Solución de Automatización de Pruebas BDD Dirigida por Metadatos y Agentes Inteligentes para la Optimización de la Mantenibilidad y Portabilidad

Dario Leonardo Narváez Jácome

Proyecto de grado entregado para obtener el título de
Magister en Ingeniería de Software

Dirigida por
Mgtr. Juan Pablo García Cifuentes

Pontificia Universidad Javeriana Cali
Facultad de Ingeniería y Ciencias
Maestría en Ingeniería de Software
Santiago de Cali
18 de junio de 2026

Santiago de Cali, 18 de Junio de 2026.

Señores
Pontificia Universidad Javeriana Cali.
Ph.D. Luisa Rincón Directora
Maestría en Ingeniería de Software.
Cali.

Cordial Saludo.

Por medio de la presente hago constar que en mi calidad de director de trabajo de grado he revisado el proyecto titulado “Solución de Automatización de Pruebas BDD Dirigida por Metadatos y Agentes Inteligentes para la Optimización de la Mantenibilidad y Portabilidad” realizado por el estudiante de Magister en Ingeniería de Software Dario Leonardo Narváez Jácome (cod: 8944265), el cual se encuentra terminado y considero que cumple con los requisitos para ser sustentado.

Atentamente,

Juan Pablo García Cifuentes

Firmado digitalmente por Juan Pablo García Cifuentes
Fecha: 2026.06.19 00:00:10 -05'00'

Msc. Juan Pablo García Cifuentes

Santiago de Cali, 19 de Junio de 2026.

Señores
Pontificia Universidad Javeriana Cali
Ph.D. Luisa Rincón
Directora Maestría en Ingeniería de Software
Cali.

Cordial Saludo.

Me permito presentar a su consideración el proyecto de grado titulado "Solución de Automatización de Pruebas BDD Dirigida por Metadatos y Agentes Inteligentes para la Optimización de la Mantenibilidad y Portabilidad" con el fin de cumplir con los requisitos exigidos por la Universidad y para que sea sometido a revisión del jurado y cumpla su aprobación, para conseguir posteriormente el título de Magister en Ingeniería de Software.

Atentamente,

Dario Leonardo Narváez Jácome
Código: 8944265

Ficha Resumen
Trabajo de Grado Maestría en Ingeniería de Software

TÍTULO: Solución de Automatización de Pruebas BDD Dirigida por Metadatos y Agentes Inteligentes para la Optimización de la Mantenibilidad y Portabilidad

1. Énfasis: Ingeniería de Software
2. Área de trabajo: Automatización de Pruebas
3. Tipo de proyecto: Aplicado
4. Estudiante: Dario Leonardo Narváez Jácome
5. Correo electrónico: novodaro@javerianacali.edu.co
6. Dirección y teléfono: Calle 1 Oeste 52-370 Apto 215. Cel. 3126708373
7. Director: Juan Pablo García Cifuentes
8. Vinculación del director: Cátedra
9. Correo electrónico del director: jpgarcia@javerianacali.edu.co
10. Palabras clave: MCP, BDD, IA, Metadatos, Automatización, QA
11. ODS que aplica al proyecto (Agenda 2030): ODS 8 - Trabajo decente y crecimiento económico.
12. Fecha de inicio: 06/Febrero/2026
13. Resumen: Esta investigación propone una solución de automatización BDD dirigida por metadatos y el protocolo MCP. Implementa un orquestador inteligente (LLM) para desacoplar la lógica de negocio de la ejecución técnica, permitiendo un funcionamiento agnóstico entre Selenium y Playwright. El objetivo final es optimizar la mantenibilidad y portabilidad de los activos de prueba bajo el estándar ISO/IEC 25010.

Agradecimientos

Hacer este proyecto fue, más que un requisito, un reto personal. Al no tener el afán de cumplir con un plazo estricto, la motivación principal fue el deseo de terminar algo que había empezado. A quienes estuvieron ahí durante este proceso, gracias.

A mi familia y a mi pareja, por entender las largas horas frente al computador, por la paciencia y por siempre confiar en que lo lograría.

A mi director, Juan Pablo García Cifuentes, por darme buenas ideas cuando el panorama no estaba claro y ayudarme a encontrarle el norte a este trabajo.

Al profesor Mario Mora, por darnos la estructura documental. Sin su guía, darle orden a todo lo que investigamos habría sido mucho más difícil.

A la profesora Luisa Rincón, por inspirarnos a retomar lo que había quedado pendiente y recordarnos que nunca es tarde para terminar un ciclo.

A los jurados que van a leer este documento, gracias por tomarse el tiempo de revisarlo y por las observaciones que seguro ayudarán a mejorarlo.

Entregar este proyecto hoy me demuestra que las metas personales, por más que se aplacen, siempre encuentran su momento cuando hay persistencia.

Resumen

Este proyecto propone y evalúa una solución de automatización de pruebas BDD dirigida por metadatos (*Meta-Driven Testing* o MDT), orquestada por agentes inteligentes locales mediante el *Model Context Protocol* (MCP). El propósito es superar la elevada carga de mantenimiento de enfoques convencionales como *Page Object Model* (POM). Validada empíricamente bajo la norma ISO/IEC 25010 sobre una aplicación transaccional (*eShop*), la solución demostró erradicar la fragilidad estructural ante mutaciones de interfaz (logrando un 100 % de auto-reparación) y brindó flexibilidad total para alternar motores de ejecución (Selenium/Playwright) sin alterar el código de prueba. Estos resultados proveen evidencia empírica inicial sobre la viabilidad técnica de integrar IA y abstracción semántica para optimizar los ciclos modernos de aseguramiento de calidad.

Palabras clave: Automatización de Pruebas, BDD, Inteligencia Artificial, Metadatos, Model Context Protocol, ISO/IEC 25010, Mantenibilidad, Auto-reparación.

Abstract

This project proposes and evaluates a metadata-driven BDD test automation solution (*Meta-Driven Testing* or MDT), orchestrated by local intelligent agents using the *Model Context Protocol* (MCP). The purpose is to overcome the high maintenance burden of conventional approaches such as *Page Object Model* (POM). Empirically validated under the ISO/IEC 25010 standard on a transactional application (*eShop*), the solution demonstrated the eradication of structural fragility against interface mutations (achieving 100 % self-healing) and provided total flexibility to alternate execution engines (Selenium/Playwright) without altering the test code. These results provide initial empirical evidence on the technical viability of integrating AI and semantic abstraction to optimize modern quality assurance cycles.

Keywords: Test Automation, BDD, Artificial Intelligence, Metadata, Model Context Protocol, ISO/IEC 25010, Maintainability, Self-healing.

Índice general

Agradecimientos **3**

1. Introducción **1**
1.1. Definición del problema . . . 1
1.1.1. Planteamiento del problema . . . 1
1.1.2. Sistematización . . . 2
1.2. Justificación del trabajo de grado . . . 2
1.3. Objetivos del proyecto . . . 3
1.3.1. Objetivo General . . . 3
1.3.2. Objetivos específicos . . . 3
1.4. Alcance del Proyecto . . . 3

2. Metodología del proyecto **5**
2.1. Fase 1: Diseño del Ecosistema de Metadatos (Objetivo Específico 1) . . . 5
2.2. Fase 2: Construcción del Orquestador Inteligente (Objetivo Específico 2) . . . 5
2.3. Fase 3: Evaluación y Validación Experimental (Objetivo Específico 3) . . . 6
2.4. Justificación Metodológica . . . 6
2.5. Resultados obtenidos . . . 6

3. Marco de referencia **7**
3.1. Bases Teóricas . . . 7
3.1.1. Modelo de Calidad de Software (ISO/IEC 25010) . . . 7
3.1.2. Arquitectura Genérica de Automatización de Pruebas (gTAA) . . . 7
3.1.3. Patrones de Diseño para la Automatización . . . 8
3.1.4. Enfoque Dirigido por Metadatos (Metadata-Driven Testing - MDT) . . . 8
3.1.5. Model Context Protocol (MCP) . . . 8
3.1.6. Behavior-Driven Development (BDD) y la Brecha del Glue Code . . . 9
3.2. Estado del Arte . . . 9
3.2.1. Evolución de las Arquitecturas: Del Scripting Lineal al Diseño Modular . . . 9
3.2.2. El Desafío del "Glue Code" en Behavior-Driven Development (BDD) . . . 10
3.2.3. Divergencia Tecnológica: Selenium vs. Playwright . . . 10
3.2.4. Enfoques Dirigidos por Metadatos y Herramientas Low-Code . . . 10
3.2.5. Resiliencia Semántica y Agentes de IA en la Industria . . . 11
3.2.6. Comparativa de Enfoques de Automatización Existentes . . . 11
3.3. Tecnologías Habilitadoras y Algoritmos Resilientes . . . 11
3.3.1. Orquestación Cognitiva: LangChain y LangGraph . . . 12
3.3.2. Generación de Localizadores Resilientes: Algoritmo Robula+ . . . 12

4. Diseño Técnico y Arquitectura **15**
4.1. Especificación de Requerimientos del Sistema . . . 15
4.1.1. Requerimientos Funcionales (RF) . . . 15
4.1.2. Requerimientos No Funcionales (RNF) . . . 16
4.2. Análisis del Protocolo de Interoperabilidad (MCP) . . . 16
4.2.1. Criterios de Selección y Gobernanza de los Servidores MCP Analizados . . . 16
4.2.2. Comparativa Arquitectónica de Servidores MCP . . . 17
4.3. Modelo de Metadatos . . . 18
4.3.1. Análisis Teórico y Justificación del Diseño . . . 18
4.3.2. Propuesta de Repositorio de Definiciones Técnicas . . . 19
4.4. Modelo de Contexto . . . 20
4.4.1. Estructura del Modelo Resuelto . . . 20
4.5. Arquitectura del Sistema (basado en el Modelo gTAA) . . . 21
4.5.1. Implementación Distribuida: Modelo Cliente-Servidor . . . 21
4.5.2. Especificación de Capas del Modelo Arquitectónico . . . 21
4.5.3. Trazabilidad Requerimientos-Arquitectura . . . 23
4.5.4. Decisión de Diseño: Estrategia de Localización Semántica vía XPath . . . 24
4.5.5. Dinámica Operativa Proyectada y Flujo de Interacción . . . 25
4.6. Selección del Modelo de Lenguaje y Estrategia de Cómputo Cognitivo . . . 27
4.6.1. Evaluación de Paradigmas de Despliegue: Modelos Abiertos frente a APIs Comerciales . . . 27
4.6.2. Evaluación y Selección del Entorno de Inferencia . . . 28
4.7. Filtrado y Preselección en Hugging Face Hub . . . 29
4.7.1. Análisis de Licencias y Conformación del Top 5 . . . 30

5. Construcción e Implementación **35**
5.1. Especificación del Ecosistema Híbrido Multiagente . . . 35
5.1.1. Symphony: Orquestador Híbrido Multi-Agente . . . 35
5.1.2. Argos: Agente Localizador y de Resiliencia . . . 37
5.1.3. Nexus: Agente de Adaptación y Validación Semántica . . . 38
5.2. Dinámica Operativa: Flujo de Interacción Híbrido Cliente-Servidor . . . 39

6. Análisis de Ejecuciones Experimentales y Evaluación ISO/IEC 25010 **43**
6.1. Diseño de Pruebas y Metodología Experimental . . . 43
6.1.1. Topología Distribuida del Entorno . . . 43
6.1.2. Automatización mediante Pipeline CI/CD . . . 44
6.1.3. Consolidación de Resultados y Auditabilidad . . . 46
6.1.4. Casos de Prueba y Estrategia de Mutación . . . 46
6.2. Alineación Operacional con ISO/IEC 25010:2023 . . . 47
6.3. Resultados Comparativos de Ejecución (MDT vs POM) . . . 48
6.4. Medición Operacional de Resultados (MDT vs POM) . . . 49

6.4.1. Resultados de Flexibilidad: Archivos y Líneas Modificadas 49
6.4.2. Resultados de Mantenibilidad: Tasa de Fragilidad ante Mutaciones 50
6.4.3. Resultados de Mantenibilidad: Intervención Humana Requerida 50
6.4.4. Usabilidad en el Mantenimiento 51
6.5. Análisis Adicional y Hallazgos Secundarios 52
6.5.1. Determinismo en la Validación de Negocio (Ejecución 8) 52
6.5.2. Análisis de Eficiencia y Consumo de Tokens (LLM) 52
6.5.3. Discusión Crítica: El Trade-off de la Automatización basada en LLM 53
6.6. Amenazas a la Validez 53
6.7. Síntesis del Capítulo y Respuesta al Objetivo 3 54

7. Conclusiones **55**
7.1. Conclusiones sobre el Diseño Arquitectónico y Metadatos (Objetivo 1) 55
7.2. Conclusiones sobre el Ecosistema Multiagente (Objetivo 2) 55
7.3. Conclusiones sobre la Validación Experimental frente a POM (Objetivo 3) 56
7.4. Trabajos Futuros 57
7.5. Lecciones aprendidas 57

Bibliografía **59**

Índice de figuras

4.1. Arquitectura multicapa propuesta basada en el modelo de referencia gTAA. 23
4.2. Diagrama de secuencia proyectado para el flujo operativo basado en el protocolo MCP. 26
4.3. Configuración de filtros base de búsqueda en Hugging Face. 30
4.4. Resultados de búsqueda en Hugging Face bajo licencia Apache 2.0. 31
4.5. Resultados de búsqueda en Hugging Face bajo licencia MIT. 32
4.6. Resultados de búsqueda en Hugging Face bajo licencia Llama 3. 32

5.1. Topología del grafo de estados para el orquestador Symphony. 36
5.2. Topología del grafo de estados para el agente localizador Argos. 37
5.3. Topología del grafo de estados para el agente de adaptación Nexus. 39
5.4. Diagrama de Actividad: Algoritmo de orquestación, manejo de caché y validación semántica. 40
5.5. Diagrama de Secuencia: Trazabilidad del flujo de comunicación entre el motor central y los agentes cognitivos especializados. 42

6.1. Arquitectura de red y flujos de comunicación en la topología distribuida del laboratorio. 44
6.2. Ejecución secuencial del Pipeline CI/CD en Jenkins orquestando el entorno experimental. 45
6.3. Histórico consolidado de ejecuciones experimentales (MDT vs POM). 48

Índice de tablas

3.1. Comparativa de Enfoques de Automatización y Diagnóstico de la Brecha Técnica . . 12

4.1. Requerimientos Funcionales 15
4.2. Requerimientos No Funcionales 16
4.3. Comparativa Arquitectónica de Servidores MCP: Playwright vs Selenium 17
4.4. Matriz de Trazabilidad Arquitectónica Conceptual 23
4.5. *Matriz Comparativa: Modelos Open Source vs. APIs Comerciales* 27
4.6. *Matriz Comparativa de Plataformas de Inferencia Local* 28
4.6. *Matriz Comparativa de Plataformas de Inferencia Local* 29
4.7. *Especificaciones Técnicas y Justificación Empírica de los Modelos Seleccionados* . . 33

6.1. Descripción de los Casos de Prueba (TC) ejecutados en el experimento. 47
6.2. Descripción de las mutaciones inyectadas en la interfaz de usuario del SUT. 47
6.3. Operacionalización de las características ISO/IEC 25010:2023 evaluadas en el experimento. 47
6.4. Matriz comparativa de resultados de ejecución por casos de prueba entre la arquitectura MDT y la línea base POM. 49
6.5. Esfuerzo de migración entre motores de ejecución: MDT vs POM. 49
6.6. Consolidado cuantitativo de resiliencia y fragilidad entre MDT y POM. 50
6.7. Cuantificación del esfuerzo de refactorización requerido ante cambios evolutivos de UI. 51
6.8. Consumo consolidado de tokens LLM por ejecución experimental. 52
6.9. Identificación y mitigación de amenazas a la validez de los resultados. 53

Capítulo 1

Introducción

1.1. Definición del problema

En el ecosistema de desarrollo de software contemporáneo, la adopción de metodologías ágiles y marcos de trabajo de **Entrega Continua (CI/CD)** ha convertido a la automatización de pruebas funcionales en una capacidad crítica. Como sostienen Humble and Farley (2010), pioneros del *Continuous Delivery*, es poco viable lograr ciclos de despliegue rápidos y fiables sin una suite de pruebas automatizada que proporcione retroalimentación inmediata, actuando como la red de seguridad que permite la integración constante de código. Según Capgemini et al. (2024), el 75 % de las organizaciones identifica la integración de la automatización de pruebas en CI/CD como un factor esencial para la entrega de calidad en equipos ágiles.

No obstante, la industria enfrenta la denominada **"Paradoja de la Automatización"**, concepto desarrollado por Graham and Fewster (2012): el esfuerzo requerido para mantener los activos técnicos aumenta de forma exponencial a medida que la suite de pruebas crece, llegando a comprometer la agilidad que el proceso pretende proporcionar. Según Graham, muchas organizaciones caen en la trampa de automatizar para ahorrar tiempo, pero terminan consumiendo ese ahorro (y recursos adicionales) en la reparación de scripts técnicos, lo que invalida el retorno de inversión (ROI) y genera un estancamiento en la calidad del software.

1.1.1. Planteamiento del problema

Esta problemática se fundamenta en tres deficiencias estructurales:

1. **Fragilidad de los Activos de Prueba y Deuda Técnica**: Existe un acoplamiento rígido entre la lógica de negocio y los identificadores técnicos de la interfaz de usuario. Meszaros (2007) tipifica este fenómeno como el Fragile Test Smell, una condición donde modificaciones menores en el sistema bajo prueba (SUT) invalidan una proporción significativa de la suite de automatización. Esta falta de mantenibilidad (según la norma ISO/IEC 25010 (ISO, 2023)) obliga a los equipos de calidad a dedicar gran parte de su capacidad técnica a la reparación de scripts, generando una ineficiencia organizacional que desvía recursos estratégicos hacia labores de corrección operativa. Adicionalmente, como se afirma en Capgemini et al. (2025) la industria ya se enfrenta a una creciente crisis de deuda tecnológica. El 56 % de las organizaciones reporta dificultades para validar integraciones con sistemas heredados. Al mismo tiempo, el 53 % menciona desafíos al probar componentes generados por IA.

2. **Limitación Técnica del 'Glue Code' en BDD:** Aunque el enfoque **Behavior-Driven Development (BDD)** busca mejorar la colaboración mediante el uso de lenguaje natural, su implementación convencional depende de una capa crítica de código de acoplamiento o *glue code*. Según Wynne et al. (2017), el mantenimiento de estas definiciones se torna inmanejable a medida que el sistema escala, exigiendo una traducción manual a código por cada sentencia de negocio definida. Si se integra la evidencia recolectada por Binamungu et al. (2018), donde se identificó que el 40 % de los profesionales realiza procesos manuales para la detección y gestión de duplicidad en la implementación de BDD, y que un 17 % abandonó dicha tarea debido a la complejidad técnica subyacente, estos datos sugieren una tendencia hacia la generación de deuda técnica relevante que podría comprometer la mantenibilidad a largo plazo. En la práctica, esto implica que aunque BDD acerque el dominio, cada nueva expresión de negocio exige código técnico personalizado. Esta barrera técnica podría limitar a los analistas funcionales del ciclo técnico, contraviniendo las metas del **ODS 8 (Trabajo Decente y Crecimiento Económico)** al subutilizar el talento humano y elevar costos operativos.

3. **Dependencia Tecnológica y Rigidez de las APIs:** Los activos de prueba suelen estar vinculados con un alto grado de acoplamiento a las APIs de una herramienta específica (ej. Selenium). La carencia de **portabilidad** impide que un mismo escenario BDD sea ejecutable en motores de ejecución más eficientes (ej. Playwright) sin desechar la inversión previa. Esta rigidez representa un riesgo de obsolescencia para el negocio, dificultando la adopción de innovaciones que mejoren la competitividad. Como sostiene Smart and Molak (2023), el éxito de una estrategia BDD radica en la capacidad de separar las reglas de negocio de los detalles de implementación técnica a través de capas de abstracción bien definidas.

1.1.2. Sistematización

En este contexto, surge la siguiente pregunta:

¿Cómo diseñar e implementar una solución de automatización de pruebas BDD dirigida por metadatos que optimice la mantenibilidad y portabilidad de los activos técnicos, facilitando la participación de analistas funcionales y fortaleciendo la productividad organizacional?

1.2. Justificación del trabajo de grado

Ante este escenario no se encuentra una solución dirigida por metadatos que permita el desacoplamiento entre la intención de negocio y las APIs de ejecución (como Selenium o Playwright), cumpliendo con los estándares de interoperabilidad de la **gTAA** (ISTQB, 2024). Esta dependencia de código rígido podría comprometer la mantenibilidad del software, convirtiendo la evolución de las pruebas en un proceso de refactorización complejo y costoso en lugar de una gestión de configuración eficiente. Además, la dificultad de externalizar la complejidad técnica no solo impacta negativamente los atributos de calidad de la norma ISO/IEC 25010, sino que perpetúa una brecha técnica que excluye a los analistas funcionales del proceso de automatización.

