluación formales sigue siendo marginal, constituyendo el vacío central que esta tesis aborda.

4.3. Métodos formales y métricas arquitectónicas

Los métodos formales sostienen la eliminación de ambigüedades y la detección temprana de errores [2]; el *model checking* se consolidó como técnica de verificación automática [53]; Alloy aporta lógica relacional y análisis acotado [54]; TLA+ formaliza propiedades temporales [55]. En el ámbito hispano, Pons et al. compendian fundamentos de lógica para informática [63]. Más recientemente,

Lean 4 combina teoría de tipos dependientes con un *runtime* eficiente, habilitando demostraciones y cálculos deterministas sobre estructuras [30].

Para juzgar calidad estructural en microservicios, marcos de métricas cuantifican cohesión, granularidad y acoplamiento, y orientan la comparación de descomposiciones [31, 32].

El valor de estos marcos de métricas reside en su capacidad para actuar como un **proxy cuantitativo** de los *olores arquitectónicos* (*architectural smells*) documentados en la literatura [14]. Por ejemplo, un *God Service* —un anti-patrón que centraliza lógica funcional disímil— [14] se manifestaría cuantitativamente como un servicio con una cohesión LCOM muy baja (valor numérico alto) y una granularidad SGM elevada. De igual modo, un acoplamiento externo excesivo (alto *Fan-Out* o *Coupling*) es un indicador de interacciones deficientes o frágiles (*Wobbly Service Interactions*) [14]. Por lo tanto, al integrar estas métricas directamente en la función de aptitud del ciclo evolutivo, ArchiGenMS no solo evalúa la calidad *a posteriori*, sino que *previene activamente* la generación de arquitecturas estructuralmente deficientes.

Sin embargo, muchas propuestas reportan métricas *sin* un eslabón formal que *rechace* automáticamente estructuras inválidas (auto-llamadas, referencias inexistentes). En sentido inverso, los métodos formales rara vez se integran a pipelines de diseño asistidos por IA a partir de requisitos textuales. Esta desarticulación motiva un enfoque que una *señales cuantitativas computables* con *verificación operativa*.

4.4. Delimitación: Calidad Estructural Estática vs. Optimización Dinámica en Runtime

Una vez analizados los enfoques de IA para el *diseño* (Sección 4.2) y la *evaluación estructural* (Sección 4.3), es metodológicamente crucial delimitar el alcance de esta tesis frente a otra robusta línea de investigación: la aplicación de IA para la **operación y optimización en tiempo de ejecución (runtime)**.

Esta rama del estado del arte, que no es el foco de este trabajo, se centra en propiedades operativas (NFRs dinámicos) y presume que una arquitectura estructuralmente *viable* ya existe. La literatura en esta área es extensa y aborda problemas como:

- **Autoescalado (Autoscaling):** Se han propuesto numerosas estrategias basadas en Machine Learning (ML) y Deep Learning (DL) para predecir la carga y ajustar dinámicamente los recursos (p.ej., réplicas de contenedores), superando a los escaladores heurísticos tradicionales [59, 58].
- **Depuración de Rendimiento:** Herramientas como SAGE [62] utilizan la telemetría (trazas distribuidas) y modelos de aprendizaje para localizar cuellos de botella y anomalías de latencia en grafos de servicios complejos en producción.
- **Tolerancia a Fallos y Auto-adaptación:** Se ha explorado el uso de Aprendizaje por Refuerzo (Reinforcement Learning) para que los sistemas aprendan políticas de auto-adaptación [60] y arquitecturas proactivas de tolerancia a fallos [61].

Contraste Metodológico. El contraste con la presente tesis es fundamental y define el *vacío* que ArchiGenMS ocupa. La literatura de *runtime* (analizada en esta sección, 4.4) persigue propiedades

operativas (SLA, latencia, *throughput*, resiliencia) y utiliza como artefacto de entrada métricas de observabilidad (logs, trazas, métricas de carga).

Por el contrario, esta tesis se circunscribe a la **calidad estructural estática**, un problema *previo* a la implementación. El objetivo es validar el *diseño* (el *blueprint*) antes de incurrir en costos de despliegue. Los artefactos de entrada son *requisitos textuales* y la validación se realiza mediante métricas *pre-implementación* (LCOM, SGM, Fan-Out) y verificación formal.

Ambos enfoques son complementarios: una optimización de *runtime* (Sección 4.4) sólo puede ser efectiva si parte de una arquitectura estructuralmente cohesiva y desacoplada, que es precisamente el objetivo que persigue esta tesis (Secciones 4.2 y 4.3).

4.5. Reproducibilidad, trazabilidad y comparación de enfoques

Las líneas anteriores convergen en dos exigencias metodológicas: *reproducibilidad* (código, datos, configuraciones abiertas) y *trazabilidad* (prompts, semillas, parámetros, artefactos intermedios). Las revisiones sobre IA generativa en arquitectura subrayan que sin estos atributos es difícil avanzar hacia confianza operativa [19]. Más aún, los problemas de granularidad y acoplamiento documentados por la literatura no se resuelven sólo con heurísticas lingüísticas o generación libre: requieren criterios formales y señales cuantitativas para *seleccionar* y *rechazar* arquitecturas.

Cuadro 4.1: Comparación sintética de líneas de trabajo relevantes

Línea / Ejemplos	**Entrada típica**	**Técnica principal**	**Salida típica**	**Validación / Reproducibilidad**
NLP/ML sobre requisitos [11, 47, 12, 49]	Historias / Casos (texto), dataset abierto [33]	Embeddings + clustering; ML supervisado/heurístico	Conjuntos candidatos de servicios	Validación limitada; reproducibilidad desigual; *sin* verificación estructural automática
LLMs para diseño [22, 23, 51, 52, 24, 50]	Historias, descripciones de dominio, *prompts*	Generación condicionada (instrucciones/patrones)	Borradores arquitectónicos, decisiones justificadas	Resultados plausibles; el *zero-shot* tiende al alto acoplamiento; verificabilidad débil sin intervención experta [50]
Práctica industrial (Heurística) [17, 6, 8]	Requisitos (texto), Conocimiento del experto	Manual (DDD, OOAD, Event Storming) [17, 6, 8]	Diagramas (UML), Documentos de diseño [6]	Subjetiva, dependiente del experto; baja reproducibilidad; *sin* verificación formal [17]
Métricas de diseño [31, 32]	Modelos de servicios (estáticos)	Cálculo de cohesión, granularidad, acoplamiento	Puntajes y reportes comparativos	Cuantificación clara; suele faltar *rechazo formal* de estructuras inválidas
Métodos formales [2, 53, 54, 55, 30, 63]	Modelos formales / grafos / lógicas	Model checking, análisis relacional, tipos dependientes	Veredictos de corrección, contraejemplos	Alta garantía; rara vez integrados a generación desde texto

4.6. Vacíos identificados y posicionamiento de esta tesis

Vacío 1 — De la plausibilidad a la corrección. Los enfoques con NLP/ML y LLMs *producen* candidatos desde texto, pero la evidencia experimental reciente [50] demuestra que incluso los modelos más avanzados (o3) fallan en métricas de acoplamiento estructural bajo condiciones de generación libre. Carecen de un *eslabón formal* que rechace automáticamente arquitecturas inválidas y que informe métricas *computables* para guiar la selección, obligando actualmente a una validación humana costosa y subjetiva.

Vacío 2 — Métricas desconectadas de la verificación. Existen marcos de métricas para comparar descomposiciones [31, 32], pero pocas soluciones las integran con un validador formal operativo que produzca un reporte determinista consumible por un algoritmo de búsqueda/selección.

Vacío 3 — Reproducibilidad y trazabilidad. Muchas propuestas no publican el paquete reproducible (código, datos, *prompts*, configuraciones, semillas) ni registran artefactos intermedios, pese a que la literatura remarca su importancia para construir confianza [19] y a la existencia de datasets abiertos [33].

Posicionamiento de la tesis. Esta tesis aborda los tres vacíos con un *pipeline* que: (i) usa IA para *proponer* arquitecturas desde requisitos textuales; (ii) computa métricas estructurales explícitas (cohesión, granularidad, acoplamiento) derivadas del propio artefacto; y (iii) integra **verificación formal operativa en Lean 4** para *rechazar* estructuras inválidas y producir un *reporte determinista* que alimenta la función de aptitud de un algoritmo evolutivo. El énfasis en reproducibilidad (código y datos abiertos, configuración declarativa, registro de *prompts* y semillas) alinea la contribución con las exigencias metodológicas actuales. Este posicionamiento dialoga con trabajos previos del autor [20, 18] y con la línea doctoral en curso [28].

4.7. Síntesis crítica

(1) La literatura de microservicios consolidó principios, patrones y retos, y mostró empíricamente que granularidad, cohesión y acoplamiento siguen siendo desafíos difíciles de gobernar [56, 57, 1, 4, 15, 37, 7, 3, 36].

La evidencia reciente ha profundizado este diagnóstico, no solo identificando las áreas de diseño más críticas (como la observabilidad y la gestión de datos) [38], sino también catalogando los *olores arquitectónicos* (*architectural smells*) resultantes de un mal diseño [14] y la consecuencia extrema de la reversión al monolito debido a la alta complejidad y costo [13].

(2) La práctica industrial ha intentado gobernar estos desafíos predominantemente mediante enfoques manuales y heurísticos, como el *Domain-Driven Design* (DDD) y talleres colaborativos [8]. Sin embargo, la evidencia empírica demuestra que la aplicación de DDD sigue siendo abstracta y altamente dependiente de la pericia del arquitecto [17], y que, en la práctica, los equipos ágiles persisten en el uso de notaciones tradicionales (ej. OOAD) que no se adaptan a la naturaleza del paradigma [6].

(3) La IA permite *proponer* arquitecturas desde texto (NLP/ML) y acelerar decisiones (LLMs), pero persisten limitaciones de verificabilidad y calidad estructural, con una marcada tendencia al sobre-acoplamiento en modelos generativos no guiados [11, 47, 12, 49, 22, 23, 51, 52, 24, 25, 19, 33, 50].

(4) Los métodos formales y las métricas ofrecen la base para garantizar corrección y evaluar calidad, aunque su integración con generación automática ha sido parcial [2, 53, 54, 55, 63, 30, 31,

32].

La tesis articula estas **cuatro** líneas en un *pipeline* reproducible que combina generación con IA, métricas computables y verificación en Lean 4 para promover la *selección* de arquitecturas con mejor calidad estructural—no sólo su producción—y contribuir a la agenda de confianza en arquitectura asistida por IA.

Capítulo 5

Enfoque Propuesto: Descubrimiento de Microservicios usando IA

Los capítulos anteriores han establecido el estado del arte (Capítulo 2) y los fundamentos teóricos (Capítulo 3) de esta investigación. El análisis de la literatura reveló un vacío metodológico central: la ausencia de *pipelines* que integren la generación de microservicios asistida por IA con la verificación formal de su calidad estructural.

Este capítulo presenta la propuesta de esta tesis para abordar dicho vacío. Primero, se detalla el marco metodológico, la *Design Science Research* (DSR), que guía el diseño y la validación del artefacto (Sección 5.1). A continuación, se describe la arquitectura y los componentes del artefacto central de esta tesis, **ArchiGenMS** (Sección 5.2 y subsiguientes).

5.1. Metodología de Investigación (Design Science Research)

La presente investigación doctoral adopta la metodología de **Investigación en Ciencia del Diseño** (Design Science Research, DSR) como marco estructurador. La DSR es un paradigma de investigación reconocido en el ámbito de los Sistemas de Información y las Ciencias de la Computación, cuyo objetivo principal no es meramente describir, explicar o predecir la realidad, sino crear y evaluar artefactos innovadores que resuelvan problemas prácticos y relevantes del mundo real [26, 27].

La elección de la DSR se justifica dado que la contribución central de esta tesis no es un hallazgo puramente descriptivo, sino un artefacto tangible y evaluable: **ArchiGenMS**. Este artefacto es un *pipeline* computacional diseñado con el propósito explícito de resolver el problema de la ambigüedad y la falta de rigor estructural en el descubrimiento de microservicios a partir de requisitos textuales en escenarios *greenfield*.

Siguiendo el modelo propuesto por Hevner et al. [26], esta tesis se articula en torno a los tres ciclos fundamentales de la DSR:

1. **El Ciclo de Relevancia (Relevance Cycle):** Este ciclo conecta el contexto del problema de investigación con la solución propuesta. La relevancia del problema —la dificultad de definir límites de servicio coherentes y desacoplados desde texto no estructurado— es un desafío práctico y persistente en la industria del software, como se ha documentado en la Sección

1.1 y 1.2. Este ciclo se cierra en el Capítulo 7, donde el artefacto (`ArchiGenMS`) se evalúa empíricamente contra dicho problema utilizando *datasets* representativos del dominio [33].

2. **El Ciclo de Rigor (Rigor Cycle):** Este ciclo asegura que la investigación se fundamenta sólidamente en la base de conocimiento existente, garantizando que el artefacto sea una contribución novedosa. El rigor de esta tesis se construye a través de:
   - Una fundamentación exhaustiva del estado del arte, materializada en el Mapeo Sistemático de la Literatura (MSL) [20] y la Revisión Sistemática de la Literatura (SLR) [18]. Estos estudios (analizados en el Capítulo 4) identificaron la brecha metodológica clave: la ausencia de enfoques que integren generación automática con verificación formal.
   - El establecimiento de los fundamentos teóricos (Capítulo 3), que anclan el artefacto en teorías consolidadas sobre métricas arquitectónicas [31, 32] y métodos formales [30].
3. **El Ciclo de Diseño (Design Cycle):** Este es el ciclo central de la tesis, donde el artefacto `ArchiGenMS` se construye y evalúa iterativamente. Este ciclo abarca múltiples capítulos:
   - La propuesta conceptual del artefacto, presentada y discutida en el simposio doctoral [21].
   - El diseño y la arquitectura del *pipeline* (Capítulos 5 y 6), donde se detalla la integración de LLMs, la ingeniería de *prompts* evolutiva y la crucial integración con el validador formal en Lean.
   - La evaluación empírica del artefacto (Capítulo 7), que valida su desempeño (convergencia) y la calidad estructural de las arquitecturas generadas, como se reporta en el trabajo de consolidación del artefacto [28].

5.2. Visión General del Enfoque

Siguiendo el marco metodológico de la *Design Science Research* (DSR) establecido en la Sección 5.1, este capítulo aborda el núcleo del **Ciclo de Diseño** (Design Cycle). Se presenta el artefacto propuesto, **ArchiGenMS**, detallando su arquitectura conceptual, los componentes de generación evolutiva y la integración de la verificación formal que buscan resolver el problema de relevancia identificado.

Los capítulos anteriores mostraron que, aunque los microservicios ofrecen ventajas significativas en escalabilidad y mantenibilidad, su diseño sigue siendo un problema abierto debido a la dificultad de identificar límites de servicio adecuados [56, 57, 1, 3]. Asimismo, se evidenció que los enfoques basados en inteligencia artificial generan arquitecturas plausibles a partir de requisitos textuales [11, 47, 12, 49, 22, 23], pero carecen de mecanismos de verificación formal. Por otro lado, los métodos formales y métricas arquitectónicas ofrecen garantías de corrección y calidad [2, 53, 54, 55, 31, 32], aunque rara vez se integran a procesos de descubrimiento de microservicios.

Esta necesidad de un enfoque híbrido se ve reforzada por hallazgos empíricos recientes. Como demuestran Pereira et al. [50], incluso los modelos de razonamiento más avanzados (p. ej., GPT-4 o3) sufren de lo que denominamos la 'ilusión de plausibilidad': generan descomposiciones funcionalmente coherentes pero estructuralmente deficientes, con un acoplamiento bidireccional que los expertos penalizan con puntuaciones de apenas 2/5. ArchiGenMS aborda esta deficiencia sustituyendo la validación manual por un oráculo formal.

El enfoque propuesto en esta tesis, denominado **ArchiGenMS**, busca cerrar esta brecha mediante un pipeline híbrido que combina:

1. **Modelos generativos de lenguaje (LLMs)** para derivar arquitecturas candidatas a partir de requisitos textuales (historias de usuario, casos de uso).
2. **Métricas arquitectónicas** que permiten evaluar cohesión, granularidad y acoplamiento de los servicios generados, garantizando criterios cuantitativos de calidad.
3. **Verificación formal en Lean 4** que asegura consistencia estructural y reproducibilidad de los resultados.

ArchiGenMS fue introducido inicialmente como una propuesta conceptual en el Simposio Doctoral de CIbSE 2025 [21], y posteriormente consolidado en su aplicación experimental en CACIC 2025 [28]. Su fundamento teórico se construyó a partir de un mapeo sistemático de literatura [20] y una revisión sistemática más exhaustiva [18], que identificaron la ausencia de metodologías integradoras capaces de unir generación automática y verificación formal en el diseño de microservicios.

La Figura 5.1 ilustra esta visión general del enfoque. La misma fue introducida originalmente en el Congreso CACIC 2025 como parte de la primera validación experimental de ArchiGenMS [28], y se reutiliza aquí como representación formal de la propuesta doctoral. En ella se observa cómo, a partir de requisitos textuales, el pipeline genera arquitecturas de microservicios mediante LLMs, las somete a evaluación métrica y verificación formal en Lean, y selecciona aquellas con mejor desempeño estructural.

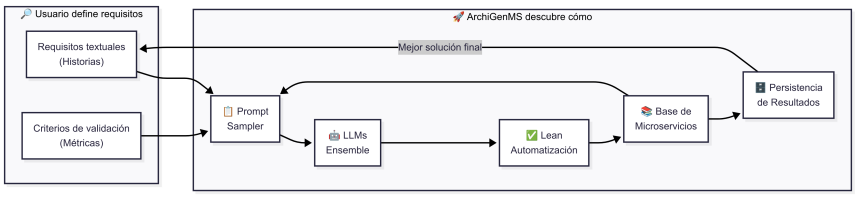


Figura 5.1: Visión general de ArchiGenMS. El pipeline integra generación evolutiva de microservicios mediante LLMs, verificación formal con Lean y retroalimentación basada en métricas arquitectónicas.

En las secciones siguientes se detallan cada uno de los componentes del enfoque: la extracción evolutiva de microservicios (Sección 5.3), la ingeniería de *prompts* (Sección 5.4), la evaluación con métricas arquitectónicas (Sección 3.6), la verificación formal en Lean (Sección 5.6) y el proceso evolutivo completo (Sección 5.7). Esta estructura refleja la transición desde los requisitos textuales hasta la obtención de arquitecturas de microservicios verificadas y evaluadas bajo criterios rigurosos.

5.3. Extracción Evolutiva de Microservicios

El punto de partida de ArchiGenMS son los requisitos textuales, típicamente representados como historias de usuario o casos de uso. Diversos trabajos han mostrado que este tipo de artefactos contienen información suficiente para identificar candidatos a microservicios, aunque su naturaleza ambigua y no estructurada dificulta una automatización completa [45, 46, 33].

Los primeros enfoques aplicaron técnicas de procesamiento de lenguaje natural y aprendizaje automático para abordar este problema. SEMGROMI, por ejemplo, utiliza agrupamiento semántico de historias de usuario basándose en similitud lingüística [11]; GreenMicro propone identificar microservicios en desarrollos desde cero (*greenfield*) a partir de casos de uso [12]; y GTMicro introduce un marco basado en grafos para descubrir microservicios desde relaciones de dependencia en requisitos [49]. Estos enfoques demostraron la factibilidad de partir de requisitos textuales, pero se mantienen fuertemente dependientes de heurísticas y carecen de mecanismos de validación estructural.

