UNIVERSIDAD INTERNACIONAL SEK

UISEK BUSINESS Y DIGITAL SCHOOL

Trabajo de Fin de Carrera Titulado:

“Diseño e Implementación de un Agente Inteligente de Pruebas Funcionales en APIs REST para Agilizar la Automatización en Calidad de Software”

Realizado por:

BRYAN ENRIQUE GARAY BENAVIDEZ

Director del Trabajo de Titulación:

ING. PATRICIO FERNANDO ALARCON LARREA MSC.

Como requisito para la obtención del título de:

INGENIERO DE SOFTWARE

Quito, Febrero de 2026

DECLARACIÓN JURAMENTADA

Yo, BRYAN ENRIQUE GARAY BENAVIDEZ, con cédula de identidad No. 0923790562, declaro bajo juramento que el trabajo aquí desarrollado es de mi autoría que no ha sido previamente presentado por ningún grado a calificación profesional y, que se ha consultado las referencias bibliográficas que se incluyen en este documento.

A través de la presente declaración, cedo mis derechos de propiedad intelectual correspondiente a este trabajo, a la UNIVERSIDAD INTERNACIONAL SEK, según lo establecido por la Ley de Propiedad Intelectual, por su reglamento y por la normativa institucional vigente.

Bryan Enrique Garay Benavidez

C.C.: 0923790562

DECLARATORIA

El Presente Trabajo de Investigación Titulado:

“Diseño e Implementación de un Agente Inteligente de Pruebas Funcionales en APIs REST para Agilizar la Automatización en Calidad de Software”

Realizado por:

BRYAN ENRIQUE GARAY BENAVIDEZ

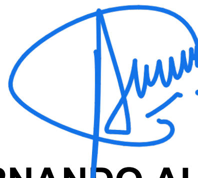
Como Requisito para la Obtención del Título de:

INGENIERO DE SOFTWARE

Ha Sido Dirigido por el Profesor:

ING. PATRICIO FERNANDO ALARCON LARREA MSC.

Quien Considera que Constituye un Trabajo Original de su Autor



ING. PATRICIO FERNANDO ALARCON LARREA MSC.

DIRECTOR

DEDICATORIA

Este trabajo va dedicado a mis padres, que me brindaron un hogar con sólidos valores como la honestidad y el amor por el trabajo duro, pilares para la construcción de esta Tesis.

También lo dedico a mi esposa Jannina que ha sido siempre el soporte emocional para afrontar todos los retos de la vida.

AGRADECIMIENTO

Al tutor de este trabajo de titulación **Ing. Patricio Fernando Alarcon Larrea Msc.** Su excelente gestión y compromiso con ayudarnos a realizar trabajos con altos estándares de calidad.

También a la coordinadora **Ing. Viviana Elizabeth Cajas Cajas, PhD.** Y a los docentes Ing. Pablo Perez Mgs. e Ing. Victo Morejon Mgs. cuyo aporte cualitativo y técnico fue fundamental para encaminar a buen rumbo este trabajo.

A la Universidad Internacional SEK, por ser el crisol para la formación de profesionales de excelencia.

RESUMEN

Este trabajo de fin de carrera propone una solución tecnológica ante el desequilibrio entre la velocidad del desarrollo ágil y la capacidad de ejecución de pruebas exhaustivas. La problemática central reside en la alta carga de tareas manuales repetitivas y el elevado costo de mantenimiento de los scripts de automatización convencionales, factores que limitan la eficiencia de los equipos de aseguramiento de la calidad de software (SQA).

El objetivo fundamental de la investigación es diseñar y construir un **agente inteligente** capaz de asistir en la planificación, diseño, ejecución y reporte de pruebas funcionales sobre APIs REST. Para ello, se implementó el sistema denominado **Omega Testing**, empleando una arquitectura modular compuesta por un frontend en React y un backend en NestJS con persistencia de datos en SQLite. El núcleo innovador del sistema es la integración de un servicio de **Inteligencia Artificial (IA)** basado en modelos de OpenAI, el cual permite transformar requerimientos en lenguaje natural en artefactos de prueba automatizados bajo el enfoque BDD (*Behavior-Driven Development*) y lenguaje Gherkin.

La implementación se realizó mediante una metodología de investigación aplicada y un marco de trabajo ágil estructurado en 12 sprints. El sistema centraliza módulos para la gestión de proyectos, inventario de endpoints, orquestación de suites de prueba y un panel de monitoreo con eventos en tiempo real mediante *Server-Sent Events* (SSE).

Como resultado, la evaluación con expertos de la industria confirmó que el **70% de los participantes** percibió una reducción de tiempo de entre el 31% y el 70% en la generación y ejecución de pruebas. En conclusión, el sistema demuestra que el uso de agentes inteligentes optimiza los tiempos de trabajo, reduce el esfuerzo manual y mejora la cobertura y precisión de las validaciones en entornos de entrega continua.

Claves: Agente Inteligente, APIs REST, Automatización de Pruebas, Calidad de Software, Inteligencia Artificial, Sprints.

ABSTRACT

This undergraduate thesis proposes a technological solution to the imbalance between the speed of agile development and the capacity to execute exhaustive testing. The central problem lies in the high burden of **repetitive manual tasks** and the high maintenance cost of conventional automation scripts, factors that limit the efficiency of **Software Quality Assurance (SQA)** teams.

The fundamental objective of the research is to design and build an **intelligent agent** capable of assisting in the planning, design, execution, and reporting of functional tests for REST APIs. To achieve this, the system named **Omega Testing** was implemented, using a modular architecture composed of a **React** frontend and a **NestJS** backend with **SQLite** data persistence. The innovative core of the system is the integration of an **Artificial Intelligence (AI)** service based on **OpenAI** models, which allows natural language requirements to be transformed into automated test artifacts under the **BDD** (Behavior-Driven Development) approach and **Gherkin** language.

The implementation was carried out using an applied research methodology and an agile framework structured into **12 sprints**. The system centralizes modules for project management, endpoint inventory, test suite orchestration, and a monitoring dashboard with real-time events via **Server-Sent Events (SSE)**.

As a result, the evaluation with industry experts confirmed that **70% of the participants** perceived a time reduction of between 31% and 70% in the generation and execution of tests. In conclusion, the sources demonstrate that the use of intelligent agents optimizes work times, reduces manual effort, and improves the coverage and precision of validations in continuous delivery environments.

Keywords: Artificial Intelligence, Intelligent Agent, REST APIs, Software Quality, Sprints, Test Automation.

ÍNDICE DE CONTENIDO

DECLARACIÓN JURAMENTADA ii
DECLARATORIA iii
DEDICATORIA iv
AGRADECIMIENTO v
RESUMEN vi
ABSTRACT vii
ÍNDICE DE TABLAS xiii
ÍNDICE DE FIGURAS xv
INTRODUCCIÓN xviii
CAPÍTULO I 1
1. ANTECEDENTES 1
1.1. EL PROBLEMA DE INVESTIGACIÓN 1
1.1.1. Planteamiento del problema 1
1.1.1.1. Diagnóstico 1
1.1.1.2. Pronóstico 2
1.1.1.3. Control de pronóstico 3
1.1.2. Justificación 4
1.1.3. Formulación del problema 5
1.1.4. Variables del problema 5
1.1.5. Sistematización del problema 5
1.2. OBJETIVOS 6
1.2.1. Objetivo General 6
1.2.2. Objetivos específicos 6
CAPÍTULO II 8
2. MARCO TEÓRICO 8

2.1. Fundamentos de la Calidad de Software ... 8

2.1.1. Concepto y evolución del Aseguramiento de la Calidad del Software ... 8

2.1.2. Importancia del SQA en entornos ágiles ... 10

2.1.3. Artefactos típicos del aseguramiento de la calidad de software ... 10

2.2. Metodologías Ágiles y el Impacto de las Pruebas ... 12

2.2.1. Principios de Scrum y Kanban aplicados al ciclo de pruebas ... 12

2.2.2. Entregas incrementales y necesidad de regresión continua ... 14

2.3. Pruebas Funcionales en APIs REST ... 15

2.3.1. Definición y características de las APIs REST. ... 15

2.3.2. Importancia de las pruebas funcionales de APIs. ... 18

2.3.3. Herramientas y enfoques tradicionales. ... 19

2.3.4. Limitaciones de los enfoques manuales y automatizados convencionales. ... 20

2.4. Automatización de Pruebas de Software ... 21

2.4.1. Concepto, beneficios y niveles de madurez en test automation. ... 21

2.4.2. Frameworks de Testing y herramientas más usadas. ... 22

2.4.3. Retos de la Automatización de Pruebas ... 25

2.5. Inteligencia Artificial en el Contexto del Testing ... 27

2.5.1. Concepto de agente inteligente y su aplicación en ingeniería de software. ... 27

2.5.2. Uso de modelos de Inteligencia Artificial en la actualidad ... 28

2.5.3. Estado del arte: propuestas recientes en AI Testing. ... 31

CAPITULO III ... 35

3. METODOLOGÍA ... 35

3.1. Tipo de Investigación ... 35

3.2. Diseño Metodológico ... 36

3.3. Tecnologías y Herramientas a Utilizar ... 37

3.4. Criterios de Validación ... 37

CAPITULO IV ... 39

4. DISEÑO ... 39

4.1. Análisis Preliminar ... 39

4.1.1. Panorama Actual de la Automatización de Pruebas ... 39

4.1.2. Necesidades Identificadas ... 39

4.1.3. Restricciones Detectadas ... 41

4.1.4. Factibilidad ... 43

4.2. Descripción del Sistema Propuesto ... 45

4.2.1. Generalidades ... 45

4.2.2. Requerimientos Funcionales ... 50

4.2.3. Requerimientos No Funcionales ... 51

4.2.4. Casos de Uso ... 52

4.3. Diseño ... 53

4.3.1. Arquitectura ... 53

4.3.2. Diagramas de Clases Simplificado ... 54

4.3.3. Diagramas de Secuencia ... 55

4.3.4. Diagrama Entidad Relación de la Base de Datos ... 58

4.3.5. Diseño de Pantallas en Figma ... 59

4.4. Planificación de Releases por Sprints ... 60

4.4.1. Estrategia de Releases ... 60

4.4.2. Historias de Usuario ... 61

CAPITULO V ... 69

5. IMPLEMENTACIÓN ... 69

5.1. Descripción del Sistema Desarrollado ... 69

5.1.1. Arquitectura general del sistema (frontend, backend y base de datos) ... 69

5.1.2. Tecnologías empleadas y estructura del monorepo ... 71

5.2. Despliegue del Software ... 79

5.2.1. Entorno de ejecución e instalación local ... 79

5.2.2. Construcción, empaquetado y CLI de ejecución 79

5.2.3. Publicación y configuración del entorno 81

5.3. Implementación del Software 82

5.3.1. Descripción general del cliente web 82

5.3.2. Módulos funcionales del sistema 83

5.3.2.1. Dashboard 83

5.3.2.2. Projects 84

5.3.2.3. Endpoints 88

5.3.2.4. Test Cases 92

5.3.2.5. Test Suites 97

5.3.2.6. Bugs 101

5.3.2.7. Test Executions 105

5.3.2.8. Settings — OpenAI Configuration 109

5.3.3. Accesibilidad y experiencia de usuario (UX/UI) 111

5.3.4. Cumplimiento de Requisitos no funcionales 115

5.4. Mantenimiento del Sistema 119

CAPITULO VI 123

6. EVALUACIÓN Y RESULTADOS 123

6.1. Pruebas Funcionales del Sistema 123

6.2. Resultados de Encuesta de Usabilidad y Desempeño 125

6.3. Beneficios del Sistema y Oportunidades de Mejora 130

CAPITULO VII 132

7. DISCUSIÓN 132

7.1. Conclusiones 132

7.2. Recomendaciones 132

BIBLIOGRAFÍA 134

ANEXOS 141

ANEXO A - Casos de Uso para el Diseño del Sistema Propuesto 141

ANEXO B – Diagramas de Secuencia para las Principales Actividades del Sistema........ 147

ANEXO C – Diseño de Pantallas de la Interfaz en Figma 153

ANEXO D – Preguntas de la Encuesta realizada 157

ANEXO E – Instrucciones Dadas Para las Pruebas de Expertos 162

ÍNDICE DE TABLAS

Tabla 1. Descripción de los métodos HTTP más comunes 16
Tabla 2. Comparativa de las características de Frameworks para API testing 24
Tabla 3. Precios referenciales de diferentes modelos de IA de OpenAI 31
Tabla 4. Recursos de hardware y software para el proyecto 44
Tabla 5. Requerimientos funcionales del sistema propuesto 50
Tabla 6. Requerimientos no funcionales del sistema propuesto 51
Tabla 7. Resumen de los componentes principales de los diagramas de secuencia creados 56
Tabla 8. Resumen de pantallas diseñadas en Figma 59
Tabla 9. Estrategia de Releases y su información clave 60
Tabla 10. Planificación del Sprint 1 — Base técnica y estructura 61
Tabla 11. Planificación del Sprint 2 — Dominios Projects y Endpoints 62
Tabla 12. Planificación del Sprint 3 — Test Cases y Test Suites 62
Tabla 13. Planificación del Sprint 4 — Ejecuciones y Bugs 63
Tabla 14. Planificación del Sprint 5 — Base de la SPA y Dashboard simulado 63
Tabla 15. Planificación del Sprint 6 — Páginas de Projects y Endpoints 64
Tabla 16. Planificación del Sprint 7 — Páginas de Test Cases y Test Suites 65
Tabla 17. Planificación del Sprint 8 — Vistas Executions y Bugs (mock) 65
Tabla 18. Planificación del Sprint 9 — Conexiones reales y sincronización 66
Tabla 19. Planificación del Sprint 10 — Ejecución E2E, SSE e IA 66
Tabla 20. Planificación del Sprint 11 — Packaging NPM y documentación técnica 67
Tabla 21. Planificación del Sprint 12 — Publicación y verificación final 67
Tabla 22. Principales tecnologías utilizadas en el código del proyecto 71
Tabla 23. Resumen general de pruebas funcionales por módulo 123
Tabla 24. CU-01 — Dashboard de entrada y navegación 141
Tabla 25. CU-02 — Crear y mantener proyectos 141
Tabla 26. CU-03 — Registrar y administrar endpoints 142
Tabla 27. CU-04 — Diseñar casos de prueba manuales (BDD) 142
Tabla 28. CU-05 — Generar casos de prueba con IA 143
Tabla 29. CU-06 — Obtener sugerencias de casos con IA 143
Tabla 30. CU-07 — Componer y mantener suites de prueba 144
Tabla 31. CU-08 — Ejecutar pruebas y monitorear en tiempo real 144
Tabla 32. CU-09 — Consultar detalle y resumen de ejecuciones 145

Tabla 33. CU-10 — Registrar y gestionar bugs 145
Tabla 34. CU-11 — Sincronizar artefactos del proyecto 146
Tabla 35. CU-12 — Configurar proveedor de IA 146

ÍNDICE DE FIGURAS

Figura 1. Gráfico ilustrativo de las ceremonias de un Sprint 12
Figura 2. Visualización de un ejemplo de tareas en un tablero Kanban 14
Figura 3. Ejemplo de una petición HTTP en Postman 19
Figura 4. Diagrama de beneficios de la automatización de pruebas de software 21
Figura 5. Gráfico de bucle percepción, decisión, acción de la IA 27
Figura 6. Diagrama de Casos de Uso del sistema propuesto 52
Figura 7. Diagrama C4 - Nivel Contenedor (Arquitectura general del MVP) 53
Figura 8. Diagrama de clases simplificado del sistema propuesto 55
Figura 9. Diagrama de entidad relación de la Base de Datos 58
Figura 10. Diagrama general de arquitectura del sistema Omega Testing MVP 70
Figura 11. Estructura general del monorepo (carpetas raíz del proyecto) 72
Figura 12. Carpetas con estructura del frontend (client/) 73
Figura 13. Estructura de carpetas del backend (src/modules) 74
Figura 14. Estructura usada en los módulos del backend (ejemplo: módulo Endpoints) 75
Figura 15. Ejemplo de código de conexión a OpenAI 77
Figura 16. Ejemplo de creación de un Assistant y Threads de Open AI en Typescript 77
Figura 17. Ejemplo de Prompt para la generación de Casos de Prueba en Typescript 77
Figura 18. Ejemplo de creación de la carpeta de proyectos y base de datos 79
Figura 19. Estructura de archivos generados para la compilación en producción 80
Figura 20. Interfaz principal del Dashboard de Omega Testing 83
Figura 21. Vista general del módulo Projects de Omega Testing 85
Figura 22. Estructura del workspace generado para un proyecto en entorno local 86
Figura 23. Vista general del módulo Endpoints de Omega Testing 88
Figura 24. Ejemplo de proyecto de testing generado con artefactos 89
Figura 25. Vista general del módulo Test Cases de Omega Testing 92
Figura 26. Vista del modo AI para casos de prueba de Omega Testing 94
Figura 27. Ejemplo de escenarios BDD en el archivo feature generado 95
Figura 28. Base de datos SQLite con los casos de prueba 96
Figura 29. Vista general del módulo Test Suites de Omega Testing 97
Figura 30. Interfaz del diálogo Create Test Suite de Omega Testing 99
Figura 31. Vista general del módulo Bugs de Omega Testing 101
Figura 32. Detalle contextual de un bug de Omega Testing 103

Figura 33. Vista general del módulo Test Executions de Omega Testing 105
Figura 34. Vista detallada de una ejecución de pruebas de Omega Testing........................... 107
Figura 35. Vista del panel de configuración de OpenAI de Omega Testing 109
Figura 36. Vista del módulo Projects en modo oscuro de de Omega Testing 111
Figura 37. Pantalla de carga inicial de Omega Testing 112
Figura 38. Diseño responsivo de Omega Testing en otro ancho de pantalla.......................... 113
Figura 39. Vista general del repositorio de Omega Testing en GitHub................................. 119
Figura 40. Página del paquete Omega Testing MVP en NPM 121
Figura 41. Registros de logs en consola de Omega Testing 121
Figura 42. Panel de OpenAI con métricas de consumo de tokens 122
Figura 43. Promedio de calificación de las heurísticas de usabilidad según Nielsen. 126
Figura 44. Percepción de reducción del tiempo de generación y ejecución de pruebas 127
Figura 45. Percepción de calidad y cobertura de Omega Testing. 128
Figura 46. Diagrama de secuencia: RF-01 — Dashboard (KPIs) 147
Figura 47. Diagrama de secuencia: RF-02 — Proyectos (CRUD) 148
Figura 48. Diagrama de secuencia: RF-03 — Endpoints por proyecto (registro, análisis y artefactos) 149
Figura 49. Diagrama de secuencia: RF-04 — Casos de prueba (manual BDD y ejecución desde la vista) 149
Figura 50. Diagrama de secuencia: RF-05 — Suites (crear, editar, listar y ejecutar) 150
Figura 51. Diagrama de secuencia: RF-06 — Ejecuciones con SSE (progreso y resultados) 150
Figura 52. Diagrama de secuencia: RF-07 — Bugs/Defectos (crear, filtrar, estadísticas)..... 151
Figura 53. Diagrama de secuencia: RF-08 — Asistente de IA (sugerir y generar BDD) 151
Figura 54. Diagrama de secuencia: RF-09 — Sincronización con workspace 152
Figura 55. Diagrama de secuencia: RF-10 — Configuración y prueba de OpenAI API Key 152
Figura 56. Diseño del Dashboard central de KPI’s en Figma 153
Figura 57. Diseño de Página de Projects.. 153
Figura 58. Diseño de Página de Endpoints 154
Figura 59. Diseño de Página de Test Cases.. 154
Figura 60. Diseño de Página de Test Suites 155
Figura 61. Diseño de Página de Bugs 155
Figura 62. Diseño de Página de Test Executions 156
Figura 63. Diseño de Página de Settings.. 156

Figura 64. Capturas de Pantalla de ejemplo como evidencia de encuesta realizada en Google Forms 158
Figura 65. Resultado de encuestas – Pregunta 1 158
Figura 66. Resultado de encuestas – Pregunta 2 159
Figura 67. Resultado de encuestas – Pregunta 3 159
Figura 68. Resultado de encuestas – Pregunta 4 159
Figura 69. Resultado de encuestas – Pregunta 5 160
Figura 70. Resultado de encuestas – Pregunta 6 160
Figura 71. Resultado de encuestas – Pregunta 7 160
Figura 72. Resultado de encuestas – Pregunta 8 161
Figura 73. Resultado de encuestas – Pregunta 9 161
Figura 74. Resultado de encuestas – Pregunta 10 161

INTRODUCCIÓN

La Ingeniería de Software moderna ha evolucionado hacia entornos altamente dinámicos donde la calidad del producto es tan prioritaria como la velocidad de entrega. De acuerdo con (Papakitsos, 2022), el Aseguramiento de la Calidad de Software (SQA) se ha consolidado como una disciplina esencial para garantizar que las soluciones digitales cumplan con los requerimientos funcionales y las expectativas del usuario final. En este contexto, las metodologías ágiles como Scrum y Kanban (Patilla et al., 2021) han transformado la manera en que los equipos desarrollan y validan software, promoviendo ciclos de entrega cortos, retroalimentación continua y adaptabilidad frente al cambio.

Sin embargo, esta agilidad ha traído consigo nuevos desafíos para los equipos de control de calidad. Las pruebas manuales, que antes bastaban para validar versiones estáticas, resultan insuficientes frente a la velocidad y complejidad de los despliegues actuales. Según (Rohit Khankhoje, 2023), la automatización de pruebas se ha convertido en un requisito indispensable para mantener la eficiencia, especialmente en las pruebas de regresión que deben ejecutarse con cada iteración. A su vez, (Wang, Mäntylä, Liu, & Markkula, 2022) destaca que los beneficios de esta automatización incluyen la reducción de tiempo y esfuerzo operativo, aunque también introducen nuevas demandas técnicas para los testers, quienes deben crear, mantener y adaptar scripts a un ritmo acelerado.

Entre los ámbitos más críticos del desarrollo contemporáneo se encuentran las interfaces de programación de aplicaciones o APIs. Estas se han convertido en el núcleo de la comunicación entre sistemas, al permitir que los servicios interactúen de forma modular y escalable (Thayer et al., 2021). Actualmente, cerca del 75% de los proyectos en la industria emplean APIs REST (Mudassir & Mushtaq, 2024), lo que evidencia su papel protagónico en la arquitectura moderna del software. Sin embargo, esta dependencia implica que cualquier falla en las APIs puede traducirse en interrupciones de servicio y pérdidas económicas.

Ante este panorama, las prácticas de API Testing se han vuelto un componente esencial dentro del aseguramiento de la calidad, aunque también representan una de las áreas con mayor carga de trabajo. Los equipos de QA enfrentan la necesidad de ejecutar múltiples validaciones en ciclos ágiles, manteniendo la coherencia de los resultados y la trazabilidad de los defectos. Según (Bhanushali, 2023), esta presión constante conduce a la saturación de los testers y a una menor capacidad para enfocarse en tareas de análisis profundo. Frente a ello, la integración de

herramientas basadas en Inteligencia Artificial emerge como una alternativa viable para aliviar esta carga y potenciar la productividad.

En este marco se inscribe la presente investigación, cuyo propósito es diseñar e implementar un agente inteligente que asista a los testers en la automatización de pruebas funcionales sobre APIs REST. Esta solución busca combinar técnicas de IA con prácticas ágiles, permitiendo una ejecución más eficiente, accesible y sostenible de los procesos de testing. De esta forma, se contribuye a cerrar la brecha entre la velocidad del desarrollo moderno y la necesidad de garantizar productos de software confiables y de alta calidad.

CAPÍTULO I

1. ANTECEDENTES

1.1. EL PROBLEMA DE INVESTIGACIÓN

1.1.1. Planteamiento del problema

1.1.1.1. Diagnóstico

En los equipos de control de calidad de software (QA), especialmente en entornos ágiles, persiste un desequilibrio entre la velocidad del desarrollo y la capacidad de ejecutar pruebas exhaustivas. A pesar de la adopción de frameworks de automatización y herramientas de integración continua (Bertolino et al., 2023), gran parte del trabajo de validación sigue dependiendo del esfuerzo manual y del mantenimiento constante de los scripts.

Esta situación genera sobrecarga operativa y limita la capacidad de los testers para concentrarse en la calidad del producto y la detección temprana de defectos. Diversos autores coinciden en que esta problemática es consecuencia directa de los ciclos de entrega acelerados y los equipos reducidos en QA frente al número de desarrolladores (Atoum et al., 2021). Los testers suelen invertir gran parte de su tiempo en tareas repetitivas, como planificar, ejecutar y reportar pruebas, mientras intentan mantener la coherencia de los escenarios de regresión ante cambios continuos en las APIs.