1.3. Objetivos del proyecto

1.3.1. Objetivo General

Desarrollar una solución de automatización de pruebas BDD dirigida por metadatos e integrada con protocolos de contexto (MCP) para optimizar la mantenibilidad y portabilidad de los activos de automatización en aplicaciones web, permitiendo la ejecución agnóstica de pruebas frente a diferentes motores tecnológicos, con el fin de facilitar la participación de analistas funcionales y fortalecer la productividad organizacional.

1.3.2. Objetivos específicos

1. Diseñar un esquema de metadatos externo y un modelo de contexto compatible con MCP que centralice la relación entre las sentencias BDD, el paso a paso de las pruebas y los elementos de la interfaz, permitiendo el desacoplamiento de la lógica de prueba frente a los identificadores técnicos.

2. Construir un orquestador inteligente (Agente) que actúe como cliente del protocolo MCP, capaz de interpretar los metadatos y utilizar el razonamiento del LLM para seleccionar y ejecutar herramientas de automatización sobre servidores de *Selenium* y *Playwright* de manera intercambiable.

3. Evaluar la eficiencia de la solución propuesta en términos de mantenibilidad y portabilidad de los activos de automatización, mediante un estudio comparativo fundamentado en el estándar **ISO/IEC 25010** frente a métodos de automatización convencionales.

1.4. Alcance del Proyecto

Incluye:

- Arquitectura de Automatización (TAS): Documento de diseño de una arquitectura de capas basada en el estándar gTAA que integre un Agente de LLM como núcleo de decisión y un servidor MCP como capa de ejecución.

- Motor de Orquestación Resiliente: Código fuente de la herramienta capaz de procesar metadatos y resolver identificadores técnicos de forma dinámica. Si un selector falla, el motor consultará el modelo de contexto para intentar localizar el elemento por su propósito semántico.

- Modelo de Metadatos: Repositorio estructurado (JSON o YAML) que define la relación entre frases BDD y el paso a paso de las pruebas.

- Integración multiconector (Selenium y Playwright): Implementación de conectividad dual que permita al agente alternar entre motores de ejecución mediante herramientas estandarizadas por el protocolo MCP.

- Informe de Validación Experimental: Documento técnico con los resultados de la comparación (basada en ISO/IEC 25010) entre el modelo propuesto y el modelo tradicional.
- Guía de Usuario: Manual técnico para la configuración de metadatos y la creación de pruebas sin código.

No incluye:

- No incluye el desarrollo o entrenamiento de Large Lenguage Model(LLM) propios
- Automatización para aplicaciones móviles o de escritorio.
- No se abordarán pruebas de carga, estrés, rendimiento o ciberseguridad.
- Interfaz gráfica avanzada para gestión de scripts.
- Soporte multilenguaje (solo español).
- Funcionalidades de analítica predictiva.
- Automatización de Web Services
- El proyecto se enfoca en la arquitectura de ejecución y no en la configuración de flujos de despliegue continuo o granjas de servidores.
- Desarrollo de herramientas para el aprovisionamiento o limpieza de datos en bases de datos.

Capítulo 2

Metodología del proyecto

El presente proyecto se enmarca como una **investigación aplicada** de carácter **cuasi-experimental**, orientada al diseño, construcción y evaluación empírica de un artefacto tecnológico en el dominio de la ingeniería de software. Para su ejecución, se adoptó un enfoque de **Investigación-Acción**, dado que el investigador interviene directamente en el contexto del problema —construyendo la solución y evaluando sus efectos de manera iterativa— lo cual es consistente con la naturaleza transformadora del proyecto, donde no se busca únicamente observar un fenómeno, sino modificar activamente las prácticas de automatización de pruebas.

La gestión del ciclo de vida se realizó bajo el marco de trabajo ágil **Scrum**, organizando el esfuerzo en *Sprints* de tres semanas. Esta estructura iterativa e incremental permitió abordar la incertidumbre técnica inherente a la integración de agentes cognitivos basados en modelos de lenguaje, refinando progresivamente tanto la arquitectura como los mecanismos de resiliencia a partir de la retroalimentación obtenida en cada iteración.

El proceso investigativo y técnico se estructuró en tres fases principales, cada una alineada explícitamente con un objetivo específico del proyecto:

2.1. Fase 1: Diseño del Ecosistema de Metadatos (Objetivo Específico 1)

Esta fase inicial se enfocó en establecer la base teórica y arquitectónica de la solución. Comprendió el estudio de los protocolos de interoperabilidad, la evaluación de las herramientas habilitadoras y la estructuración del esquema de metadatos. El propósito fue definir un repositorio centralizado que permitiera desacoplar la intención de las pruebas de negocio de los selectores técnicos de las interfaces web, sentando las bases para la abstracción funcional del sistema.

2.2. Fase 2: Construcción del Orquestador Inteligente (Objetivo Específico 2)

Durante esta etapa se llevó a cabo el desarrollo de software de la herramienta. Consistió en la programación del motor de control y la integración de los agentes cognitivos encargados de interpretar los metadatos y ejecutar las acciones sobre los navegadores. Adicionalmente, se implementaron los mecanismos de resiliencia (*self-healing*) diseñados para mitigar la fragilidad de los localizadores ante cambios dinámicos en la interfaz.

2.3. Fase 3: Evaluación y Validación Experimental (Objetivo Específico 3)

La fase final se orientó a la contrastación empírica del sistema construido. Mediante la ejecución de un diseño experimental controlado sobre una aplicación de referencia, se midieron comparativamente las métricas de **Mantenibilidad** (esfuerzo de refactorización ante mutaciones deliberadas en la interfaz) y **Portabilidad** (intercambio de motores de ejecución subyacentes) frente al enfoque tradicional *Page Object Model* (POM). Los datos recolectados posibilitaron cuantificar la reducción del esfuerzo humano y verificar el impacto sobre la deuda técnica.

2.4. Justificación Metodológica

La combinación de Investigación-Acción con una gestión iterativa bajo Scrum resultó determinante para la madurez del artefacto. La estructura en sprints permitió que hallazgos tempranos —como la necesidad de aislar la nomenclatura de herramientas entre servidores MCP o la incorporación de un protocolo de caché semántico— fueran integrados incrementalmente sin comprometer la estabilidad del sistema. De este modo, el escenario experimental de la Fase 3 se ejecutó sobre un prototipo que ya había sido refinado a lo largo de múltiples iteraciones, fortaleciendo la validez interna de los resultados. El rigor del enfoque adoptado brindó el sustento metodológico necesario para evaluar formalmente los atributos de calidad definidos por la norma ISO/IEC 25010 (ISO, 2023).

2.5. Resultados obtenidos

A través del desarrollo de este proyecto, se implementó exitosamente una arquitectura de automatización de pruebas BDD dirigida por metadatos, validada empíricamente mediante los criterios de la norma ISO/IEC 25010. Los resultados demostraron que esta solución orquestada por agentes (Symphony, Argos, Nexus) superó de manera significativa al enfoque tradicional de Page Object Model (POM). En términos de mantenibilidad, la solución exhibió capacidades autónomas de *self-healing* que evitaron la refactorización manual ante mutaciones en la interfaz gráfica, reduciendo la deuda técnica. Asimismo, en cuanto a portabilidad, la arquitectura permitió intercambiar de forma transparente los motores de ejecución (Selenium y Playwright) a través del protocolo MCP, garantizando la independencia tecnológica de los activos funcionales.

Capítulo 3

Marco de referencia

La automatización de pruebas de software se define como el uso de herramientas y procesos para ejecutar casos de prueba sin intervención manual, con el objetivo de mejorar la eficiencia y reducir errores humanos. Este proceso es fundamental en entornos ágiles y DevOps, donde la velocidad y la calidad son factores críticos.

3.1. Bases Teóricas

3.1.1. Modelo de Calidad de Software (ISO/IEC 25010)

Para operacionalizar la mejora en los activos de automatización, este proyecto adopta el estándar **ISO/IEC 25010:2023** (System and software quality models) como marco de referencia. Específicamente, la solución se centra en dos características críticas que impactan directamente el Retorno de Inversión (ROI) de la automatización:

- **Mantenibilidad (Maintainability):** Se define como la capacidad del producto software para ser modificado efectiva y eficientemente. En el contexto de los scripts de prueba, el proyecto prioriza y evalúa empíricamente la subcaracterística de **Modificabilidad** (la reducción del esfuerzo humano y la resiliencia ante mutaciones en la interfaz gráfica) (ISO, 2023). La mantenibilidad es crítica dado que el mantenimiento de scripts frágiles consume recursos significativos en entornos ágiles (Juhnke et al., 2022).
- **Portabilidad (Portability):** Referida a la capacidad general de un sistema para ser transferido de un entorno a otro. Cabe aclarar que, según la actualización 2023 del estándar ISO/IEC 25010, este atributo de alto nivel se operacionaliza y evalúa técnicamente a través de la característica de **Flexibilidad** y su subcaracterística de **Capacidad de Reemplazo** (*Replaceability*), buscando que la lógica de prueba pueda persistir incluso si el motor de ejecución subyacente (ej. Selenium) es sustituido por otro (ej. Playwright) sin afectar la definición del test (ISO, 2023).

3.1.2. Arquitectura Genérica de Automatización de Pruebas (gTAA)

El diseño de la solución sigue los lineamientos de la **Arquitectura Genérica de Automatización de Pruebas (gTAA)** definida por el *International Software Testing Qualifications Board* (ISTQB). Esta arquitectura establece una separación estricta de responsabilidades en cuatro capas horizontales orientadas a favorecer la escalabilidad y el mantenimiento (ISTQB, 2024):

1. **Capa de Generación de Pruebas:** Herramientas y lógica para diseñar casos de prueba de manera automática o manual.
2. **Capa de Definición de Pruebas:** Donde se especifican los datos y secuencias de prueba de alto nivel, separando los datos de los scripts.
3. **Capa de Ejecución de Pruebas:** El motor encargado de interpretar las definiciones, ejecutar las reglas y registrar los resultados.
4. **Capa de Adaptación de Pruebas:** La interfaz técnica que interactúa directamente con el SUT (System Under Test) a través de drivers específicos.

3.1.3. Patrones de Diseño para la Automatización

La evolución de la mantenibilidad en las pruebas se explica a través de la transición de patrones de diseño, donde el proyecto propone una evolución hacia la abstracción semántica.

- **Page Object Model (POM):** Es el patrón clásico que modela las páginas web como clases orientadas a objetos. Aunque mejora la modularidad al separar los selectores de los tests, tiende a violar el *Principio de Responsabilidad Única (SRP)*, creando "God Classes" difíciles de mantener a gran escala cuando la interfaz cambia frecuentemente (Leotta et al., 2013).
- **Screenplay Pattern (Patrón Guion):** Una evolución del POM que aplica rigurosamente los principios SOLID (especialmente SRP y Open-Closed). En Screenplay, los usuarios son modelados como **Actores** que tienen **Habilidades** (ej. `NavegarWeb`) para realizar **Tareas** (ej. `Login`, `AgregarAlCarrito`) (Smart and Molak, 2023).

3.1.4. Enfoque Dirigido por Metadatos (Metadata-Driven Testing - MDT)

El MDT representa un cambio de paradigma de la programación imperativa ("cómo hacerlo") a la declarativa ("qué hacer"). En este modelo, la lógica de control, los objetos de la interfaz y los datos de prueba se extraen del código y se almacenan como metadatos estructurados (JSON, YAML, XML) (Bäuerle et al., 2024; Brambilla et al., 2017).

Un **Motor de Interpretación** (Runtime Engine) lee estos metadatos en tiempo de ejecución para invocar las funciones necesarias. Esto permite modificar el comportamiento de las pruebas sin necesidad de recompilar el código fuente, satisfaciendo la característica de *Modificabilidad* de la ISO 25010 y permitiendo una adaptación rápida ante cambios en la UI (Abughazala and Muccini, 2025).

3.1.5. Model Context Protocol (MCP)

El *Model Context Protocol* (MCP) es un estándar de arquitectura abierta diseñado para habilitar la interoperabilidad universal entre modelos de lenguaje de gran escala (LLMs) y sistemas externos que actúan como proveedores de herramientas o datos (Anthropic PB, 2024). El protocolo

establece una separación estricta entre el cliente (modelo de razonamiento) y el servidor (proveedor de capacidades), permitiendo que las herramientas de automatización web sean expuestas como capacidades de contexto estandarizadas (Anthropic PB, 2024). A diferencia de las APIs tradicionales de *drivers*, el MCP facilita un flujo de datos bidireccional donde el contexto del entorno es tan relevante como la instrucción de ejecución.

3.1.6. Behavior-Driven Development (BDD) y la Brecha del Glue Code

BDD es una metodología ágil que utiliza lenguajes naturales estructurados (Gherkin: *Given/When/Then*) para definir el comportamiento esperado y fomentar la colaboración. Sin embargo, su implementación técnica tradicional sufre del problema del **"Glue Code" (Código de Pegamento)**: la necesidad de escribir y mantener manualmente métodos de código para cada paso de Gherkin, lo que aumenta la deuda técnica y el costo de mantenimiento (Wynne et al., 2017).

3.2. Estado del Arte

La evolución de la automatización de pruebas ha estado marcada por una búsqueda constante para reducir la fricción entre la definición de los requisitos del negocio y su implementación técnica. A continuación, se presenta una revisión crítica de los enfoques predominantes, analizando cómo la industria y la academia han intentado resolver los problemas de mantenibilidad y portabilidad, y dónde persisten las brechas que este proyecto busca abordar.

3.2.1. Evolución de las Arquitecturas: Del Scripting Lineal al Diseño Modular

Inicialmente, la automatización de pruebas se basaba en enfoques de *scripting lineal* o herramientas de "grabación y reproducción" (*record & playback*). Si bien estos métodos permitían una creación rápida de pruebas, resultaron ser insostenibles a largo plazo debido a su alta sensibilidad d ante cambios en la interfaz de usuario (UI), lo que generaba un alto costo de mantenimiento (Lehtonen, 2023).

Para mitigar estos problemas, la industria adoptó el patrón **Page Object Model (POM)**. Este enfoque desacopla la estructura de la página de la lógica de la prueba, encapsulando los localizadores en clases de objetos dedicadas. Aunque el POM mejora la reutilización del código, estudios recientes indican que a menudo conduce a la creación de "Clases Dios" (*God Classes*) que violan el Principio de Responsabilidad Única, manteniendo una alta barrera técnica que excluye a los expertos del dominio (Leotta et al., 2013). Más recientemente, el **Patrón Screenplay** ha surgido como una evolución del POM, aplicando principios SOLID para centrar las pruebas en las "tareas" del usuario en lugar de en la estructura de la página (Smart and Molak, 2023). Sin embargo, su implementación sigue requiriendo habilidades avanzadas de programación, lo que no resuelve el problema de la brecha de habilidades (*skill gap*) en equipos multidisciplinarios.

3.2.2. El Desafío del "Glue Code" en Behavior-Driven Development (BDD)

El enfoque *Behavior-Driven Development* (BDD), popularizado por herramientas como Cucumber, ha intentado cerrar la brecha de comunicación mediante el uso de lenguajes naturales como Gherkin. Sin embargo, la literatura académica identifica consistentemente un cuello de botella crítico: la dependencia del **"Glue Code"** (código de pegamento).

Investigaciones empíricas, como las de Irshad et al., demuestran que el mantenimiento de la capa de definición de pasos (*Step Definitions*) consume recursos significativos y tiende a acumular deuda técnica (Irshad et al., 2021). Por cada nuevo escenario de negocio escrito en Gherkin, se requiere una implementación manual de código subyacente, lo que a menudo duplica el esfuerzo y desincroniza la documentación viva de la realidad del sistema (Wynne et al., 2017). Aunque existen intentos de generar este código automáticamente mediante Inteligencia Artificial (IA) o Grandes Modelos de Lenguaje (LLMs) (Molison et al., 2025), estas soluciones a menudo generan código no determinista o difícil de mantener, sin resolver el problema estructural de la dependencia del código.

3.2.3. Divergencia Tecnológica: Selenium vs. Playwright

La elección del motor de ejecución es otro factor determinante en la arquitectura de pruebas. Durante años, **Selenium WebDriver** ha sido el estándar de la industria, operando bajo el protocolo HTTP (W3C WebDriver). No obstante, estudios comparativos recientes (García et al., 2024) han evidenciado sus limitaciones en términos de velocidad y estabilidad frente a herramientas modernas como **Playwright**.

Playwright utiliza una arquitectura basada en *WebSockets*, lo que permite una comunicación bidireccional y de baja latencia con el navegador, ofreciendo una ejecución más rápida y mecanismos de espera automática (*auto-wait*) que reducen drásticamente la inestabilidad de las pruebas (*flakiness*) (Tymoshchuk, 2025). A pesar de estas ventajas, la migración de suites de pruebas existentes de Selenium a Playwright es costosa debido al acoplamiento de los scripts con las APIs específicas de cada herramienta, lo que subraya la necesidad de una arquitectura agnóstica que abstraiga el motor de ejecución.

3.2.4. Enfoques Dirigidos por Metadatos y Herramientas Low-Code

Para abordar la complejidad técnica, han surgido plataformas comerciales *Low-Code/No-Code* y enfoques *Dirigidos por Metadatos (Metadata-Driven)*. Estas soluciones permiten definir pruebas mediante configuración o interfaces visuales, reduciendo la necesidad de escritura de código.

Estudios en el ámbito de la ingeniería de datos han demostrado que las arquitecturas dirigidas por metadatos pueden reducir el esfuerzo de mantenimiento en más del 50 % al centralizar la lógica de transformación y validación (Abughazala and Muccini, 2025). Sin embargo, en el contexto de la automatización de UI web, las soluciones existentes suelen ser herramientas propietarias que introducen un riesgo significativo de bloqueo del proveedor (*vendor lock-in*) y limitan la interoperabilidad (Khankhoje, 2022). Tras la investigación realizada hasta la fecha, no se identificó un marco de trabajo que combine la semántica de BDD con una arquitectura dirigida por metadatos y sea

agnóstica al controlador.

3.2.5. Resiliencia Semántica y Agentes de IA en la Industria

En el ecosistema contemporáneo de ingeniería de software, han surgido soluciones comerciales orientadas a mitigar la fragilidad inherente de las pruebas de interfaz de usuario mediante mecanismos de auto-curación (*self-healing*). Iniciativas tecnológicas emergentes como *Autosana.ai* (Autosana AI, 2025) y *Spur* (Spur Test, 2025) proponen el uso de agentes basados en modelos de lenguaje para la re-identificación dinámica de selectores técnicos cuando los identificadores rígidos (como XPath o selectores CSS) fallan ante modificaciones en el DOM (*Document Object Model*). Estos enfoques representan un avance significativo hacia la automatización resiliente, desplazando el mantenimiento manual por un razonamiento basado en la intención de negocio.

No obstante, la revisión crítica de estas soluciones permite identificar dos limitaciones estructurales que definen una brecha de investigación actual:

1. **Dependencia de Plataformas Propietarias:** La mayoría de estas herramientas operan como soluciones tipo SaaS (*Software as a Service*) cerradas, lo que genera un riesgo de bloqueo del proveedor (*vendor lock-in*) y restringe la soberanía y portabilidad de los activos de prueba de las organizaciones (Khankhoje, 2022).

2. **Persistencia del "Glue Code":** A pesar de la integración de IA, diversas implementaciones aún dependen de una capa de código intermedio manual para vincular las definiciones de negocio con las capacidades de ejecución del modelo, lo que no resuelve de manera integral la deuda técnica identificada por Irshad et al. (2021).

Adicionalmente, se han desarrollado herramientas emergentes como **ZeroStep** (ZeroStep, 2025) y **Stagehand** (Browserbase, 2025). ZeroStep propone la integración de IA directamente en las pruebas de Playwright utilizando comandos en lenguaje natural que se interpretan en tiempo de ejecución. Por su parte, Stagehand proporciona un framework de navegación web impulsado por IA, diseñado para extraer datos y realizar acciones mediante un modelo de lenguaje. Si bien estas herramientas reducen la barrera de entrada al simplificar la escritura de *scripts*, presentan limitaciones en entornos de grado empresarial: muchas de ellas operan como servicios en la nube (creando dependencia de APIs externas) y carecen de una arquitectura basada puramente en metadatos que desvincule por completo el dominio de negocio de la herramienta de ejecución subyacente.

3.2.6. Comparativa de Enfoques de Automatización Existentes

La Table 3.1 sintetiza las limitaciones de los enfoques de automatización predominantes y la brecha existente.