ArchiGenMS extiende esta línea de trabajo al **fundamentar el proceso de descubrimiento en la disciplina de la Ingeniería de Software Basada en Búsqueda (SBSE)** [35, 43]. Como se estableció en la Sección 4.2, el diseño arquitectónico es un problema de optimización combinatoria NP-hard [68], y los algoritmos evolutivos son el enfoque metaheurístico estándar para navegar el espacio de soluciones [71].

Siguiendo este paradigma, ArchiGenMS implementa un **esquema evolutivo asistido por LLMs**. Cada candidato arquitectónico es tratado como un **genotipo** (representado en JSON), compuesto por microservicios, operaciones y dependencias. El sistema aplica operadores de variación inspirados en los algoritmos evolutivos: la **selección** de los mejores individuos (elitismo) y la **variación** (análoga al cruce y la mutación [69]) guiada por el LLM. Esta combinación de IA generativa y búsqueda evolutiva se inspira en agentes de codificación recientes como AlphaEvolve [29], pero adapta el proceso al dominio arquitectónico. **A diferencia de los enfoques de generación estática (*zero-shot*) evaluados recientemente, donde se reportan niveles de acoplamiento bidireccional que comprometen la plausibilidad del diseño [50], la representación genotípica de ArchiGenMS permite que el LLM actúe como un motor de variación inteligente. En cada ciclo, el modelo no solo propone servicios, sino que refina la estructura basándose en las señales de acoplamiento y cohesión procesadas por el oráculo formal, evolucionando activamente hacia una arquitectura de mayor calidad técnica.**

El motor de esta búsqueda es la **función de aptitud (fitness function)**, que cuantifica la calidad de cada genotipo. De manera análoga a los enfoques MOO de vanguardia que optimizan cohesión y acoplamiento [72, 73], ArchiGenMS emplea un ciclo iterativo donde: (1) los LLMs *generan* candidatos, (2) las métricas arquitectónicas actúan como *función de aptitud*, y (3) el entorno formal en Lean *verifica* la validez estructural de cada iteración.

El proceso puede resumirse de la siguiente manera:

1. Se construye un *prompt* inicial a partir de los requisitos textuales, solicitando una descomposición en microservicios.

2. El LLM produce un candidato arquitectónico en un formato estructurado (JSON), que constituye un genotipo inicial.

3. Se generan variantes del candidato a través de inspiraciones y mutaciones inducidas por nuevos *prompts*.

4. Cada genotipo se evalúa mediante métricas arquitectónicas de cohesión, granularidad y acoplamiento.

5. Se seleccionan los candidatos con mejor desempeño para retroalimentar el ciclo y continuar la exploración.

La Figura 5.2 presenta el flujo evolutivo de ArchiGenMS. Se observa cómo, a partir de los requisitos textuales y la ingeniería de *prompts*, se deriva un genotipo inicial que luego se somete a procesos iterativos de variación, evaluación y selección. Las métricas arquitectónicas se calculan en Lean 4, lo que asegura consistencia formal y permite retroalimentar el ciclo evolutivo. El resultado final es un conjunto de candidatos optimizados que alcanzan un equilibrio entre cohesión, granularidad y acoplamiento.

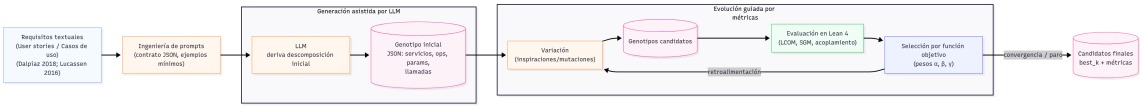


Figura 5.2: Extracción evolutiva de microservicios a partir de requisitos textuales. El proceso integra LLMs, ingeniería de *prompts*, validación en Lean y selección iterativa mediante métricas arquitectónicas.

Un aspecto clave de este enfoque es la **representación intermedia** de los candidatos, que actúa como puente entre la generación asistida por IA y la validación formal. La Figura 5.3 ilustra esta estructura genotípica: cada arquitectura se modela como un objeto compuesto por microservicios, operaciones, parámetros y dependencias, acompañado de metadatos sobre el *prompt*, el dataset y el modelo generativo empleado. Esta representación uniforme permite aplicar métricas cuantitativas y pruebas de consistencia estructural, garantizando comparabilidad entre iteraciones y reproducibilidad de los resultados.

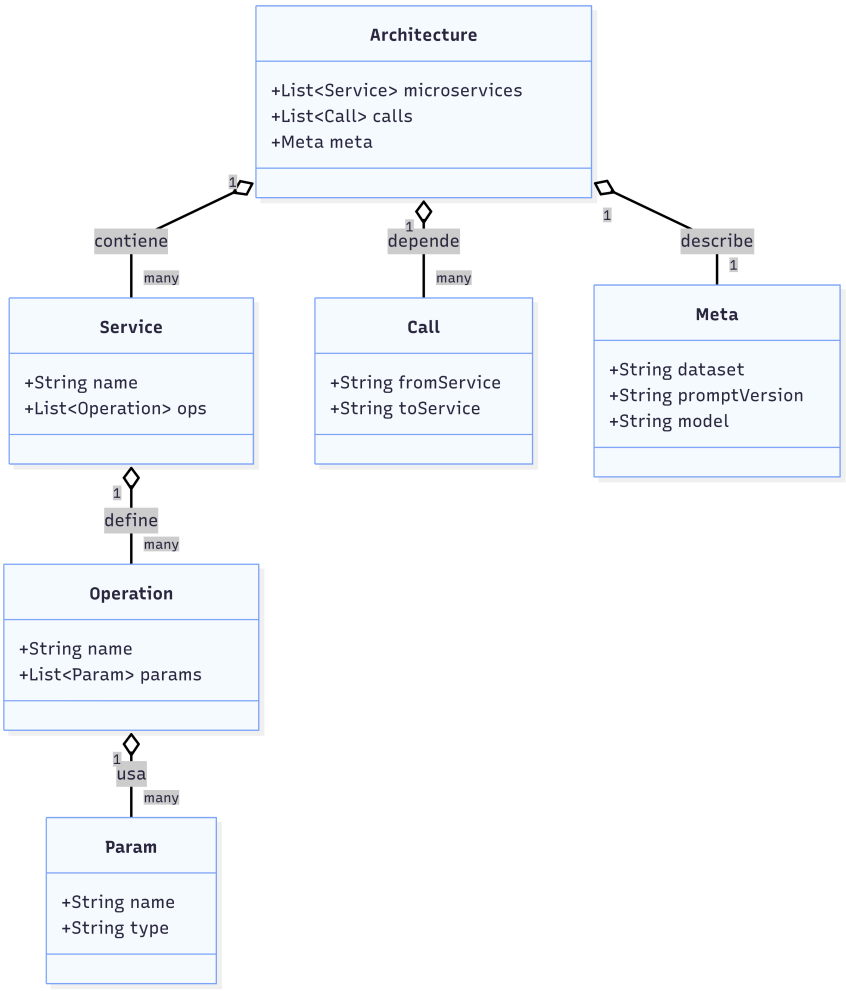


Figura 5.3: Estructura del genotipo evaluado en ArchiGenMS. La representación intermedia incluye microservicios, operaciones, parámetros, dependencias y metadatos, lo que permite su validación formal y análisis métrico.

Este procedimiento evolutivo permite no solo generar arquitecturas a partir de requisitos ambiguos, sino también explorar múltiples alternativas de diseño, lo que incrementa la diversidad de soluciones y abre la posibilidad de seleccionar arquitecturas más robustas. La combinación de generación, variación y selección guiada por métricas constituye el núcleo del proceso de extracción evolutiva en ArchiGenMS.

5.4. Ingeniería de Prompts y Modelos Generativos

La calidad de las arquitecturas candidatas generadas por modelos de lenguaje depende en gran medida de la **ingeniería de *prompts***, entendida como el diseño sistemático de instrucciones que orientan el comportamiento del modelo. En ArchiGenMS, los *prompts* no se formulan de manera libre, sino que establecen un contrato de salida explícito en formato JSON, compuesto por microservicios, operaciones, parámetros y dependencias. Esta restricción sintáctica permite reducir la ambigüedad en las respuestas y facilita su integración directa con el módulo de verificación formal en Lean.

La literatura reciente ha puesto de relieve tanto el potencial como las limitaciones de los LLMs en el diseño de microservicios. Adams et al. investigan el uso de ChatGPT para el desarrollo de microservicios, mostrando que puede asistir en la extracción de entidades y relaciones, aunque con problemas de consistencia y verificabilidad [22]. De manera similar, Stojanovic et al. analizan la identificación de microservicios con ChatGPT, concluyendo que la utilidad del enfoque depende de *prompts* cuidadosamente diseñados [23]. Más allá de aplicaciones exploratorias, PF4MD propone un marco para la descomposición en microservicios que combina *prompt engineering* con evaluación métrica, reforzando la necesidad de mecanismos de control en el uso de modelos generativos [51]. Finalmente, Esposito et al. presentan un estudio exhaustivo sobre la aplicación de IA generativa en arquitectura de software, destacando la reproducibilidad, la confianza y la verificabilidad como desafíos pendientes [19].

ArchiGenMS recoge estas lecciones y las amplía mediante un **esquema evolutivo de ingeniería de *prompts***. En cada ciclo, un *prompt* inicial derivado de los requisitos textuales genera un candidato arquitectónico. A partir de este genotipo inicial, se diseñan nuevas variantes de *prompts* que introducen modificaciones —por ejemplo, ajustes en la granularidad de servicios u operaciones—. Estas variantes generan arquitecturas alternativas que son evaluadas con métricas de cohesión, granularidad y acoplamiento, y posteriormente sometidas a verificación formal. La retroalimentación obtenida permite refinar los *prompts* y continuar explorando el espacio de soluciones de forma iterativa.

Este mecanismo se inspira en el paradigma de agentes generativos con retroalimentación evolutiva, como AlphaEvolve, un agente de codificación que combina generación automática con ciclos de validación y refinamiento en dominios científicos y algorítmicos [29]. De manera análoga, ArchiGenMS convierte los *prompts* en un instrumento dinámico para guiar la búsqueda arquitectónica, transformándolos en el eje de un proceso de optimización evolutiva.

La Figura 5.4 ilustra este proceso. A partir de los requisitos textuales, un *prompt* inicial produce un genotipo candidato; este se evalúa con métricas arquitectónicas y, según los resultados, se refina el *prompt* para generar nuevas variantes. El ciclo continúa hasta obtener candidatos que cumplen con los criterios de calidad establecidos.

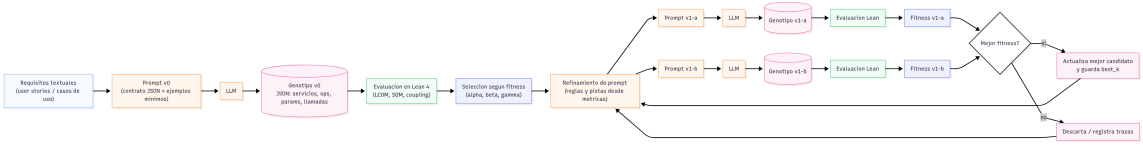


Figura 5.4: Esquema de ingeniería evolutiva de *prompts* en ArchiGenMS. Los requisitos textuales se transforman en un *prompt* inicial que genera un genotipo arquitectónico; los resultados métricos retroalimentan el proceso, guiando la creación de variantes y la selección de mejores candidatos.

En síntesis, la ingeniería de *prompts* en ArchiGenMS trasciende la mera formulación de instrucciones: constituye un mecanismo evolutivo que combina generación, variación y selección, con retroalimentación explícita basada en métricas y validación formal. Este diseño asegura que los candidatos arquitectónicos no solo sean plausibles desde el punto de vista semántico, sino también robustos y consistentes desde la perspectiva estructural.

5.5. Evaluación con Métricas Arquitectónicas

En este apartado se detalla cómo las métricas de calidad arquitectónica se aplican operativamente dentro de ArchiGenMS: a partir del artefacto JSON producido por el LLM (servicios, operaciones, parámetros y llamadas), se computan los indicadores por servicio y sus agregaciones a nivel de sistema que alimentan la función objetivo del ciclo evolutivo.

Fórmulas de referencia

Formalmente, las métricas se definen del siguiente modo:

Lack of Cohesion of Methods (LCOM).

$$\mathrm{LCOM}(s) = \frac{|P_{ns}|}{|P_{ns}| + |P_s|}$$

donde P_{ns} es el conjunto de pares de operaciones en s que no comparten parámetros, y P_s es el conjunto de pares que sí comparten al menos un parámetro [12]. Valores cercanos a 0 reflejan alta cohesión, mientras que valores cercanos a 1 indican baja cohesión.

Service Granularity Metric (SGM).

$$\mathrm{SGM}(s) = |O_s|$$

donde $|O_s|$ es el número de operaciones del servicio. Valores entre 2 y 5 suelen asociarse con una granularidad apropiada [12, 74].

Desviación estándar del tamaño granular (SGM-SD).

$$\mathrm{SGM}_{SD}(s) = \sqrt{\frac{\sum_{i=1}^{n}(|\mathrm{params}(o_i)| - \overline{|\mathrm{params}|})^2}{n}}$$

donde $|\mathrm{params}(o_i)|$ es la cantidad de parámetros de la operación o_i, y $\overline{|\mathrm{params}|}$ el promedio de parámetros por operación en s [12, 74].

Coupling (Fan-Out).

$$\mathrm{Coupling}(s) = |\{t \mid \exists \text{ una llamada desde } s \text{ hacia } t,\ t \neq s\}|$$

que corresponde al número de microservicios externos invocados por s [74, 31].

De JSON a métricas

El modelo generativo produce candidatos arquitectónicos en un formato JSON estandarizado, como se ilustra en el Listado 5.1. Cada genotipo incluye: (i) la lista de microservicios (`"microservices"`), (ii) operaciones (`.ºps"`) con sus parámetros (`"params"`), y (iii) dependencias de llamadas (`çalls"`). A partir de este objeto se derivan las métricas para cada servicio y, posteriormente, se calculan valores agregados a nivel de sistema.

```
{
  "microservices": [
    {"name": "Order", "ops": [
      {"name": "createOrder", "params": ["userId","items"]},
      {"name": "getOrder",    "params": ["orderId"]}
    ]},
    {"name": "Payment", "ops": [
      {"name": "charge", "params": ["orderId","amount"]}
    ]}
  ],
  "calls": [
    {"caller": "Order", "callee": "Payment"}
  ]
}
```

Listado 5.1: Ejemplo simplificado de salida JSON de un candidato arquitectónico.

A partir de este artefacto (Listado 5.1), el proceso de evaluación que ejecuta el validador Lean procede de la siguiente manera:

Primero, se analiza el servicio `.ºrder"` (línea 3). Se identifica que posee dos operaciones (líneas 4-5), por lo tanto, SGM($Order$) = 2. Para la cohesión, se computa el único par de operaciones posible: `(createOrder, getOrder)`. Al inspeccionar sus respectivas listas `"params"`, el validador determina que no comparten ningún elemento (el conjunto `{üserId", ïtems"}` y el conjunto `{.ºrderId"}` son disjuntos). En consecuencia, se tiene $|P_s| = 0$ y $|P_{ns}| = 1$. Aplicando la convención computable (Sección 3.6), al ser $|P_s| = 0$, el servicio es trivialmente no cohesivo, resultando en LCOM($Order$) = 1,0. Para la dispersión granular, se calculan las aridades (cardinalidad de parámetros): $|\{"userId","items"\}| = 2$ y $|\{"orderId"\}| = 1$. Sobre el vector $[2, 1]$, la media es 1,5 y la varianza poblacional es 0,25, resultando en $\text{SGM}_{SD}(Order) = \sqrt{0{,}25} = 0{,}5$.

Segundo, se analiza el servicio `"Payment"` (línea 7). Posee una sola operación (`çharge"`), por lo que SGM($Payment$) = 1. Según las convenciones (Sección 3.6), al tener menos de dos operaciones, no existen pares para comparar, resultando en LCOM($Payment$) = 0,0 y $\text{SGM}_{SD}(Payment) = 0{,}0$.

Tercero, se analiza el acoplamiento a partir de la lista `çalls"` (línea 11). Se identifica una única llamada donde `çaller"` es `.ºrder"` y `çallee"` es `"Payment"`. Por lo tanto, el grafo de dependencias asigna Coupling($Order$) = 1 y Coupling($Payment$) = 0.

Agregación a nivel sistema

A nivel de sistema, ArchiGenMS reporta las métricas agregadas que guían el proceso evolutivo:

$$\texttt{lcom_avg} = \frac{1}{|\mathcal{S}|}\sum_{s\in\mathcal{S}} LCOM(s), \quad \texttt{sgm_max} = \max_{s\in\mathcal{S}} SGM(s),$$

$$\texttt{sgm_sd_sum} = \sum_{s \in \mathcal{S}} SGM_{SD}(s), \quad \texttt{coupling_max} = \max_{s \in \mathcal{S}} Coupling(s).$$

En el ejemplo de la Figura 5.1, los valores serían: `lcom_avg` = 0, `sgm_max` = 2, `sgm_sd_sum` = 0,5, `coupling_max` = 1.

Normalización y robustez

Para garantizar comparabilidad entre iteraciones y distintos *datasets*, ArchiGenMS implementa convenciones adicionales:

1. $LCOM(s) = 0$ cuando $|Ops(s)| < 2$, al no existir pares comparables.
2. Se eliminan auto-llamadas $s \to s$ y duplicados en $\mathcal{C}$.
3. Los valores en coma flotante se redondean a tres decimales, favoreciendo estabilidad numérica en la comparación entre generaciones.

Reproducibilidad

El proceso de cálculo se encuentra implementado en el paquete reproducible ArchiGenMS [64], que asegura la disponibilidad del código, datos y scripts para ejecutar los experimentos y verificar los resultados. Cada evaluación produce un objeto JSON con las métricas agregadas y metadatos del candidato (dataset, versión del *prompt*, modelo), lo que permite trazabilidad y análisis estadístico (véase Cap. 7).

5.6. Validación Formal en Lean

La incorporación de Lean 4 como mecanismo de verificación formal no es solo una elección tecnológica, sino una respuesta al problema de la fragilidad estructural detectada en modelos generativos de última generación. Mientras que la literatura reciente confirma que los LLMs pueden identificar servicios con precisión, fallan consistentemente en garantizar la integridad relacional y el bajo acoplamiento [50]. Lean 4 actúa en ArchiGenMS como el filtro de rigor que transforma una propuesta lingüísticamente plausible en una arquitectura matemáticamente consistente.

Una contribución central de ArchiGenMS es la incorporación del asistente de pruebas Lean 4 como mecanismo de verificación formal de los candidatos arquitectónicos generados en el ciclo evolutivo. A diferencia de la simple aplicación de métricas en un lenguaje de propósito general, Lean permite formalizar las definiciones matemáticas y garantizar que cada evaluación preserve corrección y consistencia, eliminando ambigüedades que suelen estar presentes en implementaciones ad hoc [30].

El rol de Lean en ArchiGenMS es doble. Primero, se utiliza para definir de manera formal las métricas arquitectónicas presentadas en la Sección 3.6, asegurando que sus propiedades estén respaldadas por una semántica lógica precisa. Segundo, actúa como un validador automático de genotipos, descartando arquitecturas inválidas (por ejemplo, auto-llamadas o dependencias hacia servicios inexistentes) antes de ser consideradas en el proceso evolutivo. Esto asegura que la población de candidatos se mantenga dentro de un espacio de búsqueda consistente, incluso frente a la naturaleza inherentemente ruidosa de las salidas de modelos generativos.