Esta dinámica afecta no solo la eficiencia, sino también la motivación y la capacidad de innovación dentro de los equipos. En el ámbito específico de las APIs REST, la situación es aún más crítica. A medida que las organizaciones amplían su infraestructura digital, la cantidad y complejidad de las APIs crece de manera exponencial (Pargaonkar, 2023). Cada modificación o nueva integración requiere pruebas de funcionalidad, compatibilidad y seguridad, las cuales deben ejecutarse repetidamente en cada sprint.

La carga de trabajo resultante no escala de forma sostenible con los recursos humanos disponibles (Duraisamy et al., 2021), lo que provoca retrasos, errores y menor cobertura de pruebas. Aunque las pruebas automatizadas han mitigado parte del problema, también introducen nuevos retos, como el mantenimiento continuo

del código de prueba, la integración con pipelines de CI/CD y la necesidad de conocimientos avanzados de programación.

En consecuencia, muchos equipos siguen atrapados en un ciclo de automatización parcial que no resuelve de fondo la falta de agilidad en la validación de software. Este escenario justifica la necesidad de un enfoque más inteligente y colaborativo.

La incorporación de un agente inteligente basado en IA (Nembhard et al., 2023), tiene el potencial de transformar la automatización de pruebas en un proceso más accesible, dinámico y orientado al aprendizaje continuo. Este tipo de asistente no reemplaza el criterio humano, sino que potencia la capacidad de los testers para diseñar, ejecutar y analizar pruebas de manera más eficiente, reduciendo la carga repetitiva y mejorando la calidad del software entregado.

1.1.1.2. Pronóstico

Tomando en consideración el diagnóstico anterior (Wang, Mäntylä, Liu, & Markkula, 2022), en el cual se describe una problemática frecuente en los los equipos de QA y desarrollo, si estos continúan dependiendo de procesos manuales o poco automatizados para las pruebas de APIs REST, se mantendrán expuestos a errores frecuentes, retrasos en las entregas y una cobertura de pruebas insuficiente. En (Hossain, 2018) se detalla que esta situación limita la capacidad de respuesta ante cambios, incrementa la posibilidad de que defectos lleguen a producción y obliga a los testers a invertir la mayor parte de su tiempo en tareas repetitivas, relegando actividades estratégicas como el diseño de pruebas innovadoras o el análisis de riesgos.

Muchos equipos en la actualidad utilizan frameworks para pruebas automatizadas, (Bertolino et al., 2023), las cuáles evitan la laboriosa y repetitiva tarea de realizar una prueba manual, cada vez que un nuevo requerimiento lo exige, en cambio se escriben los paso mediante código en algún lenguaje de programación, permitiendo una mayor rapidez en la ejecución de pruebas de regresión ante cambios. No obstante, (Nass et al., 2021) a pesar de que las pruebas automatizadas alivian la carga manual, introducen otra tarea compleja al flujo de procesos de un equipo de QA, que consiste en escribir, mantener y actualizar los casos de prueba automatizados cada vez que algo cambia en el software. (Duraisamy et al., 2021), también existen tareas derivadas como integración con pipelines de Integración

Continua y Despliegue Continuo (CI/CD), por lo que el problema de falta de tiempo y recursos para enfocarse en la calidad persiste.

Haciendo un enfoque en API Testing, (Pargaonkar, 2023), a medida que la complejidad y el número de APIs crecen, el esfuerzo requerido escala de forma poco sostenible, afectando tanto la eficiencia como la calidad final del software. Como resultado, las organizaciones pueden enfrentar interrupciones de servicios, pérdidas económicas y daños reputacionales, especialmente en sectores donde las APIs son el eje central de la operación y la experiencia del usuario. Esto último en términos prácticos, implicaría por ejemplo que, una API con defectos no prevenidos por el equipo de QA, sale a producción y su resultado no es el esperado, entregando mal la información en la capa del cliente. Sería evidente que el cliente percibiría esto de forma negativa, generando daños a la reputación o posible pérdida de clientes.

Con el fin de buscar una solución moderna a esta problemática persistente, (Khankhoje, 2024a) expone la tesis de usar las herramientas y beneficios de la Inteligencia Artificial para poder ir cerrando la brecha entre los ritmos de desarrollo acelerados junto con plazos cortos de entrega que enfrentan los equipos de QA. En este contexto surge el concepto de Agente Inteligente, (Nembhard et al., 2023), el cual se entiende como un asistente virtual que colabora estrechamente con los testers para organizar y ejecutar tareas relacionadas con la verificación de funcionalidades, la generación de reportes y la comunicación de resultados con los desarrolladores. Así, el agente inteligente transforma la automatización en un proceso más dinámico y accesible, facilitando que actividades repetitivas y técnicas se gestionen de forma autónoma mientras los profesionales de calidad se concentran en tareas estratégicas.

1.1.1.3. Control de pronóstico

Guiado por (Kalech & Stern, 2020), se propone el desarrollo e implementación de un agente inteligente capaz de asistir en la planificación, diseño, implementación, ejecución y reporte de las pruebas funcionales de APIs REST cambiaría de forma significativa el panorama de los equipos de QA que enfrentan problemas de tiempos y recursos escasos. Esta solución permitiría automatizar tareas repetitivas, generar scripts a partir de descripciones en lenguaje natural y ejecutar validaciones de manera más rápida y precisa, reduciendo tanto los errores humanos como el esfuerzo manual.

En (Khankhoje, 2024a), se sugiere que al facilitar la integración con los flujos actuales de trabajo, este agente inteligente puede ser aprovechado por testers y desarrolladores sin necesidad de profundos conocimientos en automatización, lo que ayuda a democratizar el acceso a prácticas de calidad más avanzadas. En consecuencia, los equipos tendrían la oportunidad de enfocarse en análisis más estratégicos, ampliar la cobertura de pruebas y fortalecer la calidad final del software, alineándose con las exigencias de entrega continua y mejora constante propias de los entornos ágiles.

1.1.2. Justificación

En el ámbito de la calidad de software, (Bhanushali, 2023), especialmente dentro de equipos que trabajan bajo metodologías ágiles, se ha identificado la necesidad de agilizar y fortalecer los procesos de pruebas funcionales sobre APIs REST. Como contextualizan (Hayat et al., 2024), actualmente, muchas de estas actividades se desarrollan de manera manual o dependen de herramientas con capacidades limitadas, lo que conlleva a una sobrecarga de tareas repetitivas, lentitud en la validación de cambios y una cobertura insuficiente en escenarios críticos. Estas restricciones dificultan que los equipos puedan mantener el ritmo de entrega continua y la exigencia de calidad que demanda el entorno actual.

Esta realidad motiva la creación de un agente inteligente orientado a la automatización integral de las pruebas de APIs REST, diseñado para reducir la carga operativa, minimizar errores y potenciar la eficiencia general del proceso de testing.

La propuesta se articula a través de la arquitectura de tres componentes clave que trabajan de manera integrada:

- **Interfaz Local de Usuario:** Interfaz visual que se instalará en la computadora del usuario, que permite planificar, diseñar, implementar, ejecutar, reportar y monitorear casos de prueba, conectándose al backend y al modelo de IA ajustado para realizar operaciones avanzadas sin comprometer la privacidad de los datos locales.

- **Backend Local de Usuario:** Servicio local que se ejecuta en el servidor del usuario y procesa toda la lógica de negocio de las operaciones mediante endpoints y se conecta con el servicio de Inteligencia Artificial. Así mismo, articula la persistencia de datos mediante una base de datos portable.

- **Servicio de Inteligencia Artificial:** Responsable de interpretar descripciones en lenguaje natural y generar artefactos de testing automatizados a través de un modelo de lenguaje previamente ajustado (fine-tuned) específicamente para la generación y validación de pruebas funcionales en APIs REST.

1.1.3. Formulación del problema

¿El diseño e implementación de un agente inteligente permitirá agilizar la automatización y mejorar la calidad de las pruebas funcionales en APIs REST para equipos de QA en entornos de desarrollo ágil?

1.1.4. Variables del problema

La formulación de este problema contempla variables que guiarán el desarrollo y validación de la solución propuesta. Como variable independiente se considera el diseño e implementación del agente inteligente, el cual integra funciones de planificación, generación automática de scripts, ejecución de pruebas y reporte de resultados en APIs REST. Por su parte, como variables dependientes se definen la agilidad alcanzada en los procesos de automatización y el nivel de calidad logrado en las pruebas funcionales realizadas por los equipos de QA bajo entornos de desarrollo ágil, usando el Agente Inteligente. La relación entre estas variables permitirá evaluar en qué medida la incorporación del agente inteligente optimiza tiempos, reduce esfuerzo manual y fortalece la cobertura y precisión de las validaciones.

1.1.5. Sistematización del problema

- ¿Cómo levantar y priorizar requisitos y historias de usuario para la planificación, diseño y ejecución incremental de pruebas funcionales en APIs REST dentro de los equipos de QA?

- ¿Cómo diseñar la arquitectura y backlog del agente inteligente de forma que cada incremento aporte valor inmediato y se integre a los flujos de trabajo ágiles existentes?

- ¿Cómo implementar, sprint a sprint, los módulos del agente inteligente, asegurando generación automática de scripts, ejecución de pruebas y protección de información en cada iteración?

- ¿Cómo validar continuamente, mediante revisiones y demos, que el agente inteligente cumpla con los criterios de aceptación y requisitos de calidad definidos por los usuarios y stakeholders?
- ¿Cómo desplegar la solución de forma iterativa en entornos reales, fomentando su adopción, escalabilidad y refinamiento con feedback continuo de testers y desarrolladores?
- ¿Cómo documentar y actualizar de manera incremental el funcionamiento del agente inteligente, ofreciendo capacitación continua y monitoreo de resultados durante todo el ciclo ágil?

1.2. OBJETIVOS

1.2.1. Objetivo General

Diseñar, construir e iterar un agente inteligente que agilice la automatización y eleve la calidad en las pruebas funcionales de APIs REST, entregando incrementos funcionales a través de sprints y facilitando su adopción y mejora continua en equipos de control de calidad bajo marcos de trabajo ágiles.

1.2.2. Objetivos específicos

- Identificar y priorizar requisitos e historias de usuario claves para la automatización de pruebas funcionales sobre APIs REST dentro de los equipos de QA.
- Planificar y diseñar la arquitectura modular del agente inteligente, organizando su backlog para maximizar la entrega de valor en cada sprint.
- Desarrollar e implementar progresivamente los componentes del agente, validando su funcionalidad mediante revisiones continuas y pruebas de usuario.
- Integrar mecanismos de generación automática de scripts, ejecución de pruebas y manejo seguro de datos dentro de ciclos iterativos de mejora.
- Desplegar versiones incrementales del agente inteligente en entornos reales, recolectando feedback para ajustar su escalabilidad y compatibilidad con diversos niveles de experiencia técnica.

- Mantener documentación viva y capacitar a usuarios y desarrolladores durante todo el desarrollo ágil, garantizando comprensión, uso efectivo y sostenibilidad de la herramienta.

CAPÍTULO II

2. MARCO TEÓRICO

2.1. Fundamentos de la Calidad de Software

2.1.1. Concepto y evolución del Aseguramiento de la Calidad del Software

De la literatura científica (Ariel Menéndez-Verdecia & Noralma Aguilar-Moncayo III, 2021), se puede entender que el Software Quality Assurance (SQA) o Aseguramiento de la Calidad del Software es un proceso preventivo y de verificación que abarca todo el ciclo de vida del software, incorpora políticas, procesos, métricas, y garantiza que cada entrega satisfaga los requisitos explícitos y las expectativas implícitas del usuario al menor costo de corrección. En la práctica (Aizprúa Alfonso et al., 2019), el SQA alinea las decisiones técnicas y de negocio a través de criterios de aceptación medibles, mecanismos de trazabilidad y evidencia auditable del cumplimiento de los requisitos del negocio.

Históricamente (Lee, 2014), el SQA se ha relacionado con enfoques de inspección al final del desarrollo, con todo lo que ello implica en términos de coste de corrección tardía y opacidad sobre la causa raíz de los defectos. Con la maduración de los modelos de desarrollo de software (Kassie & Singh, 2020), surgen prácticas de verificación temprana tales como revisiones técnicas formales, walkthroughs, análisis estático y pruebas por niveles (unidad, integración, sistema, aceptación), moviendo el esfuerzo de la detección tardía a la prevención. En la forma actual de la práctica (Pardo et al., 2021), la transformación hacia metodologías ágiles y DevOps se ha establecido con estrategias shift-left (desplazamiento a la izquierda) y pruebas continuas, revisando la telemetría de producción para cerrar el ciclo de mejora con datos reales de uso.

Para clarificar el alcance (Spillner et al., 2021), distinguimos tres conceptos complementarios.

- Aseguramiento de Calidad (QA): Se enfoca en el gobierno de los procesos, prevención y establecimiento de políticas, estándares y auditorías.

- Control de Calidad (QC): Verificar el producto a través de pruebas, inspecciones y medición de resultados.
- Gestión de la Calidad: Consiste en la articulación estratégica que alinea QA y QC con objetivos de negocio, indicadores y ciclos de mejora.

Otro aspecto para evaluar la calidad del software es la combinación de pruebas manuales y automatizadas (Thooriqoh et al., 2021). Las pruebas manuales fueron la primera línea donde un humano revisa que el software desarrollado satisfaga las necesidades del negocio, por lo que certifica y verifica el desarrollo. No obstante, con el tiempo, los ciclos de vida de desarrollo de software cada vez más rápidos han hecho necesario probar más rápido con herramientas de codificación. De ahí nacen las pruebas automatizadas de software, las cuales por medio de lenguajes de programación, frameworks y arquitecturas pueden crear pruebas codificadas, scripts que permiten probar de manera más rápida y repetible.

Desde el punto de vista económico (Gutierrez Sullca, 2023), el SQA se justifica por el coste de arreglar fallos en las últimas etapas del ciclo de vida del software. El costo de arreglar se incrementa cuanto más tarde se descubre el defecto. En las últimas etapas de un software, si existen errores, más código se necesita cambiar, se tiene que volver a desplegar y probar todo de nuevo sin generar nuevos errores en los cambios. Por el contrario, ir probando cada parte en las primeras etapas de un software hace que cualquier error se solucione en el momento y no afecte a todo lo demás.

La calidad de software no solo tiene un impacto económico sino que prevenir defectos en ambientes productivos (Rohit Khankhoje, 2023), es decir en los cuales los usuarios finales ya utilizan el software desarrollado, impactan también en la reputación y la percepción del usuario del producto generado. Este impacto puede ser muy diverso y depende del contexto. En la historia de la calidad de software, se han remarcado desde ejemplos triviales como errores con bajo impacto como un usuario no pudiendo hacer click en un botón porque este tiene baja visibilidad, hasta errores críticos como fallos en hardware que provienen de un software defectuoso y ha llevado al fallo de sistemas mecánicos causando accidentes que han acabado con la vida de seres humanos.

Desde un usuario que no puede ver su estado financiero por un servidor caído hasta accidentes de cohetes provocados por fallos de software en la precisión decimal de un

campo decimal, son incidentes que se pueden prevenir o mitigar con un adecuado control de calidad del software, razón por la cual esta área de la tecnología tiene un gran impacto en los ciclos de producción de cualquier software.

2.1.2. Importancia del SQA en entornos ágiles

Para (Najihi et al., 2022), la calidad en metodologías ágiles es una forma de garantizar la entrega continua de software sin sacrificar la estabilidad. En equipos que trabajan en ciclos cortos, la validación debe ir paralela a cada cambio, por lo que el aseguramiento de la calidad deja de ser una fase final y se convierte en una práctica continua integrada en la planificación, el desarrollo y la revisión de resultados.

En la práctica (van Driel et al., 2020), el aseguramiento de la calidad se manifiesta cuando cada historia de usuario define criterios de aceptación transparentes y verificables, cuando las pruebas se ejecutan con frecuencia en un entorno de integración continua y cuando el equipo asume la responsabilidad colectiva de prevenir regresiones. Este método disminuye los tiempos de corrección, evita la acumulación de errores y permite que el producto fluya a un ritmo constante.

Para guiar este esfuerzo ágil de desarrollo (Najihi et al., 2022), es bueno clasificar los riesgos en orden de prioridad: impacto en el negocio, visibilidad para el usuario, complejidad técnica y dependencias externas. Con esta priorización, el equipo establece coberturas mínimas por historia de usuario, datos de prueba representativos y conjuntos de pruebas para diferentes propósitos, como verificación rápida, regresión o validaciones más profundas cuando sea necesario. Esta manera de trabajar hace que se pueda uno enfocar en lo esencial y aprovechar mejor el tiempo.

2.1.3. Artefactos típicos del aseguramiento de la calidad de software

En la calidad del software (Vinueza, 2009), los artefactos son documentos y registros que guían el proceso de prueba, aseguran la trazabilidad y mantienen evidencia para la toma de decisiones. Para este trabajo se adopta un conjunto mínimo y suficiente para entornos ágiles: plan de pruebas, casos de prueba, conjuntos de prueba, ejecuciones de prueba y registro de defectos.

- **Plan de pruebas:** Se entiende como, el documento que define alcance, objetivos, enfoque, recursos y responsables para validar un conjunto de

requisitos o una versión del producto. En su versión esencial, especifica qué se probará y qué no, riesgos principales, criterios de entrada y salida, calendario tentativo y roles del equipo. En marcos iterativos, el plan se mantiene en constante cambio: se actualiza por iteración y conserva solo lo necesario para no frenar el ritmo de entrega.

- **Casos de prueba:** Un caso de prueba, describe una condición verificable con entradas, pasos y resultados esperados para comprobar el cumplimiento de un requisito funcional o de calidad. Para que resulte útil, debe ser claro, reproducible, independiente en lo posible y vinculado explícitamente al requisito que valida, lo que sostiene la trazabilidad y facilita el análisis de impacto ante cambios.
- **Conjuntos de prueba (test suites o test sets):** Son agrupaciones de casos con un propósito común, por ejemplo, verificación rápida tras un cambio, regresión de un módulo o validación previa a un despliegue, que permiten planificar ejecuciones de forma eficiente y controlar tiempos.
- **Ejecuciones de prueba:** Corresponden al registro de la ejecución de uno o varios casos bajo condiciones específicas: versión del sistema, ambiente, datos utilizados y resultado obtenido. Para efectos de auditoría, deben conservar evidencias como respuestas de servicios, capturas, registros de sistema y cualquier insumo que demuestre qué se probó y con qué resultado. Integradas a la automatización, estas ejecuciones se disparan de forma regular y temprana, reduciendo el tiempo de detección de errores.
- **Bugs o defectos**: Se definen como la discrepancia entre el comportamiento observado y el esperado según requisito o especificación. Un buen registro de defecto incluye pasos para reproducir, evidencia, severidad, prioridad, ambiente y estado de seguimiento. En la gestión diaria, la severidad refleja impacto técnico o de negocio, mientras que la prioridad indica el orden de atención, lo que permite coordinar correcciones con criterio.

Dentro de todos artefactos es importante asegurar la trazabilidad entre los elementos creados (Mohamad Yusop et al., 2023), una cadena típica recomendada conecta requisito caso de prueba, conjunto de prueba, ejecución, evidencia, defecto (cuando aplica). Esta relación hace posible responder preguntas clave: qué requisito se validó, con qué caso, cuándo se ejecutó, qué resultado arrojó y qué acciones derivaron.

2.2. Metodologías Ágiles y el Impacto de las Pruebas

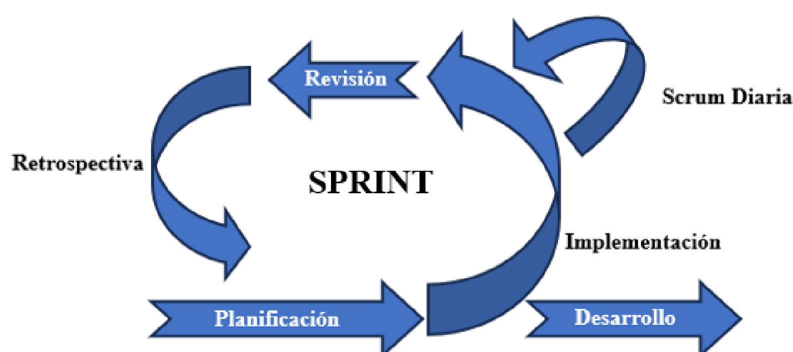
En la literatura de metodologías ágiles (López-Rodríguez & García-Peña, 2021), se afirma que trabajar en ciclos cortos, con feedback constante y capacidad de adaptación cambia la manera de verificar el software: la calidad deja de ser una etapa al final y se incorpora desde la planificación hasta la entrega. En ese contexto, la verificación se basa en criterios definidos, trazabilidad de lo que se verificó y evidencia suficiente para tomar decisiones oportunas.

2.2.1. Principios de Scrum y Kanban aplicados al ciclo de pruebas

Concebido a principios de los años 90 (Verwijs & Russo, 2023), Scrum es un marco de trabajo de tipo ágil para gestionar el desarrollo de proyectos mediante sprints (iteraciones con objetivos específicos y de duración fija), con roles, eventos y artefactos regulares que permiten la inspección y la adaptación. En el ciclo de pruebas, esto significa que cada sprint incluye pruebas planificadas desde el refinamiento de requisitos y que cada desarrollo se revisa en su respectivo sprint, para una entrega de valor continua, asegurando la calidad.

Se puede enlazar las ceremonias de Scrum (Fang, 2021), como oportunidades para realizar control de calidad a lo largo del sprint. Cada evento representa una oportunidad tangible para prevenir defectos, diseñar verificaciones y obtener evidencias en concordancia con los criterios de aceptación.

Figura 1. Gráfico ilustrativo de las ceremonias de un Sprint



Nota. Elaborado por el autor.

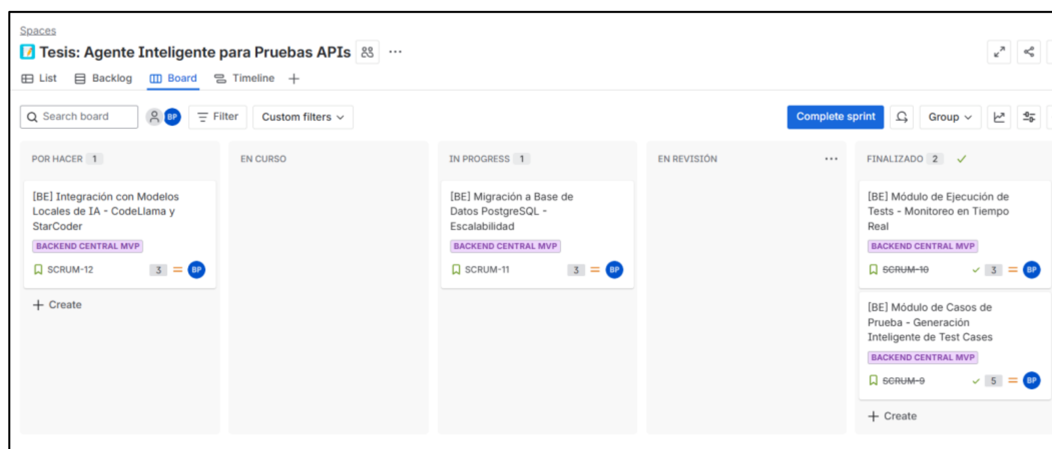
- **Refinamiento del Product Backlog**: Se negocian historias, se acuerdan criterios de aceptación verificables. Todo el equipo reduce ambigüedades o

confusiones, proponen elementos a testear, definen riesgos y sugieren datos de prueba representativos.

- **Planificación del Sprint**: Se eligen las tareas y se decide el objetivo del sprint. Por parte de QA, se estiman pruebas, se alinea cobertura mínima, se define qué se automatiza en esta iteración y se verifica que las historias estén listas para construir y probar.
- **Daily Scrum**: Se realiza un seguimiento de progreso y bloqueos. QA informa el estado de las pruebas, fallos críticos, flujos bloqueados por dependencias y negocia la priorización de casos/entornos cuando cambian los riesgos.
- **Desarrollo durante el sprint**: Es la parte de desarrollo e integración continua. El equipo de QA realiza pruebas manuales o automatizadas, pruebas exploratorias, audita contratos de APIs y monitorea métricas de calidad en el flujos de trabajo.
- **Revisión del Sprint (Sprint Review)**: Se revisa el trabajo realizado con las partes interesadas. Aquí el equipo de QA muestra evidencia tangible de cumplimiento de criterios, resultados de pruebas y defectos abiertos que afectan al negocio, y recopila comentarios para mejorar la cobertura.
- **Retrospectiva del Sprint**: Es la fase que permite mejorar los procesos en base a lo realizado. QA sugiere medidas sobre defectos significativos, estado de las pruebas, deuda de automatización, reproducibilidad de pruebas y acuerda modificaciones en los criterios para definir el nivel de calidad.