3.3. Tecnologías Habilitadoras y Algoritmos Resilientes

El desarrollo de ecosistemas multiagente orientados a la automatización requiere de una infraestructura tecnológica que permita la gestión de estados, la orquestación cognitiva y la inferencia

Tabla 3.1: Comparativa de Enfoques de Automatización y Diagnóstico de la Brecha Técnica

Criterio de Evaluación	Scripting Tradicional (Selenium + POM)	BDD Convencional (Cucumber)	Soluciones AI-SaaS (Autosana / Spur)	Brecha de Investigación Identificada
Mantenibilidad	Baja (Fragilidad ante cambios de UI)	Baja (Alta deuda de Glue Code)	Alta (Mecanismos de Self-healing)	**Optimización mediante Metadatos y Contexto**
Portabilidad	Nula (Atado a la API del Driver)	Limitada (Acoplamiento técnico en Steps)	Nula (Bloqueo de proveedor / Vendor Lock-in)	**Agnosticismo Alto vía Protocolo MCP**
Resiliencia	Mínima (Dependencia de selectores rígidos)	Mínima (Fallo ante cambios estructurales)	Alta (Modelos de IA propietarios)	**Resiliencia Semántica basada en metadatos**
Estandarización	Protocolos Web tradicionales (W3C)	DSL basado en lenguaje natural	Cerrada (Caja negra tecnológica)	**Uso de Model Context Protocol (MCP)**
Intervención Técnica	Alta (Requiere programación técnica)	Alta (Necesidad de desarrollo de Step Defs)	Baja (Enfoque No-Code comercial)	**Minima (Ejecución dirigida por Metadatos)**

resiliente sobre interfaces dinámicas. En este contexto, se han propuesto herramientas y algoritmos específicos que representan avances técnicos significativos en la actualidad.

3.3.1. Orquestación Cognitiva: LangChain y LangGraph

La integración de Grandes Modelos de Lenguaje (LLMs) en aplicaciones complejas ha sido impulsada por frameworks de abstracción como **LangChain** (Chase, 2022), el cual estandariza la interacción con modelos cognitivos, la gestión de memoria y la composición de cadenas de procesamiento. Sin embargo, para la construcción de agentes autónomos, las cadenas lineales (*pipelines*) pueden presentar limitaciones ante escenarios que demandan flujos cíclicos, reflexión y corrección autónoma de errores.

Para abordar este tipo de limitaciones arquitectónicas surge **LangGraph** (LangChain AI, 2024), una extensión oficial diseñada específicamente para modelar aplicaciones cognitivas como grafos de estados. En LangGraph, los flujos de interacción se definen mediante nodos (que encapsulan funciones o llamadas a LLMs) y aristas (condiciones de enrutamiento). Esta topología permite la ejecución cíclica y la persistencia del estado en el tiempo, distanciándose de la ejecución secuencial de los Grafos Acíclicos Dirigidos (DAGs) convencionales. Esta capacidad facilita la implementación de patrones agénticos complejos, como la auto-recuperación (*self-healing*) y la validación semántica iterativa en entornos de alta incertidumbre.

3.3.2. Generación de Localizadores Resilientes: Algoritmo Robula+

En la automatización web moderna, uno de los desafíos más recurrentes es la fragilidad de los identificadores o selectores (*locators*) frente a la evolución constante del *Document Object Model* (DOM). Tradicionalmente, los selectores basados en rutas absolutas o atributos dinámicos tienden a romperse con modificaciones en la estructura visual de la interfaz.

El algoritmo **Robula+** (*Robust Locator Algorithm*), propuesto por Leotta et al. (2016), aborda este problema de fragilidad mediante un enfoque heurístico para la generación de selectores XPath. El algoritmo inicia con un selector genérico y explora sistemáticamente el árbol del DOM, aplicando transformaciones progresivas con el objetivo de inferir un XPath corto, único y menos acoplado a la geometría de la página. La adopción de lógicas analíticas inspiradas en Robula+ dentro de agentes

localizadores se considera parte del estado del arte para mitigar fallos de localización, contribuyendo a que la automatización dependa de anclajes semánticos resilientes y no de estructuras visuales volátiles.

Capítulo 4

Diseño Técnico y Arquitectura

*Este capítulo y sus correspondientes apartados se estructuran con el propósito de alcanzar el **Objetivo Específico 1**, enfocado en el diseño de la arquitectura conceptual y del esquema de metadatos de la solución.*

4.1. Especificación de Requerimientos del Sistema

4.1.1. Requerimientos Funcionales (RF)

Para garantizar el correcto funcionamiento del sistema y dar cumplimiento a los objetivos del proyecto, se estructuraron las necesidades operacionales del software. Los requerimientos funcionales (RF) identificados para el núcleo del orquestador y sus componentes se describen detalladamente en la Tabla 4.1.

Tabla 4.1: Requerimientos Funcionales

ID	Requerimiento	Descripción
RF1	Ejecución BDD Dirigida por Metadatos	El sistema debe interpretar escenarios de prueba en lenguaje natural y resolver su ejecución mediante un repositorio de metadatos externo.
RF2	Orquestación Inteligente	Integración de un motor de razonamiento (LLM) capaz de planificar secuencias de acciones técnicas a partir de instrucciones semánticas.
RF3	Agnosticismo de Motor	Ejecución intercambiable sobre diferentes motores de automatización sin modificar los escenarios ni los metadatos.
RF4	Resiliencia (Self-healing)	Localización dinámica de elementos mediante razonamiento sobre el DOM cuando los selectores tradicionales fallan.
RF5	Interoperabilidad (MCP)	Comunicación entre orquestador y motores mediante un protocolo estándar de interoperabilidad.
RF6	Gestión de Caché	Persistencia de resoluciones técnicas exitosas para optimizar el consumo de recursos en ejecuciones recurrentes.
RF7	Trazabilidad y Reporte	Generación de reportes que permitan la auditoría técnica y funcional de cada paso realizado por los agentes.

4.1.2. Requerimientos No Funcionales (RNF)

Con el fin de asegurar que el sistema automatizado sea robusto, modular y sostenible en el tiempo, se definieron los criterios de calidad que guiarán su implementación técnica. Estos requerimientos no funcionales (RNF) se describen en la Tabla 4.2 y actúan como las directrices fundamentales para la evaluación de la arquitectura agéntica y su interoperabilidad.

Tabla 4.2: Requerimientos No Funcionales

ID	Requerimiento	Descripción (Atributo de Calidad)
RNF1	Mantenibilidad	Reducción del esfuerzo de mantenimiento mediante el desacoplamiento de elementos técnicos y lógica de negocio.
RNF2	Portabilidad	Facilidad para sustituir el motor de ejecución subyacente con un impacto mínimo en la suite de pruebas.
RNF3	Modularidad	Arquitectura compuesta por capas independientes siguiendo el modelo gTAA.
RNF4	Transparencia	Proporcionar logs detallados de las decisiones tomadas por los agentes inteligentes.
RNF5	Eficiencia	Minimizar el intercambio de datos con el LLM mediante estrategias de optimización de contexto.

4.2. Análisis del Protocolo de Interoperabilidad (MCP)

Con el propósito de diseñar una arquitectura que mitigue los riesgos de acoplamiento técnico con un motor de automatización específico, se estructuró un estudio comparativo de los servidores MCP disponibles. En la Tabla 4.3 se contrastan las características operativas esenciales de Playwright MCP y Selenium MCP, analizando cómo estos comportamientos podrían incidir sobre el diseño de la arquitectura de la solución.

4.2.1. Criterios de Selección y Gobernanza de los Servidores MCP Analizados

A continuación se describen los criterios de selección de los servidores MCP analizados, considerando su origen, madurez y modelo de gobernanza dentro del ecosistema de ingeniería de software:

- **Selección de Playwright MCP (Línea Oficial):** La adopción de este servidor se fundamenta en el paquete oficial `@playwright/mcp`, desarrollado y mantenido por el equipo central de Microsoft (Microsoft, 2026). Su gobernanza nativa garantiza una alineación estricta con las APIs internas del motor de automatización y actualizaciones síncronas con el núcleo del framework, proporcionando un estándar de estabilidad que actúa como *benchmark* de control para evaluar el protocolo MCP en condiciones óptimas.

- **Selección de Selenium MCP (Línea de Comunidad):** A diferencia de Playwright, el proyecto Selenium no provee un servidor MCP oficial de manera nativa en su núcleo actual. Ante la ausencia de una distribución centralizada, se realizó un mapeo de las soluciones emergentes en repositorios públicos de GitHub, identificando alternativas construidas sobre la máquina virtual de Java (como `selenium-mcp` (Naveen Automation Labs, 2026)) y otras variantes desarrolladas en Node.js (como `selenium-mcp-server` (TheMindMod, 2026)). Al contrastar las métricas de adopción disponibles en GitHub, se seleccionó la distribución `@angiejones/mcp-selenium` (Jones, 2026), debido a que registra los índices más elevados de descargas, valoraciones (*stars*) y bifurcaciones (*forks*), consolidándose cuantitativamente como la implementación de referencia dentro de dicha plataforma.

4.2.2. Comparativa Arquitectónica de Servidores MCP

A partir de estas consideraciones, se desglosan en la Tabla 4.3 los criterios técnicos y su impacto proyectado en el diseño:

Tabla 4.3: Comparativa Arquitectónica de Servidores MCP: Playwright vs Selenium

Característica	Playwright MCP	Selenium MCP	Criterio / Impacto Proyectado en el Diseño
Comando de Arranque	`npx @playwright/mcp`	`npx @angiejones/mcp-selenium`	Ambos servidores se basan en `StdioClientTransport`. Por lo tanto, el diseño conceptual debe prever la capacidad de inyectar variables de entorno para definir dinámicamente el motor de ejecución.
Gestión del Ciclo de Vida	**Implícito**. Inicialización del navegador vinculada al ciclo del proceso MCP.	**Explícito**. Requiere la invocación secuencial de instrucciones de apertura y cierre de sesión.	El diseño del sistema requerirá una capa de abstracción para la inicialización. Esto permitirá unificar el comportamiento bajo un modelo implícito, aislando esta disparidad operativa de las capas superiores.
Nomenclatura de Herramientas	Esquema estandarizado mediante prefijos (ej. `browser_navigate`).	Esquema directo basado en acciones atómicas (ej. `navigate`).	Debido a las discrepancias en el nombramiento de funciones, el diseño técnico contempla el aislamiento de la nomenclatura para cada servidor, mitigando el riesgo de alucinaciones semánticas por parte del modelo.

Continúa en la siguiente página...

Tabla 4.3 – Continúa desde la página anterior

Característica	Playwright MCP	Selenium MCP	Criterio / Impacto Proyectado en el Diseño
Estructura de Localizadores	**Unificado**. Admite un argumento global para la resolución de selectores CSS y XPath.	**Desacoplado**. Requiere la definición explícita del tipo de estrategia y su valor.	El diseño conceptual debe contemplar un componente de traducción semántica encargado de estandarizar la intención del modelo hacia los esquemas JSON particulares de cada motor de automatización.
Extracción del DOM	Ejecución interna mediante inyección de scripts globales en el contexto del navegador.	Ejecución externa dependiente del paso explícito de argumentos de control.	Esta diferencia operativa exige el diseño de una interfaz homogénea de extracción de la estructura HTML, definida como un requisito técnico crítico para habilitar las futuras funciones de auto-recuperación (*Self-healing*).
Cierre y Limpieza	Finalización del subproceso controlada directamente por el canal STDIO.	Riesgo latente de persistencia de binarios huérfanos del controlador en el sistema operativo.	El diseño conceptual exige proyectar una directriz de finalización y liberación de recursos orientada a la infraestructura de Selenium, mitigando así riesgos de degradación de memoria por la persistencia de procesos zombi.

4.3. Modelo de Metadatos

4.3.1. Análisis Teórico y Justificación del Diseño

Este apartado define la estructura propuesta para los metadatos que actuarán como el núcleo de la Capa de Definición en el sistema TAS. El diseño busca establecer una gramática común que permita el desacoplamiento entre la lógica de negocio (especificada en BDD) y la ejecución técnica, siguiendo los principios de la arquitectura gTAA.

La necesidad de un esquema de metadatos en un sistema de automatización inteligente se fundamenta en los siguientes principios de diseño:

- **Independencia de la Implementación**: El sistema debe ser capaz de modificar la ejecución técnica de un paso de prueba sin alterar la definición del escenario de negocio. Los metadatos actúan como el puente de traducción necesario para esta independencia.
- **Optimización del Modelo de Lenguaje**: El diseño contempla reducir la carga cognitiva de los agentes de razonamiento mediante la provisión de instrucciones pre-estructuradas, lo que incrementa el determinismo y la precisión en la ejecución.
- **Modelo de Resiliencia Basado en Contexto**: Se propone un esquema de almacenamiento de contexto que permita al sistema "aprender" y persistir resoluciones técnicas exitosas, habilitando capacidades futuras de auto-curación (self-healing) ante cambios en el entorno.

4.3.2. Propuesta de Repositorio de Definiciones Técnicas

Se propone la organización de las definiciones técnicas mediante un formato de intercambio de datos de alta legibilidad (YAML), con la intención de favorecer una gestión modular, jerárquica y reutilizable de los pasos de prueba. El objetivo de este repositorio conceptual es facilitar la transición entre las declaraciones orientadas al negocio y las operaciones técnicas del sistema.

4.3.2.1. Mapeo Semántico Técnico

Para ejemplificar esta transición, en el Listing 1 se exhibe un escenario de prueba redactado bajo el enfoque de desarrollo guiado por comportamiento (BDD), el cual describe una interacción típica de usuario en un lenguaje descriptivo y abstracto.

```
# language: es
Característica: Autenticación en el sistema de Orange
  Como un usuario registrado del sistema de Orange
  Quiero iniciar sesión en el sistema de Orange
  Para acceder a las funciones protegidas del sistema de Orange

  Escenario: Inicio de sesión exitoso en Orange
    Dado el usuario se encuentra en la plataforma de Orange
    Cuando el usuario ingresa sus credenciales correctas
    Entonces el sistema permite el acceso Orange
```

Listing 1: Prueba escrita en BDD

Como contraparte, en el Listing 2 se ilustra la propuesta de mapeo correspondiente para el escenario anterior. En este bloque se observa cómo cada una de las instrucciones de negocio se vincula con directivas técnicas básicas y detalladas. Este modelo de organización semántica pretende servir como una guía estructurada que facilite al componente de control la interpretación de la intención de la prueba, propendiendo por la reducción de ambigüedades antes de interactuar con el motor de automatización.

```
"el usuario se encuentra en la plataforma de Orange":
  instructions:
    - "Navega a la URL de la plataforma de Orange: https://opensource-demo.orangehrmlive.com
       /web/index.php/auth/login"

"el usuario ingresa sus credenciales correctas":
  instructions:
    - "Escribe 'Admin' en el campo de texto Username"
    - "Escribe 'admin123' en el campo de texto Password"
    - "Da click en el botón Login"

"el sistema permite el acceso Orange":
  instructions:
    - "Verifica que este el texto 'Dashboard'"
```

Listing 2: Propuesta de Definición de Paso

4.4. Modelo de Contexto

Se plantea la inclusión de un modelo de contexto de ejecución persistente (en adelante, modelo de contexto), concebido como un mecanismo de memoria caché para mitigar el consumo innecesario de tokens en ejecuciones iterativas de una misma prueba. La intención de este componente es almacenar de forma dinámica aquellos localizadores y acciones técnicas que ya fueron resueltos con éxito en una sesión previa. De este modo, ante ejecuciones subsecuentes del mismo escenario, el sistema propende por la reutilización de este conocimiento estructurado, reduciendo la necesidad de invocar repetidamente la capacidad de inferencia del modelo de lenguaje para instrucciones cuyo camino de resolución ya ha sido determinado históricamente.

4.4.1. Estructura del Modelo Resuelto

Con el propósito de ilustrar la organización de esta memoria de ejecución, en el Listing 3 se esquematiza la propuesta de estructura en formato JSON. Este objeto representa la abstracción de una resolución técnica exitosa que se proyecta persistir de manera lógica. Al registrar el mapeo entre la intención original del paso de prueba y los componentes técnicos finales —como el tipo de acción, el XPath exacto y los datos de entrada de la interacción— se facilita un esquema de consulta rápida. Este modelo pretende actuar como una capa de optimización operativa, de manera que el componente de control pueda replicar interacciones conocidas en ciclos repetitivos de prueba, favoreciendo la velocidad del proceso y disminuyendo potencialmente los costos asociados al procesamiento de contexto semántico.

```
{
  "stepText": "el usuario ingresa sus credenciales correctas",
  "instructions": [
    {
      "instruction": "Escribe 'Admin' en el campo de texto Username",
      "actions": [
        {
          "accion": "type",
          "xpath": "//input[@name='username']",
          "valor": "Admin",
          "metadata": {
            "selenium": { "tool": "send_keys", "arguments": { "by": "xpath", "value": "//
                input[@name='username']", "text": "Admin" } },
            "playwright": { "tool": "browser_type", "arguments": { "target": "//input[@name
                ='username']", "text": "Admin" } }
          }
        }
      ]
    }
  ]
}
```

Listing 3: Propuesta de Modelo de Contexto Resuelto

4.5. Arquitectura del Sistema (basado en el Modelo gTAA)

La arquitectura propuesta para el enfoque de **Meta-Driven Testing (MDT)** guiado por metadatos e inteligencia artificial se fundamenta en el modelo de referencia gTAA (*generic Test Automation Architecture*). El diseño conceptual busca maximizar la separación de preocupaciones mediante una estructura de cuatro capas lógicas independientes, permitiendo que la solución proyectada sea agnóstica al motor de ejecución y resiliente a cambios dinámicos en la interfaz de usuario.

4.5.1. Implementación Distribuida: Modelo Cliente-Servidor

Para llevar el modelo abstracto gTAA a una implementación técnica escalable, el framework se estructura bajo un paradigma distribuido Cliente-Servidor:

- **Cliente (MDT-Client):** Agrupa lógicamente la *Capa de Generación* y la *Capa de Definición*. Su responsabilidad radica en procesar los escenarios BDD, resolver los metadatos YAML, gestionar la memoria caché (*Golden Copy*) para la optimización de recursos, y compilar un paquete de instrucciones estructuradas. Operativamente, actúa como una herramienta CLI rápida que lanza y audita las pruebas sin requerir poder de cómputo intensivo para inferencia de IA.

- **Servidor (MDT-Server):** Materializa la *Capa de Ejecución* y la *Capa de Adaptación*. Actúa como un motor agnóstico gobernado por un Sistema Multiagente (Symphony, Argos, Nexus). Su función principal es recibir las instrucciones del cliente, inferir intenciones semánticas mediante Modelos de Lenguaje (LLMs), calcular localizadores resilientes sobre el DOM y despachar comandos físicos al sistema bajo prueba mediante el protocolo MCP.

Esta separación de despliegue refuerza el desacoplamiento estipulado por gTAA, permitiendo que la capa de negocio (el repositorio de pruebas) y la capa cognitiva (orquestación de IA) evolucionen, se escalen y se mantengan de manera autónoma.

4.5.2. Especificación de Capas del Modelo Arquitectónico

4.5.2.1. Capa de Generación (Test Generation Layer)

Esta capa se diseña como el punto de entrada del framework. Su responsabilidad teórica es la ingesta y gestión de los activos de prueba expresados en lenguaje natural (BDD).

- **Función**: Interpretar y estructurar las intenciones de negocio para su procesamiento posterior, abstrayendo las definiciones funcionales de la complejidad técnica subyacente.

- **Alineación**: Aborda conceptualmente el requerimiento **RF1** (Ejecución BDD Dirigida por Metadatos).

4.5.2.2. Capa de Definición (Test Definition Layer)

Se proyecta como el repositorio central de conocimiento y abstracción. En esta capa se plantea la gestión de los metadatos y esquemas declarativos que vincularán los términos de negocio con las directivas técnicas.

- **Función**: Proporcionar un mecanismo de mapeo semántico que desacople la lógica de prueba de la implementación física del sistema bajo prueba (SUT).

- **Alineación**: Aborda las directrices de calidad de los requerimientos **RNF1** (Mantenibilidad) y **RNF3** (Modularidad).

4.5.2.3. Capa de Ejecución (Test Execution Layer)

Constituye el núcleo de razonamiento planeado para la solución. Se diseña para integrar agentes de toma de decisiones basados en modelos de lenguaje (LLM) que transformen las directivas semánticas en planes de acción técnicos e iterativos.

- **Función**: Orquestar la planificación de comandos, la optimización de peticiones, la auditoría del razonamiento y los mecanismos de recuperación en tiempo de ejecución.

- **Alineación**: Da cobertura a los requerimientos **RF2** (Orquestación Inteligente), **RF4** (Resiliencia [Self-healing]), **RF6** (Gestión de Caché), **RF7** (Trazabilidad y Reporte), **RNF4** (Transparencia) y **RNF5** (Eficiencia).