Integración técnica

La interacción entre Python y Lean se realiza a través de la herramienta de construcción `lake`, mediante el comando:

```
lake exe validate <ruta_al_genotipo.json>
```

Cada ejecución recibe como entrada un JSON con la definición de microservicios, operaciones y llamadas (cf. Listado 5.1), y devuelve un objeto JSON que contiene el estado de validación (`OK`/`ERROR`), un mensaje asociado y, en caso de éxito, las métricas agregadas calculadas. En los experimentos reportados, la latencia promedio de validación por individuo fue de aproximadamente tres segundos, lo que resulta aceptable dentro del ciclo evolutivo.

Definiciones de servicios y métricas

El siguiente fragmento muestra la definición de las estructuras básicas y el cálculo de la métrica *Lack of Cohesion of Methods (LCOM)* en Lean 4:

```
structure Operation where
  name   : String
  params : List String
  deriving Repr, BEq

structure Service where
  name : String
  ops  : List Operation
  deriving Repr, BEq

def LCOM (s : Service) : Float :=
  let ops := s.ops
  if ops.length < 2 then 0.0 else
  let listOfLists := ops.map (fun o₁ => ops.map (fun o₂ => (o₁, o₂)))
  let allPairs := listOfLists.foldr List.append []
  let pairs := allPairs.filter (fun (o₁, o₂) => o₁.name < o₂.name)
  let (sharing, notSharing) := pairs.partition
    (fun (o₁, o₂) => o₁.params.any (fun p => o₂.params.contains p))
  let p := Float.ofNat notSharing.length
  let q := Float.ofNat sharing.length
  if p == 0.0 || q == 0.0 then 0.0 else p / (p + q)
```

Listado 5.2: Definición de estructuras y cálculo de LCOM en Lean 4.

De manera análoga se implementan las métricas de granularidad de servicios (`SGM`), desviación estándar granular (`sgmSd`) y acoplamiento (`couplingOut`), manteniendo consistencia con las fórmulas presentadas en la Sección 3.6 y con lo reportado en CACIC 2025 [28].

Validación de genotipos

Además del cálculo de métricas, Lean permite imponer restricciones estructurales duras sobre los candidatos arquitectónicos. El Listado 5.3 muestra la función `validateGenotype`, que descarta genotipos con auto-llamadas o con dependencias hacia servicios inexistentes:

```
def validateGenotype (gt : Genotype) : Except String Unit := do
  let names := gt.microservices.map (·.name)
  for c in gt.calls do
    if c.caller == c.callee then
      throw s!"Self-call: {c.caller}"
    unless names.contains c.caller do
      throw s!"Unknown caller: {c.caller}"
    unless names.contains c.callee do
      throw s!"Unknown callee: {c.callee}"
  pure ()
```

Listado 5.3: Validación de genotipos en Lean 4.

Estas validaciones eliminan candidatos inconsistentes antes de aplicar la función objetivo basada en métricas. En conjunto, las métricas formales y las restricciones estructurales se integran en el ciclo evolutivo como una forma de *fitness function* con verificaciones duras y blandas.

Reproducibilidad

El código completo, incluyendo la implementación de métricas, validación y punto de entrada, se encuentra disponible en el paquete reproducible *ArchiGenMS* [64]. Este paquete permite a otros investigadores replicar los resultados reportados, extender las métricas o adaptar el esquema de validación a diferentes dominios de aplicación, consolidando la transparencia y la trazabilidad de la propuesta.

Discusión

La elección de Lean como motor de verificación no es incidental. A diferencia de enfoques centrados exclusivamente en el cálculo numérico de métricas (e.g., [12, 74, 31]), aquí se combina generación automática, validación formal y retroalimentación evolutiva. Esto coloca a ArchiGenMS en una posición distintiva respecto de la literatura previa, al integrar *formal methods* [2, 55, 54] dentro de un pipeline de descubrimiento arquitectónico asistido por IA generativa.

Es imperativo destacar que el rol de Lean 4 trasciende el cálculo numérico. El estudio de Pereira et al. [50] sugiere que el *few-shot prompting* manual mejora la calidad percibida; ArchiGenMS evoluciona este concepto al utilizar el asistente de pruebas para automatizar dicha mejora. Esto garantiza por construcción lo que otros enfoques solo intentan inducir mediante ejemplares estáticos, eliminando la subjetividad del juicio humano en el ciclo de diseño.

5.7. Integración del Pipeline Completo

La propuesta ArchiGenMS se fundamenta en la integración coordinada de los diferentes componentes presentados en las secciones anteriores: extracción evolutiva de microservicios, ingeniería de *prompts*, métricas arquitectónicas y verificación formal en Lean. Mientras que cada uno de estos elementos tiene valor individual, su verdadero aporte surge cuando se articulan en un **pipeline completo y reproducible** que asegura tanto la generación como la validación de arquitecturas candidatas.

El proceso comienza con los **requisitos textuales**, representados típicamente como historias de usuario o casos de uso [46, 45, 33]. A partir de ellos, el *Prompt Sampler* construye una solicitud inicial que es procesada por los modelos de lenguaje generativo (LLMs), produciendo una primera descomposición en microservicios en formato JSON. Este artefacto se interpreta como un **genotipo arquitectónico**, conformado por servicios, operaciones y dependencias.

El **módulo evolutivo** aplica operadores de inspiración, mutación y selección, siguiendo esquemas $(\mu + \lambda)$ inspirados en computación evolutiva, similares a los empleados en AlphaEvolve [29], pero orientados al dominio arquitectónico. Cada iteración genera nuevas variantes de microservicios que son evaluadas por medio de métricas estructurales (LCOM, SGM, SGM-SD y Coupling) [12, 74, 31]. La función de aptitud guía la selección de los mejores candidatos, mientras que los no válidos reciben penalizaciones.

En este punto, el rol de la **verificación formal en Lean 4** es central: toda arquitectura candidata se valida automáticamente mediante el módulo `ValidateGenotype.lean`, que comprueba la ausencia de referencias inconsistentes, calcula las métricas y garantiza que los resultados cumplen las restricciones definidas. Esto otorga un nivel de rigor inexistente en enfoques previos basados solo en heurísticas o aprendizaje automático [11, 47, 23, 22].

Finalmente, el pipeline implementa mecanismos de **persistencia de resultados** y **reproducibilidad**. Cada ciclo almacena tanto los individuos intermedios como el mejor candidato, registrando métricas, estado y tiempos de ejecución. De este modo, ArchiGenMS se alinea con buenas prácticas de investigación reproducible y con las recomendaciones metodológicas previas [20, 18, 21]. El paquete abierto disponible en GitHub [64] garantiza que cualquier investigador pueda replicar los experimentos y extender el enfoque.

La Figura 5.5 resume esta integración, mostrando cómo los artefactos de entrada se transforman en arquitecturas verificadas de microservicios a través de un ciclo iterativo y formalmente validado.

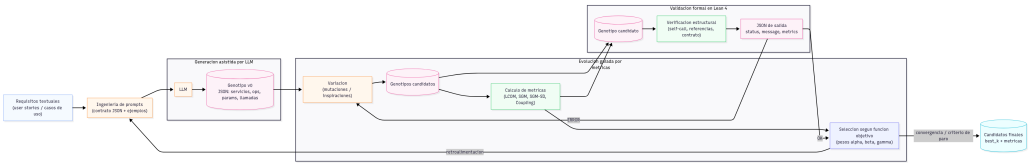


Figura 5.5: Integración del pipeline ArchiGenMS. Los requisitos textuales se transforman en arquitecturas de microservicios mediante un ciclo evolutivo de generación, evaluación y verificación formal.

Finalmente, la integración del pipeline completo convierte a ArchiGenMS en un artefacto capaz de combinar generación automática, retroalimentación métrica y garantías formales, proveyendo evidencia sólida sobre la factibilidad de automatizar el descubrimiento y diseño de microservicios desde requisitos textuales. Este marco constituye la base para los experimentos presentados en el Capítulo 7.

5.7.1. Ubicación de la propuesta dentro de AI4SE

El pipeline implementa una instancia de *AI4SE para diseño*: (1) *asistencia generativa* (LLMs) para derivar candidatos arquitectónicos; (2) *validación computable* (Lean 4) de métricas estructurales (LCOM, SGM, *fan-out*); y (3) *selección* por una función de aptitud escalar simple. Así, el

uso de IA no reemplaza la verificación, sino que la *alimenta* con propuestas diversas sometidas a criterios objetivos y reproducibles. Este enfoque difiere de ramas de AI4SE centradas en pruebas y mantenimiento [41, 40] y aproxima la línea de automatización de diseño asistida por modelos [42].

5.8. Síntesis del Enfoque Propuesto

Este capítulo ha detallado el artefacto central de esta tesis, **ArchiGenMS**, constituyendo el núcleo del **Ciclo de Diseño (Design Cycle)** de la metodología DSR, tal como se definió en la Sección 5.1. Se ha presentado un *pipeline* integral que responde directamente a los vacíos metodológicos identificados en el estado del arte (Capítulo 2).

El enfoque propuesto articula tres contribuciones clave en un flujo unificado, cada una fundamentada en las secciones precedentes:

1. **Generación Evolutiva (Secciones 5.3 y 5.4):** Se utiliza un esquema de IA generativa (LLMs) no solo para la generación inicial, sino como un motor de variación semántica dentro de un bucle evolutivo [22, 23].

2. **Evaluación Cuantitativa (Sección 5.5):** Se integra un oráculo métrico (LCOM, SGM, Coupling) como *fitness function*, asegurando que la búsqueda evolutiva se guíe por principios de calidad arquitectónica (alta cohesión, bajo acoplamiento) y no solo por heurísticas de similitud [11].

3. **Validación Formal (Sección 5.6):** Se introduce un validador en Lean 4 como un filtro de consistencia *en bucle*, que rechaza artefactos estructuralmente inválidos (p.ej., auto-llamadas) generados por el LLM.

Esta combinación de generación automática, retroalimentación métrica y garantías formales (detallada en la Sección 5.7) diferencia a ArchiGenMS de la literatura previa. Al proponer un *pipeline* reproducible y verificable [64], se establece el "planoçonceptual del artefacto. Los capítulos subsecuentes se dedicarán a describir su materialización técnica (Capítulo 6) y su validación empírica (Capítulo 7).

Capítulo 6

Arquitectura y Herramientas de Soporte

Habiendo definido el enfoque metodológico y el *pipeline* conceptual de ArchiGenMS en el Capítulo 5 (Enfoque Propuesto: Descubrimiento de Microservicios usando IA), esta sección aborda su materialización técnica. El presente capítulo describe la arquitectura de software del artefacto, detallando los componentes implementados, el entorno de ejecución y los mecanismos de orquestación que dan soporte al ciclo "generar–verificar". El objetivo es proveer una visión clara de la infraestructura que permite la integración de los modelos generativos (LLMs), la gestión de los *datasets* de requisitos y el motor de verificación formal en Lean 4, asegurando la reproducibilidad de la propuesta.

6.1. Arquitectura General del Sistema

La arquitectura de ArchiGenMS se diseñó con el objetivo de integrar de forma coherente los distintos componentes necesarios para el descubrimiento y validación de microservicios. Como se discutió en el Capítulo 5, el *pipeline* combina modelos generativos de lenguaje, métricas arquitectónicas y verificación formal para derivar arquitecturas reproducibles a partir de requisitos textuales ambiguos.

En la Figura 6.1 se muestra una visión general del sistema, donde se observan las tres capas fundamentales: (i) entrada de requisitos, (ii) motor de generación y evaluación, y (iii) persistencia y análisis de resultados. Esta representación enfatiza la relación entre los insumos textuales y los artefactos arquitectónicos que resultan del proceso iterativo.

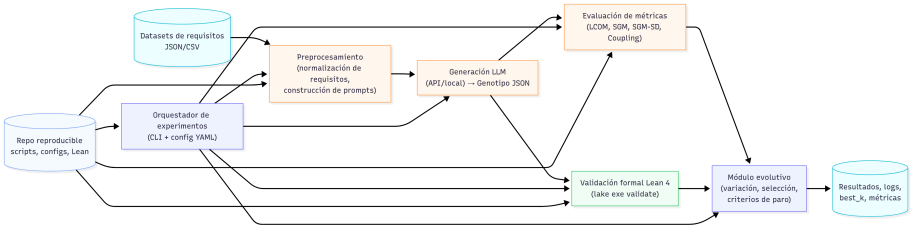


Figura 6.1: Arquitectura general de ArchiGenMS en alto nivel.

Complementariamente, la Figura 6.2 presenta una visión por capas, donde se distinguen los distintos módulos internos del *pipeline*. Este enfoque jerárquico facilita entender cómo se organiza el flujo de datos desde los requisitos textuales hasta la obtención de arquitecturas verificadas y métricamente evaluadas.

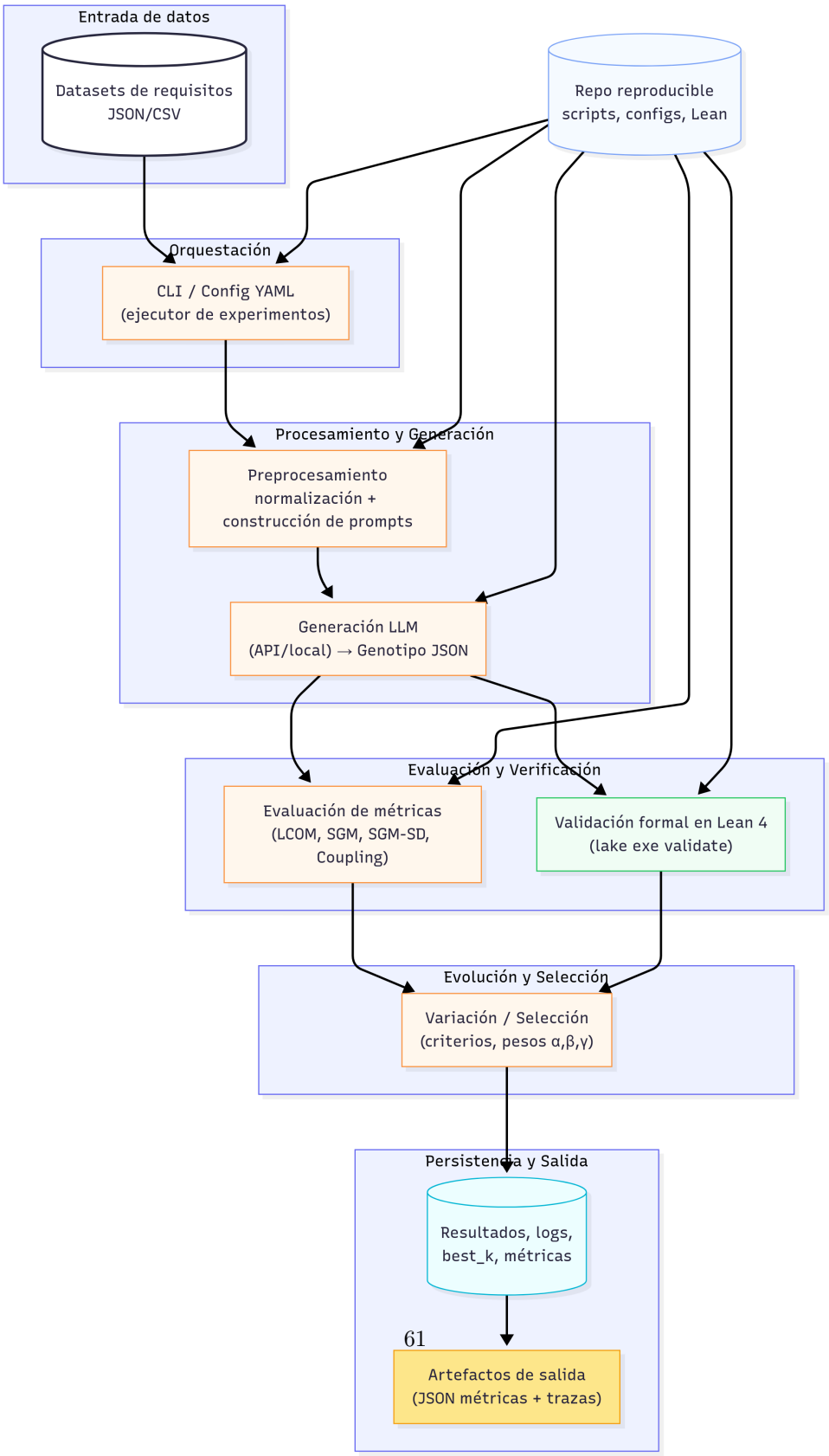


Figura 6.2: Arquitectura de ArchiGenMS organizada por capas y módulos funcionales.

Para mayor claridad, en la Tabla 6.1 se resumen las capas y módulos que conforman ArchiGenMS, junto con sus responsabilidades principales y las tecnologías empleadas en su implementación. Este desglose refuerza el principio de separación de preocupaciones y facilita la replicabilidad del enfoque.

Cuadro 6.1: Resumen de capas y módulos de ArchiGenMS

Capa/Módulo	**Descripción / Responsabilidad**	**Tecnologías / Herramientas**
Entrada de datos	Requisitos textuales en forma de historias de usuario y casos de uso, extraídos de *datasets* públicos	CSV/JSON (Dalpiaz 2018), Python parsers
Orquestación	Control de experimentos, configuración reproducible, ejecución del *pipeline*	CLI, Configuración YAML, Python 3.11
Preprocesamiento	Normalización de requisitos y construcción de *prompts* estructurados	Python (NLTK, spaCy), reglas JSON
Generación LLM	Derivación de genotipos iniciales de microservicios en formato JSON	GPT-4o-mini vía API/OpenAI
Evaluación de métricas	Cálculo de LCOM, SGM, SGM-SD y *fan-out* para cada candidato	Python + Lean 4 (`ServiceMetrics.lean`)
Verificación formal	Validación estructural y detección de inconsistencias en microservicios	Lean 4.22.0-rc3; `lake exe validate`
Evolución	Variación y selección guiada por función de aptitud, usando pesos α, β, γ	Algoritmos evolutivos ($\mu+\lambda$), Python
Persistencia de resultados	Registro de todos los individuos, métricas y arquitecturas óptimas	JSONL (`program_db.jsonl`), archivos `best_*.json`

6.2. Entorno de Ejecución y Replicabilidad

Un aspecto central de ArchiGenMS es su énfasis en la reproducibilidad de los experimentos. La integración de inteligencia artificial generativa con verificación formal demanda no solo una arquitectura conceptual clara, sino también un entorno de ejecución estandarizado que permita replicar resultados en diferentes contextos académicos e industriales.

El entorno fue diseñado con un enfoque modular y reproducible, siguiendo prácticas de ciencia abierta. En particular, todos los experimentos fueron encapsulados en un repositorio público [64], que contiene:

- **Código fuente** de los módulos principales (preprocesamiento, generación, evolución, métricas, validación formal).
- **Datasets de entrada**, incluyendo requisitos textuales como historias de usuario y casos de uso de referencia [33].
- **Scripts de orquestación** en Python para ejecutar el *pipeline* completo con configuración reproducible.
- **Artefactos Lean 4** que implementan la verificación formal de las arquitecturas generadas, siguiendo la sintaxis introducida en [30].