Por otro lado, Kanban (Tetteh, 2024), originado en prácticas de flujos de trabajo, se centra en visualizar el trabajo, delimitar el trabajo en curso y mejorarlo continuamente. Llevado al ciclo de pruebas, esto se traduce en visualizar cuántas pruebas están pendientes, identificar cuellos de botella (datos, entornos, dependencias) y adaptar políticas de trabajo para que las pruebas fluyan sin bloqueos. La representación en un tablero con estados definidos y reglas sencillas permite la intervención oportuna.

Figura 2. Visualización de un ejemplo de tareas en un tablero Kanban



Nota. Elaborado por el autor.

La mezcla de Scrum y Kanban crea un refuerzo en la calidad (Tetteh, 2024), Scrum agrega ritmo y ceremonias regulares de inspección y adaptación; Kanban, aporta control de flujo y visualización fácil del trabajo. En conjunto, los proyectos logran con ambas herramientas ciclos de desarrollo y pruebas más predecible, menos tiempo entre una actividad y otra y más visibilidad de lo que ya está probado y lo que queda pendiente.

2.2.2. Entregas incrementales y necesidad de regresión continua

Con cambios incrementales en cada sprint, cada nueva entrega puede introducir defectos a funcionalidades ya estables (Qasim et al., 2021). Aquí, las pruebas de regresión son las que permiten verificar que los módulos o componentes ya realizados, siguen funcionando con normalidad tras un cambio o una nueva característica. En metodologías ágiles, esta revisión no se hace de forma aislada, debe acompañar cada integración para identificar errores tempranamente y prevenir que escalen a fases más tardías.

En la práctica (Qasim et al., 2021), una estrategia de regresión continua mezcla conjuntos de pruebas delimitadas y de ejecución rápida con conjuntos más grandes programados en momentos específicos. Las primeras se ejecutan ante cualquier cambio significativo para dar una señal temprana; las segundas profundizan la verificación por módulo o antes del lanzamiento de una versión grande. El objetivo es balancear tiempo de respuesta con profundidad de verificación.

Para las APIs de tipo REST (Minhas et al., 2020), la regresión continua se apoya en tres elementos.

- Primero, pruebas de contrato que aseguren la estabilidad de los acuerdos de intercambio de datos.
- Segundo, datos de prueba representativos que permitan ejercitar reglas de negocio y validaciones de entrada.
- Tercero, evidencia clara y comparable entre ejecuciones (respuestas, registros y condiciones de borde) que permita identificar desviaciones con precisión.

La sincronización con la integración continua (Minhas et al., 2020), refuerza este modelo: cualquier cambio significativo inicia automáticamente las pruebas definidas y guarda sus resultados con la trazabilidad requerida. Cuando surge un error, el equipo soluciona antes de continuar, evitando acumulaciones y permitiendo mantener el flujo de entrega.

2.3. Pruebas Funcionales en APIs REST

2.3.1. Definición y características de las APIs REST.

Desde una perspectiva técnica (Golmohammadi et al., 2023), una API es una interfaz que permite que diferentes sistemas se comuniquen e interactúen entre sí de manera controlada. En la actualidad, existen múltiples tipos de API que comparten características y esquemas diferentes.

- **REST** es la opción predominante por su alineación con HTTP, amplia documentación y estilos de construcción definidos que permiten estandarizar la construcción de este tipo de servicios.
- **SOAP** se apoya en XML y contratos de tipo manual de instrucciones que indica cómo se realizará el intercambio de datos.
- **GraphQL** expone un esquema tipado y permite que el cliente pida exactamente los campos que necesita, lo que reduce sobrecarga sistemas que manejan grandes volúmenes de datos.
- **gRPC** usa Protocol Buffers sobre HTTP/2 y se caracteriza por tener baja latencia, útil al lidiar con streaming y contratos estrictos.
- **Las APIs en tiempo real** suelen trabajar con **WebSockets** para comunicación bidireccional sostenida, en las cuales, los webhooks implementan un patrón

dirigido por eventos que envía notificaciones cuando algo relevante ocurre en el sistema.

En particular (Golmohammadi et al., 2023), cuando se adhiere al estilo REST, la comunicación se estructura en torno a recursos identificados por direcciones web y se utilizan los métodos de HTTP para manipularlos. Este estilo de diseño fomenta la simplicidad, el desacoplamiento y la fácil integración entre aplicaciones de diferentes tecnologías.

En este protocolo un cliente hace solicitudes y un servidor responde con un código de estado y una representación de datos (generalmente en JSON). Los métodos más frecuentes se pueden ver en la siguiente tabla.

Tabla 1. Descripción de los métodos HTTP más comunes

Método (HTTP)	Acción típica en REST	Cuerpo en la petición	Respuesta esperada	Casos de uso comunes
GET	Recuperar recursos o colecciones	No habitual	200 OK con JSON	Listar productos, obtener detalle por id
POST	Crear un nuevo recurso bajo una colección	Sí, con el payload del nuevo recurso	201 Created con representación y/o Location	Registrar usuario, crear orden
PUT	Reemplazar un recurso completo	Sí, con la versión completa del recurso	200 OK o 204 No Content	Actualizar perfil completo
PATCH	Modificar parcialmente un recurso	Sí, con solo los campos a cambiar	200 OK o 204 No Content	Cambiar estado de una orden, actualizar un campo puntual
DELETE	Eliminar un recurso	No habitual	204 No Content o 200 OK	Borrar una dirección, remover ítems del carrito por id

Nota. Elaborado por el autor.

Una solicitud HTTP común tiene diferentes elementos esenciales (Juviler, 2023). Primero, la línea de solicitud une el método y la URL. La URL está formada por esquema (https), nombre de host, ruta que localiza al recurso y parámetros de consulta si se filtra o pagina. Segundo, los encabezados agregan información, como Content Type para especificar el tipo de cuerpo, Accept para el tipo de respuesta que se espera o Authorization para las credenciales. Tercero, el cuerpo lleva datos cuando la operación

lo exige (por ejemplo, un objeto JSON en create o update). Finalmente, el contexto de seguridad puede abarcar tokens temporales, renovaciones y restricciones de procedencia.

Por otro lado una respuesta HTTP también tiene sus partes específicas (Juviler, 2023). El código de estado informa el resultado de la operación, la cabecera de respuesta proporciona información adicional (por ejemplo, paginación o control de caché) y el cuerpo contiene la información solicitada o un mensaje de error estructurado si la solicitud es fallida.

Para organizar la interpretación de resultados, los códigos HTTP se suelen agrupar de la siguiente manera (Patni, 2023).

- Los **2xx** confirman éxito, con usos comunes como 200 para recuperación, 201 cuando se crea un recurso y 204 cuando no hay contenido que devolver.
- Los **3xx** expresan redirecciones.
- Los **4xx** señalan errores atribuibles a la petición, por ejemplo 400 por entrada inválida, 401 cuando falta autenticación, 403 por autorización insuficiente, 404 si el recurso no existe, 409 ante conflictos de estado y 422 cuando los datos no cumplen reglas de negocio.
- Los **5xx** indican fallos del lado del servidor, típicamente 500 por error interno, 502 por problemas de pasarela y 503 cuando el servicio no está disponible.

Otra característica importante para el diseño y la prueba es que REST es sin estado (Patni, 2023). Cada solicitud se maneja de forma independiente y lleva consigo la información necesaria para resolverla, lo que facilita la escalabilidad del servidor y hace que su comportamiento en condiciones de carga sea predecible. Al mismo tiempo, técnicas como idempotencia en las actualizaciones, paginación en listados grandes, filtros y ordenamientos explícitos, versionado explícito de la API, ayudan a prevenir rupturas y mantener la integración de servicios mediante APIs a lo largo del tiempo.

Finalmente, en seguridad y gobierno se sugieren controles para el uso adecuado de la API REST (Alharbi & Moulahi, 2023). La autenticación y autorización debe verificarse en escenarios de éxito de acceso y fallos de acceso, límites de tasa y control de velocidad para evitar usos indebidos, validación de entradas para prevenir errores y vulnerabilidades, política de orígenes para controlar el consumo desde aplicaciones

web. Estos elementos se añaden a las pruebas funcionales para garantizar que la API no solo responda, sino que lo haga en condiciones seguras y predecibles.

2.3.2. Importancia de las pruebas funcionales de APIs.

Al construir un servicio que expone una API, antes de promoverla a producción, ésta debe pasar por etapas de pruebas (Martin-Lopez et al., 2022), en donde se verifique que cada endpoint de la API cumpla con lo especificado en los requisitos y en la especificación: reglas de negocio, validaciones de entrada, transformaciones de datos y mensajes de error claros. La utilidad es identificar anomalías antes de que el cambio llegue a manos de usuarios internos o externos en producción.

Las pruebas funcionales validan que la semántica de cada endpoint corresponda con la regla de negocio y que siempre regrese la misma respuesta ante entradas válidas, límites y de error (Martin-Lopez et al., 2022). Prevenir errores mediante pruebas permite tener métricas que dan confianza a los equipos. Por ejemplo, se disminuye incidentes por errores de integración, se elimina retrabajos en despliegues y promueve el cumplimiento de KPIs como son la tasa de éxito de transacciones. Por ejemplo, si un cliente móvil invoca la API de órdenes de un comercio en línea, se espera que la API devuelva un 200 OK, así también el inventario, los precios y el estado de pago se modifiquen o cambien de forma correcta. Verificar estos flujos antes de promover cambios reduce incidencias en producción y mejora la experiencia del usuario final.

La integración se vuelve relevante cuando conectamos la API con el resto del sistema y con terceros, ya que en ello se basa la fiabilidad y escalabilidad del software creado (Martin-Lopez & Alonso, 2023). Probar end to end con frontends web y móviles valida que la versión expuesta se alinea con lo que esperan los clientes en términos de formatos, códigos y tiempos de respuesta. Hacerlo también entre servicios backend (con colas o eventos) permite ver si existen errores de serialización, esquemas y orden de mensajes. En arquitecturas de microservicios, estas pruebas validan contratos y previenen fallos en cadena cuando un servicio interactúa con otros. El resultado de un desarrollo de este tipo de servicios correctamente validado y verificado es una capa de APIs capaz de comunicarse con sus clientes y que permite integrarse exitosamente con el resto de los componentes de un sistema de software.

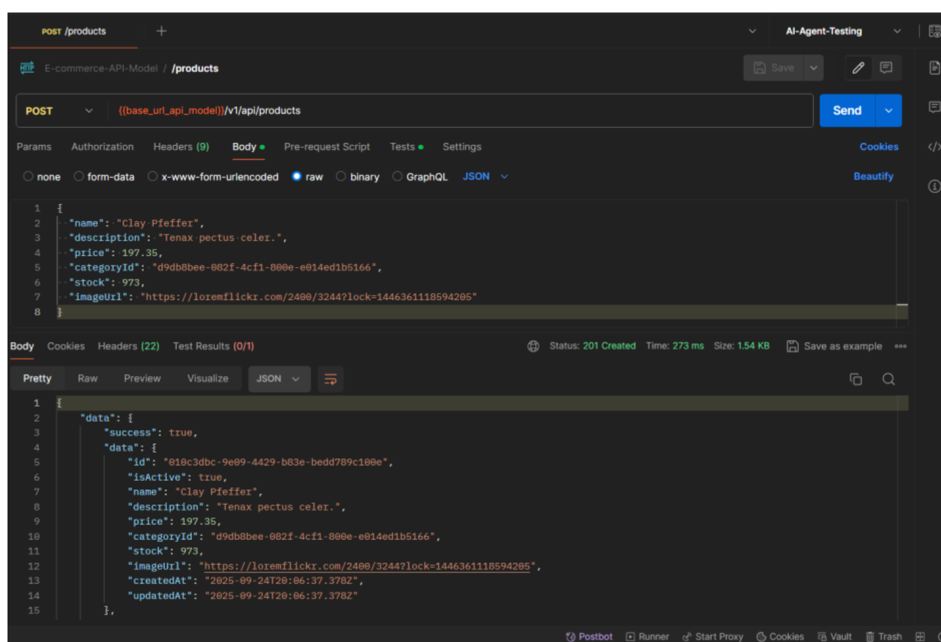
2.3.3. Herramientas y enfoques tradicionales.

Tal como recogen diversos autores y manuales técnicos (Yuda Syahidin & Randy Ramadhan, 2021), las herramientas comunes para probar APIs posibilitan estructurar peticiones, parametrizar datos, validar respuestas y automatizar ejecuciones. A continuación, se presentan tres formas usuales de cómo se prueban APIs.

Postman estructura el trabajo en espacios y colecciones (Kore et al., 2022). Se pueden configurar entornos con variables como host, keys o tokens y llamarlas en las peticiones, para poder alternar entre desarrollo, pruebas y producción sin tener que reescribir. En cada petición se pueden adjuntar scripts pre-request para preparar cabeceras o datos de prueba y pruebas con aserciones simples sobre el código de estado, el esquema de la respuesta o valores específicos.

Para ejecutar en lote, el Runner o ejecutador puede probar colecciones enteras (con múltiples iteraciones) y archivos de datos. Otra herramienta útil que suele usarse en conjunto con Postman es **Newman** (Kore et al., 2022), el cual se trata de una herramienta de línea de comando que permite ejecutar desde una terminal colecciones de Postman y tener reportes listos para adjuntar a informes de prueba y auditar los resultados obtenidos.

Figura 3. Ejemplo de una petición HTTP en Postman



Nota. Elaborado por el autor.

Para analizar el comportamiento en condiciones de carga o estrés, **JMeter** puede configurar planes de prueba con grupos de hilos que simulen varios usuarios simultáneos (Jonsson et al., 2022). La configuración de ejemplo agrega muestras HTTP para cada endpoint, temporizadores para simular tiempos de espera reales, aserciones de código/contenido y listeners para registrar resultados. El componente CSV Data Set permite la parametrización de datos y la ejecución sin interfaz hace posible su uso en procesos de integración. Aunque se le relaciona con pruebas de carga, también soporta verificaciones funcionales elementales que refuerzan la cobertura cuando el número/cantidad de usuarios y peticiones son determinantes para la forma en que realmente opera una API REST.

2.3.4. Limitaciones de los enfoques manuales y automatizados convencionales.

El enfoque de pruebas manuales de APIs es limitado cuando el ritmo de cambio es elevado (Ehsan et al., 2022), esto se debe a que repetir request HTTP una tras otra es un proceso largo y susceptible a errores al realizarse de forma manual, además es un proceso poco reproducible y muy dependiente de la persona que lo ejecute. Además, conservar datos de prueba consistentes sin ayuda adicional se convierte en una desventaja, ya que el tester tiene que registrar los resultados y adjuntarlos manualmente.

Frente a estos enfoques manuales la automatización de pruebas de APIs permite reforzar aspectos como la reproducibilidad de resultados, rapidez y consistencia, generando informes de resultados en tiempos cortos y que pueden ser integrados automáticamente a herramientas de reportes que permiten obtener métricas de calidad de software.

Sin embargo, la automatización tradicional también tiene limitaciones (Ehsan et al., 2022). Los scripts de prueba son susceptibles a cambios en rutas, esquemas o mensajes, así mismo, el mantenimiento aumenta con la cantidad de casos, y la dependencia de entornos y datos puede generar fallos esporádicos que reducen la confianza en las pruebas. Cuando no se prioriza por riesgo, los tiempos de ejecución se alargan y la integración continua pierde velocidad.

Lo que se puede entonces analizar, es que de los enfoques manuales y automatizados, surgen oportunidades de mejora que pueden permitir pruebas ágiles que se adapten a los ritmos vertiginosos de desarrollo de software o en concreto APIs. Este contexto se alinea

con la propuesta de esta tesis: un agente que ayude al tester en las tareas repetitivas y mantenga la calidad sin sobrecargar el proceso.

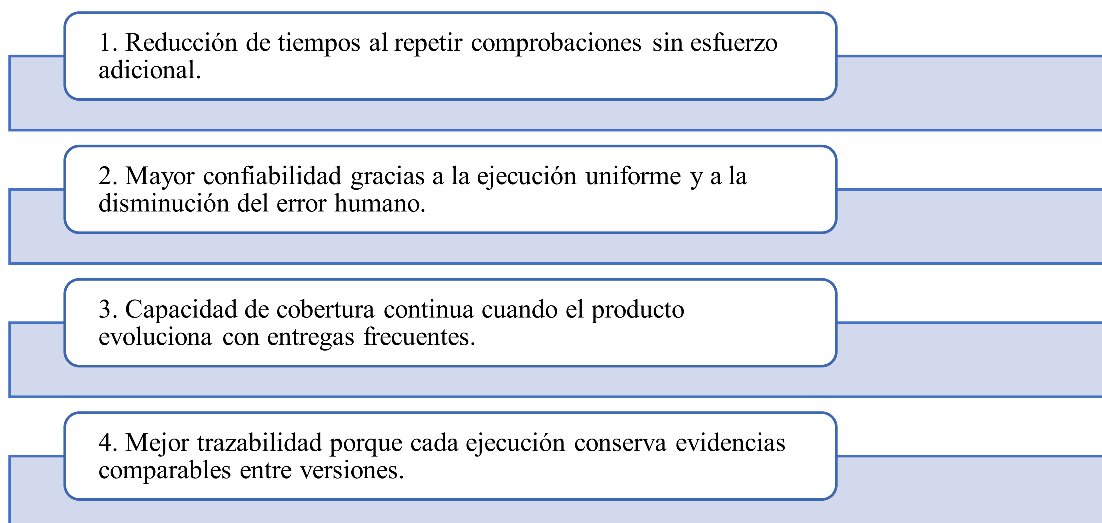
2.4. Automatización de Pruebas de Software

2.4.1. Concepto, beneficios y niveles de madurez en test automation.

La automatización de pruebas (Popov et al., 2022), se entiende como el uso de herramientas y scripts de prueba para ejecutar verificaciones de forma repetible, comparar resultados con los esperados, preparar datos y capturar pruebas sin intervención humana continua. Esta definición incluye pruebas unitarias, de integración y de regresión, con criterios de aceptación definidos para determinar si un cambio es aceptable.

En términos de beneficios, se pueden lista principalmente los que encontramos en la siguiente figura.

Figura 4. Diagrama de beneficios de la automatización de pruebas de software



Nota. Elaborado por el autor.

Para entender cómo un equipo de calidad de software llega a establecer pruebas automatizadas, es útil pensar en la automatización como un proceso de madurez creciente (Wang, Mäntylä, Liu, Markkula, et al., 2022).

- En las primeras etapas predominan las pruebas manuales y pocos scripts de prueba aislados.
- En segundo lugar, los scripts de prueba se agrupan en suites parametrizadas.

- En un tercer nivel, estas ejecuciones se acoplan con la integración continua y bloquean cambios si fallan criterios de calidad.
- Luego, el equipo prioriza por riesgo, estabiliza información y ambientes, y aplica métricas para mejorar lo que más impacta en el flujo.
- En etapas más avanzados, se incorpora observación en producción para retroalimentar las pruebas y se automatiza la creación de artefactos a partir de contratos o especificaciones.

2.4.2. Frameworks de Testing y herramientas más usadas.

La automatización de pruebas (Popov et al., 2022), puede realizarse en cualquier tecnología ya sea web, móvil o incluso de hardware. Para este trabajo el autor se enfocará en dos tipos relevantes en torno al desarrollo de software moderno como son: verificación por interfaz visual (páginas webs o aplicaciones móviles) y verificación de servicios (APIs).

Un framework de automatización de pruebas (Umar & Zhanfang, 2019), se compone de una colección de librerías, carpetas y código estructurado bajo patrones de diseño para desarrollar casos de prueba, conjuntos o suites de prueba y los artefactos que conforman todo el código para que las pruebas se ejecuten en los distintos entornos de programación, como por ejemplo Java, Python, Javascript o TypeScript.

En ese sentido, Selenium, Cypress y Playwright se posicionan entre los frameworks más usados para la automatización. Los tres ofrecen opciones robustas para suites de pruebas, con tienen distintos lenguajes, estructura de proyecto, patrones de diseño y capacidades nativas para probar servicios, las cuales son diferencias que es bueno conocer antes de elegir.

Selenium

Primero se describirá a Selenium (Gudavalli & JayaLakshmi, 2022), el cual es un conjunto de herramientas para automatizar acciones en navegadores a través de WebDriver. Soporta múltiples lenguajes: Java, Python, C# o JavaScript. Una opción común y con amplia documentación es Java, en el cual normalmente se gestionan las dependencias con Maven o Gradle y se organizan las fuentes en src/test/java para las pruebas y src/main/java para las utilidades comunes.

En cuanto a estructura de carpetas, lo normal es tener pages/ (objetos de página), tests/ (casos de prueba), data/ o fixtures/ (datos de prueba) y utils/ (funciones auxiliares). Los más comunes son el Page Object Model para encapsular localizadores y acciones, el Data Builder para crear datos de entrada y, en proyectos más grandes, Screenplay para modelar tareas y actores de manera declarativa.

En la ejecución, se suele emplear JUnit o TestNG como frameworks de pruebas (Gudavalli & JayaLakshmi, 2022), esperas explícitas para una sincronización precisa y Selenium Grid para la ejecución paralela en varios nodos. Como Selenium no proporciona un cliente HTTP integrado, a menudo se integran bibliotecas externas, como RestAssured en Java o requests en Python. Esto permite combinar o separar las pruebas de UI y de servicios en el mismo repositorio si es necesario. La configuración de los entornos se realiza a través de perfiles Maven/Gradle o archivos .properties. Para los informes se usan herramientas como Allure o Surefire.

Cypress

Por otro lado se tiene Cypress, el cual es un framework de pruebas end-to-end (Thekkan Othayoth & Anuar, 2022), basado en el navegador, escrito en JavaScript o TypeScript, permite flujos de trabajo ágiles: tiene esperas automáticas, captura evidencias de manera precisa y registra cada paso. Su estructura generalmente incluye:

- cypress/e2e/ (casos de prueba)
- cypress/fixtures/ (datos de prueba)
- cypress/support/ (comandos personalizados y hooks)
- cypress.config (configuración global)

En las pruebas se suelen emplear objetos de página, funciones reutilizables para acciones comunes, fixtures para datos repetitivos y comandos personalizados para evitar duplicaciones.

Para pruebas de APIs (Palani, 2021), comandos como cy.request() hace posible mandar peticiones HTTP directamente sin tener que pasar por la interfaz. También se puede usar cy.intercept() para simular o escuchar llamadas, aislando dependencias y controlando el entorno con variables.

Playwright

Otro Framework con amplio uso es Playwright, el cual ayuda en la automatización de las pruebas en aplicaciones modernas (Polkhovskaya, 2025), y es conocido por su estabilidad con sus esperas automáticas y aislamiento por contexto. Soporta varios lenguajes: TypeScript/JavaScript, Python, .NET, Java. En TypeScript, el más común, la configuración inicial se hace a través del archivo playwright.config.ts y una carpeta tests/.

El framework proporciona soporte para fixtures reutilizables, paralelismo de forma nativa y gestión de entornos mediante proyectos. En las pruebas de interfaz se puede utilizar la estructura pages/ con el patrón Page Object. Para las pruebas de servicios, Playwright tiene un cliente HTTP integrado (request) y contextos de solicitud para agrupar autenticación, encabezados y datos, evitando el uso de herramientas externas para las pruebas de API.

Con respecto a la ejecución e integración (Polkhovskaya, 2025), con un solo comando y distintas flags (npx playwright test) proporciona paralelismo, fragmentación por archivo, informes HTML/JUnit/JSON y el reporte para depurar y revisar resultados de ejecución. La parametrización de entornos se hace con settings en el mismo archivo y con variables para poder cambiar entre desarrollo, test y producción sin duplicar código. Este método reúne en un solo lugar utilidades, datos y ajustes.

Comparativa de frameworks enfocados en pruebas de API's

La siguiente tabla resume criterios prácticos para poder comparar los 3 frameworks (Umar & Zhanfang, 2019), mencionados anteriormente.