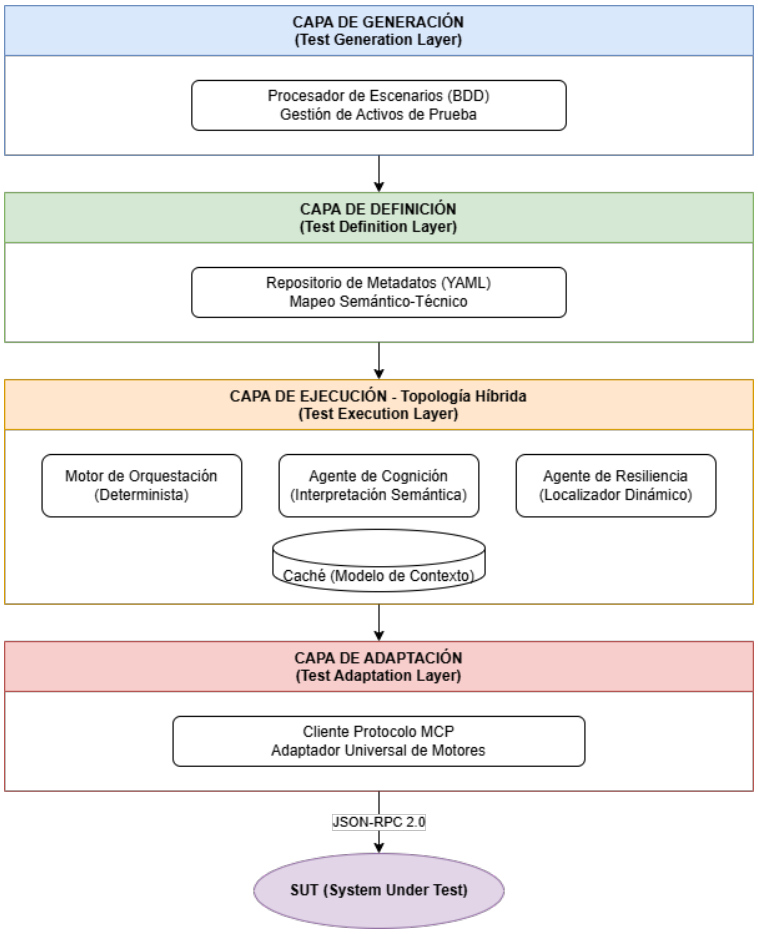
4.5.2.4. Capa de Adaptación (Test Adaptation Layer)

Se define como la interfaz de abstracción técnica necesaria para establecer la comunicación con el Sistema Bajo Prueba (SUT). El diseño estipula el uso de protocolos estandarizados para interactuar con las herramientas de automatización.

- **Función**: Traducir los planes de acción del agente cognitivo en comandos de bajo nivel, ejecutar las interacciones físicas y recuperar el estado del DOM de forma estandarizada.

- **Alineación**: Satisface los criterios de los requerimientos **RF3** (Agnosticismo de Motor), **RF5** (Interoperabilidad [MCP]) y **RNF2** (Portabilidad).

Figura 4.1: Arquitectura multicapa propuesta basada en el modelo de referencia gTAA.



Como se modela en la Figura 4.1, la subdivisión en cuatro capas independientes asegura el aislamiento conceptual entre las intenciones de negocio y las operaciones técnicas de bajo nivel.

4.5.3. Trazabilidad Requerimientos-Arquitectura

Para validar la idoneidad teórica del diseño frente a las necesidades identificadas, la Tabla 4.4 establece la correspondencia analítica entre los requerimientos formulados y las abstracciones arquitectónicas de la solución propuesta.

Tabla 4.4: Matriz de Trazabilidad Arquitectónica Conceptual

ID Requerimiento	Capa Responsable	Mecanismo o Componente de Diseño
RF1 (BDD Metadatos)	Capa de Generación	Procesador y analizador de escenarios sintácticos

Continúa en la siguiente página...

Tabla 4.4 – Continúa de la página anterior

ID Requerimiento	Capa Responsable	Mecanismo o Componente de Diseño
RF2 (Orquestación)	Capa de Ejecución	Sistema híbrido de orquestación y agentes cognitivos
RF3 (Agnosticismo Motor)	Capa de Adaptación	Capa de abstracción de motores (Patrón Estrategia)
RF4 (Resiliencia)	Capa de Ejecución	Algoritmo de relocalización semántica del DOM
RF5 (Interoperabilidad MCP)	Capa de Adaptación	Interfaz de cliente basada en *Model Context Protocol*
RF6 (Gestión de Caché)	Capa de Ejecución	Almacén relacional de resoluciones técnicas previas
RF7 (Trazabilidad)	Capa de Ejecución	Subsistema proyectado de auditoría y reporte técnico
RNF1 (Mantenibilidad)	Capa de Definición	Esquema declarativo de metadatos acoplados por semántica
RNF2 (Portabilidad)	Capa de Adaptación	Controladores de abstracción para motores externos
RNF3 (Modularidad)	Estructura del Framework	Desacoplamiento lógico de las 4 capas funcionales
RNF4 (Transparencia)	Capa de Ejecución	Registro estructurado de trazas de razonamiento lógico
RNF5 (Eficiencia)	Capa de Ejecución	Filtro y optimizador dinámico de ventanas de contexto

4.5.4. Decisión de Diseño: Estrategia de Localización Semántica vía XPath

En el contexto de la automatización web moderna, la selección del mecanismo de localización estructural resulta relevante para el diseño arquitectónico, especialmente ante la adopción de frameworks que generan clases e identificadores dinámicos.

Para este ecosistema, se ha establecido **XPath** (XML Path Language) como el estándar arquitectónico de localización e interacción, frente a alternativas tradicionales como los selectores CSS exclusivos o los identificadores directos (*IDs*). Esta decisión se fundamenta teóricamente en las investigaciones de Leotta et al. (2016), quienes cuestionan la percepción generalizada de que XPath es inherentemente más frágil que otros localizadores, aportando los siguientes argumentos estructurales:

- **Alta Expresividad Simbólica:** La gran mayoría de los métodos de localización (por clase, nombre o ID) pueden ser simulados nativamente utilizando expresiones XPath. Esto propor-

ciona un nivel base de robustez comparable al de las APIs específicas del motor, unificado bajo un único lenguaje evaluador.

- **Amplitud de Cobertura:** Mientras que métodos directos (como buscar por ID) presentan limitaciones si el elemento objetivo carece de dicho atributo en el código fuente, la expresividad lógica de XPath facilita la determinación unívoca de elementos dentro del árbol de la página web, incluso en ausencia de identificadores estáticos.
- **Navegación Multidimensional Extendida:** Aunque los selectores CSS han evolucionado para permitir ciertas validaciones ascendentes (mediante la pseudo-clase `:has()`), XPath posee ejes lógicos explícitos (como `parent::` y `preceding-sibling::`) que permiten transitar con alta flexibilidad por la topología del árbol en diversas direcciones.
- **Evaluación de Texto Nativo:** Constituye una ventaja comparativa destacable. Mientras que CSS carece de mecanismos integrados para seleccionar elementos basándose en su contenido textual, XPath incluye soporte nativo para evaluación semántica visual mediante funciones de cadenas (ej. `text()="..."` o `contains()`).
- **Inferencia Contextual para IA:** La combinación de movilidad estructural y evaluación de texto habilita patrones de razonamiento cognitivo avanzados. Un agente de IA puede anclarse lógicamente a un texto visible, ascender a su contenedor estructural, navegar hacia los nodos hermanos y localizar el `<input>` objetivo más cercano. Esta concatenación de movimientos espaciales anclados en semántica visual es viabilizada de manera eficiente por la expresividad de XPath.

La literatura sugiere que el debate sobre la fragilidad de XPath surge, en parte, porque históricamente algunas herramientas automáticas generaban rutas absolutas poco mantenibles, y no necesariamente por una debilidad intrínseca del lenguaje (Leotta et al., 2016). Al combinar la expresividad estructural de XPath con algoritmos heurísticos de agentes cognitivos, la *Capa de Adaptación* obtiene un mecanismo de localización unificado, incrementando significativamente la resiliencia del sistema frente a mutaciones en la interfaz.

4.5.5. Dinámica Operativa Proyectada y Flujo de Interacción

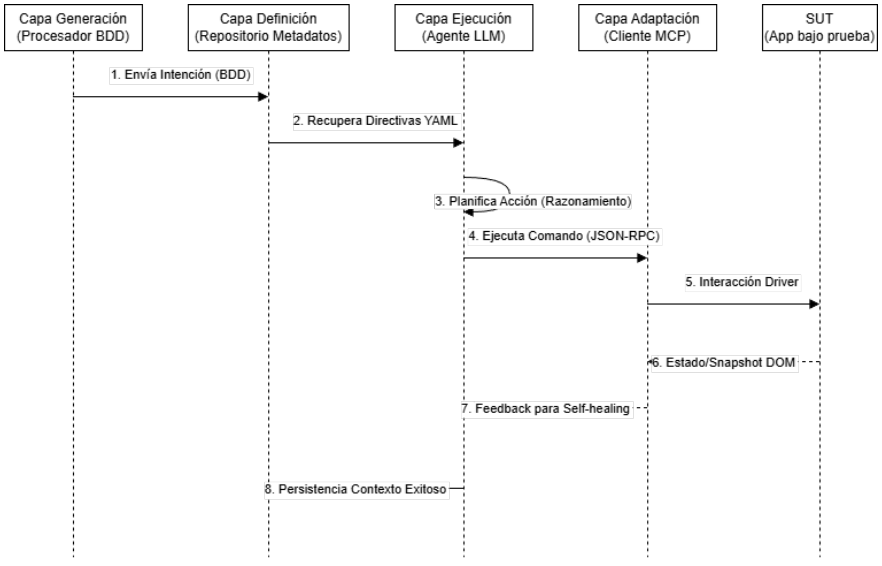
Para materializar el principio de separación de preocupaciones heredado de gTAA, el diseño del flujo operativo se concibe de forma secuencial y orientado a eventos. Este esquema teórico prevé que las especificaciones funcionales permanezcan inalteradas ante fluctuaciones en la interfaz del SUT, delegando la resiliencia en la capa cognitiva y la optimización de recursos en la lógica de persistencia intercapa.

El comportamiento dinámico idealizado para la interacción entre componentes se describe mediante la siguiente secuencia lógica (ver Figura 4.2):

1. **Inyección Semántica:** El componente de procesamiento en la Capa de Generación recibe la especificación formalizada en formato BDD (**RF1**) y solicita su traducción a la estructura de la Capa de Definición.

2. **Traducción Basada en Metadatos:** El repositorio de conocimiento mapea conceptualmente las intenciones de negocio en directivas técnicas estructuradas (como esquemas YAML), aislando la prueba de la capa física del SUT (**RNF1**, **RNF3**).
3. **Evaluación Preventiva de Costes:** Antes de invocar al servidor de ejecución, el cliente interroga el módulo de almacenamiento local (**RF6**). Si la relación acción-elemento se encuentra registrada como una resolución exitosa anterior, se inyecta el selector técnico de manera inmediata en la petición, mitigando el tráfico de red innecesario y el gasto de contexto de inferencia (**RNF5**).
4. **Planificación Cognitiva:** Ante un fallo de coincidencia en el almacén de resoluciones, el agente de planificación inteligente (**RF2**) asume la tarea de razonamiento analizando el estado actual del DOM. Cada inferencia o bifurcación en el proceso de decisión se diseña para generar un registro técnico detallado (**RNF4**).
5. **Abstracción de la Ejecución:** El plan de acción resultante es transferido a la Capa de Adaptación. El diseño estipula que este componente actúe como un cliente del protocolo MCP (**RF5**), convirtiendo el plan en mensajes normalizados *JSON-RPC 2.0* enviados de forma transparente hacia el motor automatizador configurado (**RF3**, **RNF2**).
6. **Activación del Mecanismo de Resiliencia:** En escenarios donde la interfaz del SUT cambie y el motor falle al buscar un selector, la arquitectura contempla la activación del módulo de relocalización dinámica (**RF4**) para re-evaluar el árbol del DOM en caliente empleando capacidades cognitivas alternativas.
7. **Consolidación de Datos y Cierre:** Finalizada la acción física, se extrae el estado resultante de la interfaz, se actualizan los registros de resoluciones (**RF6**) y el módulo de auditoría genera la bitácora paso a paso de la prueba (**RF7**).

Figura 4.2: Diagrama de secuencia proyectado para el flujo operativo basado en el protocolo MCP.



4.6. Selección del Modelo de Lenguaje y Estrategia de Cómputo Cognitivo

El núcleo de razonamiento del framework proyectado depende críticamente de la capacidad de un Modelo de Lenguaje para interpretar intenciones de negocio, interactuar con el protocolo MCP y localizar elementos. Por lo tanto, la selección del modelo base no solo responde a criterios de precisión lógica, sino a una evaluación multidimensional que equilibre la soberanía de los datos, la latencia de operación y la viabilidad financiera bajo escenarios de uso intensivo.

4.6.1. Evaluación de Paradigmas de Despliegue: Modelos Abiertos frente a APIs Comerciales

El primer vector de decisión radica en la naturaleza del acceso al modelo. Se requiere determinar si la arquitectura debe orientarse hacia el consumo de servicios comerciales cerrados (*Closed Source* de nivel de frontera) o hacia el despliegue autónomo de modelos de código abierto u weights compartidos (*Open Source / Open Weights*) en infraestructura controlada.

La Tabla 4.5 resume las compensaciones (*trade-offs*) arquitectónicas de ambos enfoques basándose en los atributos de calidad fundamentales del sistema.

Tabla 4.5:
Matriz Comparativa: Modelos Open Source vs. APIs Comerciales

Atributo de Calidad	**Modelos de Código Abierto (Open Source)**	**APIs Comerciales (Closed Source)**
Privacidad y Seguridad	Máxima absoluta. El modelo se despliega en infraestructura local o nube privada. Los datos nunca salen del perímetro de seguridad.	Limitada. Los datos (código, metadatos, DOM) deben viajar a servidores externos de terceros, sujeto a políticas de privacidad.
Predecibilidad de Costes	Alta (Cómputo Fijo). El coste está asociado al alquiler o compra del hardware (GPUs). Peticiones ilimitadas sin coste variable.	Baja (Gasto Variable). Esquema de pago por uso basado en tokens. En bucles agénticos (*self-healing*), el coste puede escalar exponencialmente.
Especialización (*Fine-Tuning*)	Completa. Acceso total a los pesos del modelo para entrenarlo en lenguajes específicos, formatos YAML/JSON estrictos o protocolos como MCP.	Restringida. Solo permiten capas superficiales de personalización orientadas a estilo, pero no modificaciones profundas del núcleo.
Latencia y Rendimiento	Optimizable. Permite aplicar cuantización y desplegar en el borde (*edge computing*) junto a la aplicación para reducir el TTFT.	Dependiente de Red. Añade latencia por tráfico de internet y está sujeta al *throttling* (límites de tasa de peticiones) del proveedor.
Control de Ciclo de Vida	Soberanía Total. El modelo descargado es propiedad del proyecto. Comportamiento 100 % reproducible y libre de depreciaciones a ciegas.	Riesgo de Proveedor (*Lock-in*). El proveedor puede actualizar, cambiar el comportamiento o retirar el modelo de la API de forma repentina.

Continúa en la siguiente página...

Tabla 4.5 – Continúa de la página anterior

Atributo de Calidad	Modelos de Código Abierto (Open Source)	APIs Comerciales (Closed Source)
Capacidad General	Variable. Requiere mayor esfuerzo de ingeniería de prompts, orquestación y hardware dedicado para igualar a los modelos de frontera.	Excelente. Modelos de frontera listos para usar con un rendimiento sobresaliente en lógica compleja y razonamiento general inmediato.

4.6.2. Evaluación y Selección del Entorno de Inferencia

El despliegue del framework se realiza de forma local sobre Windows 11 (Intel i5-13420H, 16 GB de RAM, RTX 4050 de 6 GB). Para mitigar riesgos de saturación y garantizar la coexistencia con los navegadores de prueba, se establece un límite operativo de alrededor de **5 GB de VRAM** para el modelo y su caché de contexto (*KV Cache*).

4.6.2.1. Evaluación de Plataformas de Inferencia Local

El análisis de las tecnologías se enfoca en la instrumentación analítica y el control de recursos bajo Windows 11, cuyos criterios se sintetizan en la Tabla 4.6:

Tabla 4.6:
Matriz Comparativa de Plataformas de Inferencia Local

Criterio	LM Studio	Ollama	Docker / vLLM
Consumo de Recursos	**Flexible.** Su interfaz basada en Electron eleva la RAM (Smith and Johnson, 2024), pero permite ejecución *headless* (CLI) para mitigar el consumo gráfico (lms, 2024).	**Bajo.** Servicio nativo en segundo plano con impacto mínimo en RAM y VRAM dinámica (oll, 2024; Gerganov, 2024).	**Alto.** Reserva estática de VRAM para *PagedAttention*; sobrecarga por virtualización en WSL2 (doc, 2024; Kwon et al., 2023).
Catálogo de Modelos	**Entendible.** Buscador visual integrado directamente con Hugging Face, facilitando la exploración frente a la gestión por comando de Ollama (lms, 2024).	**Restringido.** Biblioteca curada gestionada de forma textual por consola mediante etiquetas estáticas (*tags*) (oll, 2024).	**Complejo.** Requiere la descarga y conversión manual de archivos de peso completo (*safetensors*).
Viabilidad Local	**Semaforización.** Sistema visual de alertas (verde/amarillo/rojo) que calcula si el modelo se ajusta a la RAM/VRAM local antes de descargar (lms, 2024).	**Manual.** No previene el desbordamiento de memoria; el usuario debe estimar la viabilidad del modelo empíricamente.	**Inexistente.** Diseñado bajo el supuesto de hardware de grado servidor con recursos dedicados masivos.

Tabla 4.6:
Matriz Comparativa de Plataformas de Inferencia Local

Criterio	LM Studio	Ollama	Docker / vLLM
Telemetría	**Excelente.** Gráficas nativas en tiempo real de uso de GPU, RAM y velocidad de tokens, fundamentales para la caracterización de los SLMs (lms, 2024).	**Básica.** Monitoreo austero por consola o extracción de metadatos planos a través de peticiones HTTP a su API REST (oll, 2024).	**Compleja.** Carece de panel interno; exige la configuración de servicios externos como Prometheus.

4.6.2.2. Justificación de la Plataforma seleccionada

Las capacidades de **telemetría** y las facilidades de selección del catálogo de modelos constituyeron los factores determinantes para la selección de la plataforma a utilizar:

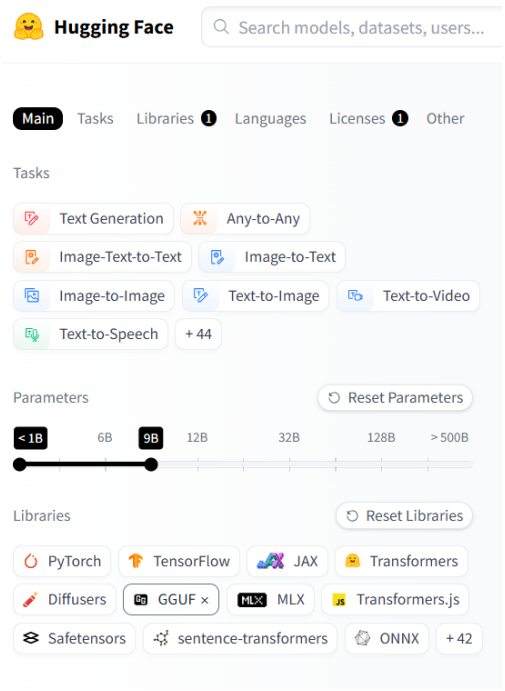
- **Herramienta Descartada – Docker / vLLM:** Excluida por la preasignación estática de VRAM que bloquea los recursos requeridos por los navegadores de automatización y por la complejidad de traducción de WSL2 (doc, 2024; Kwon et al., 2023).
- **Candidata Inicial – LM Studio (Diagnóstico y Ejecución Funcional):** Seleccionada como opción prioritaria debido a su sistema de semaforización de hardware y su catálogo intuitivo (lms, 2024). En la caracterización elemental del prototipo se explotará su telemetría visual. Posteriormente, en la fase funcional, se ejecutará en **modo *headless* (sin interfaz gráfica)** para reducir el uso de recursos, mitigando así el riesgo de degradación de hardware.
- **Alternativa de Contingencia – Ollama (Mitigación Operacional):** Se establece como el motor de respaldo estratégico. Si se identifican inconvenientes de estabilidad o conectividad con el CLI de la herramienta principal, la paridad de su API REST compatible con OpenAI y el uso compartido del formato agnóstico GGUF (oll, 2024; Gerganov, 2024) permitirán conmutar el framework hacia Ollama de forma transparente modificando únicamente el puerto de red, garantizando la resiliencia de la investigación.

4.7. Filtrado y Preselección en Hugging Face Hub

La exploración de los modelos candidatos se realizó directamente en el portal web de Hugging Face (Wolf et al., 2020). Aunque la plataforma de inferencia elegida (LM Studio) permite buscar modelos localmente, el uso directo del repositorio web se justifica metodológicamente por la necesidad de aplicar filtros cruzados avanzados antes de proceder con su importación al entorno de ejecución (lms, 2024).

Utilizando el operador de ordenamiento por adopción (*Sort: Most downloads*), el universo de modelos se decantó inicialmente bajo los siguientes tres criterios base (ver Figura 4.3):

Figura 4.3: Configuración de filtros base de búsqueda en Hugging Face.



- **Especialización Sintáctica (*Hugging Face Tags: instruct code*):** Se condicionó la búsqueda a modelos que fusionen un entrenamiento masivo en lenguajes de programación con alineación explícita para el seguimiento de instrucciones. Esto es esencial para que el agente acate directivas estrictas.
- **Formato de Inferencia (*Hugging Face Tag: GGUF*):** Se restringió exclusivamente a archivos optimizados en formato GGUF, garantizando la compatibilidad nativa para la fraccionación de capas en hardware de consumo (Gerganov, 2024).
- **Techo de Hardware (*Parameters: <9B*):** Se limitó el rango a modelos inferiores a 9 mil millones de parámetros, asegurando desde el origen que su peso estático (típicamente entre 1 y 5 GB) no exceda el umbral operacional estipulado de 5 GB de VRAM.