Cuadro 6.2: Configuración de hardware y software utilizada

Componente	**Especificación**
CPU	Intel Core i7-7500U @ 2.70–3.50 GHz (2C/4T)
Memoria	12 GB DDR4
GPU dedicada	AMD Radeon R7 M340 (2 GB)
GPU integrada	Intel HD Graphics 620 (128 MB)
Sistema operativo	Windows 10 Pro x64
Python	3.11.5
Lean 4	v4.22.0-rc3 (Lake)

- **Resultados experimentales**, almacenados en formato JSONL para permitir trazabilidad de cada ejecución.

El sistema fue desplegado en un entorno controlado de Python 3.11, con dependencias fijadas en un archivo `requirements.txt`, y acompañado de un archivo de configuración en YAML que especifica parámetros de los experimentos (tamaño de la población evolutiva, pesos de métricas α, β, γ, número de iteraciones, etc.).

La verificación formal se ejecuta mediante el compilador de Lean 4.22.0-rc3, usando el gestor de paquetes `lake` para compilar y ejecutar los módulos de validación (`lake exe validate`). Este diseño asegura que la validación de arquitecturas no dependa de entornos ad hoc, sino de una herramienta formal de propósito general, lo que incrementa la confiabilidad de los resultados.

La replicabilidad se sustenta en tres pilares:

1. **Disponibilidad de código y datos:** todo el material está publicado en un repositorio abierto.
2. **Determinismo controlado:** aunque los modelos generativos incorporan estocasticidad, los experimentos pueden replicarse fijando semillas aleatorias y registrando configuraciones exactas.
3. **Trazabilidad:** cada ejecución guarda sus resultados (artefactos JSON, métricas, *logs*) en un formato estandarizado, lo que facilita comparaciones entre ejecuciones.

Entorno de ejecución Los experimentos se corrieron en un equipo de escritorio con Windows 10 x64, Python 3.11.5 y Lean 4.22.0-rc3. La Tabla 6.2 resume la configuración utilizada. Esta información complementa la disponibilidad de código y datos, y facilita la replicación independiente.

Checklist de replicación mínima Para reproducir una ejecución de ArchiGenMS, recomendamos:

- Clonar el repositorio y fijar el intérprete a **Python 3.11.5**.
- Instalar dependencias con `pip install -r requirements.txt`.
- Preparar `.env` con las credenciales necesarias (API del LLM).
- Instalar **Lean 4.22.0-rc3** y `lake`; compilar el proyecto formal.
- Ejecutar el flujo orquestado (script Python) sobre el *dataset* elegido.
- Verificar que cada candidato pase por `lake exe validate` y que se genere el JSON de métricas {`lcom_avg`, `sgm_max`, `sgm_sd_sum`, `coupling_max`}.

Ejemplo de invocación del validador El verificador formal recibe un genotipo en JSON y retorna estado y métricas. A modo ilustrativo:

```
# Desde el directorio del proyecto Lean (formal/)
lake exe validate "experiments/best_CulRepo.json"
# Salida esperada (formato):
# {
#   "status": "OK",
#   "message": "",
#   "metrics": {
#     "lcom_avg": 0.333,
#     "sgm_max": 2.0,
#     "sgm_sd_sum": 0.707,
#     "coupling_max": 1
#   }
# }
```

Listado 6.1: Ejemplo de invocación del validador Lean.

En conjunto, estas características aseguran que ArchiGenMS pueda ser replicado por otros investigadores, siguiendo los principios de transparencia y verificación independiente que la comunidad científica demanda para investigaciones basadas en inteligencia artificial y métodos formales.

6.3. Integración con Lean 4 y Verificación Formal

Papel de Lean en el bucle generar–verificar. Cada arquitectura candidata se somete a verificación formal en *Lean 4* [30] para comprobar invariantes estructurales (p. ej., ausencia de *servicios dios*, separación de subdominios, conectividad acotada) definidas sobre el modelo abstracto de servicios y relaciones. La verificación no opera como validación tardía, sino como *oráculo* que guía la evolución descartando configuraciones plausibles pero estructuralmente deficientes, y suministra las métricas que alimentan la función de *fitness* (§ 6.4.1). Las definiciones y demostraciones formales se documentan en el *Anexo B*.

La validación formal constituye uno de los pilares distintivos de ArchiGenMS. Mientras que la mayoría de los enfoques basados en inteligencia artificial para el diseño de microservicios se limitan a la generación de candidatos arquitectónicos [11, 12, 23], nuestro enfoque incorpora un proceso explícito de verificación con Lean 4, lo que asegura que las arquitecturas propuestas cumplan propiedades estructurales fundamentales antes de avanzar en el ciclo evolutivo.

Lean 4 se ha consolidado como un asistente de demostración moderno y eficiente, diseñado tanto para la verificación matemática como para la programación funcional [30]. En ArchiGenMS se utiliza como componente externo que recibe artefactos arquitectónicos en formato JSON, generados a partir de la salida del modelo de lenguaje, y ejecuta un conjunto de validaciones computables. Estos artefactos se transforman en estructuras de datos internas (tipos `Service`, `Operation` y `Call`) sobre las cuales se calculan métricas de calidad y se verifican restricciones básicas de consistencia, tales como:

- inexistencia de llamadas recursivas directas (`self-call`);
- referencia únicamente a servicios declarados en el modelo;
- ausencia de dependencias cíclicas no deseadas.

El flujo de integración sigue un esquema reproducible: (i) se genera un archivo JSON que describe un genotipo candidato, (ii) este archivo se entrega al validador de Lean mediante `lake exe validate`, (iii) el verificador produce un objeto JSON de salida que contiene el estado de la validación y los valores de las métricas calculadas. En caso de error, el ciclo evolutivo penaliza dicho candidato, garantizando que solo progresen arquitecturas estructuralmente correctas.

A continuación, se muestra un fragmento representativo del reporte generado por Lean a partir de un candidato válido:

```
{
  "status": "OK",
  "message": "",
  "metrics": {
    "lcom_avg": 0.25,
    "sgm_max": 4.0,
    "sgm_sd_sum": 1.732,
    "coupling_max": 2
  }
}
```

Listado 6.2: Ejemplo simplificado de salida JSON del validador Lean

Este reporte es consumido por el módulo evolutivo en Python, que lo utiliza para alimentar la función de aptitud y decidir la supervivencia de cada genotipo. De este modo, Lean no solo valida la corrección de los artefactos, sino que también proporciona métricas cuantitativas que guían el proceso evolutivo.

La integración con Lean 4 refuerza así el carácter híbrido de ArchiGenMS: combina la creatividad generativa de los modelos de lenguaje con la precisión y confiabilidad de los métodos formales. Este enfoque permite reducir la brecha entre la exploración heurística y la garantía de calidad, aportando un nivel de verificación poco común en la literatura sobre microservicios [22, 19, 28].

6.4. Módulo de Métricas y Evaluación Automática

Este módulo representa el componente central del "verificador"dentro del pipeline "generar-verificar"de ArchiGenMS. Mientras que en el Capítulo 5 se describió el flujo conceptual, esta sección detalla la implementación práctica del oráculo que traduce los artefactos generados por el LLM (genotipos `JSON`) en un juicio cuantitativo, formal y reproducible.

Este módulo es, por tanto, el motor de la evaluación automática. Su función es implementar las métricas estructurales (LCOM, SGM, Coupling) que, como se discutió en la Sección 3.6, actúan como proxies cuantitativos para prevenir los *olores arquitectónicos* (*architectural smells*) documentados en la literatura [14]. La salida de este módulo alimenta directamente la función de aptitud del ciclo evolutivo (Sección 6.4.1), permitiendo que la búsqueda converja hacia soluciones estructuralmente robustas.

6.4.1. Criterio de *fitness* y métricas

Resumen operativo. La *cohesión* se estima mediante **LCOM** sobre las responsabilidades de cada servicio; una menor LCOM indica mayor cohesión [31]. La *granularidad* se cuantifica con **SGM** y su variante **SGM-SD** para captar tamaño y dispersión funcional, evitando sobredescomposición

o macroservicios [32]. El *acoplamiento* se aproxima por el ***fan-out*** inter-servicio derivado del grafo de dependencias; valores menores reducen complejidad de coordinación [31]. Estas definiciones sustentan la función de *fitness* del *pipeline* (con pesos α, β, γ) y sus umbrales de aceptación, y se formalizan en el *Anexo B*.

El cálculo de métricas arquitectónicas constituye un componente esencial para la evaluación de cada candidato generado en ArchiGenMS. Mientras que en el Capítulo 3.6 se presentaron las definiciones formales de las métricas y en el Capítulo 5.3 se describió su uso en el ciclo evolutivo, en esta sección se detalla su implementación práctica y la forma en que los valores se derivan automáticamente de los artefactos producidos por el modelo generativo.

Cada genotipo candidato se representa como un objeto `JSON` que contiene la lista de microservicios, sus operaciones y las llamadas entre ellos. A partir de esta estructura, el módulo de métricas implementado en `Lean 4` y `Python` realiza los siguientes cálculos de manera automática:

- **LCOM**: promedio de cohesión entre operaciones de un mismo servicio, detectando redundancias y fragmentación funcional [12].
- **SGM**: número de operaciones por microservicio, indicador directo de granularidad.
- **SGM-SD**: desviación estándar de los parámetros de las operaciones, midiendo consistencia interna [74].
- **Coupling (Fan-Out)**: número de dependencias salientes hacia otros microservicios, indicador de acoplamiento externo [31].

Para garantizar la trazabilidad, los valores se normalizan y empaquetan junto con el estado de validación formal en un único reporte JSON. Este diseño asegura que la evaluación sea completamente automatizada y que cada ejecución pueda registrarse en un formato reproducible y comparable entre experimentos.

A continuación, se muestra un ejemplo real tomado de los experimentos de ArchiGenMS sobre el *dataset CulRepo*. El Listado 6.3 presenta un fragmento del genotipo de entrada generado por el LLM:

```
{
  "microservices": [
    { "name": "UserService", "ops": [
        { "name": "createUser", "params": ["username","email","password"]
          },
        { "name": "deleteUser", "params": ["userId"] }
    ]},
    { "name": "RepoService", "ops": [
        { "name": "createRepo", "params": ["userId","repoName"] },
        { "name": "deleteRepo", "params": ["repoId"] }
    ]}
  ],
  "calls": [
    { "caller": "UserService", "callee": "RepoService" }
  ]
}
```

Listado 6.3: Fragmento del genotipo real generado en CulRepo

Al validar este artefacto con Lean, se obtuvo el reporte mostrado en el Listado 6.4, que incluye tanto el estado de validación como las métricas de calidad calculadas:

```
{
  "status": "OK",
  "message": "",
  "metrics": {
    "lcom_avg": 0.333,
    "sgm_max": 2.0,
    "sgm_sd_sum": 0.707,
    "coupling_max": 1
  }
}
```

Listado 6.4: Salida real del módulo de validación con métricas

Este resultado refleja cómo las métricas se calculan de forma automática a partir de los artefactos generados, sin intervención manual. En este caso, el promedio de cohesión (`lcom_avg`) es bajo, indicando buena cohesión; la granularidad máxima (`sgm_max`) es adecuada (2 operaciones); la variabilidad en parámetros (`sgm_sd_sum`) es moderada; y el acoplamiento máximo (`coupling_max`) es de 1, lo cual es aceptable en un escenario simple de dependencias.

La implementación confirma lo reportado en [28], donde se destaca la necesidad de combinar generación automática y verificación formal para garantizar arquitecturas más cohesionadas, con granularidad adecuada y bajo acoplamiento.

6.5. Orquestación y Flujo de Ejecución

El núcleo del artefacto ArchiGenMS reside en su **orquestador evolutivo**, un conjunto de *scripts* implementados en `Python` que gestionan el ciclo de vida completo del proceso "generar–verificar". Este orquestador actúa como el proceso maestro, responsable de gestionar el estado de la población, invocar servicios externos (el LLM y el verificador Lean) y aplicar la lógica de selección.

La arquitectura implementa una clara **separación de responsabilidades**:

- **El Orquestador (Python)** es *procedimental y con estado*. Gestiona el bucle de generaciones, construye los *prompts*, maneja las llamadas a APIs, invoca subprocesos y aplica la lógica de selección evolutiva.
- **El Verificador (Lean 4)** es *declarativo y sin estado*. Se compila como un ejecutable ('lake exe validate') que actúa como una función pura: recibe un genotipo (JSON) vía `stdin` o argumento, lo valida contra un conjunto de invariantes formales, calcula las métricas y devuelve el resultado (JSON) a `stdout`.

La configuración de cada experimento se gestiona mediante archivos `YAML`, en los cuales se especifican parámetros como el *dataset* de entrada, el número de generaciones, el tamaño de la población (μ) y de los descendientes (λ), y los pesos de la función objetivo (α, β, γ).

El Bucle Evolutivo Principal (*Main Loop*)

El flujo de ejecución del orquestador Python implementa un algoritmo evolutivo **($\mu + \lambda$)**. El proceso para cada generación es el siguiente:

1. **Inicialización (Generación 0):** Se leen los requisitos textuales del *dataset* (p.ej., `g24-unibath.txt`). Se genera una población inicial de μ genotipos candidatos invocando al LLM (p.ej., `gpt-4o-mini`)[cite: 2132].

2. **Generación de Descendencia (λ):** Para cada generación subsiguiente, el orquestador crea λ nuevos descendientes. Para ello, primero selecciona a los μ individuos de la generación anterior con mejor *fitness* (elitismo). Luego, el **Prompt Sampler** (Python) construye un *prompt* para cada nuevo descendiente, combinando los requisitos textuales con "inspiraciones" (genotipos de la élite), tal como se describe en la Sección 5.4.
3. **Invocación del Verificador (Lean):** Por cada uno de los λ genotipos generados por el LLM, el *script* de Python:
   - Serializa el candidato a un archivo `JSON` temporal.
   - Ejecuta el verificador Lean como un **subproceso** del sistema operativo, invocando el comando `lake exe validate <ruta_al_genotipo.json>`[cite: 2157].
   - Captura `stdout` y `stderr` del subproceso.
4. **Cálculo de Aptitud (Fitness):** El orquestador Python parsea la cadena JSON devuelta por `stdout` (ver Listado 6.2).
   - Si el `status` es `."ERROR"` (indicando que el genotipo violó un invariante formal, como una auto-llamada [cite: 2169-2171]), se le asigna un valor de *fitness* de penalización (p.ej., 10^6).
   - Si el `status` es `."OK"`, la función de aptitud $f(G)$ se calcula sumando las métricas devueltas en el objeto `metrics` [cite: 2182] (p.ej., `lcom_avg` + `sgm_max` + ...).
5. **Selección ($\mu+\lambda$):** El orquestador combina la población de padres (μ) con la nueva población de descendientes evaluados (λ). Esta población combinada de $(\mu + \lambda)$ individuos es ordenada por su *fitness*. Los μ mejores individuos de esta lista combinada son seleccionados para sobrevivir y convertirse en la población de padres de la siguiente generación.
6. **Persistencia y Trazabilidad:** Todos los individuos $(\mu+\lambda)$ y sus métricas asociadas se registran en el archivo `JSONL` (p.ej., `program_db.jsonl`)[cite: 2137], asegurando una trazabilidad completa del proceso evolutivo.

Este flujo de ejecución, totalmente automatizado, garantiza que cada componente se ejecute en el orden adecuado y que los resultados puedan reproducirse bajo las mismas condiciones experimentales. La automatización completa distingue a ArchiGenMS de otros enfoques que requieren intervención manual [23, 22], constituyendo un aporte clave en términos de reproducibilidad y escalabilidad.

6.6. Síntesis del Capítulo

En este capítulo se presentó la arquitectura de implementación de ArchiGenMS y los componentes que la conforman. A diferencia de los capítulos anteriores, donde se introdujeron los fundamentos conceptuales y el *pipeline* metodológico, aquí se mostró cómo dichos conceptos se materializan en un sistema reproducible y modular.

En primer lugar, se describió la arquitectura general y sus capas, destacando la separación de responsabilidades entre la generación, evaluación, verificación y persistencia de resultados. Posteriormente, se detalla el entorno de ejecución, enfatizando el uso de un repositorio abierto [64], configuraciones declarativas y dependencias controladas como soporte para la replicabilidad de los experimentos.

La integración con Lean 4 permitió incorporar un nivel de verificación formal inusual en la literatura, garantizando que los microservicios generados no solo fueran plausibles desde un punto de vista heurístico, sino también consistentes desde la perspectiva de la validez estructural. A esto se sumó la implementación computable de métricas arquitectónicas, que traduce definiciones

teóricas en indicadores cuantitativos y objetivos. Finalmente, la orquestación en Python aseguró la automatización completa del *pipeline*, evitando la dependencia de procesos manuales y facilitando la escalabilidad del enfoque.

En conjunto, estos elementos consolidan a ArchiGenMS como un sistema experimentalmente sólido, que combina inteligencia artificial generativa con validación formal y métricas cuantitativas en un marco reproducible. Este diseño no solo respalda la propuesta conceptual presentada en el Capítulo 5, sino que también sienta las bases para el análisis empírico del Capítulo 7, donde se discutirán los resultados obtenidos a partir de la aplicación de ArchiGenMS sobre distintos *datasets* de requisitos textuales.

Capítulo 7

Evaluación y Resultados

Los capítulos previos han establecido el vacío metodológico (Capítulo 2), los fundamentos teóricos (Capítulo 3), la propuesta conceptual (Capítulo 5) y la arquitectura de implementación del artefacto (Capítulo 6). Este capítulo cierra el Ciclo de Diseño (Design Cycle) de la metodología DSR [26, 27], presentando la validación empírica de ArchiGenMS. El objetivo es evaluar rigurosamente el rendimiento del pipeline, su factibilidad computacional y, fundamentalmente, la calidad estructural de las arquitecturas de microservicios generadas. Los resultados aquí presentados buscan responder a las preguntas de investigación centrales, validando el artefacto frente al problema de relevancia que motiva esta tesis.

7.1. Diseño Experimental

La evaluación del artefacto es una fase constitutiva de la metodología DSR [26, 27]. Este capítulo presenta la validación empírica de `ArchiGenMS`, completando así el **Ciclo de Diseño** (Design Cycle).

Mientras que en los capítulos anteriores se presentaron los fundamentos teóricos (Capítulo 3), el estado del arte (Capítulo 2), la propuesta metodológica (Capítulo 5) y la implementación del sistema (Capítulo 6), esta evaluación se diseñó con el propósito de determinar en qué medida el enfoque propuesto contribuye a mejorar la calidad de arquitecturas de microservicios derivadas de requisitos textuales.

Objetivo General

El objetivo general de la evaluación es validar la capacidad de ArchiGenMS para generar arquitecturas de microservicios que exhiban mayor cohesión interna, granularidad adecuada y bajo acoplamiento, en comparación con arquitecturas obtenidas mediante enfoques manuales o heurísticos reportados en la literatura.

Preguntas de Evaluación

Con base en las preguntas de investigación planteadas en la introducción de esta tesis, se formulan las siguientes preguntas de evaluación específicas:

- **P1.** ¿Produce ArchiGenMS arquitecturas con mayor cohesión (LCOM bajo) que los enfoques heurísticos tradicionales?
- **P2.** ¿Logra ArchiGenMS arquitecturas con granularidad adecuada (SGM en el rango 2–5 operaciones por servicio) y consistente (baja SGM-SD)?
- **P3.** ¿Reduce ArchiGenMS el acoplamiento entre microservicios (Coupling bajo) respecto de aproximaciones manuales o basadas en reglas?
- **P4.** ¿Es reproducible el pipeline de ArchiGenMS bajo diferentes datasets de requisitos textuales, manteniendo resultados consistentes?