Tabla 2. Comparativa de las características de Frameworks para API testing

Criterio	Selenium	Cypress	Playwright (TypeScript)
Lenguajes	Java, Python, C#, JS	JS, TS	TS, JS, Python, .NET, Java
Estructura base	src/test/java, pages, tests, utils	cypress/e2e, fixtures, support	tests, fixtures, utils, playwright.config.ts
Cliente HTTP para APIs	Externo (RestAssured/requests)	Nativo (cy.request)	Nativo (request con contextos)
Ejecución/CI	JUnit/TestNG, Grid	Runner integrado	Corredor único con paralelismo

Comunidad/soporte	Muy amplia y con madurez técnica	Amplia, foco JS/TS	Muy activa, documentación clara
Curva de adopción	Alta	Media	Media
Idoneidad para APIs	Requiere biblioteca externa	Buena	Muy buena

Nota. Elaborado por el autor.

2.4.3. Retos de la Automatización de Pruebas

La literatura indica que la automatización se enfrenta a ciertos tipos de problemas recurrentes (Nass et al., 2021).

- Lo primero es el mantenimiento de scripts. Cambios de ruta, de contratos, de mensajes o de interfaz implican la modificación de varios archivos, se crean duplicidades y surgen errores esporádicos por esperas mal calculadas o localizadores cambiantes. Si el desarrollo es ágil, los cambios constantes y el proyecto es grande y cuenta con miles de casos de prueba, mantener toda la base de código se vuelve complejo ante cambios pequeños.
- La segundo es la deuda técnica. En la urgencia por entregar, se dejan para después las refactorizaciones, mejoras de seguridad o rendimiento. Nuevas versiones o mejores prácticas se van acumulando, lo cual muchas veces puede forzar a los equipos de QA a priorizar entre enfocarse en la calidad técnica de los scripts de prueba o la cobertura y profundidad de las pruebas.
- En tercer lugar, la falta de cobertura útil. La cantidad de casos de prueba puede crecer, aún así, la cobertura efectiva de escenarios de negocio no, ya que hay un doble enfoque en codificar los script y mantener el framework y evaluar aspectos de calidad. Por tanto las áreas críticas pueden quedar sin cobertura de escenarios de prueba.

Desde una mirada crítica, estos retos comparten raíces. Al existir tantos frameworks de automatización, librerías herramientas, técnicas y enfoques para probar, el tester puede llegar a pasar más tiempo elaborando artefactos de prueba, haciendo su mantenimiento y ejecución que elaborando escenarios de prueba valiosos que aporten mayor cobertura de requerimientos del software bajo prueba.

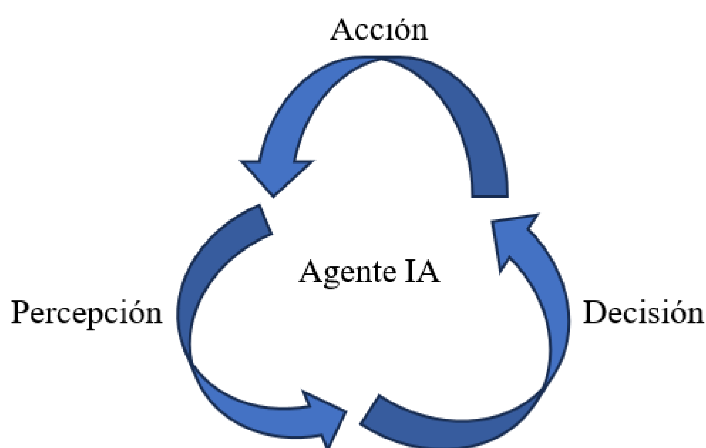
Esto último deja un espacio de oportunidad para nuevas herramientas que ayuden a optimizar tiempos de trabajo, organizar de mejor manera los artefactos de prueba y permitan reducir los tiempos de creación, mantenimiento y ejecución de casos de prueba.

2.5. Inteligencia Artificial en el Contexto del Testing

2.5.1. Concepto de agente inteligente y su aplicación en ingeniería de software.

Un agente inteligente enfocado al ámbito de la calidad de software (Haldorai, 2022), se puede conceptualizar como un sistema que recibe información del entorno (requerimientos, contratos de APIs, resultados de ejecución), razona sobre objetivos y restricciones, y actúa ejecutando acciones, todo de manera cíclica siguiendo el esquema percepción-decisión-acción que usa la retroalimentación para mejorar en la siguiente iteración.

Figura 5. Gráfico de bucle percepción, decisión, acción de la IA



Nota. Elaborado por el autor.

Ese ciclo en QA se puede manifestar de la siguiente manera: revisar artefactos (criterios de aceptación, contratos), sugerir o crear casos de prueba, ejecutar pruebas controladas y documentar evidencia (códigos, cuerpo de respuesta, tiempos, trazas) para informar al equipo.

En el plano arquitectónico (Haldorai, 2022), un agente útil para QA combinaría diferentes elementos:

- Un **orquestador** que decide el siguiente paso (diseñar, ejecutar, priorizar, reportar).
- Un conjunto de **herramientas** (cliente HTTP para APIs, ejecutor de pruebas, repositorio de casos y datos).

- Una **memoria de contexto** (de corto plazo para la sesión y de largo plazo para plantillas y lecciones aprendidas).
- **Políticas de control y auditoría** que gobiernan acceso, trazabilidad y limpieza de datos.

La forma de operar del agente permite diferenciar entre distintos tipos, entre los más relevantes están los agentes apoyados en reglas (plantillas, heurísticas, catálogos de aseveraciones) y agentes potenciados por modelos de lenguaje (intérpretes de texto y contratos). En ambos casos se deben establecer límites de autonomía: desde proponer artefactos aprobados por humanos hasta operar bajo condiciones predefinidas. En este trabajo se opta por un humano en el bucle, en el que el agente sugiere y crea código susceptible de revisión humana.

Durante el ciclo de vida de las pruebas de software (Khan et al., 2024), el agente se puede posicionar donde más retorno genere, como por ejemplo la refinación de requisitos para definir criterios de aceptación, la creación de casos de prueba basados en escenarios y datos o la ejecución priorizada en CI/CD y el análisis de resultados para agrupar fallos, sugerir acciones de mejora y predecir futuros incidentes. En cualquiera de estos escenarios la IA sería una aliada para mejorar los tiempos de QA y poder dedicarlo a la calidad y no a tareas repetitivas.

2.5.2. Uso de modelos de Inteligencia Artificial en la actualidad

Panorama general de la inteligencia artificial actual

Desde la literatura científica (Chang, 2023), la inteligencia artificial se ha consolidado como el campo que tiene como objetivo potenciar a sistemas informáticos con capacidades que les permitan percibir el entorno, razonar y actuar ante problemas complejos. Los últimos desarrollos (Sauvola et al., 2024) han desplazado el foco de los sistemas basados en reglas a los grandes modelos de lenguaje (LLM), capaces de comprender y generar lenguaje natural de forma coherente y contextualizada.

Conceptualmente (Lucas, 2023), un LLM es un modelo entrenado con grandes cantidades de datos usando la arquitectura de transformadores para encontrar relaciones entre palabras. Esta habilidad les permite tomar especificaciones en lenguaje natural y convertirlas en artefactos estructurados, lo cual puede resultar muy útil en campos como el aseguramiento de calidad de software, para pasar de requerimientos en lenguaje

natural o instrucciones humanas sobre cómo ejecutar un caso de prueba, a artefactos funcionales en forma de código para ejecutar y realizar pruebas.

Opciones de modelos disponibles y formas de consumo

En la actualidad existe una gran cantidad de formas en las que se distribuyen y comercializan los modelos de lenguaje de inteligencia artificial (de Carvalho Souza & Weigang, 2025). Algunos de los más relevantes serán descritos a continuación.

- Las APIs comerciales, como las de OpenAI, Anthropic (Claude), Google (Gemini), DeepSeek, que dan acceso a modelos altamente sofisticados mediante llamadas autenticadas y pago por uso (mediante tokens).
- La segunda son los agregadores de nube (Amazon Bedrock, Google Vertex AI), que exponen diferentes modelos bajo una misma interfaz, estandarizando la seguridad y los modelos de precios (pago por uso, por lotes o por capacidad reservada).
- En tercer lugar, los modelos abiertos auto hospedados (como los que se ejecutan con Ollama o similares), que se pueden descargar en servidores locales y evitan depender de proveedores externos, no obstante, requieren una infraestructura potente para obtener tiempos de respuesta aceptables.

Comparando estas alternativas, la elección depende de las características del proyecto, los modelos abiertos ofrecen mayor control y privacidad, sin embargo, su rendimiento está limitado por la memoria y el procesamiento, mientras que los modelos comerciales y en la nube pueden ser más apropiados por su accesibilidad y escalabilidad cuando los recursos del sistema son limitados.

El caso de OpenAI como referente práctico

En este estudio se usa como ejemplo de implementación a OpenAI por su madurez tecnológica, amplia documentación y facilidad de uso mediante APIs. De acuerdo con su documentación oficial (Zhang & Shao, 2024), la API proporciona varios componentes que es útil definir:

- **Tokens**: Esta es la unidad de coste y de contexto. Se puede clasificar en función del tipo de tokens que se utilicen (por ejemplo: prompt tokens, completion tokens, total tokens). Es relevante verificar el uso del contexto en cada solicitud para controlar los costos al usar los modelos de la API

- **Modelos**: Los modelos son las diferentes versiones con características específicas de agentes con Inteligencia Artificial que proporciona el proveedor, en este caso OpenAI. Cada modelo se diferencia en funcionamiento, cómo fueron entrenados, qué pueden generar, cuánto cuestan por tokens, etc. Por ejemplo, hay: GPT-4o, GPT-4.1, serie "mini/nano" de bajo costo. Se eligen en función de:
  - Tamaño de ventana de contexto (tokens).
  - Calidad de razonamiento vs costo/latencia.
  - Soporte de herramientas (function calling, file search, structured outputs).
- **Assistants:** Son agentes parametrizables dentro de un modelo, el usuario puede especificar sus instrucciones (system prompt), las herramientas y políticas de ejecución. Se define un rol y un propósito que se mantiene en el tiempo, por ejemplo, un "Asistente de QA" que crea y mejora artefactos.
- **Threads:** Son hilos de conversación que conservan el contexto y el historial de mensajes. Permiten dar continuidad a las tareas, trazabilidad como iterar un conjunto de casos de prueba, otro uso sería crear un hilo por característica o épica. Estos se pueden reiniciar o truncar cuando el historial crece para no generar costos.
- **Messages:** Estos son los mensajes que el usuario, el asistente o el sistema envían en un hilo, los que nos permiten conversar o interactuar con el modelo de IA, en los que proporcionamos nuestras solicitudes, adjuntamos contexto y recibimos respuestas estructuradas.
- **Runs:** Son las ejecuciones de un Assistant sobre un hilo en un momento dado. Lo que posibilitan es administrar el ciclo de vida de herramientas y procesos internos. Permiten iniciar la generación y obtener métricas como estado, tokens usados, pasos.

Es importante mencionar que lo que se cubre aquí es solo una porción de las funciones que se exponen en la documentación de Open AI, en particular las interfaces de Assistants, Threads, Messages, Runs, las cuales son suficientes para el desarrollo práctico de la solución planteada en este trabajo de investigación. Además de estas existen otras herramientas de Open AI que pueden ser de utilidad en otros contextos, tales como: Chat Completions, Responses API, Webhooks, Vector Stores, Containers, Realtime, entre muchas otras que puedes consultar en la documentación oficial: https://platform.openai.com/docs/api-reference/introduction.

Para estimar precios, la siguiente tabla resume como ejemplo seis modelos en tres rangos de precios, tomando como referencia la política de precios por tokens para modelos serie 4.1 y 5 de GPT a la fecha del 25 de septiembre de 2025.

Tabla 3. Precios referenciales de diferentes modelos de IA de OpenAI

Modelo	Entrada (USD/1M tokens)	Entrada en caché	Salida (USD/1M tokens)
GPT-5	1.25	0.125	10.00
gpt-realtime (texto)	4.00	0.40	16.00
GPT-5-mini	0.25	0.025	2.00
GPT-4o-mini	0.60	0.30	2.40
GPT-4.1-nano	0.10	0.025	0.40

Nota. Datos obtenidos de la página web oficial de precios de Open AI en la fecha: 25 de septiembre de 2025. https://openai.com/api/pricing/

Los precios actuales pueden cambiar, se debe verificar en la página oficial para saber su valor actual. Para una correcta decisión sobre cuál es el modelo más conveniente para un proyecto se debe evaluar las necesidades de modelos de IA, calidad y latencia de las respuestas pedidas, métodos de interacción, tipo de contenido a generar, presupuesto disponible entre otros factores que puedan ser relevantes a un proyecto.

2.5.3. Estado del arte: propuestas recientes en AI Testing.

Introducción conceptual al AI Testing

La aplicación de la IA en las pruebas de software emerge como una respuesta ante las limitaciones de la automatización clásica (Qazi et al., 2022). En su forma más práctica, el AI Testing usa machine learning, modelos predictivos y generación automática de artefactos para ayudar en las tareas que antes requerían codificación manual de scripts. Este cambio se debe a la creciente complejidad de los sistemas y la necesidad de entregar funcionalidades en plazos cada vez más ajustados, lo que impulsa a las empresas a buscar maneras más rápidas y eficientes de validación.

La principal diferencia entre la automatización tradicional y el testing con IA es la adaptabilidad (Khankhoje, 2024). Mientras que la primera está ligada a scripts predefinidos que deben actualizarse ante cualquier cambio en la interfaz o en la lógica de negocio, la segunda aprovecha la capacidad de los algoritmos para aprender patrones,

adaptarse a los cambios y generar nuevos casos en tiempos cortos. Contextualizado al ámbito del software, significa que un cambio en un servicio o en una interfaz ya no rompería todos los scripts, sino que la IA puede proponer cambios o incluso regenerar las pruebas afectadas con poca intervención humana.

En el estado del arte actual, se pueden identificar tres enfoques principales de uso de IA en el testing (Qazi et al., 2022):

- Lo primero es la creación automática de casos de prueba a partir de contratos como OpenAPI o descripciones en lenguaje natural para ampliar la cobertura y minimizar omisiones.
- El segundo es la predicción de fallos, donde algoritmos entrenados con datos históricos de proyectos pueden predecir las partes del sistema en las que más esfuerzo se necesita en la validación.
- El tercero es la optimización de pruebas de regresión, donde la IA determina qué conjuntos de pruebas ejecutar primero según el riesgo, los cambios recientes y las métricas de cobertura, lo que reduce el tiempo de ejecución sin sacrificar la detección de errores.

En entornos ágiles, donde las entregas son iterativas y los requisitos cambian constantemente, el testing potenciado por la IA permite a los equipos de aseguramiento de calidad mantenerse al día, liberando a los testers de tareas repetitivas para que puedan concentrarse en pruebas exploratorias o escenarios complejos que exigen razonamiento humano. Así, la IA se convierte en un aliado metodológico para acelerar la entrega continua de software confiable.

Herramientas AI para pruebas funcionales y de calidad

La creación de herramientas potenciadas por la IA para la automatización de pruebas es uno de los mayores avances concretos en la práctica de aseguramiento de calidad (Job, 2021). Estas herramientas vienen a resolver una necesidad latente: la generación y mantenimiento de pruebas funcionales, las cuales son costosas en tiempo y esfuerzo cuando se hacen manualmente o con frameworks de automatización tradicionales.

Ahora, se explicarán algunas de las herramientas actuales que se construyen integrando la IA (Job, 2021).

- **Testim**, es una plataforma de IA que acelera la creación y el mantenimiento de pruebas funcionales. Su valor diferencial es poder grabar interacciones de usuario y convertirlas en pruebas repetibles y que se actualizan automáticamente cuando cambia la interfaz. En escenarios reales, si un campo de un formulario cambia su ID, Testim puede reconocer el patrón y actualizar el script, en lugar de que el tester necesite modificar el caso de nuevo.
- Por su parte, **Functionize** es una solución en la nube para pruebas funcionales, de rendimiento y de carga. A diferencia de las herramientas tradicionales, que exigen la configuración de entornos locales y la gestión de dependencias, Functionize utiliza su infraestructura en la nube para ampliar la ejecución de pruebas y recopilar métricas de forma centralizada. Un equipo puede simular cientos de usuarios simultáneos en una aplicación sin tener que enfocar su esfuerzo en configurar servidores de prueba.
- En las interfaces gráficas, **Applitools** hace una propuesta enfocada en la IA visual, donde las validaciones no se basan en valores o textos, sino en la apariencia de las pantallas. Esto es bueno para los casos en los que la experiencia del usuario es importante porque la herramienta detecta diferencias visuales en botones, menús o diseños en distintos navegadores y dispositivos. Con esto se asegura que el sistema no solo funcione, sino que lo haga de manera consistente en diseño y apariencia.

Cuando se miran estas iniciativas juntas, se ve cómo la IA puede acortar los pasos entre el diseño de casos de prueba y su ejecución en algún framework/plataforma y cómo se puede disminuir el mantenimiento que implica trabajar con frameworks de automatización convencionales. Pese a ello, también hay importantes limitaciones técnicas. Por un lado, estas herramientas están limitadas por la calidad de los datos históricos en los que se basan para funcionar, y si el entrenamiento no es representativo, su rendimiento se ve afectado.

Caso de estudio en la industria del software

En el estudio de (Qazi et al., 2022) se exploró el efecto de las herramientas de IA en **Software Houses en Pakistán**, mostrando cómo las tecnologías emergentes están comenzando a cambiar las prácticas de aseguramiento de calidad. El estudio encontró que las organizaciones que adoptaron la IA en sus procesos de prueba aceleraron la

identificación de errores y mejoraron la eficiencia en los plazos de entrega, creando valor inmediato en proyectos con alta urgencia de resultados.

En el estudio se detalla que las empresas estudiadas utilizaron modelos de aprendizaje automático e IA para ciertas tareas de SQA. Entre sus usos se han llegado a incluir técnicas de clasificación y regresión para predecir módulos defectuosos, redes neuronales convolucionales (CNN) para analizar interfaces y métodos basados en visión por computador para verificar la consistencia visual en regresiones.

En los mismos resultados (Qazi et al., 2022) se descubrió que la implementación de IA no estuvo exenta de obstáculos. Restricciones de presupuesto, falta de personal capacitado y condiciones socioeconómicas de la región impidieron su apropiación sostenible. Este escenario confirma que este tipo de herramientas con IA, necesitan un ecosistema organizacional preparado para adoptarlas, lo que implica invertir no solo en licencias o infraestructura, sino en personal capacitado para entender y supervisar lo que entregan los sistemas inteligentes.

El informe también señala que las empresas que sí superaron estos obstáculos obtuvieron beneficios adicionales: disminución de los costes de mantenimiento, mayor estabilidad en la ejecución de regresiones y mejor sincronización entre desarrollo y QA. Porque la IA permitió la trazabilidad entre requisitos y pruebas, reduciendo las brechas de comunicación entre los roles de un equipo ágil.

Como lección, los casos de la industria demuestran que la adopción de IA en testing no es una cuestión de tecnología, sino de madurez organizacional y predisposición al cambio. Por lo tanto, los beneficios en productividad y calidad solo se logran cuando se tiene una estrategia que alinee personas, procesos y tecnología hacia una meta compartida de mejora continua.

CAPITULO III

3. METODOLOGÍA

3.1. Tipo de Investigación

Para fines de desarrollo de la documentación de esta tesis se aplicará el tipo de metodología conocido como investigación aplicada (Hernández-Sampieri et al., 2016), esta será estructurada mediante fases. El objetivo que persigue esta metodología es recopilar el cúmulo de teoría y documentación técnica con el fin proponer una solución práctica a una problemática relevante al campo de estudio de la tesis.

Otro componente metodológico clave que se incorpora en este trabajo es el estudio mediante entrevistas y encuestas con expertos (Hernández-Sampieri et al., 2016). Estos permiten una visión actual y relevante del estado actual del campo de investigación, así como también pueden dar una visión crítica que permita validar la solución implementada.

Este tipo de investigación se acompaña por una metodología de desarrollo de proyectos ágil. Esto permite que el desarrollo sea por periodos cortos llamados sprints (López-Rodríguez & García-Peña, 2021). Durante estos sprints se trabajará de forma ordenada las diferentes historias de usuario que recopilan los requerimientos tanto funcionales como no funcionales de la solución propuesta. El marco ágil concreto a aplicar es Scrum, por lo que se facilitará el trabajo gracias a ceremonias individuales como:

- Sprint Planning: Se planificarán las historias a trabajar en cada sprint para realizar un desarrollo incremental basado en las historias de usuario
- Releases: El release será una forma de completar un milestone o conjunto importante de cambios como sería el completar el Backend, el Frontend o la conexión entre ambos componentes.
- Sprint Retroscpective: A pesar de no ser una ceremonia formal ya que es un desarrollo individual, la retroalimentación constante se incorpora como una forma de autoevaluación crítica que permite determinar qué se hizo bien, qué se hizo mal y qué se debe continuar haciendo.

Además de estas prácticas el diseño e implementación contempla un desarrollo mediante prototipado rápido que tiene como objetivo final la realización de un MVP’s (Producto Mínimo Viable). Estos prototipos permiten ir realizando desarrollos incrementales sobre los

cuales se puede mejorar el desarrollo incorporando nuevas funcionalidades que vayan haciendo crecer de forma incremental el software, hasta cumplir con todas las funcionalidades esperadas. Por su parte el MVP priorizaría funciones y permitiría una primera versión mínima totalmente funcional de la solución planteada como ejemplo del alcance que se podría llegar a tener si se continua mejorando el producto para casos de uso reales en empresas.

3.2. Diseño Metodológico

Esta tesis será realizada mediante diferentes entregables que componen las partes de este documento como son:

- **Marco Teórico:** Es la recopilación documental y teórica que permite tener presente de forma estructurada la información clave para tomar decisiones técnicas con respecto al diseño, construcción e implementación práctica. En esta fase se definen conceptos clave, se listan las teorías más importantes y se contextualiza con los ejemplos más recientes y el estado del arte del campo de investigación.
- **Metodología:** Aquí se declara cuál es la forma técnica en la cual se va a desarrollar de manera estructurada tanto la parte documental como la ejecución práctica de este trabajo. Se explican las metodologías de investigación y de desarrollo a usar de acuerdo con las prácticas actuales.
- **Análisis y Diseño:** Es la primera fase de la implementación práctica en donde se diagrama y planifica todos los elementos a desarrollar. Así mismo se analiza cuáles son los antecedentes, las necesidades encontradas y cómo se planea resolver la pregunta de investigación. Al mismo tiempo, se esboza cómo debe lucir el proyecto, cuál es la arquitectura y los diagramas clave para entender el funcionamiento del sistema.
- **Implementación:** En esta fase se describe el sistema realizado, se implementa en diferentes dispositivos y se despliega para que pueda ser usado por los diferentes usuarios finales, así mismo se propone cómo se realizaría el mantenimiento de la solución tecnológica.
- **Evaluación:** Una vez implementado el sistema puede ser probado primero mediante pruebas funcionales como no funcionales para verificar el funcionamiento básico.

Luego se puede validar con la ayuda de usuarios expertos que pueden aportar su visión crítica sobre el valor de la solución tecnológica construida.

3.3. Tecnologías y Herramientas a Utilizar

La realización de este trabajo en su totalidad requirió de los siguientes componentes:

- Bibliografía especializada y técnica para documentar los conceptos y guiar las decisiones prácticas.
- Laptops para el desarrollo del código del proyecto, despliegue y pruebas.
- Lenguajes de programación como Typescript y Shell, frameworks como Nest y React con Vite y librerías open source para la realización del código del proyecto, versionamiento y despliegue.
- Repositorio gratuito de almacenamiento de código: GitHub.
- Despliegue en servidor remoto de pago para almacenar una API para pruebas del proyecto: Render.
- Herramienta de diseño y maquetado de las interfaces web y pantallas del proyecto: Figma.
- Hub de publicación de librerías en el entorno de Javascript como despliegue para entregar la solución propuesta: NPM.
- Técnicas de recolección de datos cualitativas mediante encuestas y entrevistas a expertos en materia de Calidad de Software.

3.4. Criterios de Validación

Se propone el siguiente conjunto de criterios como forma de validar el éxito de la solución propuesta, así como para ayudar a responder la pregunta de investigación detallada en el Capitulo 1.

- **Usabilidad del sistema**

  Métrica aplicada: Encuesta basada en las 10 heurísticas de Nielsen.

  Descripción: Se evaluarán dimensiones de usabilidad como la visibilidad del estado del sistema, consistencia visual, control del usuario, eficiencia, flexibilidad y prevención de errores, mediante una escala Likert de 1 a 5.