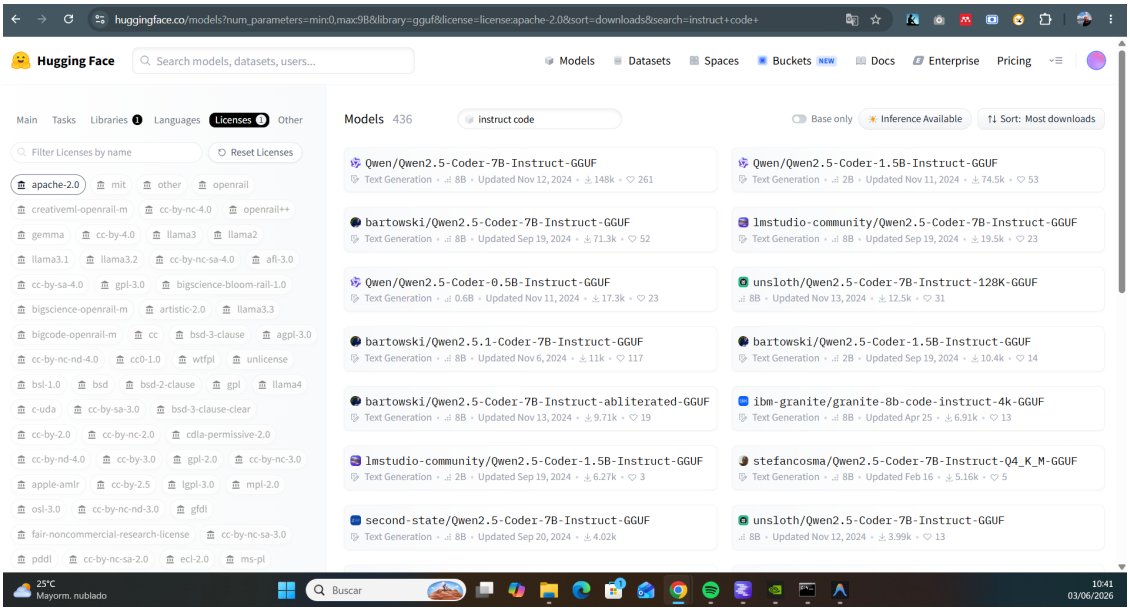
4.7.1. Análisis de Licencias y Conformación del Top 5

Sobre la base de los filtros técnicos anteriores, se aplicó un cuarto criterio correspondiente al **Tipo de Licencia** para garantizar la soberanía de los datos procesados localmente. Al segmentar los resultados entre licencias permisivas (*Apache 2.0*, *MIT* y *Llama 3*), la evidencia gráfica reveló tendencias asimétricas de adopción que dictaminaron la selección definitiva:

1. **Dominio de la Licencia Apache 2.0:** Como se evidencia en la Figura 4.4, esta licencia concentra el mayor volumen de adopción. El modelo *Qwen/Qwen2.5-Coder-7B-Instruct-GGUF*

lidera globalmente con 148k descargas, seguido de la variante de *1.5B* con 74.5k y la de *0.5B* con 17.3k. Además, *ibm-granite/granite-8b-code-instruct-4k-GGUF* (6.91k) se posiciona como el modelo de distinta arquitectura más descargado. Estos cuatro modelos conforman la base de la selección por su comprobado volumen empírico.

Figura 4.4: Resultados de búsqueda en Hugging Face bajo licencia Apache 2.0.



2. **Aporte de la Licencia MIT:** Para garantizar variabilidad técnica y evitar sesgos de un único proveedor, se analizó el filtro MIT (ver Figura 4.5). Excluyendo derivados de Qwen presentes en esa lista, *unsloth/Seed-Coder-8B-Instruct-GGUF* destaca como la principal alternativa con 2.77k descargas, ganándose un lugar para actuar como un control metodológico de otro origen.

3. **Descarte de la Licencia Llama 3:** Los resultados evidencian una adopción marginal de la comunidad hacia las variantes *Llama 3* orientadas a código en formato GGUF menores a 9B (ver Figura 4.6). El modelo más descargado de esta lista (*Codellama-3-8B-Finetuned-Instruct-i1-GGUF*) apenas alcanza las 178 descargas. Por este motivo, se descartó formalmente su inclusión.

Una vez consolidado el catálogo de los cinco modelos finalistas presentados en la Tabla 4.7, se procedió a su descarga en el entorno de ejecución. En este punto intervino **LM Studio**, donde su sistema de semaforización interactiva local (lms, 2024) evaluó de manera anticipada si la carga de los archivos generaría un desbordamiento hacia la memoria RAM. Al realizar la comprobación empírica, los cinco modelos elegidos marcaron semáforo en verde, superando exitosamente este último filtro confirmatorio de viabilidad física en la tarjeta NVIDIA RTX 4050 (6 GB de VRAM).

Figura 4.5: Resultados de búsqueda en Hugging Face bajo licencia MIT.

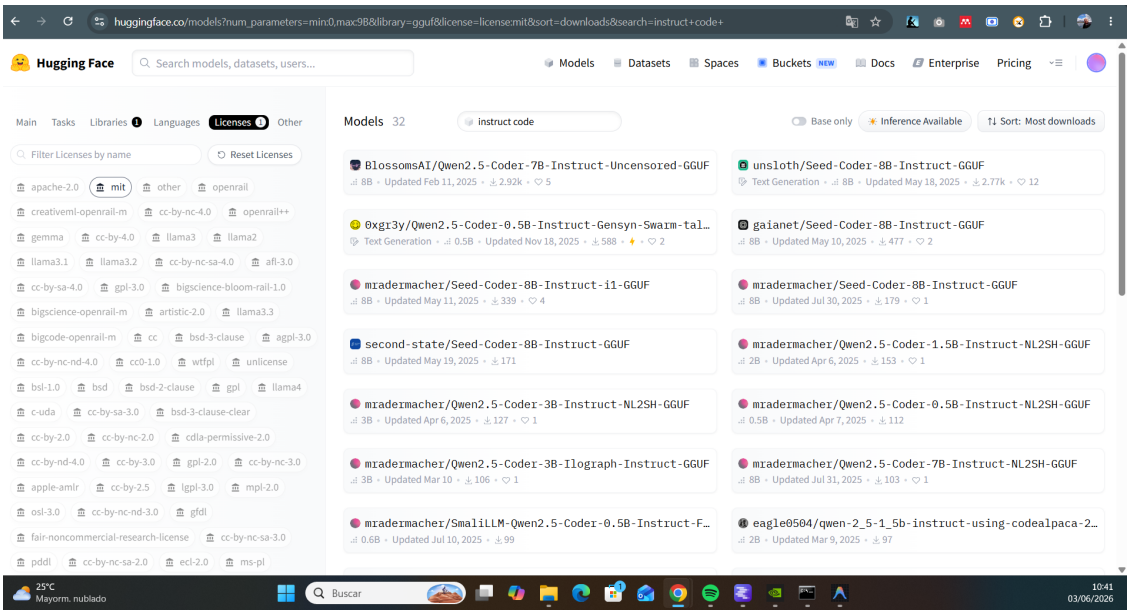


Figura 4.6: Resultados de búsqueda en Hugging Face bajo licencia Llama 3.

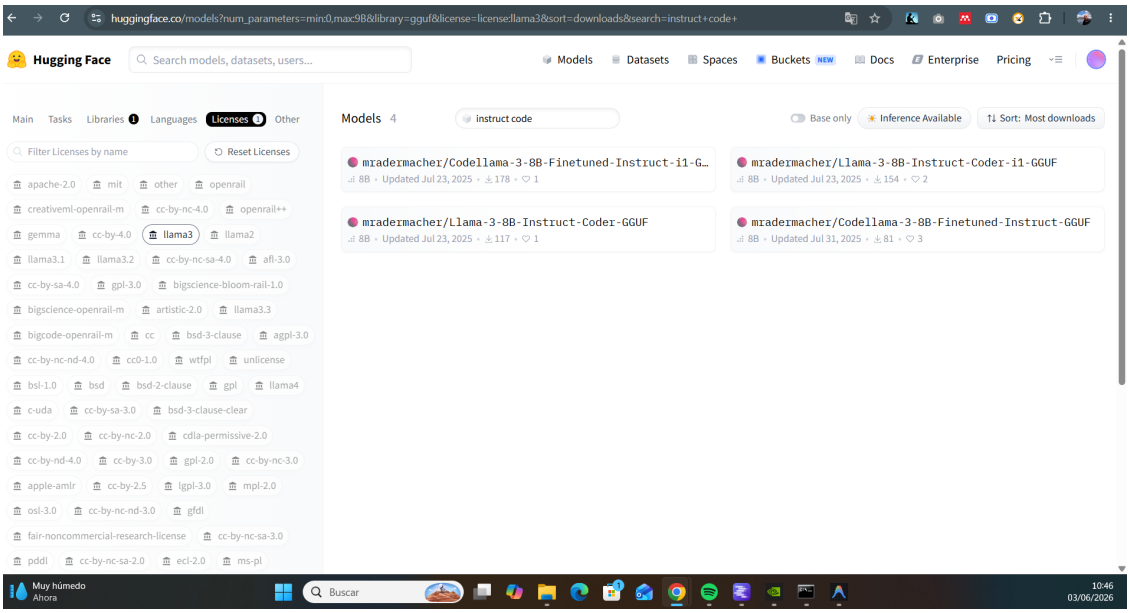


Tabla 4.7:
Especificaciones Técnicas y Justificación Empírica de los Modelos Seleccionados

Modelo	Descargas	Licencia	Parám.	Peso	Justificación Basada en la Evidencia (Hugging Face)
Qwen2.5-Coder-7B-Instruct	148k	Apache 2.0	7B	∼4.3 GB (Q4_K_M)	**Líder Absoluto de Descargas:** Evidenciado como el modelo número uno en la captura bajo los filtros de *instruct code* y $GGUF$ ($< 9B$).
Qwen2.5-Coder-1.5B-Instruct	74.5k	Apache 2.0	1.5B	∼1.8 GB (Q8_0)	**Segundo Modelo Más Adoptado:** Seleccionado directamente por ocupar la segunda posición global visible en la captura de la licencia Apache 2.0.
Qwen2.5-Coder-0.5B-Instruct	17.3k	Apache 2.0	0.5B	∼0.6 GB (Q8_0)	**Tercera Variante Qwen Más Descargada:** Visible en los resultados principales de la captura Apache 2.0, se incluye por su alto volumen empírico.
IBM Granite-8B-Code-Instruct-4k	6.91k	Apache 2.0	8B	∼4.8 GB (Q4_K_M)	**Mayor Adopción Fuera de Qwen (Apache 2.0):** Evidenciado en la primera captura como el modelo de otra arquitectura más descargado dentro de la licencia dominante.
Seed-Coder-8B-Instruct	2.77k	MIT	8B	∼4.8 GB (Q4_K_M)	**Alternativa Principal de Licencia MIT:** Representa la adopción más alta en la captura de la licencia MIT (excluyendo derivados de Qwen), sirviendo como control.

Capítulo 5

Construcción e Implementación

*Este capítulo se desarrolla para dar cumplimiento al **Objetivo Específico 2**, centrado en la construcción e implementación técnica del ecosistema multiagente y su integración con el protocolo MCP.*

5.1. Especificación del Ecosistema Híbrido Multiagente

Para la materialización del núcleo operativo definido en la arquitectura gTAA (el cual abarca transversalmente las Capas de Ejecución y Adaptación), se ha optado por implementar una topología híbrida que combina un Motor de Orquestación determinista con un Sistema Multiagente (MAS). La motivación principal detrás de esta decisión arquitectónica radica en el Principio de Responsabilidad Única y en la necesidad de favorecer la estabilidad del proceso (**RNF5**). Al delegar la orquestación a un núcleo puramente estructural, se mitiga el riesgo de que el flujo de control sufra desviaciones o alucinaciones, mientras que el razonamiento dinámico y la interacción física se aíslan en servicios especializados.

Cabe señalar una adaptación frente al planteamiento original del proyecto. Aunque en el **Objetivo Específico 2** y la **Fase 2** se mencionó un único "orquestador inteligente (Agente)" para actuar como cliente MCP, durante el diseño se identificó que concentrar múltiples responsabilidades (orquestación, interpretación semántica y localización visual) en un solo componente podría resultar ineficiente. Esta decisión técnica se sustenta además en el uso de modelos de lenguaje de menor escala (*Small Language Models*), los cuales tienden a experimentar degradación cognitiva y mayor riesgo de alucinación si se saturan con múltiples tareas en su *prompt*. Por ello, el concepto original se orientó hacia un ecosistema distribuido: un motor central coordina el flujo, mientras que agentes especializados asumen tareas específicas. Este enfoque modular se alinea con la meta original de integrar razonamiento LLM y MCP, buscando favorecer una mayor estabilidad, precisión y viabilidad económica.

En este ecosistema colaborativo interactúan un sistema de control central y tres agentes de Inteligencia Artificial, cada uno con roles claramente delimitados y mapeados a las capas funcionales del diseño:

5.1.1. Symphony: Orquestador Híbrido Multi-Agente

Symphony se concibe como el cerebro direccional del flujo de ejecución, construido en Python utilizando el framework LangGraph. A diferencia de un orquestador puramente determinista, Symphony actúa como el puente cognitivo inicial: emplea un Modelo de Lenguaje Pequeño/Mediano

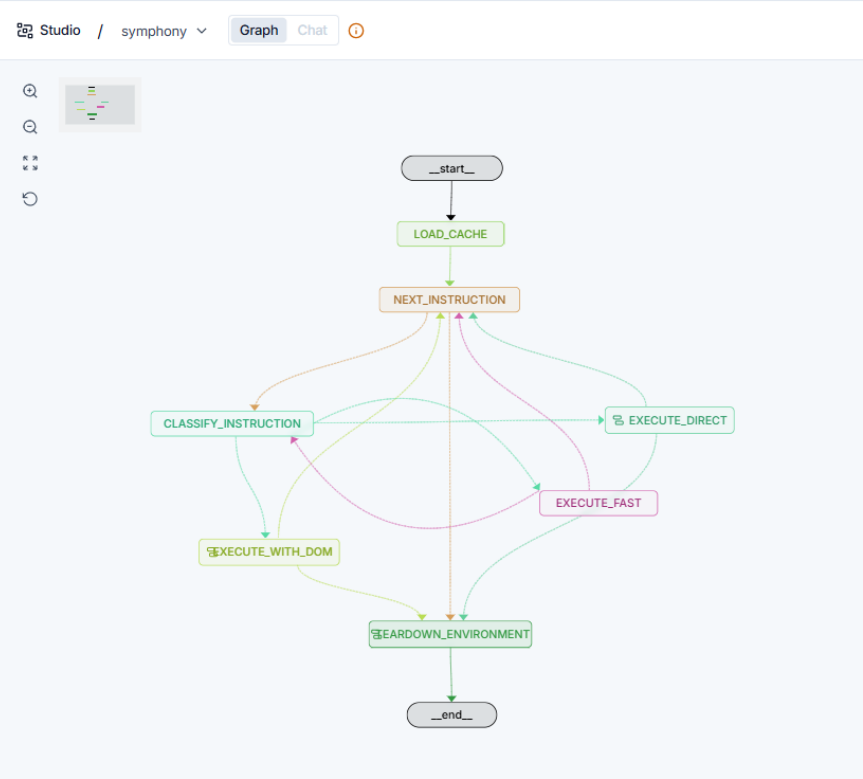
(LLM) en su nodo de clasificación para evaluar semánticamente las instrucciones en lenguaje natural, determinando si requieren iteración con el DOM, si corresponden a aserciones lógicas, o si son de ejecución rápida. A partir de este análisis de intención, delega dinámicamente la tarea al agente especializado correspondiente.

Trazabilidad Arquitectónica:

- **Capa gTAA:** Núcleo cognitivo de la Capa de Ejecución.
- **Requerimientos Cubiertos: RF2** (Orquestación Inteligente) y **RNF4** (Transparencia, a través de justificaciones estructuradas durante la clasificación).
- **Repositorio oficial:** `https://github.com/narvaezdario91/mdt-server`

Topología LangGraph:

Figura 5.1: Topología del grafo de estados para el orquestador Symphony.



Como se ilustra en la **Figura 5.1**, la topología del orquestador se estructura como un grafo de estado condicional que bifurca el flujo según la evaluación semántica. El grafo de Symphony está compuesto por los siguientes nodos principales:

- `LOAD_CACHE`: Interroga el almacenamiento local para recuperar rutas de ejecución históricas antes de invocar modelos cognitivos.

- `CLASSIFY_INSTRUCTION`: Analiza semánticamente la instrucción y decide la estrategia de enrutamiento óptima (*Fast Route*, *DOM Route* o *Direct Route*).
- `EXECUTE_*`: Ramificaciones de ejecución dinámicas que comisionan la interacción a los subagentes según el nivel de abstracción requerido.

5.1.2. Argos: Agente Localizador y de Resiliencia

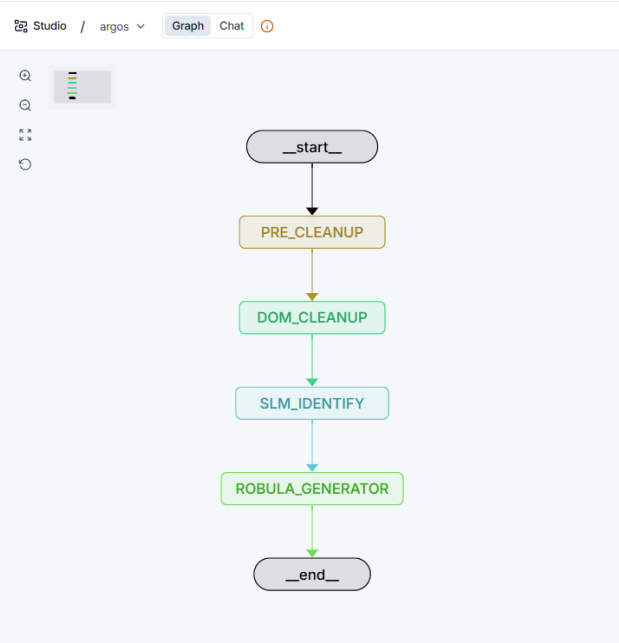
Argos funge como el componente visual y analítico de la ejecución. Es un agente localizador dotado con capacidades de auto-recuperación semántica (*self-healing*). Su tarea es recibir la descripción del elemento objetivo, analizar el Modelo de Objetos del Documento (DOM) actual, e inferir un localizador XPath técnico robusto capaz de soportar cambios estructurales imprevistos en la interfaz.

Trazabilidad Arquitectónica:

- **Capa gTAA:** Capa de Ejecución (Módulo de Visión/Localización).
- **Requerimientos Cubiertos: RF4** (Resiliencia / *Self-healing*).
- **Repositorio oficial:** `https://github.com/narvaezdario91/mdt-server`

Topología LangGraph:

Figura 5.2: Topología del grafo de estados para el agente localizador Argos.



Como se observa en la **Figura 5.2**, la topología de este agente exhibe un diseño lineal y directo, orientado estrictamente al procesamiento y extracción de datos visuales. El flujo de procesamiento de Argos se compone de los siguientes nodos:

- `PRE_CLEANUP` y `DOM_CLEANUP`: Filtro y sanitización del HTML crudo para eliminar etiquetas invisibles o innecesarias, mitigando el consumo de tokens.

- `SLM_IDENTIFY`: Identificación visual de los atributos del elemento deseado empleando razonamiento contextual.

- `ROBULA_GENERATOR`: Ensamblaje algorítmico de un selector XPath resiliente.

5.1.3. Nexus: Agente de Adaptación y Validación Semántica

Nexus representa el componente de acción directa sobre el Sistema Bajo Prueba (SUT) y el evaluador final de las operaciones. Actúa como un *driver* técnico inteligente y agnóstico, compatible con motores como Playwright y Selenium. Mediante el uso del protocolo *Model Context Protocol* (MCP), efectúa la interacción física requerida. Crucialmente, Nexus posee un nodo interno de validación (*validate_result*) basado en un Modelo de Lenguaje (LLM) que contrasta la salida cruda del navegador (textos de error o estados parciales del DOM) contra la intención original en lenguaje natural para verificar semánticamente si la acción o aserción realmente se cumplió de forma lógica, y no solo técnica.

Trazabilidad Arquitectónica:

- **Capa gTAA:** Capa de Adaptación.

- **Requerimientos Cubiertos: RF3** (Agnosticismo de Motor), **RF5** (Interoperabilidad MCP) y **RNF2** (Portabilidad).