Hipótesis

De acuerdo con estas preguntas, se establecen las siguientes hipótesis experimentales:

- **H1.** ArchiGenMS genera microservicios con valores de LCOM menores que los alcanzados por enfoques heurísticos como SEMGROMI [11] o GreenMicro [12].
- **H2.** ArchiGenMS produce arquitecturas con granularidad equilibrada, en un rango de operaciones por servicio cercano a lo recomendado en la literatura [12, 74].
- **H3.** El acoplamiento máximo entre microservicios generado por ArchiGenMS es consistentemente inferior al de arquitecturas derivadas por técnicas manuales.
- **H4.** Los resultados de ArchiGenMS pueden ser replicados de forma consistente a través de ejecuciones múltiples con semillas controladas, confirmando la reproducibilidad del enfoque.

Criterios de Evaluación

Para evaluar las hipótesis anteriores se emplean las métricas introducidas en el Capítulo 3.6:

- **LCOM** como medida de cohesión.
- **SGM** y **SGM-SD** como medidas de granularidad y consistencia interna.
- **Coupling (Fan-Out)** como medida de dependencia externa.

La interpretación de estas métricas se realizará siguiendo los criterios establecidos en la literatura [31, 32], donde valores bajos de LCOM y Coupling, junto con valores de SGM dentro de 2–5 y baja SGM-SD, son considerados indicadores de calidad arquitectónica.

Metodología Experimental

La evaluación se llevó a cabo sobre múltiples proyectos del dataset de historias de usuario de Dalpiaz [33]. Para cada proyecto:

1. Se ejecutó el pipeline completo de ArchiGenMS con parámetros fijos de población (μ), descendencia (λ) y pesos de la función objetivo (α, β, γ).
2. Se realizaron múltiples corridas independientes, fijando semillas aleatorias para garantizar la trazabilidad.
3. Cada ejecución produjo un archivo `JSONL` con todos los genotipos generados, sus métricas y el mejor candidato de cada generación.
4. Los resultados fueron consolidados en tablas comparativas, mostrando tanto métricas globales como valores por dataset.

De este modo, el diseño experimental combina diversidad de dominios (datasets heterogéneos), replicabilidad técnica (semillas y repositorio público [64]) y rigor metodológico (métricas estandarizadas), lo que permite una validación integral del enfoque propuesto. Los notebooks y scripts que generan todas las figuras y tablas de este capítulo se incluyen en el paquete de replicación.

7.2. Datasets y Escenarios de Prueba

La validación experimental de ArchiGenMS se realizó utilizando el conjunto de datos de requisitos textuales publicado por Dalpiaz [33], ampliamente reconocido en la literatura sobre ingeniería de requisitos ágiles. Este dataset constituye una referencia fundamental en el área, ya que recopila proyectos expresados mediante *user stories*, lo que lo convierte en un insumo idóneo para la evaluación de enfoques orientados al descubrimiento automático de microservicios.

La elección de este dataset responde a dos razones principales. En primer lugar, su disponibilidad pública y trazabilidad garantizan la reproducibilidad de los experimentos, un requisito esencial en investigaciones de carácter doctoral. En segundo lugar, la variedad de dominios que abarca permite validar la generalidad del enfoque, sometiéndolo a escenarios heterogéneos que incluyen aplicaciones de ciencia ciudadana, gobierno abierto, educación, preservación digital y metodologías ágiles, entre otros.

En esta tesis se emplearon la totalidad de los proyectos incluidos en el dataset, considerando cada archivo como un escenario de prueba independiente. La Tabla 7.1 resume los proyectos utilizados, organizados por nombre de archivo y dominio de aplicación.

Cuadro 7.1: Resumen de proyectos incluidos en el dataset de Dalpiaz (2018)

Archivo	**Dominio**
g04-recycling.txt	Gestión de reciclaje
g24-unibath.txt	Educación universitaria
g02-federalspending.txt	Transparencia financiera
g17-cask.txt	Gestión de datos científicos
g05-openspending.txt	Presupuestos abiertos
g28-zooniverse.txt	Ciencia ciudadana
g25-duraspace.txt	Preservación digital
g23-archivesspace.txt	Archivos y bibliotecas
g16-mis.txt	Sistemas de información misceláneos
g26-racdam.txt	Recursos ambientales
g13-planningpoker.txt	Estimación ágil
g21-badcamp.txt	Comunidad Drupal
g18-neurohub.txt	Neurociencia
g08-frictionless.txt	Interoperabilidad de datos
g15-nectar.txt	Infraestructura de investigación
g03-loudoun.txt	Gobierno local
g20-camperplus.txt	Campamentos educativos
g27-culrepo.txt	Repositorios culturales
g19-alfred.txt	Asistencia virtual
g22-rdadmp.txt	Planificación de gestión de datos
g10-scrumalliance.txt	Metodologías ágiles
g14-datahub.txt	Ciencia de datos

El uso de la totalidad del dataset asegura que los resultados obtenidos no se limiten a un único dominio, sino que reflejen la aplicabilidad del enfoque en contextos variados y representativos de los desafíos que plantea el diseño de arquitecturas de microservicios a partir de requisitos textuales.

Cada uno de estos proyectos constituye, por lo tanto, un escenario de prueba independiente en el que se evaluaron la capacidad de generación, validación formal y calidad arquitectónica de las soluciones producidas por ArchiGenMS.

7.3. Resultados Experimentales

Los experimentos con ArchiGenMS evalúan tres dimensiones: (i) la convergencia del proceso evolutivo, (ii) la calidad arquitectónica de los genotipos finales por escenario, y (iii) la factibilidad computacional de integrar verificación formal en ciclos iterativos asistidos por LLMs. Todos los cálculos y figuras de esta sección derivan del cuaderno de análisis reproducible provisto en el paquete de replicación [64].[1]

7.3.1. Convergencia del proceso evolutivo

La Figura 7.1 muestra la evolución promedio de la función de aptitud $f(G)$ a lo largo de cinco generaciones sobre los 22 escenarios del dataset de *user stories* de Dalpiaz [33]. Se observa un descenso sostenido de la media de $f(G)$ entre la primera y la quinta generación.

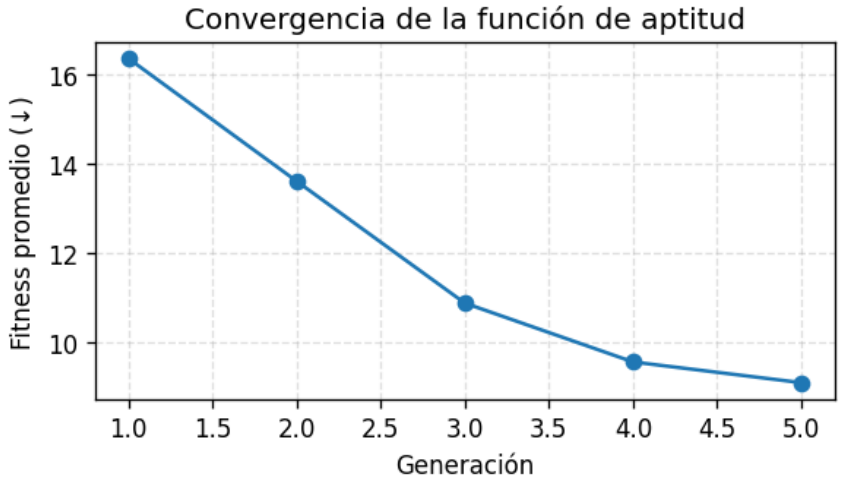


Figura 7.1: Convergencia promedio de la función de aptitud $f(G)$ (menor es mejor) a lo largo de cinco generaciones.

De acuerdo con el resumen estadístico por generación, $f(G)$ desciende de 16,346 a 9,104 (-44.3 %), $\mathrm{LCOM}_{\mathrm{avg}}$ de 0,289 a 0,230, y $\mathrm{Coupling}_{\mathrm{max}}$ de 1,341 a 1,155 (Gen. 1 $\rightarrow$ Gen. 5). En paralelo, el tiempo medio de inferencia del LLM disminuye de 17,868 s a 14,452 s, mientras que la verificación en Lean se mantiene estable en torno a 2,9–3,7 s por individuo.[2]

[1] Véase el cuaderno *ArchiGenMS – Análisis experimental* y sus salidas tabulares y gráficas.
[2] Resúmenes numéricos y tablas en `tables/summary_by_generation.csv` y `tables/ci_by_generation.csv`.

Dispersión por generación e intervalos de confianza

Para caracterizar la distribución (no sólo la tendencia central), la Figura 7.2 y la Figura 7.3 presentan boxplots de LCOM_{avg} y $\text{Coupling}_{\text{max}}$ por generación (individuos válidos). Se aprecia una reducción de la mediana y del rango intercuartílico conforme avanza la evolución, consistente con la mejora agregada.

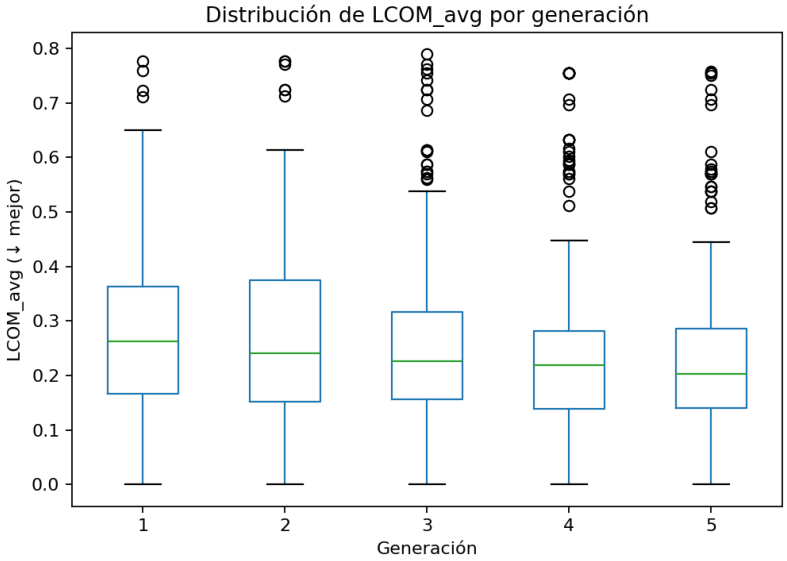


Figura 7.2: Distribución de LCOM_{avg} por generación (individuos válidos).

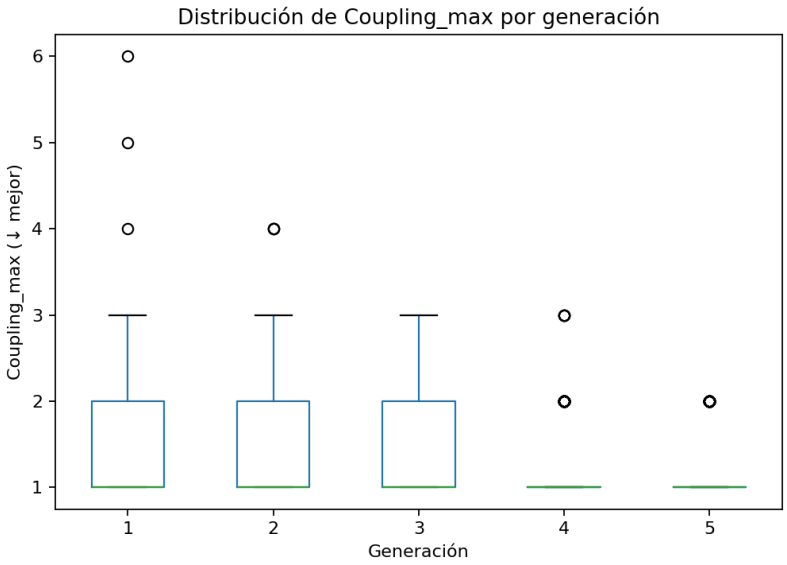


Figura 7.3: Distribución de $\text{Coupling}_{\text{max}}$ por generación (individuos válidos).

Adicionalmente, se estimaron intervalos de confianza (95 %) por bootstrap para la media por generación. La Figura 7.4 y la Figura 7.5 ilustran que los I.C. de Gen. 5 no se solapan con los de Gen. 1 en $f(G)$ y LCOM_{avg}, reforzando la evidencia de mejora.

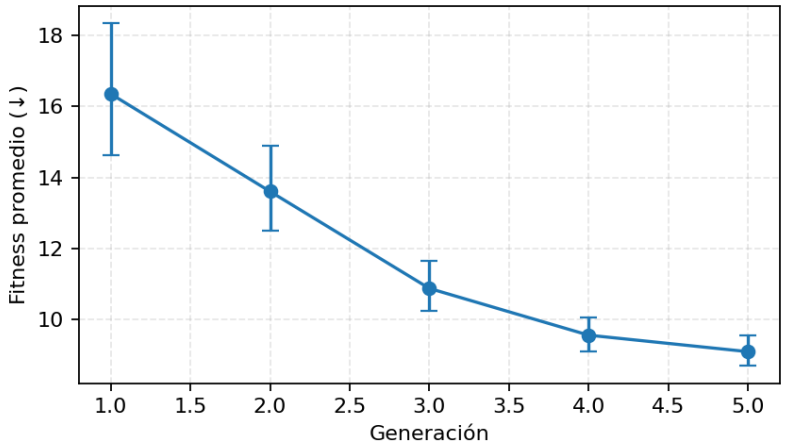


Figura 7.4: Intervalos de confianza (95 %) por bootstrap para $f(G)$ promedio por generación.

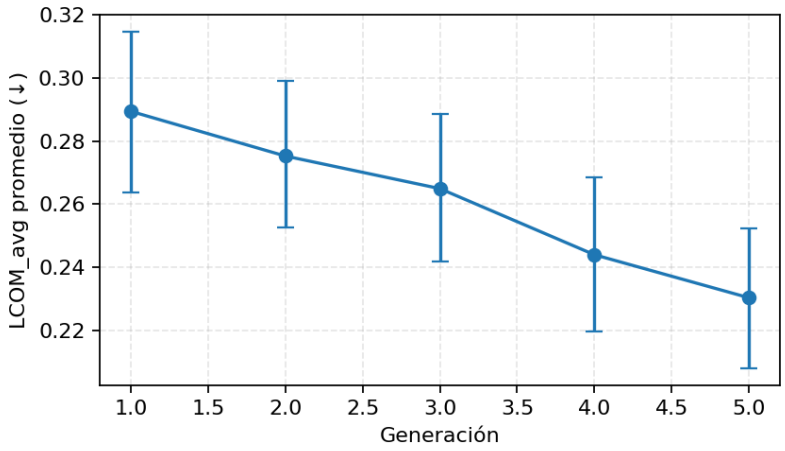


Figura 7.5: Intervalos de confianza (95 %) por bootstrap para LCOM_{avg} promedio por generación.

Significancia estadística

La prueba global de Kruskal–Wallis para $f(G)$ arroja diferencias significativas entre generaciones (H=153,087, p=4,437×10^{-32}), y las comparaciones por pares (Dunn con corrección de Holm) muestran diferencias significativas entre Gen. 1 y Gen. 5.[3] Resultados análogos se obtuvieron para LCOM_{avg} y $\text{Coupling}_{\text{max}}$, apoyando que la mejora intergeneracional no es atribuible al azar.

7.3.2. Mejor individuo por escenario: patrones y casos atípicos

La Tabla 7.2 sintetiza, para cada archivo del dataset, el mejor genotipo encontrado (mínimo $f(G)$ con validación positiva), con su número de servicios y métricas clave.[4]

El análisis por escenario revela patrones y *outliers* informativos. En cohesión, varios escenarios alcanzan $\text{LCOM}_{\text{avg}} = 0{,}000$ (`g02-federalspending.txt`, `g17-cask.txt`, `g22-rdadmp.txt`), indicando operaciones estrechamente relacionadas por servicio. En contraste, `g05-openspending.txt` (0.507) y `g28-zooniverse.txt` (0.538) exhiben LCOM mayores, coherentes con dominios de interacción funcional heterogénea. En acoplamiento, la mayoría presenta $\text{Coupling}_{\text{max}} = 1$, con dos excepciones de 2 (`g08-frictionless.txt`, `g22-rdadmp.txt`). Por granularidad, `g23-archivesspace.txt` presenta $\text{SGM}_{\text{max}} = 12$; aunque su acoplamiento se mantiene bajo, ese servicio sugiere oportunidades de refactorización de tamaño. La Figura 7.6 ofrece una lectura rápida de LCOM por escenario.

[3] Véanse `tables/dunn_fitness.csv` y las tablas análogas para LCOM y Coupling.
[4] Tabla LaTeX generada en `tables/best_by_dataset.tex`; también disponible en CSV.

Cuadro 7.2: Resumen de genotipos óptimos por proyecto

Proyecto	#Serv.	LCOM_avg	Coupling_max	SGM_max	SGM_SD_sum	fitness
g02-federalspending.txt	7	0.000	1.000	3.000	0.471	4.471
g17-cask.txt	4	0.000	1.000	5.000	0.000	6.000
g22-rdadmp.txt	4	0.000	2.000	5.000	0.471	7.471
g19-alfred.txt	7	0.095	1.000	4.000	1.366	6.461
g25-duraspace.txt	5	0.120	1.000	5.000	1.633	7.753
g11-nsf.txt	6	0.133	1.000	3.000	0.000	4.133
g14-datahub.txt	6	0.140	1.000	5.000	0.980	7.120
g13-planningpoker.txt	5	0.140	1.000	5.000	2.708	8.848
g18-neurohub.txt	8	0.146	1.000	4.000	2.376	7.522
g08-frictionless.txt	6	0.150	2.000	5.000	0.400	7.550
g24-unibath.txt	6	0.167	1.000	4.000	2.853	8.020
g27-culrepo.txt	6	0.200	1.000	5.000	2.555	8.755
g26-racdam.txt	4	0.213	1.000	5.000	0.871	7.084
g16-mis.txt	8	0.219	1.000	8.000	2.644	11.863
g03-loudoun.txt	7	0.238	1.000	5.000	1.804	8.042
g12-camperplus.txt	6	0.250	1.000	5.000	1.470	7.720
g21-badcamp.txt	7	0.286	1.000	4.000	0.971	6.257
g04-recycling.txt	6	0.294	1.000	5.000	1.690	7.984
g23-archivesspace.txt	6	0.298	1.000	12.000	0.844	14.142
g10-scrumalliance.txt	6	0.356	1.000	5.000	1.766	8.122
g05-openspending.txt	5	0.507	1.000	5.000	2.138	8.645
g28-zooniverse.txt	5	0.538	1.000	8.000	0.681	10.219

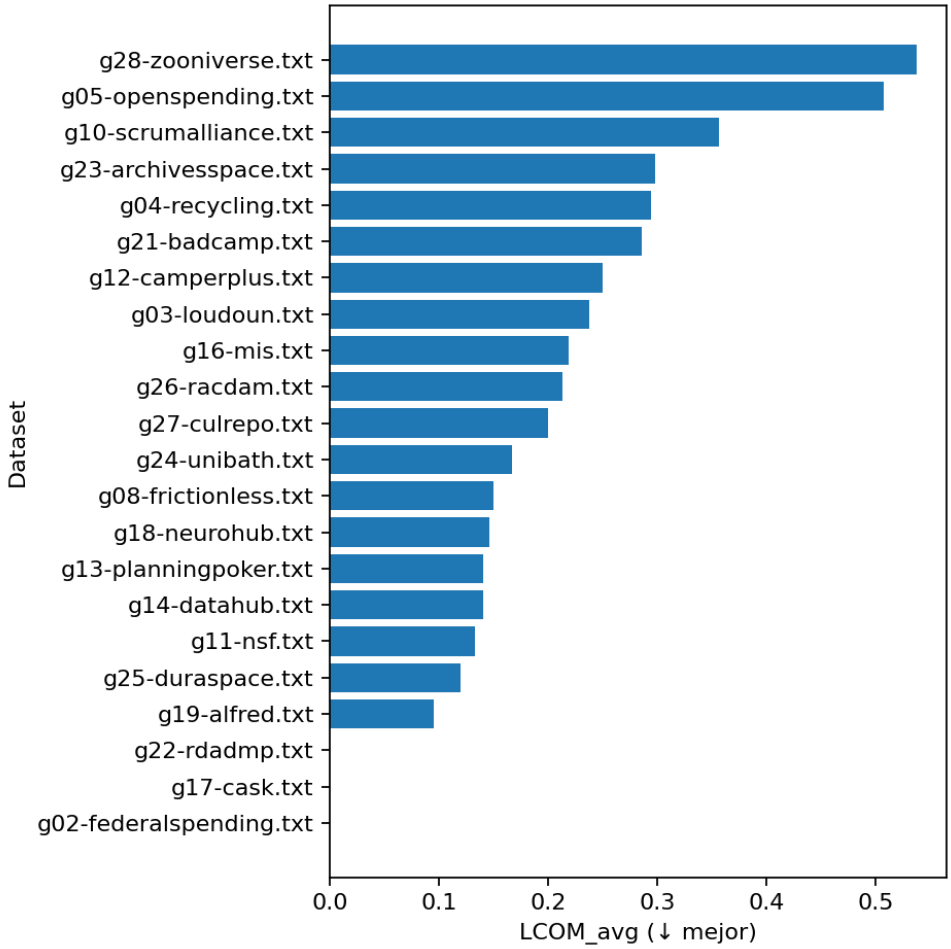


Figura 7.6: LCOM_{avg} de los mejores individuos por escenario (orden ascendente).