- **Reducción en el tiempo de diseño y ejecución de pruebas**

  Métrica aplicada: Pregunta de desempeño orientada a medir la reducción del tiempo de generación y ejecución de pruebas.

  Descripción: Los participantes seleccionarán rangos porcentuales de ahorro percibido (0–10 %, 11–30 %, 31–50 %, 51–70 % y más del 70 %).

- **Percepción sobre calidad y cobertura de pruebas**

  Métrica aplicada: Pregunta de desempeño enfocada en la mejora percibida de calidad, considerando cobertura, detección de errores y consistencia.

  Descripción: Los encuestados evaluaron la magnitud del incremento percibido en la calidad del proceso de pruebas.

- **Percepción cualitativa de expertos en QA**

  Métrica aplicada: Retroalimentación abierta incluida en la encuesta.

  Descripción: Se recopilaron comentarios de profesionales con amplia experiencia en testing funcional y automatización.

CAPITULO IV

4. DISEÑO

4.1. Análisis Preliminar

4.1.1. Panorama Actual de la Automatización de Pruebas

Como se describió en el Capítulo 2, en el marco teórico, la automatización de pruebas como herramienta clave en los procesos de calidad de software, actualmente está experimentando cambios importantes debido a la introducción del uso de la Inteligencia Artificial en el año 2025. Estos avances han llevado a los equipos a progresar en diferentes ritmos.

Aquellos equipos de QA que cuentan con recursos disponibles para pagar licencias de grado empresarial para proteger los datos privados de la empresa ya usan modelos avanzados de IA para integrarlos en sus flujos de prueba. Sin embargo, otros equipos en entornos medianos o pequeños que no cuentan con suficientes recursos económicos tienen como opción el uso de modelos gratuitos o de bajo coste de IA, aunque con ellos se sacrifica privacidad y confidencialidad de los datos de un proyectos.

Por otro lado existe la limitante de conocimientos tecnológicos necesarios para implementar eficientemente nuevas herramientas de automatización y de IA, razón por la cual existen equipos que aun en la actualidad hacen sus pruebas de software de forma manual sin scripts de prueba.

Bajo este escenario y para obtener una visión crítica de las necesidades reales, se entrevistó a 5 expertos en calidad de software que actualmente trabajan en el sector y automatizan sus pruebas con diferentes herramientas y ya han empezado a probar fases iniciales de herramientas de IA. En la siguiente sección se describen

4.1.2. Necesidades Identificadas

Para tener una visión realista se entrevistó a 5 expertos en Calidad de Software de Latinoamérica a los cuales se les hizo una pregunta en común:

¿Cuáles considera que son las necesidades actuales en términos de herramientas que ayuden a optimizar tiempo utilizando Inteligencia Artificial en los procesos de Calidad de Software?

Respuesta de los expertos:

- **Valeria Muner – QA Lead, Argentina (17 años de experiencia)**
  "Lo que más nos quita tiempo en proyectos grandes es el llevar los requerimientos a pruebas, se va mucho tiempo y recursos. Tengo mis dudas del alcance de la IA, pero si pudiera ayudarnos a generar esos casos ejecutables desde el inicio, ganaríamos un buen tiempo."
- **Betina Rodriguez – QA Manual, Uruguay (8 años de experiencia)**
  "En mi día a día lo que más me pesa es la repetición: escribir casos a mano, mantenerlos al día cuando cambian las historias, actualizar los tickets en Jira. Para mi sería de gran ayuda algo que nos permitiera organizar todos esos artefactos de manera más visual e intuitiva. No me refiero solo a generar pruebas, sino a tener un espacio donde pueda ver qué está aprobado, qué está pendiente, qué se ejecutó y qué falló."
- **Joel Torres – Test Automation Engineer, Ecuador (6 años de experiencia)**
  "En automatización lo que más demora, es hacer el código de las pruebas. Ahí veo una oportunidad para la IA, que lea una historia de Jira y me de directamente un set de casos ejecutables, incluso con datos de prueba sugeridos. Incluso se podría integrar con el pipeline de CI/CD para revisar la calidad del código, en lugar de estar repitiendo lo mismo cada sprint."
- **Cesar Tipan – QA Mobile Engineer, Ecuador (5 años de experiencia)**
  "En mobile el reto no siempre es escribir las pruebas, sino lo que viene después, manejar la enorme cantidad de incidentes cuando algo falla en ciertos dispositivos o versiones. Ahí es donde una herramienta con IA podría marcar la diferencia, podría quizás automatizar la generación de los reportes de incidentes después de una ejecución. Si la misma plataforma detecta patrones en los errores y genera reportes claros para el equipo de desarrollo, sería muy interesante."
- **Christopher Ochoa – Test Automation Engineer, Ecuador (5 años de experiencia)**
  "Yo lo que veo es que cada equipo usa distintas herramientas y al final la información queda dispersa. Me gustaría muchas veces una sola herramienta que permita administrar todo el flujo: versionamiento, ejecución, resultados,

> defectos y métricas. Todo en un mismo tablero. Tener todo esa información a la mano puede ser útil para cuando nos reunimos el equipo de QA para evaluar aspectos de diferentes features."

Lo que podemos analizar como patrón en común de las respuestas de los 5 expertos es que se necesita una herramienta que centralice las funciones del testing ya que como es común los diferentes equipos trabajan con diferentes herramientas, Frameworks y entornos que muchas veces pueden generar desgaste al tener que transferir la información de una plataforma a otra.

También podemos destacar funcionalidades clave que pueden enriquecer el diseño de una solución de calidad de software de tipo agente inteligente:

- Traducir requisitos en lenguaje natural a casos de prueba ejecutables o scripts dentro de un framework de automatización.
- Soporte visual de todos los artefactos del ciclo de vida del testing en una plataforma única como son: Casos de Prueba, Suites, Bugs o Ejecuciones.
- Reporte automático de defectos al ejecutar casos de prueba en caso de que algo falle durante el proceso.

Todos estos hallazgos serán tomados en cuenta para el diseño de la solución propuesta para esta tesis.

4.1.3. Restricciones Detectadas

Antes de empezar con el desarrollo formal, se realizaron pruebas exploratorias para ver las opciones prácticas de integrar modelos de inteligencia artificial en el flujo de trabajo de una herramienta para la calidad de software, los hallazgos permitieron encontrar los delimitantes del proyecto.

Limitaciones de Hardware

Este proyecto al ser netamente de desarrollo de software no requiere de otros instrumentos o materiales físicos más que dispositivos de tipo laptop o computadora, tanto para desarrollar como probar el sistema propuesto.

Sin embargo las características en cuanto a recursos de los dispositivos a utilizar si son relevantes a la hora de considerar el uso de modelos de inteligencia artificial. Como fase

exploratoria se probaron diferentes modelos de inteligencia artificial, tanto gratuitos como de pago mediante API's.

De esas pruebas exploratorias se pudieron encontrar restricciones de carácter de recursos ya que para usar modelos gratuitos de IA como son la serie de modelos de Ollama se necesita como mínimo para los modelos más livianos como **codegemma 2b**, 16GB de RAM y 256GB de almacenamiento libre. Probando con un laptop Dell de estas características, el modelo local gratuito de ollama demoraba alrededor de 5 min en responder una request simple para transformar requisitos en lenguaje natural a código, eso sumado al tiempo de descarga del modelo que pesaba alrededor de 5GB, lo cual con una red local de internet para uso domestico de 500mbps, demoró una hora en la descarga.

Por tanto, se descartó la posibilidad de usar modelos gratuitos locales debido a que requieren de bastantes recursos de hardware para tener tiempos de respuesta óptimos, así como asegurar la calidad de las respuestas. Un sistema que demore 5 min en responder una petición simple no es viable.

Luego de explorar modelos locales se probaron modelos de IA que ofrecen sus servicios mediante API's. El proveedor mejor documentado y que ofrecía mayor velocidad, control e incluso fiabilidad en los modelos más baratos fue Open AI. Por lo que se procedió a explorar los diferentes modelos que difieren en precio lo cual da paso a explorar la siguiente limitante.

Restricciones Económicas

Al ser una herramienta pensaba en principio como un MVP para aplicaciones pequeñas en API's, se debe considerar que el presupuesto típicamente será limitado para el usuario final. Así mismo, para el desarrollo de este proyecto se contaba con un presupuesto definido y limitado de $500 por lo que había que optar por un balance calidad precio.

Como se describió en la subsección **"2.5.2. Uso de modelos de Inteligencia Artificial en la actualidad"** del Marco Teórico, de los diferentes modelos que provee la API de Open AI uno de los más económicos es a fecha de Septiembre de 2025 el **GPT-4.1-nano**. Se compararon sus respuestas con los modelos de la serie GPT-4 y GPT-5 y los resultados fueron similares en términos de calidad, sin embargo esos modelos cuestan

mucho más por request, entonces para fines del MVP se determinó que el modelo óptimo sería el GPT-4.1-nano por su relación calidad precio.

4.1.4. Factibilidad

Reconociendo las limitantes mencionadas se procede a determinar que la viabilidad del proyecto es positiva ya se cuenta con opciones viables tanto para la parte de hardware, software y la parte económica.

Para la parte de factibilidad tecnológica se utilizarán herramientas de desarrollo gratuitas o de pago que están al alcance del presupuesto como son:

- **Hardware**: 2 Laptops personales con las cuales ya se contaba para la realización del proyecto por lo que no se considera un costo adicional.
- **IDE**: Como entorno de desarrollo para la realización del código del proyecto se utiliza **Cursor**, una potente herramienta actual que permite integrar modelos de Inteligencia Artificial al desarrollo ágil de software, el cual fue usado de forma responsable y supervisada para garantizar la calidad y el funcionamiento del software.
- **Lenguaje de Programación**: TypeScript, lenguaje que viene con librerías gratuitas para el desarrollo de Frontend con React y Vite y para el backend el framework Nest del mismo entorno.
- **Base de Datos**: Ya que esta es una herramienta pensaba para el uso local y personal, la portabilidad y velocidad es importante por eso se eligió la base de datos gratuita SQLite.
- **Despliegue**: La solución se despliega como librería en el entorno gratuito de librerías del ecosistema de Javascript NPM.
- **Versionamiento de Código**: Se utiliza GitHub como repositorio gratuito para alojar las diferentes versiones del código.
- **Servidores**: Se usa Render como hosting de pago una API de prueba que se usa para realizar verificaciones de funcionamiento del sistema propuesto.
- **Servicio de Inteligencia Artificial**: Como se describió anteriormente se utilizará como proveedor mediante API en la aplicación el servicio de Open AI.

Al tener claro cuáles son las vías tecnológicas a utilizar podemos evaluar la factibilidad económica del proyecto mediante la siguiente tabla que resume los costos que implica la realización de este proyecto.

Tabla 4. Recursos de hardware y software para el proyecto

Cantidad	Descripción	Tipo	Valor Total
2 unidades	Laptops para realizar el desarrollo y pruebas (Dell Latitude 7430 y Acer Aspire 3)	Hardware	-
6 meses	Internet de Banda Ancha Claro	Hardware	$126
6 meses	Licencia Personal IDE Cursor	Software	$120
6 meses	Servidor Cloud Render para API de Pruebas	Software	$30
1 unidad	Servicio de Open AI mediante el ingreso de Saldo prepagado	Software	$30
	Total		$366

Nota. Elaborado por el autor.

Dado que el total de gastos se encuentra dentro del presupuesto personal para el proyecto se puede decir que esta implementación cumple con la factibilidad económica.

4.2. Descripción del Sistema Propuesto

4.2.1. Generalidades

El sistema se implementará como una plataforma web full-stack para gestionar y automatizar pruebas funcionales de APIs REST. El backend se construirá en NestJS con arquitectura modular y persistencia en SQLite (entidades y migraciones SQL), exponiendo las rutas bajo el prefijo /v1/api. El frontend se desarrollará con React 18 + TypeScript + Vite + Tailwind, organizado por vistas, componentes por dominio, servicios HTTP y hooks. Se integrará IA de OpenAI para generar y sugerir casos de prueba en formato BDD, y durante las ejecuciones se publicará progreso en tiempo real.

Descripción funcional por módulos

1) Dashboard y menú Lateral

Propósito: Se trata de un punto de acceso de la aplicación como página de bienvenida donde se pueden ver métricas generales como cantidad de endpoints, proyectos, tasa de éxitos de casos de prueba entre otros que pueden añadirse de acuerdo con la conveniencia. También habrá un menú lateral para el acceso fácil a cada uno de los módulos mencionados anteriormente.

2) Projects (Proyectos)

Propósito: El sistema organizará el trabajo por iniciativas de prueba.

Qué permitirá:

- Crear proyectos con sus metadatos visibles y prepararlos como raíz del resto de componentes.
- Listar, buscar y ver el detalle de cada proyecto con indicadores clave (KPIs, Key Performance Indicators; indicadores de desempeño).
- Actualizar información del proyecto y eliminarlo de forma controlada.
- Acceder rápidamente a vistas relacionadas (endpoints, casos, suites, ejecuciones, bugs) y a acciones como “Run tests”.

Entradas/Salidas esperadas: Datos descriptivos del proyecto / confirmación y vista resumida con KPIs.

Relaciones: Punto de entrada para Endpoints, Test Cases, Test Suites, Test Executions, Bugs y AI Assistant.

Restricciones visibles: Se garantizará la unicidad por nombre y se bloqueará la eliminación si hay recursos en uso.

3) Endpoints (Catálogo de APIs)

Propósito: El sistema mantendrá un inventario de endpoints a probar, organizados por proyecto, sección y entidad.

Qué permitirá:

- Listar y filtrar endpoints (por texto, proyecto, método y estado).
- Registrar endpoints con su ruta, métodos y descripciones funcionales.
- Editar y eliminar endpoints con control de impacto en pruebas vinculadas.
- Consultar un detalle integral con pestañas de información, métodos y artefactos asociados.

Entradas/Salidas esperadas: Definiciones funcionales del endpoint / listado, detalle y artefactos vinculados.

Relaciones: Se conectará directamente con Test Cases.

Restricciones visibles: Validaciones según el tipo de datos y método definidos.

4) Test Cases (Casos de Prueba)

Propósito: El sistema permitirá definir pruebas funcionales como escenarios con pasos claros en formato BDD (*Behavior-Driven Development*).

Qué permitirá:

- Listar y filtrar casos de prueba por criterios como proyecto, sección, entidad, método o estado.
- Crear casos de forma manual mediante un editor BDD.
- Generar casos de manera asistida con IA, a partir de descripciones funcionales o endpoints registrados.
- Sugerir casos de manera asistida con IA, a partir de descripciones funcionales o endpoints registrados.

- Editar escenarios, metadatos y pasos; duplicar o eliminar casos existentes.
- Ejecutar un caso de prueba directamente desde su vista.

Entradas/Salidas esperadas: Escenarios y pasos definidos / casos creados, actualizados o ejecutados.

Relaciones: Se vinculará con Endpoints y alimentará Test Executions; contará con apoyo del AI Assistant.

Restricciones: En generación por IA se exigirá formato estricto; los pasos se ordenarán sin duplicaciones.

5) Test Suites (Conjuntos de Pruebas)

Propósito: El sistema agrupará casos de prueba en conjuntos organizados según objetivos de validación.

Qué permitirá:

- Crear y editar suites de tipo test set (conjunto de casos de pruebas) o test plans (conjunto de test sets) seleccionando casos y/o suites existentes, con etiquetas y metadatos.
- Listar, filtrar y eliminar suites de manera controlada.
- Ejecutar una suite en un solo paso.

Entradas/Salidas esperadas: Conjunto de casos seleccionados / suite lista para ejecución y su identificador al correrla.

Relaciones: Se relacionará con Test Cases y alimentará Test Executions.

6) Test Executions (Ejecuciones de Pruebas)

Propósito: El sistema orquestará corridas de pruebas y mostrará resultados en tiempo real mediante SSE (*Server-Sent Events*).

Qué permitirá:

- Iniciar ejecuciones por proyecto o por suite específica.
- Monitorear el progreso en tiempo real con eventos de inicio, avance, finalización o error.

- Consultar el detalle de ejecución con escenarios, pasos, errores y métricas consolidadas.
- Listar y filtrar ejecuciones por estado, fechas o tipo de prueba. Obtener un resumen global de ejecuciones con métricas agregadas.

Entradas/Salidas esperadas: Parámetros de ejecución / resultados con métricas y estadísticas.

Relaciones: Consumirá Test Cases y Test Suites; sus fallos podrán generar Bugs.

7) Bugs (Defectos)

Propósito: El sistema registrará y permitirá dar seguimiento a los defectos detectados en las ejecuciones.

Qué permitirá:

- Crear, editar y eliminar incidencias, registrando su contexto funcional.
- Listar y filtrar bugs por proyecto, tipo, severidad, prioridad, estado, sección y entidad.
- Visualizar estadísticas por estado, severidad, tipo y prioridad.

Entradas/Salidas esperadas: Datos de un incidente / registros de detalle y evolución del bug.

Relaciones: Se conectará con Test Cases, Test Executions y Endpoints.

8) AI Assistant (Asistente con IA)

Propósito: El sistema acelerará la creación de pruebas al generar y sugerir casos con apoyo de inteligencia artificial.

Qué permitirá:

- Inicializar, verificar o eliminar el asistente de cada proyecto.
- Generar casos en bloques estructurados (escenarios y pasos) listos para revisión.
- Sugerir casos de prueba en formato estricto y mantener un historial por proyecto.

Entradas/Salidas esperadas: Descripciones funcionales / propuestas de casos listos para integrar.

Relaciones: Se vinculará con Projects, Endpoints y Test Cases.

Restricciones visibles: Requerirá tener configurada la API Key en Settings.

9) Sync / Workspace (Sincronización)

Propósito: El sistema mantendrá la coherencia entre lo registrado y los artefactos de prueba en el espacio de trabajo.

Qué permitirá:

- Ejecutar sincronización por proyecto o de manera masiva.
- Generar reportes con elementos sincronizados, errores y tiempos de procesamiento.
- Identificar desalineaciones y proponer correcciones.

Entradas/Salidas esperadas: Selección de proyectos / reportes de sincronización.

Relaciones: Transversal a Projects, Endpoints y Test Cases.

10) Settings (Configuración)

Propósito: El sistema habilitará la configuración de parámetros mínimos para operar la IA.

Qué permitirá:

- Guardar y probar la clave del proveedor de IA.
- Confirmar el estado de conexión antes de usar el asistente.

Entradas/Salidas esperadas: Credencial de IA / confirmación y estado de conexión.

Relaciones: Será requisito para el uso de AI Assistant.

Restricciones visibles: La clave se almacenará en el archivo de configuración del espacio de trabajo.

4.2.2. Requerimientos Funcionales

Con la descripción general provista anteriormente, podemos agruparlos en requerimientos funcionales que servirán para guiar el desarrollo y no dejar ninguna funcionalidad fuera de la programación.

Tabla 5. Requerimientos funcionales del sistema propuesto

Código de Requerimiento	Requerimiento
RF-01	Visualizar un tablero con KPIs como punto de entrada de la aplicación
RF-02	Gestionar proyectos de prueba (crear, listar, actualizar y eliminar) garantizando unicidad y acceso a módulos relacionados.
RF-03	Mantener un catálogo de endpoints por proyecto con registro, filtros y detalle funcional para su uso en casos de prueba.
RF-04	Diseñar casos de prueba funcionales (manuales y generados con IA en formato BDD), con edición y ejecución desde su vista.
RF-05	Agrupar casos en suites ejecutables, permitiendo su creación, edición, listado y lanzamiento.
RF-06	Orquestar ejecuciones de prueba y presentar progreso y resultados en tiempo real con métricas consolidadas.
RF-07	Registrar y gestionar defectos vinculados a ejecuciones y casos, con filtros y estadísticas de seguimiento.
RF-08	Inicializar y operar un asistente de IA por proyecto para generar y sugerir casos de prueba evitando duplicidades.
RF-09	Sincronizar artefactos de prueba con el espacio de trabajo y emitir un reporte de cambios, errores y tiempos.
RF-10	Configurar y verificar la clave del proveedor de IA para habilitar las funciones asistidas del sistema.

Nota. Elaborado por el autor.

4.2.3. Requerimientos No Funcionales

Los requerimientos no funcionales se tratan de aquellas características que el usuario final no usa directamente como si fuera una herramienta o botón, sin embargo ayudan a su experiencia de uso de distintas formas, desde el aspecto de usabilidad hasta temas de seguridad. Así mismo hay aspectos que ayudan al software a ser mantenible y robusto de manera que puede crecer en funcionalidades de forma segura.

Los requerimientos no funcionales más relevantes que se tendrán a consideración serán:

Tabla 6. Requerimientos no funcionales del sistema propuesto

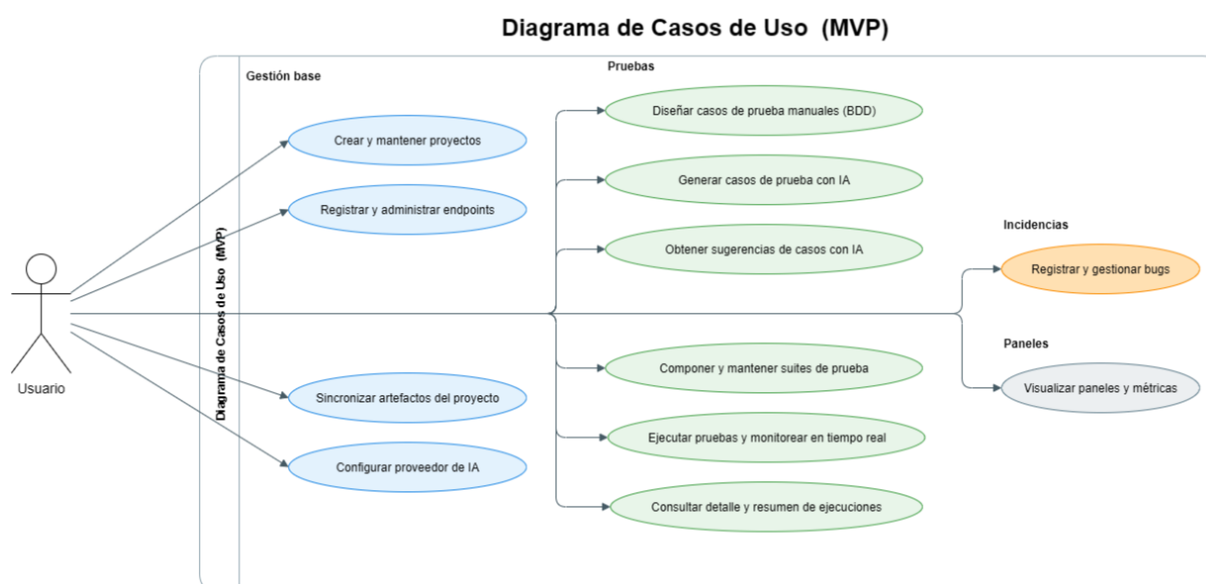
Código	Requerimiento No Funcional
RFN-01	Optimizar la eficiencia en el uso de la inteligencia artificial, limitando y reciclando el contexto para reducir consumo de *tokens* y registrando métricas de tiempo y uso real por ejecución.
RFN-02	Garantizar la observabilidad del sistema mediante transmisión en tiempo real de las ejecuciones con SSE, mostrando detalle consolidado de pasos, errores, métricas y KPIs filtrables.
RFN-03	Fortalecer la confiabilidad del sistema mediante manejo robusto de errores y estados transitorios, respondiendo con códigos adecuados en operaciones críticas y aplicando flujos tolerantes a fallos.
RFN-04	Asegurar la seguridad operativa del sistema gestionando las claves y secretos en entornos protegidos, sin exponerlos en la interfaz pública y validando su conexión de forma controlada.
RFN-05	Asegurar la mantenibilidad y escalabilidad del sistema mediante una arquitectura modular, organizada por dominios y con separación clara de responsabilidades en backend y frontend.
RFN-06	Garantizar la usabilidad y consistencia de la interfaz mediante componentes reutilizables, formularios con validaciones visibles, toasts de retroalimentación y estados de carga claros.

Nota. Elaborado por el autor.

4.2.4. Casos de Uso

A continuación se presenta el diagrama de casos de uso con estilo UML que permite visualizar de forma gráfica los requerimientos funcionales que presenta el sistema, considerando la perspectiva del usuario en el sistema. El desglose de los casos de uso detallados se encuentra en el ANEXO A.

Figura 6. Diagrama de Casos de Uso del sistema propuesto



Nota. Elaborado por el autor.