- **Repositorio oficial:** `https://github.com/narvaezdario91/mdt-server`

Topología LangGraph:

De acuerdo con la topología presentada en la **Figura 5.3**, este agente se diseña con un enfoque iterativo y cíclico para manejar la incertidumbre de la interacción física. Nexus implementa un ciclo de ejecución con capacidad de autorecuperación local mediante los siguientes nodos:

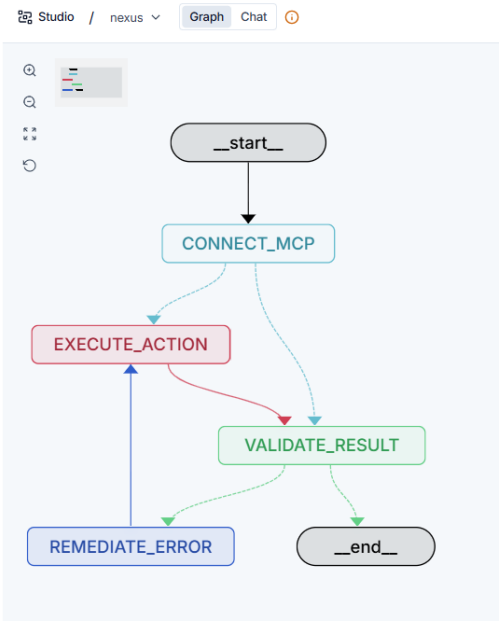
- `CONNECT_MCP`: Inicialización y negociación del protocolo de contexto con el motor de pruebas.

- `EXECUTE_ACTION`: Emisión del comando físico crudo sobre la interfaz del navegador.

- `VALIDATE_RESULT`: Auditoría cognitiva post-ejecución mediante LLM para dictaminar si el estado resultante satisface semánticamente la instrucción original.

- `REMEDIATE_ERROR`: Bucle de recuperación reactivo diseñado para corregir anomalías o fallas de estado.

Figura 5.3: Topología del grafo de estados para el agente de adaptación Nexus.



5.2. Dinámica Operativa: Flujo de Interacción Híbrido Cliente-Servidor

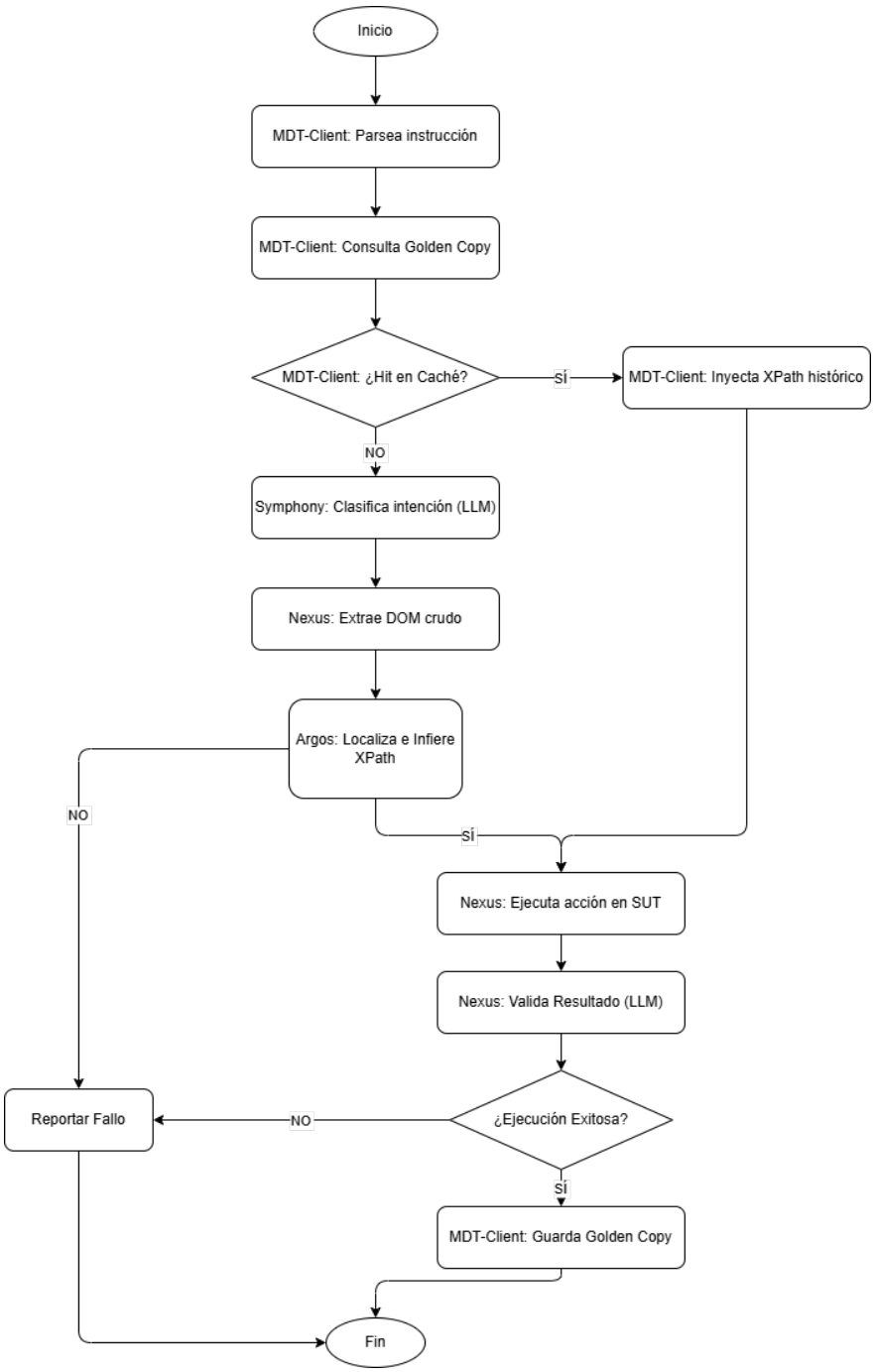
Para promover la estabilidad funcional y optimizar el consumo de contexto (LLM), el ecosistema se articula mediante una separación de responsabilidades entre el cliente (MDT-Client) y el servidor (MDT-Server). Como se ilustra en el diagrama de actividad (Figura 5.4), la arquitectura incorpora mecanismos lógicos de decisión temprana directamente en la herramienta cliente. Esta evaluación preventiva del modelo de contexto (*Cache Replay*) permite eludir procesos computacionalmente costosos a nivel de servidor cuando existen resoluciones históricas estables.

Este flujo algorítmico general se operacionaliza mediante la siguiente secuencia de interacciones, la cual queda resumida cronológicamente en el diagrama de secuencia (Figura 5.5) presentado al final de la sección:

1. **Evaluación Preventiva de Caché (MDT-Client):** Antes de invocar capacidades de inteligencia artificial, el cliente (MDT-Client) interroga su almacén de resoluciones local (*Golden Copy*). Si la instrucción posee una resolución técnica históricamente exitosa (Caché Hit), el cliente inyecta el XPath utilizado previamente, optimizando la latencia y el consumo de LLM (**RF6, RNF5**).

2. **Invocación y Clasificación Semántica:** Ante la ausencia de conocimiento previo (Caché Miss), el cliente envía la instrucción en lenguaje natural a la API del MDT-Server. El orquestador **Symphony** intercepta el mensaje y, mediante un LLM, evalúa semánticamente

Figura 5.4: Diagrama de Actividad: Algoritmo de orquestación, manejo de caché y validación semántica.



la intención para determinar la estrategia de ejecución (*Fast Route, DOM Route* o *Direct Route*). Si Symphony detecta que la instrucción es una aserción o validación (`is_assertion = True`), se omite el uso de cualquier caché local (evadiendo la *Fast Route*) para priorizar la extracción del estado real y actual de la interfaz.

```
Entrada a Symphony: "Escribe 'Admin' en el campo de texto Username"

Salida del Nodo de Clasificacion (JSON):
{
  "requires_dom": true,
  "is_assertion": false,
  "action": "type",
  "attributes": {"value": "Admin"}
}
```

Listing 4: Ejemplo de clasificación de intención por el orquestador Symphony.

3. **Extracción del Estado de la Interfaz:** Al determinar que se requiere interacción con el DOM, Symphony delega en el agente **Nexus** la tarea de extraer el Modelo de Objetos del Documento actual. Nexus interactúa con el navegador subyacente y devuelve el HTML crudo minimizado de la página.

4. **Delegación de Ubicación Resiliente:** Con el DOM recuperado, Symphony ensambla el paquete de petición y lo envía al agente **Argos**. Este escanea el árbol DOM para deducir y retornar el selector XPath más estable que referencie al elemento deseado (**RF4**).

```
Entrada Argos (JSON):
{
  "originalHtml": "<!DOCTYPE html><html>...</html>",
  "description": "campo de texto Username"
}

Salida Argos (JSON):
{
  "robustXpath": "//input[@name='username' or @placeholder='Username']",
  "reasoning": "Se identifico el input asociado al placeholder Username",
  "status": "success",
  "targetId": "temp-id-42",
  "errors": []
}
```

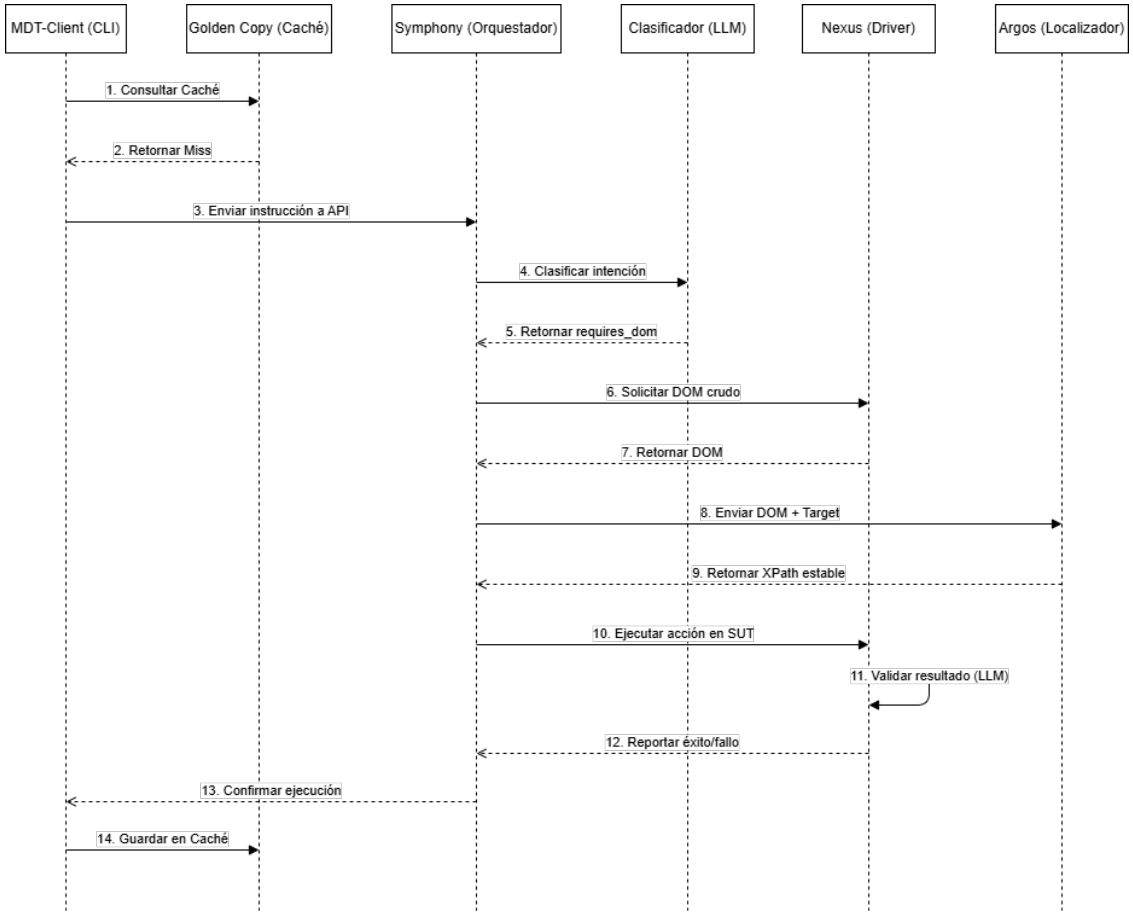
Listing 5: Ejemplo de inferencia de localizador por el Agente Argos.

5. **Ensamblaje y Ejecución Técnica:** Con el conjunto completo de datos (acción, XPath y atributos), Symphony comisiona la interacción técnica final al agente **Nexus**. Este adapta la solicitud a los requerimientos del servidor MCP e interactúa físicamente con el SUT (**RF3**, **RF5**).

6. **Validación Semántica de Resultados:** Tras la ejecución, el nodo interno *validate_result* de Nexus procesa la salida arrojada por el navegador (ej. códigos HTML, alertas, mensajes de error) empleando un LLM. El agente contrasta cognitivamente si el estado resultante satisface la instrucción original en lenguaje natural (paso crítico en escenarios de aserción), determinando así el estado final de la instrucción.

7. **Consolidación y Auditoría:** Confirmado el éxito de la operación (y superada la validación semántica) por el servidor, el MDT-Client recibe la respuesta, inserta la nueva resolución técnica en su Modelo de Contexto (*Golden Copy*) para optimizar iteraciones futuras, y genera la traza respectiva en su bitácora de auditoría de reportes (**RF7**).

Figura 5.5: Diagrama de Secuencia: Trazabilidad del flujo de comunicación entre el motor central y los agentes cognitivos especializados.



Capítulo 6

Análisis de Ejecuciones Experimentales y Evaluación ISO/IEC 25010

El presente apartado da respuesta al **Objetivo Específico 3** de la investigación, presentando un análisis empírico y crítico basado en los resultados consolidados de las ejecuciones experimentales. El propósito es contrastar el comportamiento del ecosistema multiagente Meta-Driven Testing (MDT) frente a la línea base de la industria (POM), evaluando la eficiencia de la solución propuesta en términos de **Mantenibilidad** y **Portabilidad**[1], según los criterios de la norma de calidad ISO/IEC 25010.

6.1. Diseño de Pruebas y Metodología Experimental

El diseño experimental se estructuró para someter al ecosistema MDT a condiciones exigentes frente a cambios evolutivos de la interfaz de usuario, empleando como Sistema Bajo Prueba (SUT) a *eShop*, una aplicación web transaccional representativa de comercio electrónico basada en *.NET Aspire*. El entorno de ejecución cognitivo estuvo soportado por el modelo de lenguaje de tamaño reducido `qwen:2.5-coder-instruct-8b`, desplegado mediante inferencia local para garantizar aislamiento y reducir latencias de red. Para establecer la línea base corporativa tradicional, los flujos paralelos de control (POM) se codificaron bajo *Selenium WebDriver 4.27* y *Cucumber Java 7.20*.

6.1.1. Topología Distribuida del Entorno

El laboratorio se estructuró dividiendo las responsabilidades lógicas y de hardware en dos nodos físicos de cómputo local interconectados:

- **Host 1 (Orquestación y Ejecución):** Aloja los componentes principales del experimento: la instancia de *Jenkins* encargada de automatizar el *pipeline* de Integración Continua (CI/CD), el entorno *Docker* para instanciar el ecosistema *.NET Aspire* de *eShop*, y el entorno de ejecución (*Node.js* y *Python*) para los agentes de MDT (*MDT-Client* y *MDT-Server*).

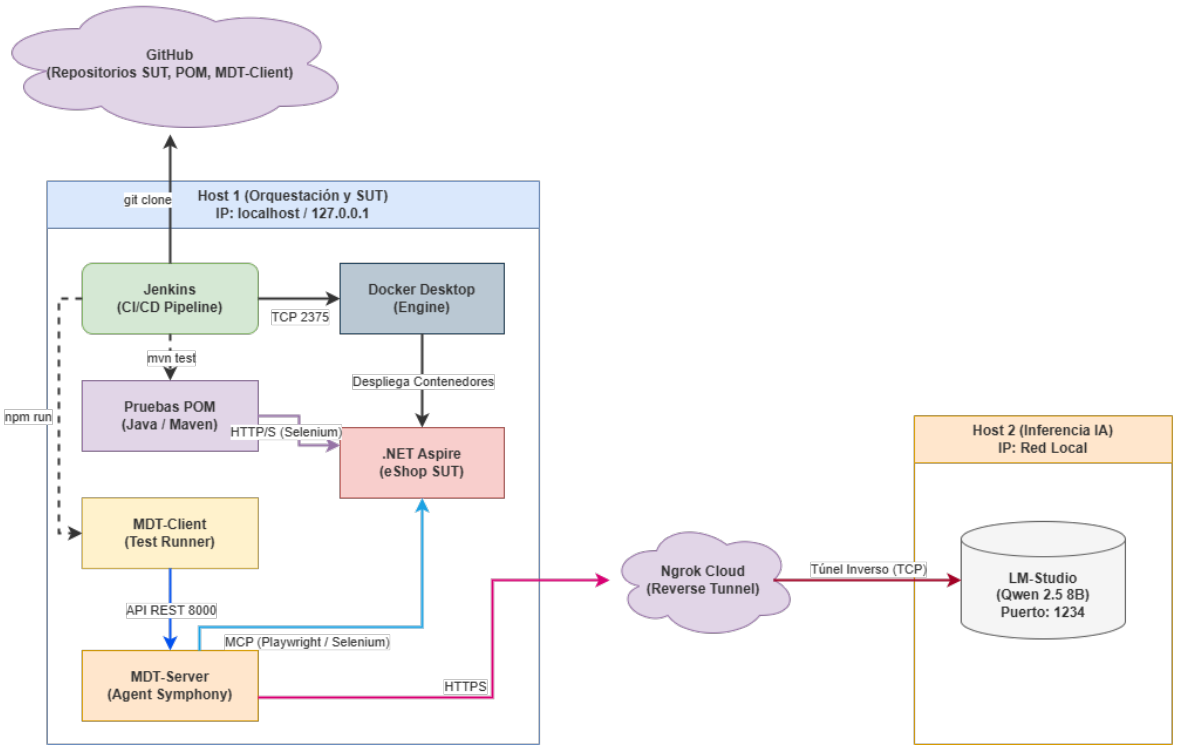
[1] Cabe aclarar que, si bien el Objetivo 3 se formuló utilizando el término original de "Portabilidad", en la actualización 2023 del estándar ISO/IEC 25010 esta dimensión fue reestructurada. En el presente análisis, la portabilidad esperada se evalúa y operacionaliza formalmente a través de la característica de **Flexibilidad** y su subcaracterística de **Capacidad de Reemplazo** (*Replaceability*), midiendo la viabilidad de sustituir el motor de automatización subyacente.

- **Host 2 (Inferencia Cognitiva):** Dedicado exclusivamente al cómputo de inferencia del LLM. Este nodo ejecuta *LM-Studio* sirviendo el modelo `qwen:2.5-coder-instruct-8b`. Para asegurar la conectividad transparente desde el Host 1, los servicios locales del Host 2 se exponen a través de un túnel inverso seguro aprovisionado mediante *Ngrok*.

Esta separación de responsabilidades asegura que las pruebas no sufran degradación de rendimiento a causa del alto consumo de recursos físicos (GPU/CPU) provocado por la inferencia del LLM.

La Figure 6.1 expone esta arquitectura de red, describiendo el flujo desde la obtención de los repositorios en GitHub mediante Jenkins, la inicialización del entorno Docker, y la ejecución de los marcos de prueba (POM y MDT). En el esquema se documenta la comunicación inter-nodos y la adopción del estándar Model Context Protocol (MCP) que emplea el servidor MDT para interactuar con el sistema bajo prueba, mientras consulta de forma remota al nodo de inferencia.

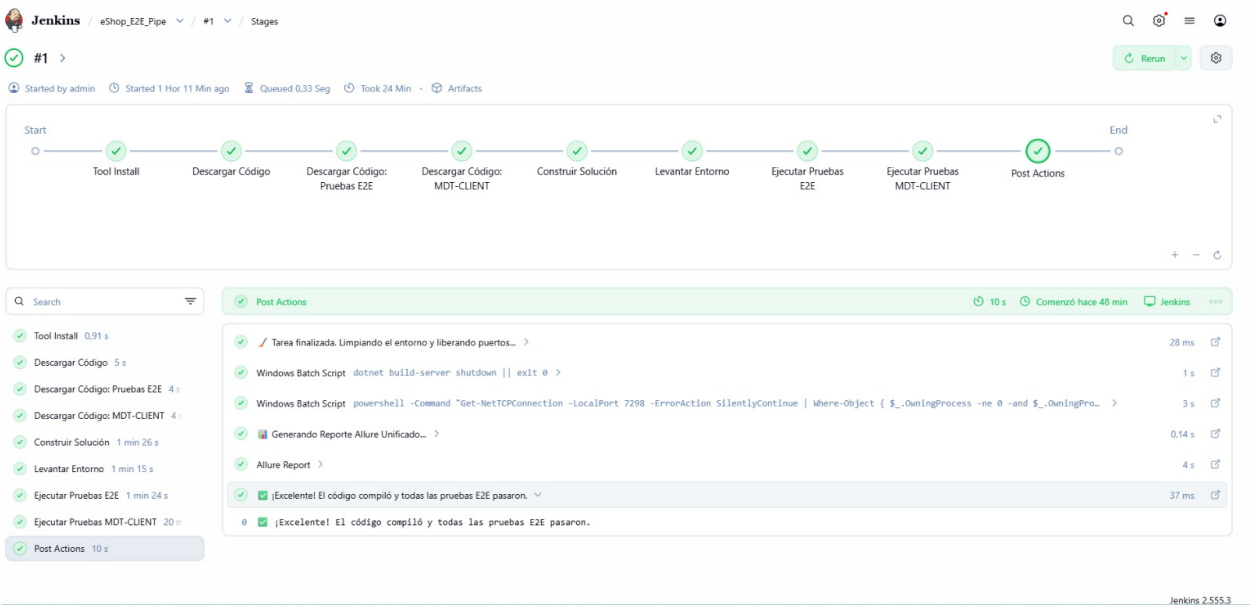
Figura 6.1: Arquitectura de red y flujos de comunicación en la topología distribuida del laboratorio.