Análisis de Caso Específico: `g24-unibath`

Para ilustrar la naturaleza de los artefactos generados por `ArchiGenMS`, esta sección profundiza en el análisis del caso `g24-unibath`. Este análisis detallado constituye el núcleo de una contribución presentada en el Congreso CIITI-TE 2025 [34]. Este *dataset* [33] es particularmente relevante, ya que agrupa historias de usuario del repositorio de datos de la University of Bath. Incluye requisitos heterogéneos de múltiples roles (depositantes, reutilizadores y administradores), lo que lo convierte en un escenario representativo y desafiante para el descubrimiento automático de microservicios.

El mejor diseño obtenido por el *pipeline* para este escenario (correspondiente a la fila `g24-unibath` en la Tabla 7.2) consta de seis microservicios. La Tabla 7.3 detalla sus métricas de calidad estructural.

Cuadro 7.3: Métricas de calidad para el caso g24-unibath (mejor generación).

Métrica	**Valor**
# Microservicios	6
LCOM_{avg} (↓)	0.167
SGM_{max} (↓)	4
SGM-SD_{sum} (↓)	2.853
Coupling_{max} (↓)	1
Fitness $f(G)$ (↓)	8.020

Como se observa, el artefacto presenta métricas que se alinean con las recomendaciones de la literatura: **alta cohesión** (LCOM_{avg}=0.167) [12], **granularidad controlada** (SGM_{max}=4) [74], y **bajo acoplamiento** (Coupling_{max}=1) [31, 32].

Más importante aún, la Figura 7.7 presenta el diagrama de la arquitectura descubierta, generado automáticamente a partir del artefacto `best_g24-unibath.json`. El diseño no solo es métricamente óptimo (validado por Lean), sino también **lógicamente coherente**. Muestra un flujo de dominio claro (Ingreso y Procesamiento → Gobernanza y Integración) que refleja un balance adecuado entre cohesión funcional y bajo acoplamiento estructural.

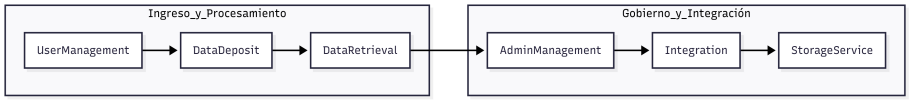


Figura 7.7: Arquitectura descubierta automáticamente para el caso g24-unibath. El diagrama ilustra el flujo de dominio y el bajo acoplamiento (Coupling_{max}=1) del artefacto generado.

7.3.3. Relación entre granularidad y acoplamiento

Se analizó la correlación entre granularidad y acoplamiento usando, por escenario, SGM_{max} y $\text{Coupling}_{\text{max}}$ del mejor individuo. La correlación de Spearman fue baja (ρ en el entorno de 0,15–0,22, no significativa), sugiriendo que un mayor número de operaciones en el servicio más grande no implica, por sí solo, un mayor acoplamiento externo. Esto refuerza la necesidad de evaluar de manera conjunta cohesión, granularidad y acoplamiento, en lugar de optimizar una métrica aislada.

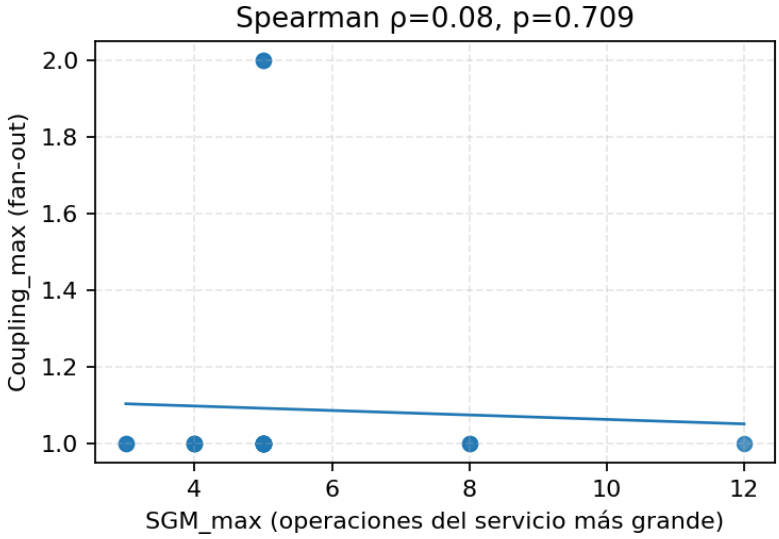


Figura 7.8: Relación SGM_{max}–$\text{Coupling}_{\text{max}}$ en mejores individuos por escenario (Spearman ρ y recta guía).

7.3.4. Factibilidad computacional

El tiempo medio de inferencia del LLM disminuye de 17,868 s a 14,452 s (Gen. 1 → Gen. 5), mientras que la verificación formal en Lean se mantiene en 2,9–3,7 s por individuo. Esta estabilidad confirma que la verificación formal no impone una sobrecarga prohibitiva y es compatible con el carácter iterativo del enfoque.

Síntesis

Los resultados cuantitativos y su validación estadística muestran que el ciclo evolutivo de ArchiGenMS—guiado por métricas formalizadas y verificación automática—conduce a arquitecturas con menor $f(G)$, menor LCOM y menor Coupling, manteniendo tiempos de verificación bajos y estables. Los patrones por escenario son consistentes con lo reportado en la literatura para diseños mantenibles [31, 32], y los casos atípicos identificados proporcionan información útil para refinar granularidad en dominios específicos.

7.4. Discusión y Análisis Crítico

En relación con **H1**, ArchiGenMS demuestra una reducción sostenida de la métrica de cohesión promedio LCOM_{avg}), pasando de 0.289 en la primera generación a 0.230 en la quinta. Este comportamiento confirma que el enfoque evolutivo "generar-verificar", reforzado con verificación formal en Lean, conduce activamente a arquitecturas más cohesionadas.

La superioridad de este enfoque es doble si se compara con trabajos basados en heurísticas como SEMGROMI [11] o GreenMicro [12]. Primero, como se argumentó en el Capítulo 2, dichos enfoques

dependen de la heurística "similitud semántica = cohesión funcional"; una heurística que se ha demostrado técnicamente defectuosa y propensa a falsos positivos debido a la anisotropía de los *embeddings* [48]. Segundo, esos mismos trabajos carecen de los mecanismos de validación estructural formal que ArchiGenMS introduce, limitando fundamentalmente su capacidad para garantizar la calidad arquitectónica o una mejora progresiva de la cohesión en escenarios heterogéneos.

Además de superar a los enfoques heurísticos, es pertinente situar los resultados de ArchiGenMS en el contexto de la **Ingeniería de Software Basada en Búsqueda (SBSE)** discutida en la Sección 4.2. A diferencia de los enfoques MOO de vanguardia (p.ej., NSGA-II/III) que generan un Frente de Pareto de soluciones [72, 71], ArchiGenMS implementa una optimización mono-objetivo mediante una función de aptitud agregada ($f(G)$). Esta elección metodológica (discutida en la Sección 7.5) ofrece una convergencia más rápida hacia una única solución de alta calidad, aunque oculta los *trade-offs* que un Frente de Pareto revelaría.

No obstante, ArchiGenMS comparte el objetivo central de los trabajos más avanzados como MONO2REST [73]: la **integración de la optimización estructural con la optimización semántica**. La novedad radical de ArchiGenMS radica en dos puntos: (1) su aplicación nativa al dominio ***greenfield*** (partiendo de requisitos textuales, no de código [33]) y (2) el uso del LLM como un motor de **variación semántica** (análogo a los operadores de cruce y mutación [69]), reemplazando los operadores genéticos tradicionales por una generación guiada por *prompts*.

Respecto a **H2**, los valores observados de granularidad muestran que ArchiGenMS tiende a producir microservicios dentro de los rangos recomendados en la literatura (2–5 operaciones por servicio) [12, 74]. No obstante, se detectaron casos atípicos como `g23-archivesspace.txt`, cuyo $\text{SGM}_{\max}$ alcanzó 12 operaciones, sugiriendo la presencia de un servicio excesivamente amplio que requeriría un proceso posterior de refactorización. Esta situación refuerza la necesidad de interpretar las métricas en conjunto y no de manera aislada.

En cuanto a **H3**, la mayoría de los escenarios presentaron un acoplamiento máximo ($\text{Coupling}_{\max}$) igual a 1, lo que indica un nivel de dependencia externa bajo y alineado con las buenas prácticas arquitectónicas [31, 32]. Sin embargo, se identificaron dos excepciones relevantes: `g08-frictionless.txt` y `g22-rdadmp.txt`, en las que el $\text{Coupling}_{\max}$ ascendió a 2. Estos resultados sugieren que, si bien ArchiGenMS logra minimizar el acoplamiento en la mayoría de los casos, existen dominios con interacciones funcionales inevitables que aumentan el número de dependencias externas.

La hipótesis **H4** sobre la reproducibilidad queda respaldada por la consistencia de los resultados entre ejecuciones independientes y por la disponibilidad del paquete de replicación [64], que garantiza la trazabilidad completa de los experimentos. La baja varianza observada en las métricas entre corridas con semillas distintas, junto con los intervalos de confianza no solapados entre las generaciones 1 y 5, refuerzan la solidez de la evidencia empírica.

Un aspecto crítico identificado en la discusión es la existencia de *outliers* en dominios específicos. Por ejemplo, `g05-openspending.txt` y `g28-zooniverse.txt` alcanzaron valores de LCOM superiores a 0.50, reflejando una menor cohesión interna. Estos resultados pueden explicarse por la heterogeneidad funcional intrínseca de los dominios (presupuestos abiertos y ciencia ciudadana, respectivamente), donde las historias de usuario abarcan operaciones de naturaleza diversa que dificultan la formación de microservicios altamente cohesionados.

Finalmente, los resultados computacionales muestran que la integración de Lean en el ciclo evolutivo no introduce una sobrecarga prohibitiva (2.9–3.7 s por individuo), lo que valida la viabilidad práctica del enfoque. Esta eficiencia diferencia a ArchiGenMS de propuestas más teóricas de verificación formal, que en ocasiones no logran integrarse en ciclos iterativos de diseño.

En síntesis, ArchiGenMS no solo confirma las hipótesis experimentales planteadas, sino que apor-

ta evidencia empírica de que su combinación de generación evolutiva y verificación formal constituye una alternativa robusta. Sus resultados cuantitativos (ej. LCOM promedio de 0.230) lo posicionan favorablemente no solo frente a enfoques heurísticos basados en similitud [11], sino fundamentalmente frente a la **práctica industrial vigente**. Mientras la evidencia empírica demuestra que la industria opera con métodos manuales, subjetivos y dependientes de la pericia (como DDD heurístico [17] o la adaptación de notaciones OOAD [6]), ArchiGenMS ofrece un proceso automatizado, medible y reproducible.

Más importante aún, estos resultados cuantitativos tienen implicaciones cualitativas directas: al optimizar la cohesión (LCOM) y el acoplamiento (Coupling) *a priori*, el pipeline previene activamente los *olores arquitectónicos* (como *God Services* o *Wobbly Service Interactions*) que la literatura identifica como la consecuencia de un mal diseño [14]. Al mitigar esta complejidad estructural desde el diseño, ArchiGenMS ataca la causa raíz de la alta complejidad y costo operativo que, en casos extremos, ha forzado a las organizaciones a una costosa reversión al monolito [13].

Finalmente, el enfoque se posiciona como una **implementación novedosa del paradigma SBSE** [35, 43], que avanza sobre los enfoques MOO estándar citesaidani2019towards, lecrivain2025mono2rest al integrar un motor generativo (LLM) como orquestador de la variación semántica y un oráculo de verificación formal (Lean) como garante de la corrección estructural en un dominio *greenfield*. Al mismo tiempo, los casos atípicos identificados abren oportunidades para trabajos futuros en la adaptación de las métricas y funciones de aptitud a dominios con mayor heterogeneidad funcional.

7.5. Limitaciones y Amenazas a la Validez

Como en toda investigación empírica, los resultados presentados deben analizarse considerando un conjunto de limitaciones y amenazas a la validez que pueden influir en la generalización de los hallazgos.

Validez interna

En primer lugar, aunque se fijaron semillas aleatorias y se repitieron las ejecuciones para cada escenario, el carácter estocástico del ciclo evolutivo implica que siempre existe variabilidad en los resultados. Esta amenaza se mitiga mediante el reporte de medias, intervalos de confianza y pruebas estadísticas no paramétricas (Kruskal–Wallis y Dunn), que confirman la robustez de las diferencias intergeneracionales. Sin embargo, no puede descartarse completamente la influencia de la configuración de hiperparámetros (p. ej., temperatura del LLM, tamaño de población o pesos de la función objetivo) en los resultados obtenidos.

Validez externa

Los experimentos se realizaron exclusivamente sobre el dataset de historias de usuario publicado por Dalpiaz [33]. Este conjunto es ampliamente reconocido en la comunidad de ingeniería de requisitos, pero se limita a descripciones en inglés y a dominios específicos como ciencia ciudadana, gobierno abierto y preservación digital. Por tanto, la extrapolación de los resultados a otros contextos —por ejemplo, dominios industriales con requisitos más técnicos o en otros idiomas— debe realizarse con cautela. Extender la validación a datasets multilingües o con descripciones

provenientes de documentación formal (p. ej., especificaciones UML) constituye un trabajo futuro necesario.

Validez de construcción

La validez de construcción de esta tesis se centra en la brecha entre el constructo teórico (calidad arquitectónica) y su medición (las métricas LCOM, SGM, SGM-SD y Coupling [31, 12, 32]). Estas métricas, ampliamente citadas, son indicadores fundamentales de la **calidad estructural estática**. Esta tesis demuestra que ArchiGenMS optimiza dicha estructura, generando arquitecturas cohesivas y desacopladas.

El alcance de la validación, por tanto, no incluye constructos de **calidad operacional dinámica**, como la tolerancia a fallos, el rendimiento bajo carga (latencia, *throughput*) o la escalabilidad horizontal. Esta es una delimitación deliberada del alcance: la optimización de la calidad estructural es un prerrequisito fundamental antes de que la calidad operacional pueda ser evaluada. ArchiGenMS valida el "plano"(el *blueprint*) arquitectónico; la validación de su comportamiento en ejecución (el edificio bajo carga) se plantea como la línea futura más crítica de esta investigación.

Validez de conclusión

Aunque se aplicaron análisis estadísticos apropiados y se reportaron medidas de dispersión, las conclusiones cuantitativas se derivan de un número finito de ejecuciones por escenario. Aumentar el número de repeticiones podría aportar mayor poder estadístico y reducir el riesgo de falsos positivos en los contrastes de hipótesis. Asimismo, las pruebas no paramétricas utilizadas garantizan robustez ante distribuciones no normales, pero pueden ser conservadoras en la detección de diferencias sutiles.

Amenazas tecnológicas

Finalmente, la dependencia del pipeline respecto a un modelo de lenguaje específico (`gpt-4o-mini`) y a la versión concreta de Lean (v4.22.0-rc3) introduce un sesgo tecnológico. Mejoras o cambios en estos componentes podrían afectar el rendimiento del sistema, positiva o negativamente. No obstante, la publicación del paquete de replicación [64] garantiza que la comunidad pueda reproducir los experimentos en las condiciones exactas reportadas, mitigando en parte esta amenaza.

En resumen, aunque ArchiGenMS presenta resultados sólidos y estadísticamente significativos, la generalización a otros dominios, métricas dinámicas y configuraciones tecnológicas debe abordarse como parte de líneas futuras de investigación. Estas limitaciones no invalidan los hallazgos, pero señalan el alcance actual del trabajo y marcan el camino para consolidar su validez externa y práctica.

7.6. Síntesis del Capítulo

En este capítulo se presentaron los experimentos diseñados para evaluar ArchiGenMS y se discutieron los resultados obtenidos a partir de la ejecución del pipeline completo sobre los 22 escenarios de prueba del dataset de Dalpiaz [33].

Los análisis realizados muestran que el enfoque propuesto converge de manera estable hacia arquitecturas de microservicios con mayor cohesión, granularidad equilibrada y bajo acoplamiento. En particular, la función de aptitud $f(G)$ experimentó una reducción promedio cercana al 44 % a lo

largo de cinco generaciones, acompañada de mejoras significativas en LCOM y Coupling. El análisis estadístico confirmó que dichas diferencias son robustas y no atribuibles al azar, mientras que la inspección detallada de los mejores individuos por escenario permitió identificar tanto patrones de diseño consistentes como casos atípicos valiosos para orientar futuras refactorizaciones.

Desde el punto de vista computacional, los tiempos de inferencia de los modelos generativos y de verificación en Lean demostraron ser compatibles con la naturaleza iterativa del enfoque, evidenciando la factibilidad de integrar verificación formal en ciclos evolutivos rápidos.

A pesar de estas fortalezas, se identificaron limitaciones asociadas al alcance del dataset, a la naturaleza estática de las métricas consideradas y a la dependencia tecnológica del pipeline. Estas limitaciones no restan validez a los resultados obtenidos, pero señalan la necesidad de ampliar la validación a otros contextos, incorporar métricas dinámicas y explorar la portabilidad del sistema a nuevos modelos y entornos.