El **diagrama de casos de uso del sistema propuesto** representa la interacción principal del usuario con el sistema, organizado en cuatro grandes bloques funcionales: **Gestión base**, **Pruebas**, **Incidencias** y **Paneles**. El actor principal es el **usuario**, quien centraliza todas las acciones del flujo, desde la configuración inicial hasta la visualización de resultados. Este enfoque permite comprender de manera clara las funcionalidades esenciales del producto mínimo viable y la relación entre los procesos que lo componen.

En el apartado de **Gestión base**, el usuario podrá crear y mantener proyectos, registrar y administrar endpoints, sincronizar artefactos del proyecto y configurar el proveedor de inteligencia artificial. Estas funciones conforman la capa de administración técnica del sistema, asegurando que el entorno de pruebas esté correctamente configurado y conectado a los servicios necesarios para su ejecución.

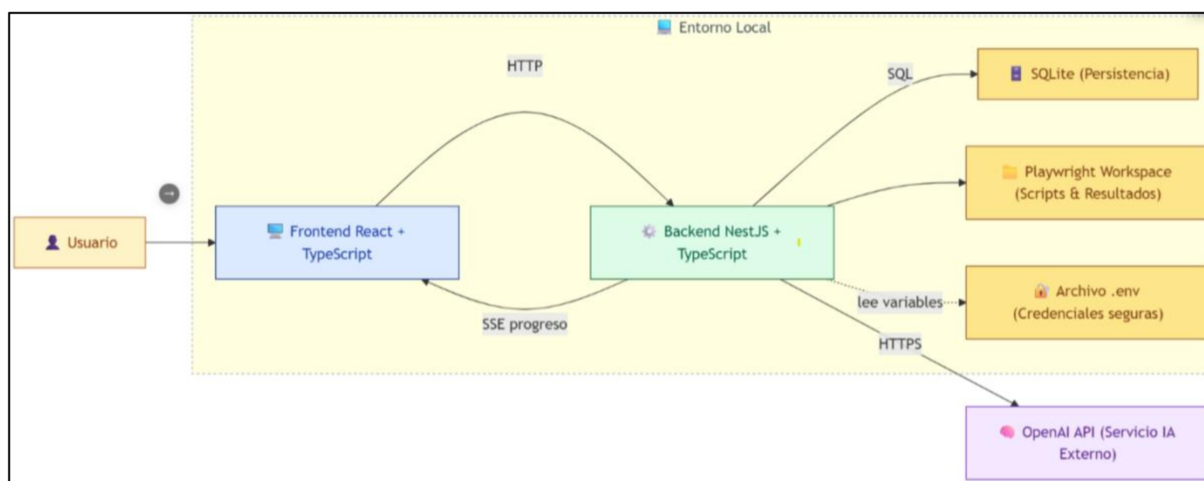
El bloque de **Pruebas** concentra las actividades centrales del sistema: diseñar casos de prueba manuales bajo el enfoque BDD, generar casos de prueba con IA y obtener sugerencias automatizadas. A esto se suman funciones operativas como componer suites, ejecutar pruebas y monitorear resultados en tiempo real, además de consultar resúmenes y métricas de ejecución. Finalmente, las secciones de **Incidencias** y **Paneles** complementan el ciclo de calidad, permitiendo registrar y gestionar bugs, así como visualizar paneles e indicadores que faciliten el análisis del desempeño del sistema y la trazabilidad de los resultados.

4.3. Diseño

4.3.1. Arquitectura

La solución propuesta será desarrollada como un MVP, estructurada bajo una arquitectura monolítica por capas con un modelo cliente–servidor local, orientado en su primera fase a un uso personal y de entorno controlado.

Figura 7. Diagrama C4 - Nivel Contenedor (Arquitectura general del MVP)



Nota. Elaborado por el autor.

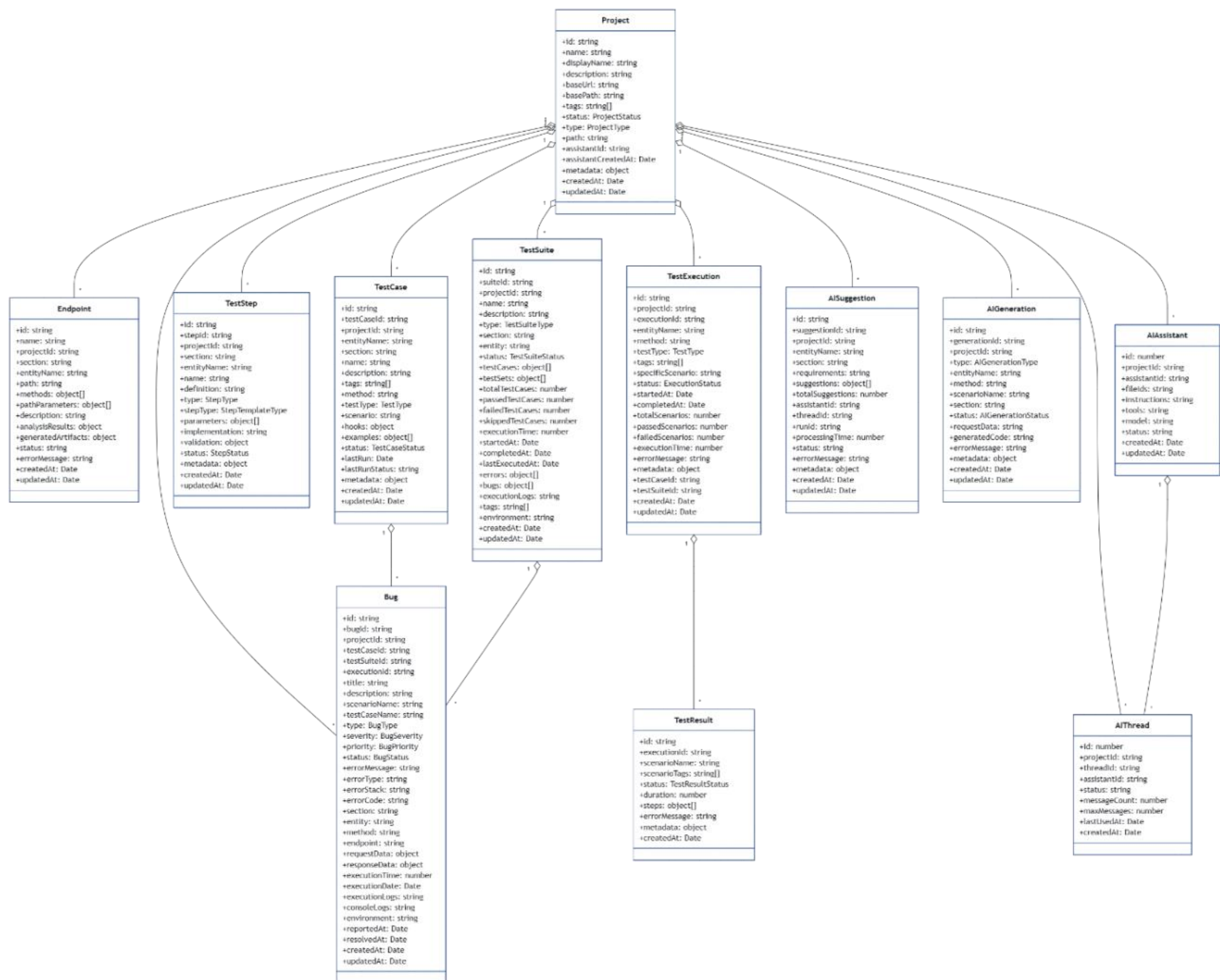
El sistema se compondrá de un frontend en React con TypeScript, encargado de ofrecer una interfaz intuitiva con un dashboard de métricas generales y navegación hacia los distintos módulos, y de un backend local implementado en NestJS con TypeScript, responsable del procesamiento lógico de las solicitudes. Ambos componentes se comunicarán mediante peticiones HTTP y un canal SSE (Server-Sent Events), el cual permitirá transmitir en tiempo real el progreso de las ejecuciones y la generación dinámica de resultados.

La persistencia de datos se gestionará con SQLite, complementada con un workspace local de archivos Playwright para el almacenamiento estructurado de scripts y resultados de prueba, junto con un archivo .env que mantendrá de forma segura las claves y credenciales necesarias. De manera controlada, el backend establecerá comunicación externa con el servicio de IA (OpenAI API) a través de solicitudes HTTPS, con el propósito de generar y sugerir casos de prueba automáticos, manteniendo la modularidad, trazabilidad y seguridad del sistema dentro de su entorno local.

4.3.2. Diagramas de Clases Simplificado

El diagrama de clases constituye una herramienta fundamental en el diseño del sistema, ya que permite representar de forma estructurada las entidades principales, sus atributos y las relaciones que existen entre ellas. Este modelo visual facilita comprender la organización interna del software, identificar dependencias entre módulos y anticipar cómo se comportarán los objetos durante la ejecución.

Figura 8. Diagrama de clases simplificado del sistema propuesto



Nota. Elaborado por el autor.

El diagrama presentado refleja una versión simplificada de las clases centrales del sistema y sus relaciones más relevantes. Se incluyen entidades como proyectos, endpoints, casos de prueba, ejecuciones, resultados, bugs y los componentes de inteligencia artificial, cada uno con sus atributos esenciales. También se muestran las cardinalidades que evidencian cómo un proyecto agrupa múltiples elementos relacionados, desde endpoints hasta ejecuciones y reportes de errores.

4.3.3. Diagramas de Secuencia

Ahora se procede a diseñar los diagramas de secuencia que consisten en representaciones de clase UML que permiten ilustrar cómo interactúan los diferentes componentes del sistema propuesto a lo largo del tiempo. Su utilidad radica en mostrar,

de manera ordenada y cronológica, el intercambio de mensajes entre actores, interfaces y módulos internos, lo que permite comprender con claridad el flujo de una funcionalidad específica, identificar dependencias y visualizar posibles puntos de fallo o mejora en la comunicación entre procesos.

Tabla 7. Resumen de los componentes principales de los diagramas de secuencia creados

RF	Nombre	Propósito principal	Principales Componentes involucrados
RF-01	Dashboard (KPIs)	Consolidar y visualizar indicadores clave del sistema mediante consultas agregadas a la base de datos.	Frontend Dashboard.tsx, ProjectsController, EndpointsController, GlobalTestExecutionController, SQLite.
RF-02	Proyectos (gestión CRUD con validación de unicidad)	Permitir la gestión de proyectos (crear, editar, listar y eliminar) asegurando nombres únicos.	ProjectsController, ProjectsService, SQLite.
RF-03	Endpoints por proyecto (registro, análisis y artefactos)	Gestionar endpoints asociados a un proyecto y generar artefactos automáticos (features, schemas, types, client).	ProjectEndpointsController, EndpointsService, AnalysisService, ArtifactsGenerationService, SQLite.
RF-04	Casos de prueba (manual BDD y ejecución desde la vista)	Facilitar la gestión de casos de prueba manuales en formato BDD y su ejecución directa desde el frontend.	ProjectTestCasesController, GlobalTestExecutionController, SQLite.
RF-05	Suites (gestión y ejecución de pruebas agrupadas)	Gestionar suites de prueba y ejecutar grupos de casos relacionados dentro de un proyecto.	TestSuitesController, GlobalTestExecutionController, SQLite.
RF-06	Ejecuciones con SSE (progreso y resultados)	Mostrar en tiempo real el progreso y resultados de las ejecuciones mediante Server-Sent Events.	GlobalTestExecutionController, TestExecutionService, SQLite, SSE /events.
RF-07	Bugs / Defectos (gestión y estadísticas)	Gestionar el registro y seguimiento de bugs detectados, generando KPIs por estado, tipo y severidad.	BugsController, BugsService, SQLite.
RF-08	Asistente de IA (sugerir y generar BDD)	Integrar IA generativa para sugerir y generar automáticamente casos de prueba en formato BDD.	AIController, TestCaseSuggestionService, , CodeManipulationService, OpenAI API, Workspace FS, SQLite.

RF-09	Sincronización con workspace	Sincronizar entidades locales (proyectos, endpoints, test cases) con el sistema de archivos del workspace.	SyncController, SyncService, SQLite, Workspace FS.
RF-10	Configuración y prueba de OpenAI API Key	Permitir registrar y validar la API Key del servicio de IA desde el panel de configuración.	AIGeneralController, OpenAIConfigService, Workspace .env.

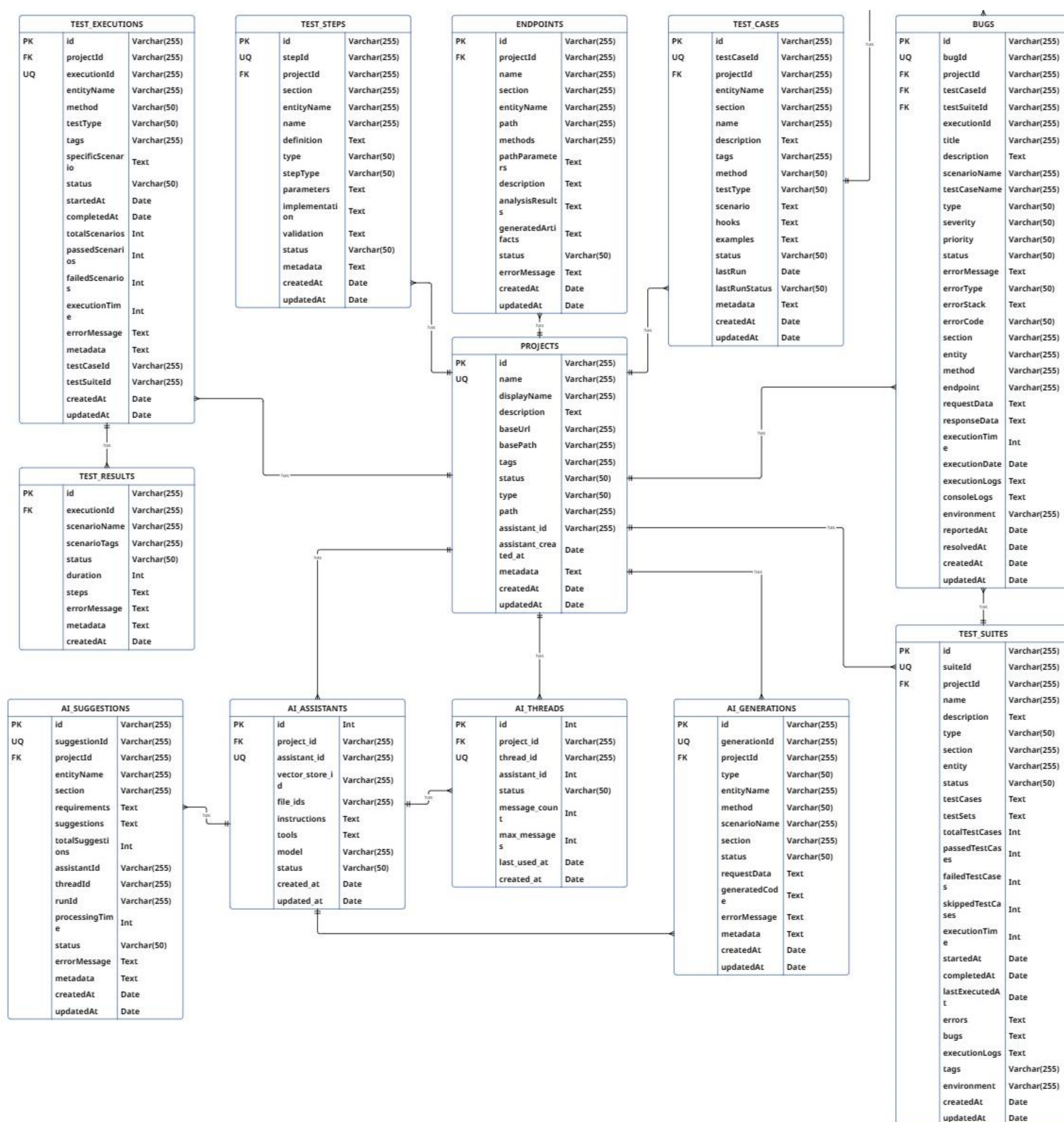
Nota. Elaborado por el autor.

Estos diagramas serán esenciales para el diseño del sistema, ya que permitirán definir con precisión los flujos de interacción entre módulos y los límites de responsabilidad de cada componente. Desde el punto de vista del frontend, los diagramas ayudarán a estructurar los servicios, hooks y elementos visuales en función de los endpoints disponibles. En el backend, servirán para definir controladores, servicios y operaciones en la base de datos, asegurando consistencia entre los flujos de negocio y los requerimientos. El desglose completo de los diagramas se presenta en el **ANEXO B**.

4.3.4. Diagrama Entidad Relación de la Base de Datos

El esquema de bases de datos constituye una representación estructurada de las entidades y relaciones que sostienen el sistema. A través de él se definen las tablas, atributos principales y vínculos lógicos, lo que garantiza consistencia en el almacenamiento de la información y facilita la trazabilidad entre proyectos, pruebas, ejecuciones y reportes.

Figura 9. Diagrama de entidad relación de la Base de Datos



Nota. Elaborado por el autor.

4.3.5. Diseño de Pantallas en Figma

El diseño de pantallas en Figma se propone como una forma de planificar el desarrollo frontend, mediante una diagramación clara de la interfaz de usuario propuesta para el sistema.

Mediante prototipos se representan las vistas principales y la forma en que se disponen los elementos de navegación y control, asegurando coherencia visual y usabilidad. En el siguiente enlace se puede encontrar los diseños en el portal de Figma: https://www.figma.com/design/JVjVc4NxskJLBFSHt1tmZQ/Omega-Testing?node-id=0-1&p=f . También se encuentran capturas de evidencia en el **ANEXO C**.

Tabla 8. Resumen de pantallas diseñadas en Figma

Pantalla	Propósito de diseño	Componentes clave principales
Dashboard	Reunirá indicadores globales y accesos rápidos para la administración.	Sidebar de navegación; tarjetas KPIs: Total Projects, Total Endpoints, Success Rate con barra, Passed Tests, Failed Tests, Average Execution Time
Projects	Permitirá gestionar proyectos y abrir configuraciones específicas.	Buscador de Projects; botón Refresh Data; botón New Project; Tarjetas de Projects con opciones, menú contextual y botón de Open Project en la tarjeta.
Endpoints	Centralizará la gestión de endpoints y sus artefactos.	Buscador; botón Refresh Data; botón Register Endpoint; tarjetas de endpoint, menú contextual y botón View Details.
Test Cases	Facilitará la gestión y ejecución de casos con soporte de IA.	Buscador; botón Refresh Data; botón Create Test Case; tarjetas de casos con estado, etiquetas, menú contextual y botón Run Test.
Test Suites	Organizará casos en suites para ejecución agrupada.	Buscador; botón Refresh Data; botón Create Test Suite; tarjetas de suites con contadores, menú contextual y botón Execute Suite.
Bugs	Registrará y dará seguimiento a defectos con métricas de estado.	Buscador; botón Refresh Data; botón Create Bug; tarjetas de bugs con prioridad, severidad, estado y menú contextual.
Test Executions	Mostrará resultados y métricas agregadas de ejecuciones.	Buscador; botón Refresh Data; tarjetas de ejecuciones con estado, métricas, duración y botón View Results.
OpenAI Configuration	Configurará y verificará la conexión con el proveedor de IA.	Campo API Key; botones Save Key, Test Connection y Refresh Status; alertas de validación y mensajes de conexión.

Nota. Elaborado por el autor.

4.4. Planificación de Releases por Sprints

4.4.1. Estrategia de Releases

El desarrollo del prototipo se articula en entregas funcionales (releases) que consolidan, en cada etapa, una capa técnica completa y verificable antes de avanzar a la siguiente. Esta organización favorece la validación temprana de la arquitectura, la detección oportuna de ajustes de diseño y un ritmo de avance controlado bajo prácticas ágiles.

La duración total estimada de trabajo en la solución práctica es de seis meses, dividido en sprints de diez días (12 sprints en total). La estimación del esfuerzo utiliza la serie de Fibonacci (1, 2, 3, 5 y 7 puntos como máximo por historia), lo que permite dimensionar la complejidad relativa y equilibrar la cantidad de trabajo de cada sprint. Al cierre de cada release se aplican criterios de completitud técnica: funcionalidad operativa, pruebas funcionales y documentación.

Tabla 9. Estrategia de Releases y su información clave

Release	Duración (meses)	Sprints	Objetivo general	Entregables clave
R1 – Capa de servicios del backend	2	4	Establecer API REST modular, entidades y persistencia local; validar contratos y reglas.	Endpoints CRUD por dominio (Projects, Endpoints, Test Cases, Test Suites, Test Executions, Bugs), DTOs y validaciones, respuestas/errores consistentes, colección Postman y guía de uso de API.
R2 – Interfaz frontend	2	4	Desarrollar SPA por dominios con datos simulados; asegurar navegación, estados y componentes reutilizables.	Vistas (Dashboard, Projects, Endpoints, Test Cases, Test Suites, Test Executions, Bugs, Settings), filtros y formularios, feedback de carga/errores, librería UI común y prototipos validados.
R3 – Integración frontend–backend	1	2	Conectar UI con API real y habilitar flujos E2E (crear, ejecutar, resultados, bugs) con SSE.	Servicios HTTP reales, SSE de progreso en ejecuciones, generación/sugerencias con IA operativas desde UI, pruebas E2E, estabilización y corrección de defectos priorizados.
R4 – Empaquetado y despliegue del MVP	1	2	Preparar arquitectura monolítica para distribución como paquete NPM.	Build y empaquetado, README/Quickstart, versionado inicial (SemVer), checklist de publicación, verificación de instalación local e instrucciones de integración.

Nota. Elaborado por el autor.

Esta distribución refleja el peso relativo de cada etapa. Cada release entrega valor demostrable y trazable, reduce incertidumbre técnica y deja el producto listo para el siguiente incremento hasta culminar en un MVP reutilizable y distribuible.

4.4.2. Historias de Usuario

La planificación se estructura en 12 sprints de 10 días alineados a los releases definidos. Dado que se trata de un proyecto unipersonal, las historias se redactan con roles desde el punto de vista de la tarea a realizar, es decir: Desarrollador para la generación de código, Tester para validar el funcionamiento y Administrador de la aplicación para tareas de gestión del proyecto. La verificación de calidad no aparece como historias independientes: queda reflejada en los criterios de aceptación y en la referencia explícita a los requerimientos funcionales (RF) y no funcionales (RFN) establecidos en las secciones 4.2.2 y 4.2.3. Todas las historias se estiman con la serie de Fibonacci.

A continuación se presenta la planificación de cada sprint por releases.

Release 1 — Backend y capa de servicios (S1–S4)

Tabla 10. Planificación del Sprint 1 — Base técnica y estructura

Historia de Usuario	Descripción	Puntos	Criterios de Aceptación
HU-01	Como desarrollador, quiero inicializar arquitectura modular Nest y preparar dominios base y configuración común para disponer de una columna vertebral mantenible y extensible.	5	Dado un repo vacío, cuando creo la app Nest con módulos “core”, habilito logging/errores y agrego scripts y README, entonces el servidor inicia sin fallos y la estructura documentada cumple RFN-05.
HU-02	Como desarrollador, quiero configurar SQLite y migraciones mínimas para persistir entidades iniciales de forma reproducible.	3	Dado el proyecto Nest, cuando defino conexión y ejecuto migraciones/seed y el comando de reset, entonces las tablas existen, el arranque es estable y el proceso queda documentado (RFN-05).
HU-03	Como tester, quiero una colección Postman con health-check y rutas base para verificar rápidamente la salud del servicio.	2	Dado el backend en ejecución, cuando corro la colección y consulto /health y rutas base, entonces obtengo 200 con payload consistente y dejo evidencia exportada en el repo (RFN-03).

Nota. Elaborado por el autor.

Se establece la base del Backend NestJS modular con configuración centralizada (env, logging, filtros), base de datos SQLite con pipeline de migraciones .sql y scaffolding de dominios principales.

Tabla 11. Planificación del Sprint 2 — Dominios Projects y Endpoints

Historia de Usuario	Descripción	Puntos	Criterios de Aceptación
HU-04	Como desarrollador, quiero el módulo de Projects con unicidad y validaciones para gestionar iniciativas sin colisiones ni datos inválidos.	5	Dado el esquema de Projects, cuando creo/leo/actualizo/elimino y pruebo duplicar name, entonces recibo 2xx en casos válidos y 409 en conflicto con mensajes uniformes (RF-02, RFN-03).
HU-05	Como desarrollador, quiero el módulo de Endpoints por proyecto para registrar rutas, métodos y artefactos asociados.	5	Dado un proyecto existente, cuando registro/edito/listo endpoints y aplico filtros por texto/método/estado, entonces el detalle refleja artefactos y las validaciones responden coherentemente (RF-03, RFN-03).
HU-06	Como tester, quiero validar respuestas y códigos de error en Postman para comprobar coherencia y manejo de bordes.	2	Dado CRUD de Projects/Endpoints, cuando ejecuto casos nominales y 404/409, entonces los códigos/payloads son consistentes y se adjunta evidencia (RFN-03).