6.1.2. Automatización mediante Pipeline CI/CD

Para ejecutar la validación experimental de manera controlada y mitigar el sesgo de intervención humana, se construyó un pipeline de Jenkins automatizado que gestiona la descarga, construcción, ejecución y recolección de resultados de ambos enfoques (convencional POM y el propuesto MDT).

Figura 6.2: Ejecución secuencial del Pipeline CI/CD en Jenkins orquestando el entorno experimental.



En lugar de depender de configuraciones manuales, la conectividad inter-nodos se inyecta dinámicamente. El fragmento de código (6) ilustra la sección de variables de entorno, donde se consolida la conexión TCP entre Jenkins y Docker local, y se direcciona el tráfico de IA al túnel Ngrok hacia el Host 2 (`OPENAI_API_BASE`).

```
environment {
    // Permite que Jenkins se conecte a Docker Desktop por TCP
    DOCKER_HOST = 'tcp://127.0.0.1:2375'
    ASPNETCORE_URLS = 'https://0.0.0.0:7298;http://0.0.0.0:5298'

    // Tunel Ngrok conectando al Host 2 (LM-Studio local)
    OPENAI_API_BASE = 'https://09b9-190-1-227-82.ngrok-free.app/v1'
    DEFAULT_MODEL = 'qwen:2.5-coder-instruct-8b'
}
```

Listing 6: Configuración de variables de entorno en Jenkinsfile.

Una vez configurado el entorno, el orquestador clona de forma independiente los repositorios fuente y levanta el contenedor del sistema *eShop*. Posteriormente, las etapas del *pipeline* inician de forma secuencial la ejecución de ambas baterías de prueba, encapsuladas mediante bloques `catchError` (7) para asegurar que el proceso de auditoría y recolección de datos no se interrumpa ante fallos inducidos por mutaciones de UI.

```
stage('Ejecutar Pruebas E2E') {
    steps {
        dir('automatizacion_qa') {
            catchError(buildResult: 'UNSTABLE', stageResult: 'FAILURE') {
                bat 'mvn clean test'
            }
        }
    }
}
stage('Ejecutar Pruebas MDT-CLIENT') {
    steps {
        dir('mdt-client') {
            catchError(buildResult: 'UNSTABLE', stageResult: 'FAILURE') {
                bat 'npm install'
                bat 'npm run dev run -- --features scenarios/features/eShop --report allure'
            }
        }
    }
}
```

Listing 7: Fases de ejecución de pruebas automatizadas.

6.1.3. Consolidación de Resultados y Auditabilidad

Para garantizar la fiabilidad del análisis empírico, se estandarizó la recolección de evidencias de ambos ecosistemas a través del framework analítico *Allure*. Como se observa en la fase *post* del pipeline (8), el sistema agrupa los directorios de resultados de Maven (Java) y Node (TypeScript) en un único reporte centralizado.

```
post {
    always {
        // ... limpieza concurrente de procesos ...
        echo 'Generando Reporte Allure Unificado...'
        allure includeProperties: false, jdk: '', results: [
            [path: 'automatizacion_qa/target/allure-results'],
            [path: 'mdt-client/allure-results']
        ]
    }
}
```

Listing 8: Fase post de consolidación de reportes unificados Allure.

Esta infraestructura dota de rigor operativo a las mediciones del estudio, aislando las iteraciones del experimento de la manipulación humana directa y garantizando que los datos de Modificabilidad e Intervención Humana, tabulados en los resultados, provengan de un entorno controlado y objetivo.

6.1.4. Casos de Prueba y Estrategia de Mutación

Para validar la tolerancia a fallos y la resiliencia del marco propuesto, se seleccionaron cuatro Casos de Prueba (TC) dirigidos intencionalmente a flujos funcionales críticos para el negocio

(autenticación y carrito de compras), detallados en la Tabla 6.1. Esta selección busca probar el desempeño del sistema en rutas donde un falso positivo resulta inaceptable.

Tabla 6.1: Descripción de los Casos de Prueba (TC) ejecutados en el experimento.

Identificador	Descripción del Caso de Prueba
TC1	Agregar producto al carrito exitosamente
TC2	Inicio de sesión exitoso
TC3	Eliminar producto del carrito exitosamente
TC4	Visualización de detalle de producto

Sobre estos flujos críticos, se aplicó una estrategia de inyección controlada de mutaciones (M1 a M6) en el código fuente del *frontend*. La selección de estas alteraciones no es aleatoria; fueron diseñadas para simular la degradación natural que sufren los localizadores técnicos en un ciclo de vida de desarrollo ágil (cambios de texto cosmético, reestructuración semántica y alteración de jerarquía del DOM). La Tabla 6.2 detalla la alteración exacta realizada en cada iteración del experimento.

Tabla 6.2: Descripción de las mutaciones inyectadas en la interfaz de usuario del SUT.

Mutación	Descripción del Cambio Aplicado	Componente / Elemento Afectado
M1	Renombrar el atributo `aria-label` de "Sign in.ª "Log in"	Menú de usuario (Navegación)
M2	Cambiar el texto guía (*placeholder*) de usuario	Formulario de Autenticación
M3	Modificar el texto visible del botón a .Ádd to cart"	Catálogo de Productos
M4	Cambiar mensaje de bienvenida por "Welcome to eShop"	Página Principal (Aserción de negocio)
M5	Envolver el botón de Login dentro de una etiqueta `<div>`	Formulario de Autenticación
M6	Convertir la etiqueta `<button>` a `<input>`	Formulario de Autenticación

6.2. Alineación Operacional con ISO/IEC 25010:2023

Para asegurar que la evaluación del experimento sea estructurada y auditable contra estándares internacionales, las métricas extraídas de las ejecuciones se han mapeado formalmente a características del modelo de calidad de producto de la norma ISO/IEC 25010:2023. La Tabla 6.3 presenta este mapeo, especificando para cada subcaracterística evaluada la métrica concreta y el instrumento de medición utilizado en el presente experimento.

Tabla 6.3: Operacionalización de las características ISO/IEC 25010:2023 evaluadas en el experimento.

Característica ISO	Subcaracterística	Métrica Operacionalizada	Instrumento de Medición
Flexibilidad	Capacidad de Reemplazo	Archivos y líneas modificadas para migrar de motor	Inspección del código base y configuración
Mantenibilidad	Modificabilidad	Tasa de Fragilidad (%) ante mutaciones de UI	Reportes unificados de ejecución (Allure)
Mantenibilidad	Modificabilidad	Intervención humana requerida (archivos, tiempo)	Registro de correcciones post-mutación

Este mapeo garantiza que el análisis empírico expuesto a continuación responda directamente al comportamiento de estas subcaracterísticas de calidad, vinculando cada hallazgo con un criterio formal del estándar.

6.3. Resultados Comparativos de Ejecución (MDT vs POM)

El experimento constó de 10 ejecuciones iterativas, combinando el motor físico (Playwright o Selenium), el uso del Caché Semántico y las mutaciones descritas. La Tabla 6.4 documenta los resultados consolidados de los cuatro Casos de Prueba (TC), contrastando el comportamiento de MDT con POM tal como se evidenció en los registros experimentales.

Nota terminológica: Se utiliza `FAIL` cuando el caso de prueba finaliza la ejecución pero reporta una violación a la regla de negocio esperada (falla legítima). Se utiliza `BROKE` cuando el script colapsa prematuramente debido a la incapacidad de ubicar los selectores en el DOM (fragilidad del script).

Adicionalmente, la Figure 6.3 evidencia visualmente el histórico de estas ejecuciones, mostrando la trazabilidad de los reportes consolidados a lo largo de las iteraciones del experimento.

Figura 6.3: Histórico consolidado de ejecuciones experimentales (MDT vs POM).

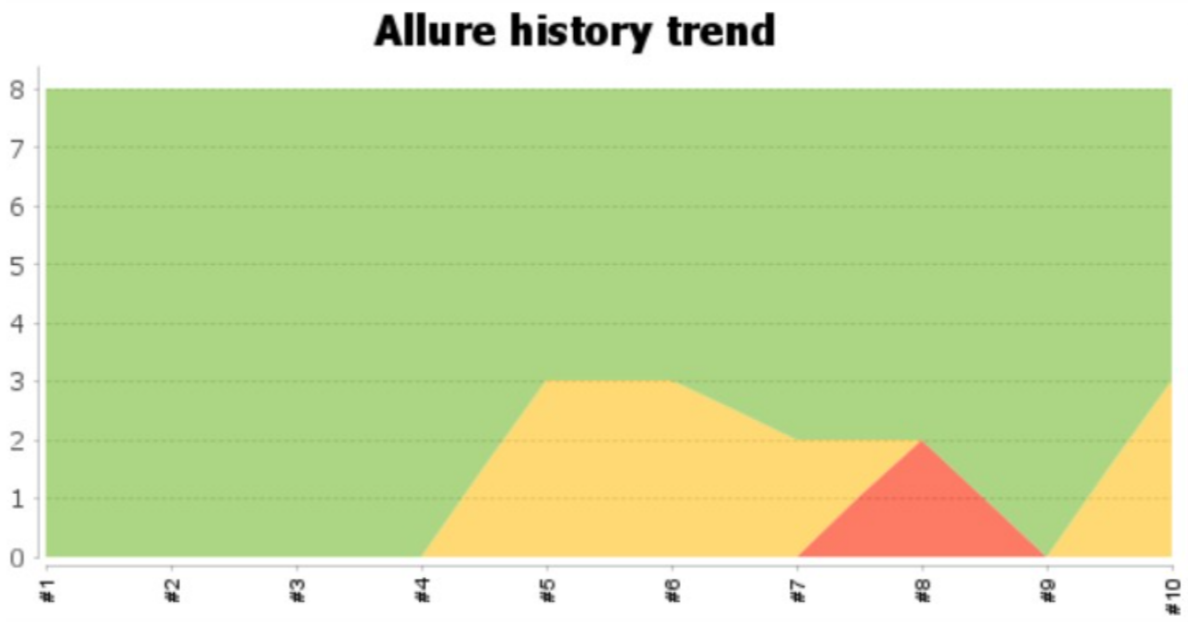


Tabla 6.4: Matriz comparativa de resultados de ejecución por casos de prueba entre la arquitectura MDT y la línea base POM.

Ejecución	Configuración MDT	Caché	Estado SUT	Resultados MDT				Resultados POM			
				TC1	TC2	TC3	TC4	TC1	TC2	TC3	TC4
1	MCP Playwright	NO	Sin Mutaciones	OK	OK	OK	OK	OK	OK	OK	OK
2	MCP Playwright	SÍ	Sin Mutaciones	OK	OK	OK	OK	OK	OK	OK	OK
3	MCP Selenium	NO	Sin Mutaciones	OK	OK	OK	OK	OK	OK	OK	OK
4	MCP Selenium	SÍ	Sin Mutaciones	OK	OK	OK	OK	OK	OK	OK	OK
5	MCP Selenium	SÍ	Mutación M1	OK	OK	OK	OK	BROKE	BROKE	BROKE	OK
6	MCP Selenium	SÍ	Mutación M2	OK	OK	OK	OK	BROKE	BROKE	BROKE	OK
7	MCP Selenium	SÍ	Mutación M3	OK	OK	OK	OK	BROKE	BROKE	OK	OK
8	MCP Selenium	SÍ	Mutación M4	FAIL	OK	OK	OK	FAIL	OK	OK	OK
9	MCP Selenium	SÍ	Mutación M5	OK	OK	OK	OK	OK	OK	OK	OK
10	MCP Selenium	SÍ	Mutación M6	OK	OK	OK	OK	BROKE	BROKE	BROKE	OK

6.4. Medición Operacional de Resultados (MDT vs POM)

Al observar los datos consolidados en la Tabla 6.4, el comportamiento de las ejecuciones se analiza conectando las métricas empíricas con el mapeo operacional de la norma ISO/IEC 25010 (Flexibilidad y Mantenibilidad):

6.4.1. Resultados de Flexibilidad: Archivos y Líneas Modificadas

En los escenarios sin mutaciones (1 a 4), se ejecutaron los mismos casos de prueba utilizando metadatos idénticos, pero alternando el motor de ejecución físico entre `MCP Playwright` y `MCP Selenium`. El estado `OK` generalizado demuestra una alta paridad de efectividad, pero más importante aún, evidencia la **Flexibilidad** (Capacidad de Reemplazo) del ecosistema.

Para cuantificar el esfuerzo de migración entre motores, la Tabla 6.5 contrasta las intervenciones requeridas en cada enfoque.

Tabla 6.5: Esfuerzo de migración entre motores de ejecución: MDT vs POM.

Métrica de Migración	MDT	POM Convencional
Archivos de prueba modificados	0	Reescritura de la suite
Líneas de código técnico alteradas	0	∼200+ líneas
Componente intervenido	1 variable (`.env`)	APIs nativas acopladas
Esfuerzo estimado	$<$ 1 minuto	Días / Semanas

En síntesis, mientras que en un modelo POM tradicional la migración de motor exige una reescritura del código técnico acoplado (clases Page Object, configuración del driver, dependencias Maven), la arquitectura MDT, soportada por el protocolo MCP, permite el intercambio de motores de ejecución con una intervención mínima en los activos de prueba.

6.4.2. Resultados de Mantenibilidad: Tasa de Fragilidad ante Mutaciones

Al inyectar las mutaciones orientadas a modificar textos y atributos semánticos, la matriz revela una diferencia significativa. En POM, las mutaciones M1, M2 y M6 provocan un estado de colapso sistemático (`BROKE`) en los Casos de Prueba 1, 2 y 3. De igual forma, la mutación M3 generó rupturas (`BROKE`) en los Casos de Prueba 1 y 2.

Para cuantificar el impacto de la arquitectura frente a esta fragilidad tradicional, la Tabla 6.6 consolida las métricas globales de los 40 escenarios ejecutados. Se define la *Tasa de Finalización Estructural* como la capacidad del orquestador para completar el flujo de navegación sin colapsar por pérdida de selectores (suma de estados `OK` y `FAIL` legítimo), y la *Tasa de Fragilidad* como el porcentaje de escenarios que fueron interrumpidos prematuramente (`BROKE`).

Tabla 6.6: Consolidado cuantitativo de resiliencia y fragilidad entre MDT y POM.

Métrica Global (Base de 40 Escenarios)	**Arquitectura MDT**	**Línea Base (POM)**
Total de escenarios superados (`OK`)	39	28
Total de fallos legítimos (`FAIL`)	1	1
Total de rupturas estructurales (`BROKE`)	0	11
Tasa de Finalización Estructural	**100 %** (40/40)	**72.5 %** (29/40)
Tasa de Fragilidad (Deuda Técnica)	**0 %** (0/40)	**27.5 %** (11/40)

Los datos muestran que, mientras el modelo POM base presentó una tasa de fragilidad del 27.5 %, la arquitectura MDT alcanzó una tasa de finalización estructural del 100 % en el contexto del experimento. La siguiente subsección complementa este análisis al examinar el esfuerzo humano derivado de dicha fragilidad.

6.4.3. Resultados de Mantenibilidad: Intervención Humana Requerida

La propensión al fallo en el modelo tradicional exige intervención manual constante de un ingeniero para actualizar selectores, afectando negativamente la mantenibilidad. En POM, las 11 pruebas interrumpidas (`BROKE`) requieren refactorización manual de selectores, demandando inversión de tiempo y esfuerzo humano para restaurar la suite operativa.

Para cuantificar este esfuerzo operativo, la Tabla 6.7 contrasta el volumen de modificaciones técnicas requeridas para estabilizar la suite de automatización tras la inyección de mutaciones. Es importante destacar que, gracias a la reutilización de código inherente al patrón POM (centralización de localizadores), los 11 escenarios interrumpidos en realidad se originan a partir de solo 4 mutaciones críticas únicas (M1, M2, M3, M6) en los elementos de la interfaz.

Tabla 6.7: Cuantificación del esfuerzo de refactorización requerido ante cambios evolutivos de UI.

Métrica de Mantenimiento	Arquitectura MDT	Línea Base (POM)
Total de pruebas interrumpidas (`BROKE`)	0 escenarios	11 escenarios
Elementos raíz mutados que exigieron corrección	0 elementos	4 elementos únicos
Líneas de código (LoC) refactorizadas por elemento	0 líneas	1 línea (cadena del selector)
Esfuerzo Total de Refactorización (LoC)	**0 líneas**	**4 líneas de código**
Factor de amplificación de fallos en cascada	**0x**	**2.75x** (11 / 4)
Tasa de auto-reparación ante mutación	**100 %** (4/4)	**0 %** (0/4)

Como se evidencia en los resultados, la arquitectura MDT eliminó por completo la necesidad de intervención manual en los escenarios mutados, logrando una tasa de auto-reparación del 100 % frente a las alteraciones estructurales. Por el contrario, la línea base (POM) evidenció un factor de amplificación de fallos de 2.75x, lo que significa que por cada elemento mutado se interrumpieron en cascada casi tres escenarios de prueba. Desde la perspectiva del estándar de calidad, la mitigación de este impacto en cascada y la eliminación del esfuerzo de refactorización (0 LoC alteradas) representan una mejora sustancial de la **Modificabilidad**, erradicando el micro-mantenimiento de selectores que convencionalmente se requiere ante cambios en la interfaz.

6.4.4. Usabilidad en el Mantenimiento

Además de la mitigación de fragilidad estructural, la arquitectura MDT favorece activamente la inclusión de analistas funcionales y expertos de dominio. Al interactuar con el ecosistema, el analista no sacrifica usabilidad frente a marcos tradicionales; por el contrario, su experiencia mejora al abstenerse de manipular código imperativo. Para ilustrar cómo se registra un cambio evolutivo desde la perspectiva del analista funcional, considérese el siguiente escenario: el negocio requiere añadir una validación adicional en la pantalla de acceso. En el enfoque POM, este cambio exigiría localizar la clase Page Object correspondiente, identificar el selector CSS o XPath del nuevo elemento, y programar la aserción en código. En MDT, el analista solo necesita editar el archivo de metadatos YAML, como se muestra en el Listing 9.

```
# ANTES: Definicion original del paso de verificacion
"el sistema permite el acceso Orange":
  instructions:
    - "Verifica que este el texto 'Dashboard'"

# DESPUES: El analista funcional agrega una nueva validacion
"el sistema permite el acceso Orange":
  instructions:
    - "Verifica que este el texto 'Dashboard'"
    - "Verifica que aparezca el mensaje 'Bienvenido' en la cabecera"
```

Listing 9: Registro de un cambio evolutivo desde la perspectiva del analista funcional.

Como se observa, el cambio consiste en añadir una instrucción en lenguaje natural al repositorio

de metadatos. El analista no necesita conocer selectores del DOM, XPaths ni la API del motor de automatización; el ecosistema multiagente se encarga de delegar al agente LLM la localización e interacción con el nuevo elemento en tiempo de ejecución. Este enfoque facilita el sostenimiento de las pruebas directamente desde los metadatos semánticos, apoyando la mantenibilidad sin exigir conocimientos en lenguajes de programación.

6.5. Análisis Adicional y Hallazgos Secundarios

6.5.1. Determinismo en la Validación de Negocio (Ejecución 8)

La Ejecución 8 (mutación M4) representa el único escenario donde se alteró explícitamente la aserción final esperada por el negocio. En este punto, la tabla muestra que tanto MDT como POM adoptan idéntico comportamiento: ambos finalizan el flujo de la prueba y emiten un estado `FAIL` en el Caso de Prueba 1. Esto evidencia empíricamente que la alta tolerancia a modificaciones visuales de MDT no compromete el rigor analítico de la aserción final; el agente falla de manera determinista cuando la regla funcional de aceptación no se cumple.

6.5.2. Análisis de Eficiencia y Consumo de Tokens (LLM)

Para evaluar la viabilidad técnica de la arquitectura Meta-Driven Testing, se consolidó la telemetría referente al consumo de tokens del Modelo de Lenguaje Grande (LLM) durante las 10 ejecuciones experimentales. La Tabla 6.8 detalla el consumo desagregado en tokens de entrada (*prompts* y contexto del DOM) y salida (acciones y razonamiento del agente).

Tabla 6.8: Consumo consolidado de tokens LLM por ejecución experimental.