En síntesis, los resultados de este capítulo aportan evidencia empírica clara de que ArchiGenMS cumple con los objetivos planteados: generar arquitecturas de microservicios más cohesionadas, menos acopladas y verificadas formalmente, constituyendo un aporte significativo frente a los enfoques tradicionales. Este balance sienta las bases para las conclusiones generales de la tesis y la identificación de líneas futuras de investigación, que se desarrollarán en el siguiente capítulo.

Conclusiones y Líneas Futuras

Habiendo completado la construcción (Capítulo 6) y la validación empírica (Capítulo 7) del artefacto ArchiGenMS, este capítulo final cierra la presente investigación doctoral. El propósito es sintetizar la totalidad del trabajo, articular las contribuciones teóricas, metodológicas y prácticas alcanzadas, discutir con transparencia las limitaciones inherentes al estudio y, finalmente, delinear las líneas futuras de investigación que se desprenden de esta contribución. El capítulo se estructura en torno a la síntesis general, el detalle de las aportaciones, el análisis de validez y la prospectiva de trabajos futuros.

7.7. Síntesis general de la investigación

Esta tesis completó un ciclo metodológico de *Design Science Research* (DSR) [26]. Se partió del **Ciclo de Relevancia** al abordar un problema central de la ingeniería de software contemporánea: el *descubrimiento y validación* de microservicios a partir de requisitos textuales. Frente a enfoques manuales no verificables, el **Ciclo de Diseño** entregó el artefacto **ArchiGenMS**: un *pipeline* que integra (i) generación evolutiva asistida por modelos de lenguaje, (ii) evaluación mediante métricas arquitectónicas formalizadas y (iii) verificación estructural automática con Lean. La propuesta se fundamentó en el **Ciclo de Rigor**, inscribiéndose en la confluencia de tres líneas: el diseño de microservicios (principios de calidad), el uso de IA generativa (derivación desde texto) y el empleo de métodos formales (garantía de corrección y trazabilidad) [31, 32, 19, 18, 30].

Empíricamente, el Capítulo 7 mostró que ArchiGenMS converge de manera estable en 22 escenarios del *dataset* de Dalpiaz [33], reduciendo en torno a un 44 % la función de aptitud global en cinco generaciones, con disminuciones concomitantes de la cohesión media (LCOM) y del acoplamiento máximo. El análisis no paramétrico (Kruskal–Wallis y Dunn con corrección de Holm) confirmó la significancia de la mejora intergeneracional, y la estabilidad temporal evidenció la factibilidad de integrar verificación formal en ciclos evolutivos rápidos. Adicionalmente, el análisis cualitativo de artefactos específicos (como el caso `g24-unibath` detallado en la Sección 7.3.2) demostró que las arquitecturas generadas no solo son **métricamente óptimas** (validadas por Lean), sino también **lógicamente coherentes** y alineadas con el flujo del dominio funcional. Este balance valida el objetivo general: *generar arquitecturas de microservicios más cohesionadas, menos acopladas y verificadas formalmente a partir de requisitos textuales*, consolidando el aporte metodológico y práctico de la tesis frente al estado del arte [11, 12, 19].

7.8. Aportaciones principales

Contribuciones teóricas

En primer lugar, se ofreció una **formalización operativa** de métricas arquitectónicas de uso extendido—LCOM, SGM, SGM-SD y *Fan-Out*—como funciones totales sobre representaciones estructuradas de microservicios y sus llamadas. La elección de estas métricas responde tanto a su relevancia para el diseño de microservicios como a su claridad semántica y computabilidad [31, 32, 12]. La formalización en Lean [30] mejoró la precisión semántica y habilitó pruebas *machine-checkable* de propiedades estructurales (p. ej., inexistencia de auto-llamadas y referencias inválidas), estableciendo un puente explícito entre definiciones matemáticas y cálculo efectivo.

Contribuciones metodológicas

Se definió un **esquema evolutivo asistido por LLMs** que orquesta: muestreo incremental de *prompts* con "inspiraciones" (reciclaje de soluciones previamente validadas), generación de candidatos en formato JSON, evaluación métrica y verificación formal, y selección/variación guiadas por función objetivo. Esta integración aporta dos elementos diferenciales frente a trabajos previos en microservicios asistidos por IA [11, 12, 51, 18]: (i) la *verificabilidad* iterativa—cada generación queda sujeta a chequeos estructurales y métricos reproducibles—y (ii) la *sinergia* entre exploración generativa y restricciones formales, que reduce la deriva semántica habitual de los LLMs y estabiliza la convergencia (en línea conceptual con agentes evolutivos recientes [29], pero aterrizado al diseño arquitectónico).

Contribuciones prácticas

Se implementó un **artefacto reproducible**—ArchiGenMS—que automatiza extremo a extremo el proceso: del texto a candidatos de arquitectura, de candidatos a verificación y métricas, y de métricas a selección evolutiva. El paquete público de replicación [64] incluye código, *notebooks*, figuras y tablas; las salidas intermedias (`program_db.json`) y finales (`best_*.json`) permiten auditoría y reutilización. La evaluación en 22 escenarios del *dataset* de referencia [33] aportó evidencia amplia y trazable, más allá de casos ad hoc, y se mantuvo consistente con los reportes previos desarrollados durante la investigación (CACIC 2024, CIbSE 2025-Doctoral Symposium, y MDPI Software 2025 [20, 21, 18]).

Contribuciones académicas

La trayectoria de publicaciones durante el doctorado—mapeo y revisión sistemáticos [20, 18], comunicación temprana del enfoque [21] y consolidación de ArchiGenMS—permitió **tejer continuidad** entre diagnóstico del vacío (falta de verificabilidad en generación asistida), definición del método y validación empírica. Esta tesis integra y amplía dichas contribuciones, articulando un marco coherente que vincula IA generativa, métricas y verificación formal para el diseño de microservicios.

7.9. Limitaciones del estudio

Convención. Usamos AI4SE (*Artificial Intelligence for Software Engineering*) para referirnos al campo de IA aplicada a IS a lo largo de esta sección.

Las conclusiones deben interpretarse a la luz de varias limitaciones y amenazas a la validez, que definen el alcance de esta investigación y marcan el camino para trabajos futuros.

Validez de Construcción (Alcance de Métricas)

La validez de construcción de esta tesis se centra en la brecha entre el constructo teórico (calidad arquitectónica) y su medición (las métricas estáticas LCOM, SGM, SGM-SD y Coupling [31, 12, 32]). Estas métricas, ampliamente citadas, son indicadores fundamentales de la **calidad estructural estática**. Esta tesis demuestra que ArchiGenMS optimiza dicha estructura, generando arquitecturas cohesivas y desacopladas.

El alcance de la validación, por tanto, no incluye constructos de **calidad operacional dinámica**, como la tolerancia a fallos, el rendimiento bajo carga (latencia, *throughput*) o la escalabilidad horizontal [31]. Esta es una **delimitación metodológica deliberada**. La optimización de la calidad estructural es un prerrequisito fundamental antes de que la calidad operacional pueda ser evaluada. ArchiGenMS valida el *plano* (el `blueprint`) arquitectónico; la validación de su comportamiento en ejecución (el edificio bajo carga) se plantea como la línea futura más crítica de esta investigación.

Delimitación respecto a métricas de pruebas

Esta tesis no utiliza *cobertura* como proxy de calidad del diseño arquitectónico. La evidencia empírica sugiere que la cobertura no está fuertemente correlacionada con la efectividad del conjunto de pruebas [41]. Dado que el foco aquí es la *calidad estructural estática* previa a la implementación, priorizamos métricas arquitectónicas (LCOM, SGM, *fan-out*) y verificación estructural, dejando las métricas de pruebas para etapas de validación del comportamiento en ejecución. Esta delimitación dialoga con las líneas futuras de simulación dinámica de NFRs planteadas en la Sección 7.10.

Validez Externa (Cobertura de Datos)

Los experimentos se realizaron exclusivamente sobre el dataset de historias de usuario publicado por Dalpiaz [33]. Este conjunto es ampliamente reconocido en la comunidad de ingeniería de requisitos, pero se limita a descripciones en inglés y a dominios específicos (ciencia ciudadana, gobierno abierto, *etc.*). Por tanto, la extrapolación de los resultados a otros contextos —por ejemplo, dominios industriales con requisitos más técnicos o en otros idiomas— debe realizarse con cautela.

Validez Interna (Hiperparámetros y Estocasticidad)

Aunque se fijaron semillas aleatorias y se repitieron las ejecuciones, el carácter estocástico del ciclo evolutivo implica variabilidad en los resultados. Esta amenaza se mitiga mediante el reporte de medias, intervalos de confianza y pruebas estadísticas no paramétricas (Kruskal–Wallis y Dunn), que confirman la robustez de las diferencias. Sin embargo, no puede descartarse la influencia de la configuración de hiperparámetros (p. ej., temperatura del LLM, pesos de la función objetivo), cuya búsqueda sistemática (*grid*/*bayesiana*) queda fuera del alcance de esta validación inicial.

Amenazas Tecnológicas (Dependencia)

Finalmente, la dependencia del pipeline respecto a un modelo de lenguaje específico (`gpt-4o-mini`) y a la versión concreta de Lean (v4.22.0-rc3) introduce un sesgo tecnológico. El paquete de replicación [64] garantiza la reproducibilidad en las condiciones exactas reportadas, mitigando esta amenaza.

Delimitación disciplinar dentro de AI4SE

El trabajo se circunscribe a AI4SE para *diseño arquitectónico greenfield*. No abordamos líneas consolidadas de AI4SE como la reparación automática de programas [39], la predicción de defectos con aprendizaje profundo [40] o la evaluación de modelos de lenguaje entrenados en código orientada a tareas de programación [42]. Estas áreas son complementarias y coherentes con la visión de un ciclo futuro donde señales estructurales guíen estrategias de prueba, mantenimiento y refactorización automática.

7.10. Líneas futuras de investigación

A partir de los hallazgos y de las limitaciones discutidas en la Sección 7.9, se abren varias direcciones promisorias que extienden lógicamente el trabajo realizado:

1. **Extensión de Validez Externa (Datasets y Multilingüe).** Abordar la limitación de cobertura de datos mediante la incorporación de repositorios industriales y *datasets* multilingües (p. ej., español/portugués). Esto permitirá evaluar la robustez de ArchiGenMS con historias de usuario de distinta granularidad y estilo, y afinar la función objetivo para dominios con alta heterogeneidad funcional (como `openspending` y `zooniverse`, que mostraron LCOM más altos).

2. **De la Verificación Estructural a la Simulación Dinámica (NFRs).** Esta es la línea de extensión más crítica, abordando directamente la delimitación de las métricas estáticas. Se propone evolucionar el pipeline "generar–verificar" hacia un pipeline "generar–verificar–simular". Habiendo establecido la capacidad de ArchiGenMS para optimizar la *calidad estructural estática* (el foco de esta tesis), el siguiente paso es evaluar la *calidad operacional*. Los candidatos que superen la verificación formal en Lean servirían como entrada para una segunda etapa de evaluación que podría (a) instanciar la arquitectura en un entorno de orquestación (ej., Kubernetes) o (b) modelarla en un simulador de sistemas distribuidos. Esto permitiría evaluar NFRs dinámicos (latencia, tolerancia a fallos) y usar dichos resultados para retroalimentar la función de aptitud.

3. **Optimización del *prompting* y Ensamblados de Modelos.** Explorar *ensembles* de LLMs y técnicas de *prompt chaining/self-refine* para reducir la varianza y las *hallucinations*. La estrategia de *inspiraciones* puede reforzarse con memoria a largo plazo de patrones arquitectónicos válidos y detección de "antipatrones" antes de invocar la verificación, acercándose a un ciclo *critic–refine*.

4. **Automatización Formal Avanzada.** Profundizar el uso de Lean para demostrar propiedades globales (p. ej., aciclicidad en el grafo de dependencias, invariantes de separación de

contextos), e investigar marcos algebraicos para la composición segura de servicios. La co-evolución "métrica + prueba" es una vía para que el propio proceso evolutivo aprenda a proponer candidatos "más demostrables".

5. **Transferencia Tecnológica y Validación Industrial.** La validación final de un artefacto DSR reside en su aplicación práctica. Esto implica dos desafíos: (a) un desafío de *ingeniería*, codificando ArchiGenMS como un servicio (ej., un *GitHub Action*) integrable en *pipelines* CI/CD industriales; y (b) un desafío de *investigación*, extendiendo el *Prompt Sampler* para manejar la complejidad de los artefactos de requisitos del mundo real, que raramente son *user stories* limpias, sino una mezcla de documentos extensos, diagramas UML y especificaciones técnicas.

7.11. Cierre

La tesis ha mostrado que la **combinación** de generación evolutiva asistida por LLMs, métricas arquitectónicas formalizadas y verificación automática ofrece una *ruta pragmática y rigurosa* para pasar de requisitos textuales a arquitecturas de microservicios de mayor calidad. ArchiGenMS no sustituye la labor del arquitecto, pero la potencia: hace visible la estructura, acota la ambigüedad y documenta decisiones con evidencias reproducibles. Al situar métodos formales en el corazón del ciclo generativo—sin penalizar la iteración—esta investigación contribuye a tender un puente entre la flexibilidad de la IA generativa y las garantías que exige la ingeniería de software. Consideramos que este es un paso necesario hacia herramientas de diseño *auditables*, capaces de sostener arquitecturas confiables en entornos cada vez más distribuidos y complejos [19, 18].

Finalmente, el trabajo invita a la comunidad a explorar—con datos abiertos, paquetes reproducibles y evaluación rigurosa—cómo la sinergia entre IA y verificación formal puede transformar no sólo *qué* arquitecturas construimos, sino *con qué grado de certeza* las aceptamos para sistemas críticos del mundo real.

Bibliografía

[1] Sam Newman. *Building microservices: designing fine-grained systems*. .º'Reilly Media, Inc.", 2021.

[2] Edmund M Clarke y Jeannette M Wing. "Formal methods: State of the art and future directions". En: *ACM Computing Surveys (CSUR)* 28.4 (1996), págs. 626-643.

[3] Victor Velepucha y Pamela Flores. "A survey on microservices architecture: Principles, patterns and migration challenges". En: *IEEE access* 11 (2023), págs. 88339-88358.

[4] Davide Taibi, Valentina Lenarduzzi y Claus Pahl. "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation". En: *IEEE Cloud Computing* 4.5 (2017), págs. 22-32.

[5] Davide Taibi, Valentina Lenarduzzi, Claus Pahl y Andrea Janes. "Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages". En: *Proceedings of the XP2017 Scientific Workshops*. 2017, págs. 1-5.

[6] Hüseyin Ünlü, Dhia Eddine Kennouche, Görkem Kılınç Soylu y Onur Demirörs. "Microservice-based projects in agile world: A structured interview". En: *Information and Software Technology* 165 (2024), pág. 107334.

[7] Saša Baškarada, Vivian Nguyen y Andy Koronios. "Architecting microservices: Practical opportunities and challenges". En: *Journal of Computer Information Systems* (2020).

[8] Nicolás Battaglia, Agustín Nicolás García y Augusto Congiusti. "Descubrimiento de Microservicios en Metodologías Ágiles: un mapeo sistemático de la literatura". En: *XXX Congreso Argentino de Ciencias de la Computación (CACIC)(La Plata, 7 al 11 de octubre de 2024)*. 2024.

[9] Anup K Kalia, Jin Xiao, Rahul Krishna, Saurabh Sinha, Maja Vukovic y Debasish Banerjee. "Mono2micro: a practical and effective tool for decomposing monolithic java applications to microservices". En: *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 2021, págs. 1214-1224.

[10] Vikram Nitin, Shubhi Asthana, Baishakhi Ray y Rahul Krishna. "Cargo: Ai-guided dependency analysis for migrating monolithic applications to microservices architecture". En: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 2022, págs. 1-12.

[11] Fredy H Vera-Rivera, Eduard Gilberto Puerto Cuadros, Boris Perez, Hernán Astudillo y Carlos Gaona. "SEMGROMI—a semantic grouping algorithm to identifying microservices using semantic similarity of user stories". En: *PeerJ Computer Science* 9 (2023), e1380.

[12] Deepali Bajaj, Anita Goel y Suresh Chand Gupta. "GreenMicro: identifying microservices from use cases in greenfield development". En: *IEEE Access* 10 (2022), págs. 67008-67018.

[13] Ruoyu Su, Xiaozhou Li y Davide Taibi. "Back to the future: From microservice to monolith". En: *arXiv preprint arXiv:2308.15281* (2023).

[14] Davide Neri, Jacopo Soldani, Olaf Zimmermann y Antonio Brogi. "Design principles, architectural smells and refactorings for microservices: a multivocal review". En: *SICS Software-Intensive Cyber-Physical Systems* 35.1 (2020), págs. 3-15.

[15] Nuha Alshuqayran, Nour Ali y Roger Evans. "A systematic mapping study in microservice architecture". En: *2016 IEEE 9th international conference on service-oriented computing and applications (SOCA)*. IEEE. 2016, págs. 44-51.

[16] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[17] Chenxing Zhong, Shanshan Li, Huang Huang, Xiaodong Liu, Zhikun Chen, Yi Zhang y He Zhang. "Domain-driven design for microservices: An evidence-based investigation". En: *IEEE Transactions on Software Engineering* 50.6 (2024), págs. 1425-1449.

[18] Daniel Narváez, Nicolas Battaglia, Alejandro Fernández y Gustavo Rossi. "Designing microservices using ai: A systematic literature review". En: *Software* 4.1 (2025), pág. 6.

[19] Matteo Esposito, Xiaozhou Li, Sergio Moreschini, Noman Ahmad, Tomas Cerny, Karthik Vaidhyanathan, Valentina Lenarduzzi y Davide Taibi. "Generative AI for Software Architecture. Applications, Trends, Challenges, and Future Directions". En: *arXiv preprint arXiv:2503.13310* (2025).

[20] Daniel Narváez, Gustavo Héctor Rossi y Nicolás Battaglia. "Aplicación de Inteligencia Artificial en el Diseño de Microservicios". En: *XXX Congreso Argentino de Ciencias de la Computación (CACIC)(La Plata, 7 al 11 de octubre de 2024)*. 2024.

[21] Daniel Narváez. "Explorando el Uso de Inteligencia Artificial en el Descubrimiento y Diseno de Microservicios". En: *Congresso Ibero-Americano em Engenharia de Software (CIbSE)*. SBC. 2025, págs. 224-231.

[22] Lauren Adams, Francis Boyle, Patrick Boyle, Dario Amoroso d'Aragona, Tomas Cerny y Davide Taibi. "ChatGPT for Microservice Development: How far can we go?" En: *International Conference on Microservices*. 2023.

[23] Tatjana Stojanovic y Saša D Lazarević. "The application of ChatGPT for identification of microservices". En: *E-business technologies conference proceedings*. Vol. 3. 1. 2023, págs. 99-105.

[24] Rudra Dhar, Karthik Vaidhyanathan y Vasudeva Varma. "Can llms generate architectural design decisions?-an exploratory empirical study". En: *2024 IEEE 21st International Conference on Software Architecture (ICSA)*. IEEE. 2024, págs. 79-89.

[25] Lenz Belzner, Thomas Gabor y Martin Wirsing. "Large language model assisted software engineering: prospects, challenges, and a case study". En: *International conference on bridging the gap between AI and reality*. Springer. 2023, págs. 355-374.

[26] Alan R Hevner, Salvatore T March, Jinsoo Park y Sudha Ram. "Design science in information systems research". En: *MIS quarterly* (2004), págs. 75-105.