Nota. Elaborado por el autor.

Se habilita el CRUD REST para proyectos y endpoints. En projects se habilita la creación de los proyectos playwright con comandos automatizados y templates para la generación de los archivos básicos de un framework de prueba.

Tabla 12. Planificación del Sprint 3 — Test Cases y Test Suites

Historia de Usuario	Descripción	Puntos	Criterios de Aceptación
HU-07	Como desarrollador, quiero el módulo de Test Cases (manual BDD) vinculado a endpoints para diseñar escenarios trazables.	7	Dado endpoints válidos, cuando creo/edito un caso con escenario/tags/ejemplos y campos obligatorios, entonces persiste con referencia a endpoint y formato validado (RF-04, RFN-05).
HU-08	Como desarrollador, quiero el módulo de Test Suites con composición por casos/suites para planificar ejecuciones agrupadas.	5	Dado casos existentes, cuando creo/edito suites y selecciono/reordeno su composición, entonces la integridad se mantiene y el conteo refleja la selección (RF-05, RFN-05).
HU-09	Como tester, quiero validar la relación proyecto, caso, suite para asegurar integridad de extremo a extremo.	2	Dado proyectos, casos y suites creados, cuando recorro el flujo CRUD y consulto vínculos, entonces no hay huérfanos y las FK son válidas con evidencia registrada (RF-02/04/05, RFN-03).

Nota. Elaborado por el autor.

Se introduce la capacidad de diseñar pruebas (BDD) y agruparlas en Test Suites. También se crea la lógica y las tablas de Test Steps que soportan la creación de casos de prueba. Las Test Suites se habilitan para que sean de tipo Test Set o Test Plan.

Tabla 13. Planificación del Sprint 4 — Ejecuciones y Bugs

Historia de Usuario	Descripción	Puntos	Criterios de Aceptación
HU-10	Como desarrollador, quiero el módulo de Test Executions con SSE de progreso y resultados para observar ejecución en tiempo real.	7	Dado casos o suites listos, cuando inicio una ejecución y me suscribo al SSE, entonces recibo started/progress/completed/failed y se persisten KPIs (passed/failed/skipped, duración) (RF-06, RFN-02).
HU-11	Como desarrollador, quiero el módulo de Bugs vinculado a ejecuciones y casos para documentar incidencias con contexto técnico.	5	Dado una ejecución fallida, cuando creo/edito/listo un bug enlazado a executionId/testCaseId, entonces queda filtrable con estructura de error uniforme (RF-07, RFN-03).
HU-12	Como desarrollador, quiero documentar la API (Swagger/README) para facilitar consumo y mantenimiento.	2	Dado endpoints estables, cuando accedo a la documentación, entonces las rutas y ejemplos están actualizados con notas de config/limitaciones (RFN-05/06).

Nota. Elaborado por el autor.

Se cierra el Release con todos los módulos requeridos implementados, validaciones, migraciones y endpoints listos en sus respectivos controladores. Los Test Executions incorporan un SSE que permite saber si una ejecución ya acabó y cuál es su estado.

Release 2 — Frontend (S5–S8)

Tabla 14. Planificación del Sprint 5 — Base de la SPA y Dashboard simulado

Historia de Usuario	**Descripción**	**Puntos**	**Criterios de Aceptación**
HU-13	Como desarrollador, quiero inicializar la SPA (React+Vite+Tailwind) con layout, rutas y estados base para una UI consistente y escalable.	5	Dado el arquetipo cliente, cuando configuro routing, header/sidebar y estados de carga/errores, entonces la navegación funciona sin rutas rotas con feedback uniforme (RFN-06).
HU-14	Como desarrollador, quiero un Dashboard con KPIs mockeados como punto de entrada para orientar al usuario.	5	Dado datos simulados, cuando cargo el dashboard, entonces veo KPIs de proyectos/endpoints/ejecuciones y puedo navegar a los módulos desde tarjetas (RF-01, RFN-06).

HU-15	Como tester, quiero validar navegación y accesibilidad base para garantizar uso fluido desde el inicio.	2	Dado la SPA, cuando recorro rutas y formularios con teclado/lector, entonces la estructura es accesible y los estados vacíos están definidos (RFN-06).

Nota. Elaborado por el autor.

Se inicializa el proyecto frontend con una interfaz responsiva que permita la navegación y un dashboard inicial. Se dejan listas las cards para conectarlas con los datos reales.

Tabla 15. Planificación del Sprint 6 — Páginas de Projects y Endpoints

Historia de Usuario	Descripción	Puntos	Criterios de Aceptación
HU-16	Como desarrollador, quiero la página de Projects con crear/ver/editar/eliminar para gestionar iniciativas desde la UI.	7	Dado datos simulados, cuando uso formularios con validaciones y filtros, entonces las tarjetas reflejan cambios con toasts y mensajes claros (RF-02, RFN-06).
HU-17	Como desarrollador, quiero la página de Endpoints con crear/ver/editar/eliminar para administrar el catálogo por proyecto.	5	Dado endpoints simulados, cuando filtro por texto/método/estado y abro detalle, entonces veo datos y acciones esperadas con validaciones visibles (RF-03, RFN-06).
HU-18	Como tester, quiero evaluar la usabilidad de formularios y mensajes de error para mejorar claridad de interacción.	1	Dado los formularios activos, cuando ingreso errores y corrijo, entonces la UI muestra ayudas inline y estados coherentes (RFN-06).

Nota. Elaborado por el autor.

Se llevan a UI los primeros dominios con datos simulados. Se crean los botones para poder crear las cards. Las cards contienen las acciones para editar, visualizar o borrar proyectos o endpoints. Hay filtros para buscar las cards y formularios con validaciones.

Tabla 16. Planificación del Sprint 7 — Páginas de Test Cases y Test Suites

Historia de Usuario	Descripción	Puntos	Criterios de Aceptación
HU-19	Como desarrollador, quiero la página de Test Cases con editor BDD para crear y mantener escenarios reutilizables.	7	Dado el editor BDD, cuando inserto Given/When/Then, tags y ejemplos y guardo, entonces el caso se valida y aparece en el listado con sus metadatos (RF-04, RFN-06).
HU-20	Como desarrollador, quiero la página de Test Suites con composición por casos para planificar ejecuciones.	5	Dado casos simulados, cuando creo/edito una suite y selecciono casos, entonces el conteo y la composición se conservan al reabrir (RF-05, RFN-06).

Nota. Elaborado por el autor.

Se agregan las páginas de Test Cases y Test Suites con botones para crear cada ítem respectivamente, las cards contienen la información necesaria básica como datos de identificación y status y dentro de los detalles se pueden visualizar aspectos específicos.

Tabla 17. Planificación del Sprint 8 — Vistas Executions y Bugs (mock)

Historia de Usuario	Descripción	Puntos	Criterios de Aceptación
HU-21	Como desarrollador, quiero la página de Test Executions con listado y detalle para revisar KPIs y resultados.	5	Dado ejecuciones simuladas, cuando filtro/abro detalle, entonces veo pasos, estados y métricas con feedback de carga y empty-states (RF-06, RFN-06).
HU-22	Como desarrollador, quiero la página de Bugs con filtros y detalle para gestionar incidencias visualmente.	5	Dado bugs simulados, cuando aplico filtros por severidad/estado y navego al detalle, entonces la información es completa y coherente (RF-07, RFN-06).
HU-23	Como desarrollador, quiero la página de configuraciones y OpenAI para centralizar ajustes.	1	Dado la sección Settings, cuando accedo a OpenAI, entonces la UI es consistente y queda lista para conectar con backend (RFN-06).
HU-24	Como tester, quiero validar la UI contra prototipos Figma para asegurar coherencia.	1	Dado el prototipo, cuando comparo flujos, entonces la implementación respeta estructura y microinteracciones clave (RFN-06).

Nota. Elaborado por el autor.

Se completa el frontend con las páginas restantes de Executions y Bugs. Además se agrega la página de configuración. Por último se validan el diseño creado comparando con el prototipado en Figma.

Release 3 — Integración frontend–backend (S9–S10)

Tabla 18. Planificación del Sprint 9 — Conexiones reales y sincronización

Historia de Usuario	Descripción	Puntos	Criterios de Aceptación
HU-25	Como desarrollador, quiero conectar Projects/Endpoints a la API real para operar con datos persistentes y errores controlados.	5	Dado el backend local, cuando consumo servicios tipados y simulo errores, entonces la UI muestra reintentos/feedback y refleja payloads reales (RF-02/03, RFN-03/06).
HU-26	Como desarrollador, quiero conectar Test Cases/Suites a la API real para mantener escenarios y composición persistentes.	5	Dado endpoints de casos/suites, cuando creo/edito/listo desde la UI, entonces las validaciones coinciden entre capas y la mensajería es uniforme (RF-04/05, RFN-06).
HU-27	Como desarrollador, quiero sincronizar artefactos del workspace para alinear base de datos y archivos.	3	Dado proyectos con artefactos, cuando ejecuto la sincronización desde la UI, entonces recibo un reporte con conteos, errores y tiempos (RF-09, RFN-03).

Nota. Elaborado por el autor.

Se reemplazan los datos simulados por API real, se incorporan los endpoints del backend desde el puerto local seleccionado. Se realiza la sincronización de workspace.

Tabla 19. Planificación del Sprint 10 — Ejecución E2E, SSE e IA

Historia de Usuario	Descripción	Puntos	Criterios de Aceptación
HU-28	Como tester, quiero ejecutar pruebas desde la UI con SSE para ver progreso y resultados en tiempo real.	3	Dado casos o suites reales, cuando lanzo la ejecución y me suscribo, entonces recibo eventos SSE y se actualizan KPIs y resultados por escenario (RF-06, RFN-02).
HU-29	Como desarrollador, quiero generación y sugerencias con IA para acelerar diseño y evitar duplicados.	5	Dado la API Key configurada, cuando solicito generación/sugerencias, entonces recibo formato estricto, se advierten duplicidades y no se expone la clave (RF-04/08, RFN-01/04).
HU-30	Como desarrollador, quiero verificar errores y registrar bugs desde fallos para dejar trazabilidad de incidencias.	5	Dado una ejecución con fallos, cuando creo el bug desde el detalle, entonces queda vinculado a executionId/testCaseId y visible en Bugs (RF-07, RFN-03).

Nota. Elaborado por el autor.

Se hacen pruebas E2E para validar ejecuciones de prueba con status provisto por el backend en tiempo real y se validan si se crean bugs automáticamente ante fallos.

Release 4 — Empaquetado y despliegue (S11–S12)

Tabla 20. Planificación del Sprint 11 — Packaging NPM y documentación técnica

Historia de Usuario	Descripción	Puntos	Criterios de Aceptación
HU-31	Como desarrollador, quiero empaquetar como librería NPM con build reproducible para facilitar distribución y reutilización.	7	Dado el proyecto estable, cuando genero artefactos y configuro package.json (main/types/scripts), entonces el build es determinista y existe versión SemVer inicial (RFN-05).
HU-32	Como administrador, quiero documentar instalación y mantenimiento para reducir fricción de adopción.	3	Dado el paquete construido, cuando sigo README/Quickstart, entonces puedo instalar, ejecutar un ejemplo mínimo y configurar secretos (RFN-06/04).
HU-33	Como tester, quiero probar la instalación local del paquete para asegurar consumibilidad en entorno limpio.	1	Dado un proyecto vacío, cuando instalo y ejecuto el ejemplo, entonces todo funciona y queda evidencia registrada (RFN-03/05).

Nota. Elaborado por el autor.

Se transforma la arquitectura en un monolito con scripts de compilación en ambiente de desarrollo y producción para producir un paquete instalable, con documentación que explique instalación y uso.

Tabla 21. Planificación del Sprint 12 — Publicación y verificación final

Historia de Usuario	Descripción	Puntos	Criterios de Aceptación
HU-34	Como administrador, quiero publicar la librería en registro privado y etiquetar versión para habilitar consumo controlado.	3	Dado los artefactos generados, cuando publico y creo el tag, entonces puedo instalar desde un consumidor y registrar el changelog inicial (RFN-05).
HU-35	Como desarrollador, quiero ejecutar el checklist de seguridad y secretos para mitigar riesgos de operación.	3	Dado dependencias/secretos configurados, cuando reviso licencias, vulnerabilidades y conectividad, entonces se cumplen controles y queda constancia de revisión (RFN-04/05).
HU-36	Como tester, quiero verificar RF-01..10 y RFN-01..06 para cerrar el MVP con evidencia trazable.	3	Dado la release publicada, cuando ejecuto la validación final, entonces documento cumplimiento y hallazgos menores con cierre formal (RF/RFN).

Nota. Elaborado por el autor.

Se publica el paquete en el hub de librerías de NPM y se realiza la verificación completa contra RF/RFN priorizados. Con esto se cierra el MVP con trazabilidad y evidencia.

Esta secuencia de historias de usuario se propone para mantener coherencia entre incrementos funcionales y calidad: cada sprint consolida un subconjunto de RF, explicita el cumplimiento de RFN y documenta evidencias. Cada historia de usuario aporta valor con respecto al objetivo de la solución planteada y permiten construir la interfaz visual y el Backend esperado. De esta manera, el MVP puede avanzar de forma trazable y comprobable hasta llegar a su empaquetado y distribución.

CAPITULO V

5. IMPLEMENTACIÓN

5.1. Descripción del Sistema Desarrollado

Por motivos de de despliegue e identificar el trabajo del autor y la solución creada, se le dio un nombre, por lo cual se puede referir al sistema como **Omega Testing**. La herramienta Omega Testing una vez ya desarrollada cuenta con diferentes elementos que serán explicados más a detalle en este capítulo.

5.1.1. Arquitectura general del sistema (frontend, backend y base de datos)

El sistema **Omega Testing MVP** fue diseñado bajo una arquitectura modular que sigue el principio de separación por responsabilidades, lo cual facilita su mantenimiento, escalabilidad y despliegue independiente de cada capa.

El proyecto está estructurado en un **monorepo**, compuesto por dos aplicaciones principales: un frontend desarrollado en React 18 con Vite y TypeScript, y un backend construido en NestJS, también con TypeScript. Ambas aplicaciones interactúan de manera síncrona a través de una API REST expuesta en el puerto 3000.

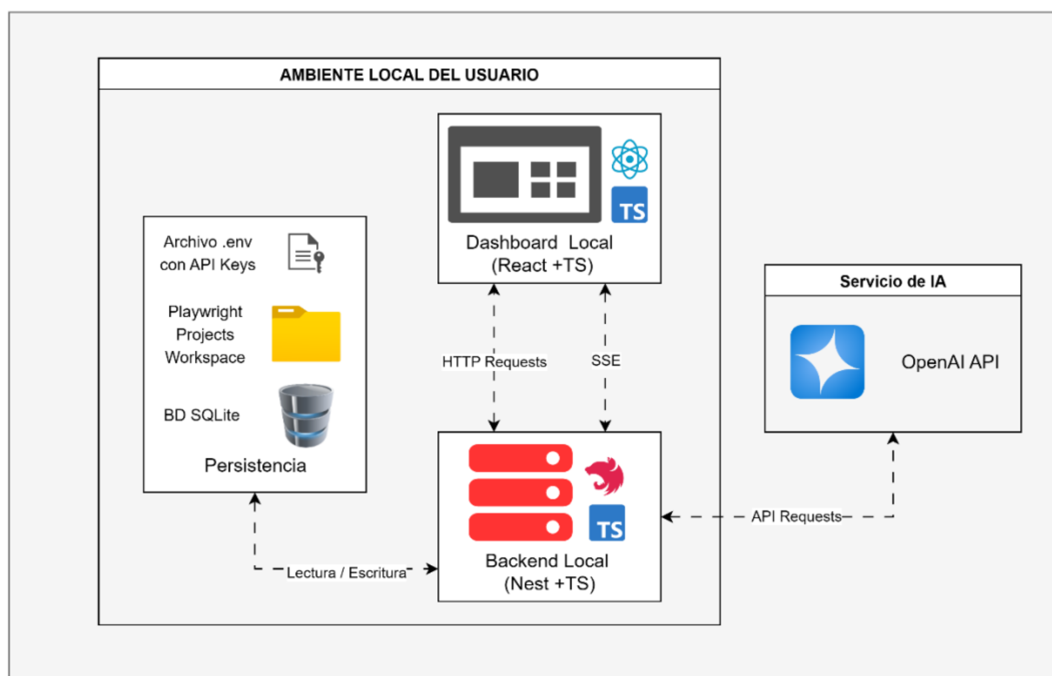
El **frontend** cumple la función de capa de presentación, ofreciendo una interfaz web que permite gestionar proyectos de prueba, endpoints, casos de prueba, suites, ejecuciones y reporte de bugs. Emplea tecnologías modernas de renderizado rápido y componentes reutilizables con TailwindCSS, TanStack Query y Radix UI, garantizando una experiencia fluida y responsiva.

El **backend**, por su parte, constituye la capa lógica del sistema. Gestiona la creación de entidades, la ejecución de pruebas automatizadas y la persistencia de datos. Está basado en **NestJS** (versión 11) e implementa un ORM mediante TypeORM, utilizando **SQLite** como base de datos embebida para simplificar la ejecución local del MVP. El backend expone sus endpoints bajo el prefijo /v1/api y cuenta con documentación interactiva generada automáticamente mediante **Swagger**.

A nivel de almacenamiento, el sistema utiliza una base **SQLite** llamada central-backend.sqlite, ubicada dentro del directorio de trabajo de Playwright Workspaces. Esta base registra los proyectos, endpoints, test cases, ejecuciones y bugs. Los *workspaces*

generados por el sistema almacenan la estructura de pruebas generadas en **Playwright y Cucumber (BDD)**, permitiendo ejecutar los casos directamente desde CLI o desde la interfaz.

Figura 10. Diagrama general de arquitectura del sistema Omega Testing MVP



Nota. Elaborado por el autor.

La figura representa el flujo de componentes del sistema en un entorno local del usuario. En la capa superior se encuentra el Dashboard Local (React + TypeScript), que gestiona la interacción del usuario y se comunica con el backend mediante HTTP y SSE para mostrar resultados en tiempo real.

El Backend Local (NestJS + TypeScript) procesa las solicitudes del frontend, coordina la ejecución de pruebas y gestiona la persistencia de datos. Puede además conectarse con el servicio de IA de OpenAI para generar o analizar casos de prueba.

La sección de Persistencia incluye el archivo .env con las claves de configuración, la base de datos SQLite para el almacenamiento de entidades y el workspace de Playwright, donde se guardan los proyectos y artefactos generados.

El esquema evidencia un modelo cliente–servidor local con comunicación directa entre frontend y backend, sin requerir infraestructura externa para su funcionamiento.

5.1.2. Tecnologías empleadas y estructura del monorepo

Tecnologías y frameworks principales

El sistema se desarrolló sobre un stack completamente basado en **TypeScript**, priorizando la coherencia tipada, la modularidad y la ejecución local sin dependencias externas. La arquitectura combinó un enfoque **monolítico modular**, donde el frontend y el backend compartieron una misma base de código dentro de un monorepo estructurado con npm workspaces.

La siguiente tabla resume las tecnologías empleadas y su propósito dentro del MVP:

Tabla 22. Principales tecnologías utilizadas en el código del proyecto

Componente	Tecnología / Librería	Propósito principal
Frontend	React 18, TypeScript, Vite, TailwindCSS, TanStack Query, Radix UI	Interfaz gráfica, gestión de estado y diseño responsivo
Backend	NestJS 11, TypeORM, SQLite, Swagger, Helmet, Compression	API REST, persistencia y documentación
Testing	Playwright, @cucumber/cucumber	Ejecución automatizada de pruebas BDD
CLI / Builds	Node.js, npm workspaces	Control de versiones y empaquetado
AI opcional	OpenAI SDK	Generación automática de test cases (configurable)

Nota. Elaborado por el autor.

Cada tecnología fue seleccionada por su compatibilidad y facilidad de integración. React y Vite ofrecieron velocidad en desarrollo y renderizado, mientras que NestJS permitió mantener una arquitectura escalable, con separación por dominio y control tipado. SQLite facilitó la persistencia sin necesidad de servicios externos, ideal para un MVP local. Playwright y Cucumber brindaron soporte BDD (Behavior-Driven Development), y la integración con OpenAI SDK se utilizó para la generación automática de pruebas cuando el usuario habilitaba la función inteligente.

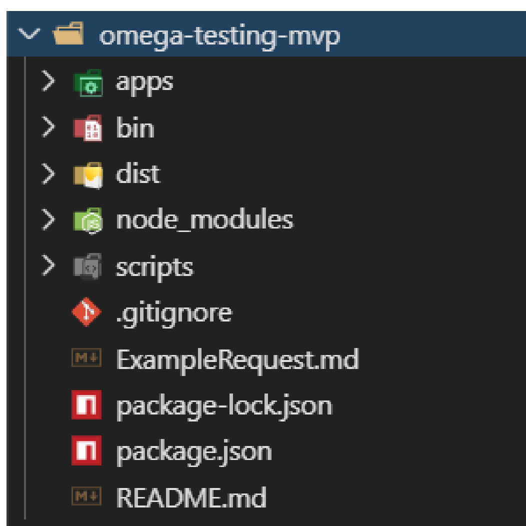
Estructura general del monorepo

El proyecto se implementó bajo un esquema monorepo, alojando tanto el frontend como el backend en un mismo repositorio. Esto permitió mantener dependencias sincronizadas, compartir tipos y facilitar los procesos de build y empaquetado.

En el nivel raíz se incluyeron los siguientes directorios y archivos:

- **apps**/: contiene los subproyectos de frontend y backend.
- **bin**/: scripts ejecutables del CLI local (omega-testing.js, static-frontend.js).
- **dist**/: carpeta generada por el proceso de build (npm run build:dist), donde se empaquetaron los artefactos finales.
- **scripts**/: utilitarios de compilación y copia, como copy-builds.js.
- **package.json**: archivo principal que define las *workspaces*, dependencias globales y comandos de desarrollo.

Figura 11. Estructura general del monorepo (carpetas raíz del proyecto)



Nota. Elaborado por el autor.

Estructura del frontend (apps/frontend)

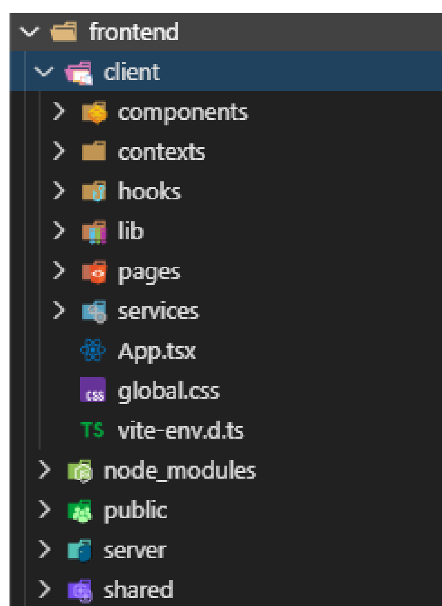
El frontend fue diseñado como una Single Page Application (SPA) desarrollada en React + TypeScript, con navegación modular y comunicación directa con el backend a través de peticiones HTTP y un canal SSE (Server-Sent Events).

El directorio client/ concentró la mayor parte de la lógica de la interfaz, organizada de la siguiente forma:

- **pages**/: alojó las vistas principales de la aplicación (*Dashboard, Projects, Endpoints, Test Cases, Test Suites, Test Executions, Bugs* y *Settings*).
- **components**/: agrupó componentes por dominio (formularios, *dialogs*, *cards*, *filters*) y elementos transversales del sistema de diseño.
- **contexts**/: manejó el estado global mediante contextos como AuthContext y ExecutionContext.
- **services**/: implementó los clientes HTTP que consumieron la API REST del backend.
- **hooks**/ y **lib**/: contuvieron lógica reutilizable y utilidades comunes.

La estructura permitió mantener una clara separación de responsabilidades y facilitar la reutilización de componentes visuales y lógicos.

Figura 12. Carpetas con estructura del frontend (client/)



Nota. Elaborado por el autor.