Ejecución	Configuración (Caché / Mutación)	Input Tokens	Output Tokens	Total Tokens	Tiempo POM	Tiempo MDT
1	Playwright / NO / Sin Mutaciones	35,892	4,821	40,713	52s	19m 15s
2	Playwright / SÍ / Sin Mutaciones	19,547	2,637	22,184	29s	7m 20s
3	Selenium / NO / Sin Mutaciones	36,390	4,902	41,292	32s	20m 03s
4	Selenium / SÍ / Sin Mutaciones	19,547	2,631	22,178	48s	5m 36s
5	Selenium / SÍ / Mutación M1	19,547	2,640	22,187	1m 12s	5m 33s
6	Selenium / SÍ / Mutación M2	19,547	2,640	22,187	1m 15s	5m 37s
7	Selenium / SÍ / Mutación M3	19,547	2,641	22,188	1m 24s	6m 08s
8	Selenium / SÍ / Mutación M4	19,547	2,644	22,191	1m 06s	6m 39s
9	Selenium / SÍ / Mutación M5	19,547	2,640	22,187	45s	5m 49s
10	Selenium / SÍ / Mutación M6	21,560	2,899	24,459	1m 21s	8m 18s

El contraste entre la Ejecución 3 (sin caché semántico, 41,292 tokens) y la Ejecución 4 (con caché habilitado, 22,178 tokens) muestra una disminución del **46.3 %** en el uso de tokens. Las Ejecuciones 5 a 9, donde el SUT sufrió mutaciones moderadas con el caché activo, mantuvieron un consumo estable (˜22,187 tokens), confirmando que las resoluciones históricas almacenadas absorbieron las variaciones de la interfaz sin requerir reprocesamiento por parte del LLM. En contraste, la Ejecución 10, donde la mutación M6 invalidó el caché y activó el protocolo de *self-healing*, incrementó el consumo a 24,459 tokens, lo que representa un recargo del **10.3 %** frente a la línea base con caché.

6.5.3. Discusión Crítica: El Trade-off de la Automatización basada en LLM

Los resultados revelan un balance de compensación (*trade-off*). La arquitectura MDT reduce el esfuerzo manual dedicado a la actualización de selectores, transfiriendo esta carga al entorno computacional mediante el consumo de tokens de inferencia del LLM. La viabilidad a largo plazo de la solución depende, por tanto, no solo de su robustez arquitectónica, sino también de la optimización de los algoritmos de caché y la transición hacia modelos de lenguaje más eficientes (como los SLMs) que reduzcan el costo computacional.

Cabe señalar que la inferencia del LLM local introduce una **penalidad temporal** frente a POM. Durante la validación empírica, la ejecución completa de la suite (4 casos de prueba) mediante el enfoque tradicional (POM) tomó en promedio **40 segundos** (˜10 segundos por caso de prueba individual). En contraste, el procesamiento guiado por agentes cognitivos locales con caché semántico habilitado requirió un promedio de **6,5 minutos** por suite (˜1,6 minutos por caso de prueba), y sin caché este tiempo ascendió a **19,6 minutos** por suite. Este incremento —de un orden de magnitud aproximado de 10x por caso de prueba— constituye el *trade-off* central de la solución: se intercambia velocidad de ejecución por adaptabilidad y reducción del esfuerzo de mantenimiento. Dado que esta latencia depende fuertemente de las capacidades de hardware (GPU/RAM) y de la optimización del modelo de lenguaje (LLM), la mejora de la *Eficiencia de Desempeño* mediante modelos más pequeños (SLMs) y algoritmos avanzados de inferencia se perfila como una línea crítica de trabajo futuro.

6.6. Amenazas a la Validez

Como parte del rigor investigativo, se identificaron variables intervinientes que pudieron inducir sesgos durante la evaluación empírica. La Tabla 6.9 documenta las amenazas evaluadas y los mecanismos de mitigación aplicados.

Tabla 6.9: Identificación y mitigación de amenazas a la validez de los resultados.

Tipo	Amenaza Evaluada	Estrategia de Mitigación
Interna	Sesgo temporal	Extracción automatizada desde Jenkins, garantizando aislamiento al reversar mutaciones entre iteraciones.
Interna	Aleatoriedad LLM	Instanciación de agentes con `temperature: 0` estricto y registro del modelo evaluado.
Interna	Muestra reducida	Delimitación a 4 pruebas y 6 mutaciones en un SUT, sin buscar generalización estadística amplia.
Externa	Generalización de dominios	Uso de *eShop*, un arquetipo con flujos representativos de aplicaciones web transaccionales.
Constructo	Subjetividad de esfuerzo	Operacionalización formal (Tabla 6.3) con métricas físicas: archivos, LoC y reportes (Allure).

6.7. Síntesis del Capítulo y Respuesta al Objetivo 3

El presente capítulo ha abordado la evaluación empírica de la solución propuesta, contrastando la arquitectura Meta-Driven Testing (MDT) frente al enfoque convencional Page Object Model (POM) bajo los criterios de la norma ISO/IEC 25010:2023. A partir de los datos recolectados en las 10 ejecuciones experimentales y los 40 escenarios evaluados, se presentan las siguientes observaciones organizadas según las métricas operacionalizadas (Tabla 6.3) que responden al **Objetivo Específico 3**:

- **Mantenibilidad — Modificabilidad (Tasa de Fragilidad):** La arquitectura MDT registró una Tasa de Fragilidad del 0 % frente al 27.5 % observado en POM (Tabla 6.6), alcanzando una Tasa de Finalización Estructural del 100 % en los 40 escenarios evaluados. Esta diferencia se atribuye al mecanismo de *self-healing* semántico y al caché de resoluciones de los elementos, que absorbieron las mutaciones inyectadas sin colapsar los flujos de navegación.
- **Mantenibilidad — Modificabilidad (Intervención Humana Requerida):** MDT eliminó por completo la necesidad de refactorización manual, registrando 0 líneas de código alteradas y una tasa de auto-reparación del 100 % ante las 4 mutaciones que afectaron selectores (Tabla 6.7). En contraste, POM requirió la corrección de 4 elementos únicos y evidenció un factor de amplificación de fallos en cascada de 2.75x (11 escenarios interrumpidos a partir de 4 mutaciones raíz).
- **Flexibilidad — Capacidad de Reemplazo (Archivos y Líneas Modificadas):** La migración entre los motores Playwright y Selenium en MDT se logró modificando una única variable de entorno (0 archivos de prueba y 0 líneas de código técnico alteradas), según lo evidenciado en la Tabla 6.5. En contraste, en POM esta migración implicaría una reescritura considerable de la suite técnica (clases Page Object, configuración del driver y dependencias acopladas al motor).

De forma complementaria, el análisis de eficiencia reveló que el caché semántico redujo el consumo de tokens del LLM en un 46.3 %, y que el recargo computacional del protocolo de auto-reparación se limitó a un 10.3 % adicional (Tabla 6.8), lo cual sugiere que el costo de inferencia se mantiene dentro de rangos operativamente viables en el contexto evaluado.

Estos hallazgos, acotados al alcance y las limitaciones del experimento descritas en la Tabla 6.9, proveen evidencia empírica favorable respecto a la capacidad de un enfoque dirigido por metadatos e integrado con agentes LLM para mejorar la mantenibilidad y flexibilidad de los activos de automatización de pruebas, en comparación con los patrones convencionales de la industria.

Capítulo 7

Conclusiones

La presente investigación logró materializar el objetivo general propuesto, demostrando la viabilidad técnica y operativa de una arquitectura de automatización de pruebas BDD dirigida por metadatos (MDT) e integrada mediante el Modelo de Protocolo de Contexto (MCP). A lo largo del diseño, construcción y validación experimental, se evidenció que la delegación de la ejecución a un ecosistema multiagente permite un desacoplamiento efectivo entre las reglas de negocio y la infraestructura técnica. A partir del análisis empírico frente a la norma de calidad ISO/IEC 25010, se exponen las siguientes conclusiones fundamentales que perfilan el impacto de esta solución:

7.1. Conclusiones sobre el Diseño Arquitectónico y Metadatos (Objetivo 1)

Reducción del código de integración (*Glue Code*) en BDD: La abstracción estructural mediante metadatos externos disminuyó la necesidad de programar artefactos intermediarios de código, una de las principales cargas técnicas en esquemas convencionales como el *Page Object Model* (POM). En los escenarios validados, esto se tradujo en una menor cantidad de código imperativo acoplado a la UI, favoreciendo activamente la inclusión de analistas funcionales y expertos de dominio en la creación y sostenimiento de las pruebas.

Evaluación según el origen de las herramientas: La comparación entre una solución impulsada corporativamente (Playwright) y una fundamentada en la comunidad (Selenium) demostró la viabilidad técnica de alternar motores subyacentes de forma agnóstica a través del protocolo MCP. Esta estrategia validó la portabilidad de la propuesta frente a arquitecturas dispares. Los hallazgos confirman que las directrices de diseño planteadas facilitan la interoperabilidad, mitigando de raíz el riesgo de dependencia tecnológica.

7.2. Conclusiones sobre el Ecosistema Multiagente (Objetivo 2)

Control y optimización mediante Agentes Atomizados: El ecosistema construido evidenció que la delegación de responsabilidades hacia agentes atomizados (*Symphony, Argos, Nexus*) permite mantener un control estructurado sobre el flujo de trabajo. Esta segregación resultó efectiva para optimizar los recursos computacionales y acotar el margen de alucinación del LLM, configurando un patrón metodológico robusto.

Resiliencia preservando el rigor analítico: Pese a la naturaleza probabilística intrínseca a los modelos fundacionales, el estudio demostró que la aplicación de restricciones sistémicas (como la temperatura en cero y *prompts* rígidamente estructurados) asegura un comportamiento orientado

al determinismo. El agente identificó correctamente las inyecciones de defectos sobre aserciones críticas de negocio, confirmando que es viable incorporar el razonamiento de la IA sin erosionar la confiabilidad de la automatización.

Viabilidad de modelos locales y soberanía de datos: Se comprobó la viabilidad operativa de emplear Modelos de Lenguaje Pequeños (SLM) ejecutados de forma 100 % local. Este hallazgo constituye un pilar esencial para la adopción empresarial corporativa, al ofrecer una arquitectura capaz de blindar la privacidad de los datos de prueba y evadir la exposición de información transaccional hacia APIs externas en la nube.

Eficiencia computacional mediante Caché Semántico: La implementación de una memoria de resoluciones históricas demostró ser fundamental para la viabilidad a largo plazo del ecosistema. Al reducir el consumo de tokens de inferencia en casi un 50 % durante escenarios estables, se mitigó el principal cuello de botella de la automatización basada en IA, demostrando que es posible equilibrar las ventajas del auto-mantenimiento con costos operativos sostenibles.

7.3. Conclusiones sobre la Validación Experimental frente a POM (Objetivo 3)

Mitigación empírica de la carga de mantenimiento: En el contexto de la validación experimental fundamentada en la norma ISO/IEC 25010, los resultados comprobaron una superioridad notable en mantenibilidad al contrastar la solución MDT frente a la línea base tradicional (POM). Ante mutaciones controladas en la interfaz de usuario, los esquemas POM demandaron refactorización manual obligatoria del código fuente de sus objetos, incrementando la deuda técnica. En marcado contraste, la orquestación semántica del MDT logró sobreponerse a la caída de selectores tradicionales mediante capacidades de *self-healing*, asumiendo autónomamente la localización de los elementos sin intervención humana. Esta evidencia empírica posiciona al enfoque propuesto como un mecanismo sustancialmente más efectivo para contrarrestar la alta carga de mantenimiento en los ciclos modernos de QA.

Validación de la Flexibilidad y Adaptabilidad Tecnológica: El marco de evaluación ISO/IEC 25010 permitió corroborar que la arquitectura MDT exhibe un grado de flexibilidad significativamente mayor que el patrón POM. Mientras que un cambio de motor de ejecución en POM exige reescribir o adaptar extensamente la capa de integración (*Glue Code*) y la gestión de *drivers*, el ecosistema MDT demostró la capacidad de conmutar la ejecución entre servidores de *Selenium* y *Playwright* alterando únicamente el enrutamiento del protocolo MCP. Esta adaptabilidad intrínseca certifica que los activos funcionales (escenarios BDD y metadatos) permanecen agnósticos a las tecnologías subyacentes, otorgando a la organización la flexibilidad estratégica de evolucionar su infraestructura de pruebas sin incurrir en reingeniería masiva.

7.4. Trabajos Futuros

A partir de los hallazgos y limitaciones documentadas en esta investigación, se identifican diversas líneas de trabajo que podrían extender el alcance y la madurez de la arquitectura propuesta:

- **Evaluación sobre aplicaciones dinámicas:** Extender el protocolo experimental hacia *Single Page Applications* (SPA) con alta carga de renderizado asíncrono y *Shadow DOM*, para validar la resiliencia semántica del agente frente a estructuras complejas de carga tardía.
- **Exploración de Modelos Multimodales:** Incorporar capacidades de visión artificial (*Vision-Language Models*) en la capa de razonamiento, posibilitando que el sistema analice la interfaz gráficamente y reduzca la dependencia exclusiva del código fuente HTML.
- **Integración con Gestores de Ciclo de Vida (ALM):** Extender la abstracción del protocolo MCP para conectar la recolección de métricas y la generación de auditorías de forma bidireccional con herramientas corporativas de gestión ágil.
- **Generación autónoma de metadatos:** Investigar algoritmos de exploración heurística que permitan a un agente cognitivo rastrear la aplicación y proponer automáticamente los diccionarios iniciales de metadatos y escenarios base.
- **Evaluación formal de la Eficiencia Temporal:** Realizar un estudio comparativo del *Comportamiento Temporal* (ISO/IEC 25010) bajo distintas configuraciones de hardware y modelos de lenguaje, cuantificando el impacto de la latencia de inferencia en escenarios de integración continua.
- **Estudio de viabilidad económica multimodelo:** Evaluar la arquitectura integrando diversos Modelos de Lenguaje (LLMs comerciales vs. SLMs de código abierto) para analizar métricas de costo-beneficio, variaciones en el consumo de tokens y determinar la factibilidad financiera del enfoque en despliegues a gran escala.

7.5. Lecciones aprendidas

El desarrollo de la arquitectura híbrida MDT y la validación empírica con Modelos de Lenguaje de Gran Escala dejaron varias lecciones clave:

- **Agentes Especializados vs. Agentes Genéricos:** Distribuir responsabilidades en roles atomizados (*Symphony*, *Argos* y *Nexus*) resulta más eficiente que centralizar la orquestación en un único LLM genérico. Acotar el contexto por agente mitiga las alucinaciones y facilita el uso de modelos locales más pequeños (SLM).
- **Estructuración estricta del prompt:** La IA aplicada a la automatización de pruebas requiere restricciones operativas severas. Ajustar la temperatura a cero y obligar al modelo a emitir respuestas en formatos rígidamente estructurados (como JSON) resultó ser fundamental para garantizar el determinismo necesario en pruebas funcionales.

- **La memoria caché es fundamental para la viabilidad económica:** La implementación empírica evidenció que sin un mecanismo de almacenamiento de caché, los costos de tokens y la latencia de inferencia son prohibitivos para integraciones continuas. La abstracción de metadatos facilitó esta estrategia de almacenamiento.
- **Adopción de Frameworks Especializados:** La utilización de herramientas diseñadas específicamente para el ecosistema LLM resultó vital. Estas librerías proporcionan patrones de diseño probados que abstraen la complejidad del manejo de estado y el enrutamiento condicional, permitiendo estructurar el proyecto agéntico de manera coherente, modular y organizada frente a integraciones construidas desde cero.
- **Validación semántica de aserciones:** La delegación de las validaciones (*assertions*) a agentes cognitivos es una estrategia clave para mitigar la fragilidad. A diferencia de las verificaciones rígidas de código (que buscan textos o atributos exactos y suelen generar "falsos positivos"de fallo ante cambios cosméticos), el análisis semántico permite que el LLM comprenda el contexto y determine el éxito real de la prueba con mayor resiliencia.

Bibliografía

(2024). *Docker Desktop: Resource management and GPU acceleration guides*. Docker Inc.

(2024). *LM Studio Docs*. LM Studio Team.

(2024). *Ollama API and Architecture Documentation*. Ollama Open Source Project.

Abughazala, M. and Muccini, H. (2025). Dqgen: Scalable metadata-driven automation for data quality validation in data-intensive applications. In *Software Architecture. ECSA 2025 Tracks and Workshops: Limassol, Cyprus, September 15–19, 2025, Proceedings*, page 363–380, Berlin, Heidelberg. Springer-Verlag.

Anthropic PB (2024). Model context protocol (mcp): An open standard for connecting ai models to data and tools. Technical report. Accessed: 2026-01-27.

Autosana AI (2025). Autosana: Ai-powered test automation and self-healing. Accessed: 2026-01-29.

Bäuerle, S., Schmidt, T., and Jurack, S. (2024). Model-driven testing in agile environments: Bridging the gap between declarative models and imperative execution. *Journal of Software: Evolution and Process*, 36(2):e2541.

Binamungu, L., Konstantinou, N., and Embury, S. M. (2018). An empirical study of the challenges of specifying and maintaining Behavior-Driven Development (BDD) scenarios. In *Proceedings of the 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 352–363, Campobasso, Italy. IEEE.

Brambilla, M., Cabot, J., and Wimmer, M. (2017). *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2nd edition.

Browserbase (2025). Stagehand: An ai web browsing framework focused on simplicity and extensibility. Accessed: 2026-06-18.

Capgemini, Sogeti, and OpenText (2024). World quality report 2024-25: New futures in focus. Technical report, Capgemini.

Capgemini, Sogeti, and OpenText (2025). World quality report 2025-26: Adapting to emerging worlds. Technical report, Capgemini.

Chase, H. (2022). Langchain: Building applications with llms through composability. https://github.com/langchain-ai/langchain.

García, B., del Alamo, J. M., Leotta, M., and Ricca, F. (2024). Exploring browser automation: A comparative study of selenium, cypress, puppeteer, and playwright. *Communications in Computer and Information Science*, 2178 CCIS:142–149.

Gerganov, G. (2024). llama.cpp: Inference of llms in pure c/c++.

Graham, D. and Fewster, M. (2012). *Experiences of Test Automation: Case Studies of Software Test Automation*. Addison-Wesley Professional.

Humble, J. and Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional.

Irshad, M., Britto, R., and Petersen, K. (2021). Adapting behavior driven development (bdd) for large-scale software systems. *Journal of Systems and Software*, 177:110944.

ISO (2023). Iso/iec 25010:2023 systems and software engineering — systems and software quality requirements and evaluation (square) — product quality model.

ISTQB (2024). Certified tester advanced level: Test automation engineer syllabus. Technical report, International Software Testing Qualifications Board.

Jones, A. (2026). mcp-selenium: An MCP implementation for Selenium WebDriver.

Juhnke, K., Törngren, M., and Chen, D. (2022). Quality models for test case specifications: An iso 25010 perspective. *IEEE Access*, 10:55689–55704.

Khankhoje, R. (2022). Beyond coding: A comprehensive study of low-code, no-code and traditional automation. *Journal of Artificial Intelligence & Cloud Computing*, pages 1–5.

Kwon, W., Li, Z., Zhang, S., Zhang, L., Yu, Y., Yu, C., Constantine, G., Moon, K., Suarez, J., Song, D., Stoica, I., and Joseph, A. D. (2023). Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*, pages 611–626. ACM.

LangChain AI (2024). Langgraph. https://github.com/langchain-ai/langgraph.

Lehtonen, R. (2023). Test automation in agile software development. Master's thesis, Turku University of Applied Sciences.

Leotta, M., Clerissi, D., Ricca, F., and Spadaro, C. (2013). Improving test suites maintainability with the page object pattern: An industrial case study. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 108–113.

Leotta, M., Stocco, A., Ricca, F., and Tonella, P. (2016). Robula+: An algorithm for generating robust xpath locators for web testing. *Journal of Software: Evolution and Process*, 28(3):177–204.

Meszaros, G. (2007). xunit test patterns: Refactoring test code. *Chemistry & ...*, page 833.

Microsoft (2026). Playwright MCP.

Molison, A. S., Moraes, M., Melo, G., Santos, F., and Assuncao, F. (2025). Is LLM-Generated Code More Maintainable & Reliable Than Human-Written Code? . In *2025 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 151–162, Los Alamitos, CA, USA. IEEE Computer Society.

Naveen Automation Labs (2026). selenium-mcp.

Smart, J. F. and Molak, J. (2023). *BDD in Action, Second Edition: Behavior-Driven Development for the Whole Software Lifecycle*. Manning Publications, 2nd edition.

Smith, J. and Johnson, R. (2024). Analyzing resource consumption and performance overhead in electron-based desktop applications. *Journal of Systems and Software Architecture*, 142:102–115.

Spur Test (2025). Spur: Autonomous agent for end-to-end testing. Accessed: 2026-01-29.

TheMindMod (2026). selenium-mcp-server.

Tymoshchuk, A. (2025). The evolution of test automation: From selenium to playwright. a comparison of automation tools. *International Journal of Computer (IJC)*, 54(1):37–45.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Le Scao, T., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. (2020). Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45.

Wynne, M., Hellesøy, A., and Tooke, S. (2017). *The cucumber book : behaviour-driven development for testers and developers*. Pragmatic Bookshelf.

ZeroStep (2025). Zerostep: Ai-powered playwright testing. Accessed: 2026-06-18.