[27] Aline Dresch, Daniel Pacheco Lacerda y José Antônio Valle Antunes Jr. "Design science research". En: *Design science research: A method for science and technology advancement*. Springer, 2014, págs. 67-102.

[28] Daniel Narváez, Nicolás Battaglia, Alejandro Fernández y Gustavo Héctor Rossi. "Descubrimiento automático de microservicios mediante modelos generativos y verificación formal". En: *Libro de actas - XXXI Congreso Argentino de Ciencias de la Computación - CACIC 2025*. Ed. por Mauro Germán Cambarieri, Carlos Fabián Lugani y Nicolás García Martínez. Viedma, Argentina: Universidad Nacional de Río Negro (UNRN), 2025, págs. 597-606. ISBN: 978-987-8258-99-7. URL: http://sedici.unlp.edu.ar/handle/10915/191127.

[29] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian et al. "AlphaEvolve: A coding agent for scientific and algorithmic discovery". En: *arXiv preprint arXiv:2506.13131* (2025).

[30] Leonardo de Moura y Sebastian Ullrich. "The lean 4 theorem prover and programming language". En: *International Conference on Automated Deduction*. Springer. 2021, págs. 625-635.

[31] Omar Al-Debagy y Peter Martinek. "A metrics framework for evaluating microservices architecture designs". En: *Journal of Web Engineering* 19.3–4 (2020), págs. 341-370.

[32] Davide Taibi y Kari Systä. *A Decomposition and Metric-Based Evaluation Framework for Microservices*. 2019. arXiv: 1908.08513 [cs.SE]. URL: https://arxiv.org/abs/1908.08513.

[33] Fabiano Dalpiaz. *Requirements data sets (user stories)*. Mendeley Data, jul. de 2018. DOI: 10.17632/7zbk8zsd8y.1. URL: https://doi.org/10.17632/7zbk8zsd8y.1.

[34] Daniel Narváez, Nicolás Battaglia, Alejandro Fernández y Gustavo Rossi. "Aplicación de Inteligencia Artificial Generativa y Verificación Formal en el Descubrimiento de Microservicios". En: *Revista Abierta de Informática Aplicada* 9.1 (2025), págs. 2-24.

[35] Mark Harman, S Afshin Mansouri y Yuanyuan Zhang. "Search-based software engineering: Trends, techniques and applications". En: *ACM Computing Surveys (CSUR)* 45.1 (2012), págs. 1-61.

[36] Mehdi AIT SAID, Abdellah EZZATI, Soukaina MIHI y Lahcen BELOUADDANE. "Microservices adoption: An industrial inquiry into factors influencing decisions and implementation strategies". En: *International Journal of Computing and Digital Systems* 15.1 (2024), págs. 1417-1432.

[37] Hulya Vural, Murat Koyuncu y Sinem Guney. "A systematic literature review on microservices". En: *International Conference on Computational Science and Its Applications*. Springer. 2017, págs. 203-217.

[38] Stefan Haselböck, Rainer Weinreich y Georg Buchgeher. "An expert interview study on areas of microservice design". En: *2018 IEEE 11th Conference on service-oriented computing and applications (SOCA)*. IEEE. 2018, págs. 137-144.

[39] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury y Satish Chandra. "Semfix: Program repair via semantic analysis". En: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, págs. 772-781.

[40] Xinli Yang, David Lo, Xin Xia, Yun Zhang y Jianling Sun. "Deep learning for just-in-time defect prediction". En: *2015 IEEE International conference on software quality, reliability and security*. IEEE. 2015, págs. 17-26.

[41] Laura Inozemtseva y Reid Holmes. "Coverage is not strongly correlated with test suite effectiveness". En: *Proceedings of the 36th international conference on software engineering*. 2014, págs. 435-445.

[42] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever y Wojciech Zaremba. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: `2107.03374 [cs.LG]`. URL: `https://arxiv.org/abs/2107.03374`.

[43] Aldeida Aleti, Barbora Buhnova, Lars Grunske, Anne Koziolek e Indika Meedeniya. "Software architecture optimization methods: A systematic literature review". En: *IEEE Transactions on Software Engineering* 39.5 (2012), págs. 658-683.

[44] Imen Trabelsi, Manel Abdellatif, Abdalgader Abubaker, Naouel Moha, Sébastien Mosser, Samira Ebrahimi-Kahou y Yann-Gaël Guéhéneuc. "From legacy to microservices: A type-based approach for microservices identification using machine learning and semantic analysis". En: *Journal of Software: Evolution and Process* 35.10 (2023), e2503.

[45] Garm Lucassen, Fabiano Dalpiaz, Jan Martijn EM van der Werf y Sjaak Brinkkemper. "Improving agile requirements: the quality user story framework and tool". En: *Requirements engineering* 21.3 (2016), págs. 383-403.

[46] Garm Lucassen, Fabiano Dalpiaz, Jan Martijn EM van der Werf y Sjaak Brinkkemper. "The use and effectiveness of user stories in practice". En: *International working conference on requirements engineering: Foundation for software quality*. Springer. 2016, págs. 205-222.

[47] Takwa Kochbati, Shuai Li, Sébastien Gérard y Chokri Mraidha. "From user stories to models: A machine learning empowered automation". En: *MODELSWARD 2022-9th International Conference on Model-Driven Engineering and Software Development*. Vol. 1. SCITEPRESS-Science y Technology Publications. 2021, págs. 28-40.

[48] Gabriela Pérez, Catalina Mostaccio y Leandro Antonelli. "Análisis comparativo de arquitecturas de NLP para detectar similitudes entre escenarios en español". En: *Workshop on Requirements Engineering (WER)*. 2025.

[49] Deepali Bajaj, Urmil Bharti, Isha Gupta, Priya Gupta y Asha Yadav. "GTMicro—Microservice identification approach based on deep NLP transformer model for greenfield developments". En: *International Journal of Information Technology* 16.5 (2024), págs. 2751-2761.

[50] Jose Renan A Pereira, Danyllo Albuquerque, Mirko Perkusich, Guillermo Rodríguez, Jorge Andrés Díaz-Pace, Kyller Gorgônio y Angelo Perkusich. "Toward Generating Microservice Architectures from Textual Requirements with Large Language Models". En: *Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS)*. SBC. 2025, págs. 79-89.

[51] Yulong Li, Zhi Li, Yitao Bu, Hongbin Xiao y Yajun Deng. "Pf4md: A microservice decomposition tool combining problem frames". En: *2023 IEEE 31st International Requirements Engineering Conference (RE)*. IEEE. 2023, págs. 359-360.

[52] J Andrés Díaz-Pace, Antonela Tommasel y Rafael Capilla. "Helping novice architects to make quality design decisions using an llm-based assistant". En: *European Conference on Software Architecture*. Springer. 2024, págs. 324-332.

[53] EM Clarke, Orna Grumberg y Doron A Peled. "Model checking the mit press". En: *Cambridge, Massachusetts, London, UK* 988 (1999).

[54] Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[55] Leslie Lamport. *Specifying systems*. Vol. 388. Addison-Wesley Boston, 2002.

[56] James Lewis y Martin Fowler. "Microservices: a definition of this new architectural term (2014)". En: *URL: http://martinfowler. com/articles/microservices. html (cit. on p. 26)* (2014).

[57] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin y Larisa Safina. "Microservices: yesterday, today, and tomorrow". En: *Present and ulterior software engineering* (2017), págs. 195-216.

[58] Alireza Goli, Nima Mahmoudi, Hamzeh Khazaei y Omid Ardakanian. "A Holistic Machine Learning-based Autoscaling Approach for Microservice Applications." En: *Closer* 1 (2021), págs. 190-198.

[59] Mahmoud Imdoukh, Imtiaz Ahmad y Mohammad Gh Alfailakawi. "Machine learning-based auto-scaling for containerized applications". En: *Neural Computing and Applications* 32.13 (2020), págs. 9745-9760.

[60] BASEL Magableh y M Almiani. "Deep q learning for self adaptive distributed microservices architecture". En: *IEEE Acess* (2019).

[61] Alexander Power y Gerald Kotonya. "A microservices architecture for reactive and proactive fault tolerance in iot systems". En: *2018 IEEE 19th International Symposium on." World of Wireless, Mobile and Multimedia Networks"(WoWMoM)*. IEEE. 2018, págs. 588-599.

[62] Yu Gan, Mingyu Liang, Sundar Dev, David Lo y Christina Delimitrou. "Sage: practical and scalable ML-driven performance debugging in microservices". En: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2021, págs. 135-151.

[63] Claudia Pons, Ricardo Rosenfeld y Clara Patricia Smith. "Lógica para informática". En: *Series: Libros de Cátedra* (2017).

[64] Daniel Narváez. *ArchiGenMS: Reproducible Package*. `https://github.com/jdanieln/archi-ai`. Accedido: 22 jul. 2025. 2025.

[65] Fernando Hernández Hernández. *Teoría de conjuntos*. Sociedad Matematica Mexicana, 1998.

[66] Jonathan L Gross, Jay Yellen y Mark Anderson. *Graph theory and its applications*. Chapman y Hall/CRC, 2018.

[67] Kawin Ethayarajh. "How contextual are contextualized word representations". En: *Comparing the geometry of BERT, ELMo, and GPT-2 Embeddings* 2 (2019).

[68] Brian S Mitchell y Spiros Mancoridis. "On the evaluation of the bunch search-based software modularization algorithm". En: *Soft Computing* 12.1 (2008), págs. 77-93.

[69] Mark Harman, Robert M Hierons y Mark Proctor. "A New Representation And Crossover Operator For Search-based Optimization Of Software Modularization." En: *GECCO*. Vol. 2. 2002, págs. 1351-1358.

[70] Mark Harman y Xin Yao. "Software module clustering as a multi-objective search problem". En: *IEEE Transactions on Software Engineering* 37.2 (2010), págs. 264-282.

[71] Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb y Ali Ouni. "Many-objective software remodularization using NSGA-III". En: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24.3 (2015), págs. 1-45.

[72] Islem Saidani, Ali Ouni, Mohamed Wiem Mkaouer y Aymen Saied. "Towards automated microservices extraction using muti-objective evolutionary search". En: *International conference on service-oriented computing*. Springer. 2019, págs. 58-63.

[73] Matthéo Lecrivain, Hanifa Barry, Dalila Tamzalit y Houari Sahraoui. "MONO2REST: Identifying and Exposing Microservices: a Reusable RESTification Approach". En: *2025 IEEE/ACM 22nd International Conference on Software and Systems Reuse (ICSR)*. IEEE. 2025, págs. 33-43.

[74] M. Abouahmed y O. Ahmed. *Machine Learning in Microservices: Productionizing Microservices Architecture for Machine Learning Solutions*. Packt Publishing, Limited, 2023. ISBN: 9781804617748. URL: `https://books.google.com.ni/books?id=29WrzwEACAAJ`.

Apéndice A

Notación y convenciones

Este anexo fija la notación empleada a lo largo de la memoria:

- Un *genotipo arquitectónico* se denota por $\mathcal{G}$, e incluye el conjunto de microservicios $\mathcal{S}(\mathcal{G})$, las operaciones por servicio $\mathcal{O}(S)$, y el conjunto de llamadas $\mathcal{C}(\mathcal{G})$.
- Para cada servicio $S \in \mathcal{S}(\mathcal{G})$, denotamos su conjunto de operaciones por $O_s := \mathcal{O}(S)$.
- La cohesión de un servicio S se representa por $\mathrm{LCOM}(S) \in [0, 1]$, donde valores menores indican mayor cohesión (definido en §3.6) [31].
- La granularidad de un servicio S se representa por $\mathrm{SGM}(S) = |O_s|$ (definido en §3.6) [32].
- La dispersión funcional se aproxima con $\mathrm{SGM_SD}(S)$ (definido en §3.6).
- El acoplamiento saliente (*fan-out*) de un servicio S es $\mathrm{Coupling}(S) := |N^+(S)|$; el acoplamiento máximo del genotipo es $\mathrm{Coupling}_{\mathrm{máx}}(\mathcal{G}) = \mathrm{máx}_{S \in \mathcal{S}(\mathcal{G})} \mathrm{Coupling}(S)$ (definido en §3.6) [31].
- Denotamos por $\mathsf{Valid}(\mathcal{G})$ el predicado de *validez formal* devuelto por el verificador en Lean 4 [30].

Cuando se requiera normalizar una magnitud X al intervalo $[0, 1]$ se empleará la notación $\widehat{X}$.

Convenciones tipográficas. Código y fragmentos JSON/CLI se tipan en `monoespaciado` mediante el paquete `listings`. Las ecuaciones clave llevan etiqueta y son referenciadas desde el cuerpo principal.

Apéndice B

Definiciones operativas y función de evaluación

En este anexo se formalizan las métricas, la función objetivo (*fitness*) del pipeline evolutivo y las invariantes verificadas en Lean, referidas en los Capítulos 5 y 6.

Definiciones operativas

Cuadro B.1: Definiciones operativas utilizadas a lo largo de la tesis

Término	**Definición (Ver §3.6 para detalles formales)**
Cohesión (LCOM)	Medida de co-uso de parámetros entre operaciones en un servicio. Se usa una variante acotada en $[0, 1]$; valores menores indican mayor cohesión [31].
Granularidad (SGM)	Tamaño funcional del servicio, medido como el número de operaciones ($\lvert O_s \rvert$) por microservicio [32].
Dispersión (SGM-SD)	Desviación estándar de la cardinalidad de parámetros de las operaciones dentro de un servicio; alta dispersión sugiere heterogeneidad funcional.
Acoplamiento (Fan-Out)	Número de dependencias salientes (*fan-out*) de un servicio hacia otros; $\text{Coupling}_{\text{máx}}$ es el máximo por genotipo [31].
Verificación en Lean	Chequeo de invariantes estructurales e integridad de referencias mediante Lean 4 [30].

Función de Aptitud (Fitness)

Tal como se introdujo en la Sección 3.8 y se utilizó en la evaluación del Capítulo 7, la calidad de cada arquitectura $\mathcal{G}$ se evalúa mediante la función de aptitud $\mathcal{F}$.

La búsqueda está configurada para **minimizar** la suma lineal directa de las métricas estructurales clave:

$$\mathcal{F} = \text{LCOM_avg} + \text{SGM_max} + \text{SGM_SD_sum} + \text{Coupling_max}.$$

Si el genotipo es inválido estructuralmente (es decir, $\mathsf{Valid}(\mathcal{G})$ es falso tras la verificación en Lean), se le asigna una penalización grande (un valor de $\mathcal{F}$ artificialmente alto) para descartarlo de la selección.

Esta definición, que prescinde de normalizaciones y pesos (como α, β, γ), se eligió deliberadamente. Evita el *tuning* de hiperparámetros de la función objetivo y facilita un análisis de sensibilidad más claro sobre cómo cada métrica estructural (cohesión, granularidad, acoplamiento) contribuye al resultado final.

Invariantes verificadas en Lean 4

El verificador formal compila el modelo JSON a tipos internos y comprueba las siguientes invariantes [30], tal como se ilustra en el Listado 3.3:

1. **I1 (Integridad de referencias).** Toda llamada $(\texttt{caller}, \texttt{callee}) \in \mathcal{C}(\mathcal{G})$ refiere a servicios declarados en $\mathcal{S}(\mathcal{G})$.
2. **I2 (No `self-call`).** No existen llamadas (S, S) para ningún $S \in \mathcal{S}(\mathcal{G})$.
3. **I3 (Aciclicidad básica).** Aunque no implementado en el experimento final, el marco permite rechazar grafos de dependencias con ciclos.

Opcionalmente, cuando se declaran umbrales formales en YAML, el verificador añade *checks* de umbral:

$$\text{Coupling}_{\text{máx}}(\mathcal{G}) \leq C_{\text{máx}}^{\text{formal}}, \qquad \max_{S \in \mathcal{S}(\mathcal{G})} \text{SGM}(S) \leq S_{\text{máx}}^{\text{formal}}.$$

En caso de violación, se devuelve estado de error y el ciclo evolutivo penaliza el individuo.

Trazabilidad experimental

Cada evaluación produce un JSON con `status`, `message` y `metrics`, como el mostrado en el Listado B.1.

```
{
  "status": "OK",
  "message": "",
  "metrics": {
    "lcom_avg": 0.333,
    "sgm_max": 2.0,
    "sgm_sd_sum": 0.707,
    "coupling_max": 1
  }
}
```

Listado B.1: Ejemplo de artefacto JSON de salida del validador Lean.

Estos campos (específicamente, los valores del objeto `metrics`) se suman directamente para calcular la función de aptitud $\mathcal{F}$ que el algoritmo evolutivo minimiza. La configuración (semillas, umbrales, etc.) queda registrada en los artefactos de orquestación, asegurando replicabilidad (Cap. 6; [64]).

Agradecimientos

La realización de esta tesis doctoral ha sido un trayecto extenso y complejo. Aunque la investigación es, en esencia, un esfuerzo individual, su culminación no habría sido posible sin el apoyo, la guía y la visión de personas e instituciones a quienes deseo expresar mi más profundo agradecimiento.

En primer lugar, a mi director, el Dr. Gustavo Rossi. Su guía incansable y su rigor metodológico han sido un pilar constante en cada etapa de este proceso. Le agradezco su enorme paciencia, su disponibilidad permanente y sus revisiones detalladas, que fueron cruciales para darle forma y rigor a esta investigación, asegurando la coherencia metodológica y la calidad de las publicaciones que la sustentan. Su tutela me proporcionó el andamiaje académico necesario para llevar este trabajo a término.

A mi co-director, el Dr. Alejandro Fernández, le agradezco su contribución decisiva en un punto pivotal de la investigación. Ante una encrucijada metodológica, su visión estratégica fue esencial para discernir y seleccionar la línea de trabajo que demostró ser la más innovadora. Esta orientación definió el rumbo y se consolidó como la contribución metodológica central que da identidad a esta tesis.

Un agradecimiento especial merece el Dr. Carlos Neil. Su guía en las etapas iniciales de mi formación doctoral fue invaluable, sentando las bases metodológicas sobre las que se construyó este trabajo. Le agradezco, asimismo, por ser el catalizador y el nexo fundamental que propició el inicio de este proyecto doctoral.

Agradezco también a mis colegas, en especial a Nicolás Battaglia, por el fructífero trabajo colaborativo que resultó en varias de las publicaciones que fundamentan este manuscrito. Asimismo, extiendo mi gratitud a los docentes del Programa de Doctorado en Ciencias Informáticas; sus aportes en las distintas asignaturas enriquecieron indirectamente mi formación y la perspectiva de esta tesis. A los revisores anónimos de la revista *Software* (MDPI) y de los congresos CACIC y CIBSE, gracias por sus comentarios críticos que ayudaron a robustecer los argumentos aquí presentados.

Agradezco el soporte institucional de la Universidad Abierta Interamericana (UAI) y al Centro de Altos Estudios en Tecnología Informática (CAETI). El reconocimiento otorgado por el CAETI al galardonar este trabajo con el Primer Lugar en la Categoría de Trabajos de Posgrado en el XVI Congreso Internacional de Innovación Tecnológica (CIITI-TE 2025) constituyó un valioso incentivo y una importante validación temprana sobre la relevancia de esta línea de investigación.

Finalmente, en un plano más personal, esta tesis no habría sido sostenible sin la paciencia y el aliento de mi familia y mi compañera de vida, Helen Ampié. Su comprensión incondicional durante las largas jornadas de investigación, experimentación y redacción de este manuscrito fue el soporte fundamental que me permitió perseverar hasta alcanzar este objetivo.