Estructura del backend (apps/backend)

El backend se construyó sobre NestJS 11, adoptando una arquitectura modular por dominio. Cada módulo encapsuló su propia lógica, controladores, servicios, DTOs y plantillas, facilitando la extensión del sistema sin romper dependencias internas.

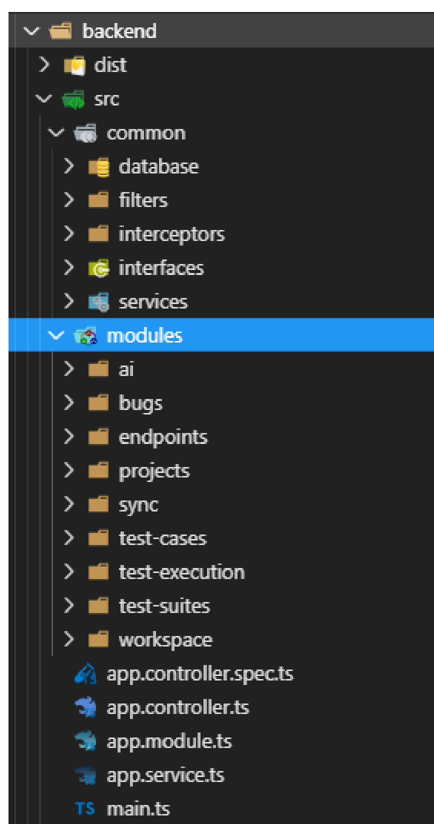
La estructura principal dentro de backend/src/ fue la siguiente:

- **modules**/: agrupó los módulos de negocio (projects, endpoints, test-cases, test-suites, test-execution, bugs) y los módulos transversales (ai, sync, workspace).

- **common**/: concentró elementos compartidos como filtros, interceptores, servicios generales y configuración de base de datos.
- **controllers**/: definieron las rutas HTTP y validaciones de entrada.
- **services**/: implementaron la lógica de negocio y la interacción con la base de datos.
- **dto**/: definieron los objetos de transferencia de datos.
- **templates**/: incluyeron archivos base para la generación automatizada de artefactos de prueba (por ejemplo, schema.template, hooks-endpoints.template).

El módulo endpoints sirvió como ejemplo representativo de la estructura modular, conteniendo controladores, DTOs, servicios especializados y plantillas para la generación de configuraciones y clientes de API.

Figura 13. Estructura de carpetas del backend (src/modules)

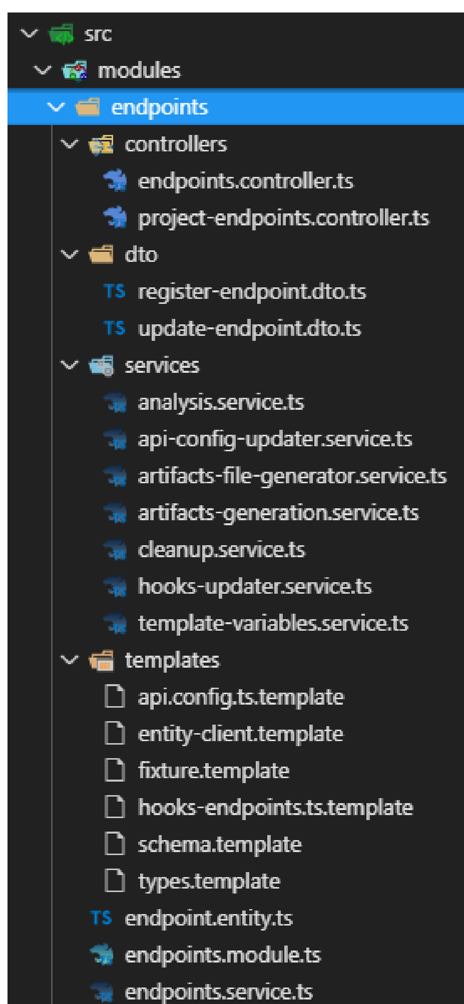


Nota. Elaborado por el autor.

Estructura interna de los módulos del backend

La Figura 13 muestra la organización interna típica de los módulos del backend, tomando Endpoints como ejemplo representativo. Cada módulo siguió la misma convención de estructura definida por NestJS, asegurando una clara separación de responsabilidades entre controladores, servicios, validaciones, entidades y plantillas.

Figura 14. Estructura usada en los módulos del backend (ejemplo: módulo Endpoints)



Nota. Elaborado por el autor.

Dentro de cada módulo se incluyeron las siguientes carpetas y archivos:

- **controllers**/: contiene los controladores encargados de manejar las rutas HTTP del dominio correspondiente. En el caso de Endpoints, por ejemplo, endpoints.controller.ts y project-endpoints.controller.ts gestionaron las operaciones CRUD y las consultas por proyecto.
- **dto**/: almacena los objetos de transferencia de datos, como register-endpoint.dto.ts y update-endpoint.dto.ts, que definen los esquemas de validación y las estructuras de entrada esperadas por cada operación.
- **services**/: agrupa la lógica de negocio y los procesos automatizados. En Endpoints incluyó servicios como artifacts-generation.service.ts o api-config-updater.service.ts, dedicados a generar, limpiar y sincronizar los artefactos de prueba.

- **templates**/: centraliza las plantillas de código utilizadas por el sistema para generar dinámicamente archivos de configuración, esquemas o hooks de prueba, manteniendo un formato consistente entre proyectos.

Finalmente, los archivos .entity.ts, .module.ts y .service.ts definen respectivamente el modelo de datos persistente, la configuración del módulo y el servicio principal que coordina sus componentes.

Esta estructura modular permitió mantener independencia funcional entre dominios como Projects, Test Cases o Bugs, facilitando el mantenimiento, las pruebas unitarias y la escalabilidad del sistema.

Capa de Inteligencia Artificial con OpenAI

La capa de inteligencia artificial se integró como un componente esencial del backend dentro del módulo apps/backend/src/modules/ai/. Su función fue automatizar la generación y análisis de casos de prueba BDD (Behavior Driven Development) a partir de la información técnica de los endpoints registrados, proporcionando asistencia contextual y aprendizaje continuo a través de la API de OpenAI.

El módulo se estructuró en servicios especializados y controladores que interactúan entre sí bajo una arquitectura modular:

- openai-config.service.ts gestiona la conexión con la API y valida la clave almacenada en el archivo .env.
- assistant-manager.service.ts administra los asistentes y su contexto.
- thread-manager.service.ts gestiona los hilos de conversación, permitiendo mantener coherencia entre solicitudes.
- test-case-generation.service.ts coordina la generación automática de escenarios en lenguaje Gherkin.
- test-case-suggestion.service.ts propone escenarios de prueba basados en las propiedades del endpoint.
- ai-generation.service.ts centraliza el registro de cada ejecución y su trazabilidad.

El proceso inicia cuando un usuario solicita la generación de un caso de prueba. El sistema asegura la conexión con OpenAI, valida las credenciales y crea el cliente necesario:

Figura 15. Ejemplo de código de conexión a OpenAI

```
import OpenAI from "openai";
@Injectable()
export class OpenAiConfigService {
  async ensureClient() {
    const apiKey = await this.getOpenAIKey();
    return new OpenAI({ apiKey });
  }
}
```

Nota. Elaborado por el autor.

Una vez validada la conexión, el backend crea o reutiliza un asistente y un hilo asociado al proyecto:

Figura 16. Ejemplo de creación de un Assistant y Threads de Open AI en Typescript

```
const assistant = await openai.beta.assistants.create({
  name: "Omega Testing Assistant",
  model: "gpt-4o-mini",
});
const thread = await openai.beta.threads.create();
```

Nota. Elaborado por el autor.

El servicio test-case-generation.service.ts envía un prompt contextualizado que incluye información del endpoint, parámetros y ejemplos de pruebas previas. La IA genera los escenarios en formato Gherkin y el sistema los almacena junto a los artefactos en el workspace de Playwright.

Figura 17. Ejemplo de Prompt para la generación de Casos de Prueba en Typescript

```
const run = await openai.beta.threads.runs.create(thread.threadId, {
  assistant_id: assistant.assistantId,
  tool_choice: "auto",
  truncation_strategy: { type: "auto" },
});
const result = await this.waitForRunCompletion(thread.threadId, run.id);
await this.aiGenerationService.markAsCompleted(generationId, {
  feature,
  steps,
});
```

Nota. Elaborado por el autor.

El flujo completo queda registrado en la tabla ai_generations, donde se almacenan estado, tokens utilizados, modelo empleado y duración de la operación.

Por otra parte, el servicio test-case-suggestion.service.ts analiza el método, la ruta y la entidad del endpoint para proponer escenarios potenciales antes de la generación:

Los controladores ai.controller.ts y ai-general.controller.ts exponen los endpoints necesarios para inicializar asistentes, validar la conexión y generar o sugerir casos. Esto permite que la interfaz consuma directamente los servicios del módulo y visualice las sugerencias y resultados generados.

En conjunto, esta capa garantiza una integración continua entre las pruebas automatizadas y la generación inteligente de artefactos, manteniendo la trazabilidad completa de cada interacción con la IA y fortaleciendo la productividad del proceso de testing dentro del entorno local.

Integración entre componentes

El monorepo permitió ejecutar y empaquetar ambos subproyectos mediante comandos centralizados. El CLI, alojado en la carpeta bin/, coordinó la ejecución conjunta del frontend y backend bajo el comando **npx omega-testing-mvp start-local**. Los scripts del directorio scripts/ automatizaron el proceso de construcción, trasladando los builds finales a la carpeta dist/.

Una vez compilado, el sistema se ejecutó completamente en entorno local, utilizando una base de datos SQLite y una carpeta de Playwright Workspaces donde se almacenaron los proyectos, archivos .env y reportes de prueba. Esta integración garantizó independencia del entorno, simplicidad en la instalación y total control sobre la persistencia y ejecución del sistema.

5.2. Despliegue del Software

5.2.1. Entorno de ejecución e instalación local

El despliegue del sistema **Omega Testing MVP** se realizó principalmente en entorno local, con el objetivo de garantizar independencia del usuario frente a servicios externos y permitir una instalación simple y reproducible.

El proyecto requiere únicamente **Node.js** y **npm** para su ejecución. Una vez clonado el repositorio, el proceso de instalación se lleva a cabo mediante:

```
npm install
```

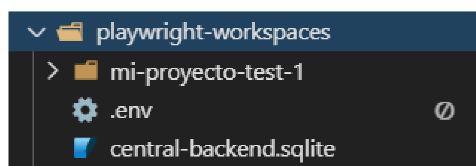
Con este comando se instalan todas las dependencias definidas en el monorepo, tanto del frontend como del backend. Para iniciar el entorno de desarrollo se ejecuta:

```
npm run dev
```

Este script inicia ambos servicios en paralelo, habilitando logs unificados. Por defecto, el **frontend** se ejecuta en el puerto **5173** y el **backend** en el **3000**, comunicándose por peticiones **HTTP** y un canal **SSE (Server-Sent Events)** que transmite en tiempo real el progreso de las ejecuciones.

Durante la primera ejecución, el sistema genera automáticamente la carpeta **playwright-workspaces/**, la cual contiene los proyectos de prueba, los artefactos y la base de datos **SQLite** (central-backend.sqlite). Esta estructura permite que el usuario ejecute, inspeccione y repita pruebas sin configuración adicional, garantizando persistencia local de los resultados y compatibilidad multiplataforma.

Figura 18. Ejemplo de creación de la carpeta de proyectos y base de datos



Nota. Elaborado por el autor.

5.2.2. Construcción, empaquetado y CLI de ejecución

El proyecto integra un **CLI interno** que permite ejecutar el sistema como una herramienta completa a través de un solo comando. Desde el monorepo, la compilación se realiza con:

```
npm run build:dist
```

Este proceso genera los builds del frontend y backend, trasladándolos al directorio **dist/** mediante el script scripts/copy-builds.js. Allí se consolidan todos los artefactos necesarios para la distribución, incluyendo el dashboard compilado, el API backend, los binarios del CLI, y los scripts de inicio.

El sistema puede ejecutarse de dos formas:

- **Modo monolítico (por defecto)**: inicia frontend y backend integrados en un mismo proceso, simplificando el uso personal.

  ```
  npx omega-testing-mvp start-local
  ```

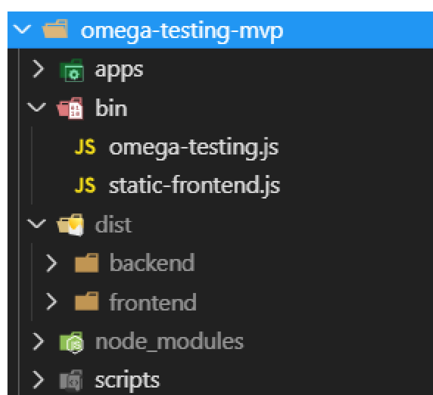
- **Modo dividido (split mode)**: permite ejecutar ambos componentes en puertos separados, ideal para pruebas o despliegues híbridos.

  ```
  node bin/omega-testing.js split --frontend-port 5173 --port 3000
  ```

El comando npm run health puede ejecutarse en cualquier momento para comprobar el estado del sistema, las rutas configuradas y la conexión con la API.

La estructura generada tras la compilación queda organizada de la siguiente forma:

Figura 19. Estructura de archivos generados para la compilación en producción



Nota. Elaborado por el autor.

Este diseño asegura que el sistema funcione como una aplicación independiente, capaz de ejecutarse localmente o redistribuirse como herramienta empaquetada sin requerir infraestructura adicional.

5.2.3. Publicación y configuración del entorno

El despliegue final del software se realizó mediante su **empaquetado y publicación como librería npm**, bajo el nombre omega-testing-mvp. Esto permitió instalar el sistema globalmente o integrarlo como dependencia dentro de otros proyectos:

```
npm i omega-testing-mvp
```

El proceso de publicación se gestiona con versionamiento semántico y control de cambios desde el CLI:

```
npm version patch -m "fix(cli) message"

git push && git push --tags

npm publish --access public
```

El paquete publicado incluye el CLI (bin/omega-testing.js), los builds del backend y frontend, y los scripts necesarios para su ejecución directa. De esta forma, el sistema puede iniciarse desde cualquier entorno Node con el comando npx omega-testing-mvp start-local, reproduciendo automáticamente el entorno completo de ejecución.

La configuración se administra mediante el archivo **.env**, que define las variables necesarias para el funcionamiento del backend y la integración de IA. La variable que se debe definir es:

- OPENAI_API_KEY: clave privada que habilita la capa de inteligencia artificial.

Si alguna variable no se encuentra definida, el sistema adopta valores predeterminados para garantizar su ejecución. Las claves sensibles, como la de OpenAI, se validan exclusivamente desde el backend, evitando cualquier exposición en el cliente.

En conjunto, este mecanismo de empaquetado y despliegue convierte a Omega Testing MVP en una solución portable, autoejecutable y lista para ser instalada en cualquier entorno local o corporativo con Node.js, manteniendo la trazabilidad de las pruebas y el control total del flujo de datos dentro del propio equipo del usuario.

5.3. Implementación del Software

5.3.1. Descripción general del cliente web

El cliente web de **Omega Testing** materializa el ciclo completo de pruebas sobre APIs REST desde una interfaz única. Su objetivo es **centralizar** la creación y administración de proyectos, el registro de endpoints, la generación y edición de casos BDD, la orquestación de ejecuciones y el seguimiento de resultados y bugs con trazabilidad.

En términos de composición, la aplicación está implementada como SPA (React + TypeScript) y se organiza en una barra lateral de navegación (Dashboard, Projects, Endpoints, Test Cases, Test Suites, Test Executions, Bugs, Settings), una barra superior con acciones globales (tema oscuro, sincronización, accesos a ajustes) y un área de trabajo que cambia según el módulo seleccionado. La experiencia se apoya en componentes accesibles (Radix), tipado estricto y estados remotos gestionados con TanStack Query.

La comunicación con el servidor es **bidireccional**:

- Para operaciones CRUD y consultas históricas se emplean endpoints REST del backend NestJS.
- Para progreso en vivo de ejecuciones se utiliza SSE (Server-Sent Events); la UI se suscribe al canal del proyecto y actualiza en tiempo real estados, pasos y KPIs.
- Las funciones inteligentes de generación y sugerencia de casos se integran mediante la capa de IA del backend, que expone puntos de entrada controlados desde *Settings , OpenAI Configuration* y servicios específicos de generación.

A nivel de flujo, el usuario puede operar el sistema de la siguiente forma: crea un proyecto, registra endpoints, genera/edita casos (manual o asistido por IA), agrupa en suites, ejecuta y analiza resultados; ante fallos, registra o vincula bugs con la ejecución y el caso correspondiente. La persistencia se realiza en SQLite (proyectos, endpoints, casos, suites, ejecuciones y bugs) y los artefactos de prueba se gestionan en el *workspace* de Playwright (features, steps, schemas, clients), lo que garantiza trazabilidad entre UI, API y archivos generados.

Acciones globales disponibles desde la interfaz

- **Crear proyecto** y **sincronizar** artefactos con el workspace local.

- **Buscar** y **filtrar** elementos por módulo (estado, método, entidad, tags, fechas).
- **Ejecutar** casos y suites, con **progreso en tiempo real** y detalle de steps.
- **Registrar y actualizar bugs** derivados de ejecuciones fallidas.
- **Configurar OpenAI** (guardar y probar la clave) para habilitar generación/sugerencia.
- **Cambiar tema** (claro/oscuro) y mantener preferencias de UI entre sesiones.

En síntesis, el cliente web provee una capa de orquestación visual que refleja el estado del motor de pruebas, integra señales en vivo y ofrece una experiencia consistente y accesible para documentar, ejecutar y auditar pruebas de extremo a extremo.

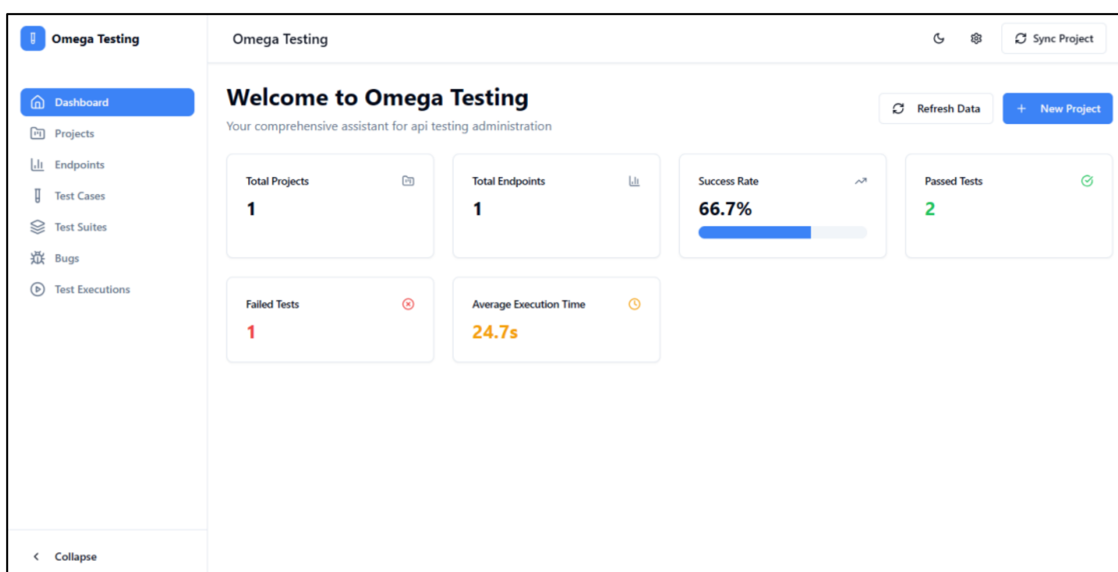
5.3.2. Módulos funcionales del sistema

5.3.2.1. Dashboard

El módulo **Dashboard** constituye la página principal de **Omega Testing**, donde se centraliza la información general del entorno de pruebas y el estado global de las ejecuciones recientes. Su propósito es ofrecer una visión resumida y actualizable de la actividad del sistema, facilitando al usuario la comprensión del desempeño del entorno sin necesidad de navegar por cada módulo individual.

En la Figura 20 se observa la interfaz del Dashboard, compuesta por tarjetas métricas que muestran los indicadores clave de rendimiento (KPIs) del sistema:

Figura 20. Interfaz principal del Dashboard de Omega Testing



Nota. Elaborado por el autor.

Entre las métricas visualizadas se incluyen:

- **Total Projects:** cantidad de proyectos activos registrados en el sistema.
- **Total Endpoints:** número de endpoints administrados en todos los proyectos.
- **Success Rate:** porcentaje de ejecuciones exitosas, representado mediante una barra de progreso visual (en el ejemplo, 66.7%).
- **Passed Tests y Failed Tests:** resumen de los resultados más recientes de pruebas ejecutadas, permitiendo identificar rápidamente fallos activos.
- **Average Execution Time:** tiempo promedio de ejecución por prueba, expresado en segundos, útil para analizar eficiencia y detectar posibles cuellos de botella.

En la parte superior derecha se encuentran las **acciones globales** del módulo:

- **Refresh Data:** actualiza los datos de las métricas en tiempo real, sincronizando con la API backend.
- **New Project:** abre el diálogo de creación de proyectos (*ProjectRegisterDialog*), permitiendo registrar un nuevo entorno de pruebas.
- **Sync Project:** ejecuta la sincronización manual de artefactos del proyecto (endpoints, casos, steps y escenarios) con el workspace local.

La interfaz mantiene coherencia con el resto del sistema, utilizando badges de color para diferenciar estados (por ejemplo, verde para éxito, rojo para error) y tipografía de alto contraste. Además, es totalmente responsiva, ajustando la disposición de las tarjetas según el tamaño de la pantalla, y respeta las preferencias de tema claro u oscuro configuradas por el usuario.

El Dashboard cumple así una doble función: supervisar de manera continua la salud del entorno de pruebas y servir como punto de partida para acceder a los módulos operativos (Projects, Endpoints, Test Cases, etc.), garantizando una experiencia de uso fluida, informativa y accesible.

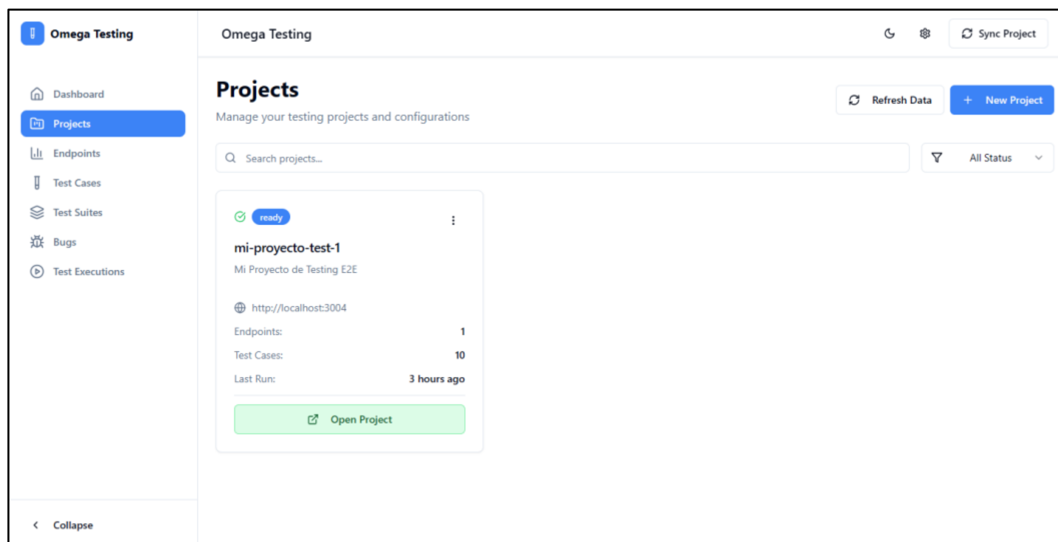
5.3.2.2. Projects

El módulo Projects permite administrar los proyectos de prueba registrados dentro de la plataforma, funcionando como el punto de partida para estructurar cada entorno de automatización. Desde esta vista, el usuario puede crear, editar, eliminar y

sincronizar proyectos, además de visualizar su estado operativo y los artefactos generados en backend y frontend.

En la siguiente figura se muestra la interfaz del módulo, donde cada proyecto aparece representado mediante una tarjeta con su nombre, estado y metadatos principales.

Figura 21. Vista general del módulo Projects de Omega Testing



Nota. Elaborado por el autor.

Información mostrada en cada proyecto

Cada tarjeta de proyecto presenta los siguientes elementos clave:

- **Nombre y Display Name:** identifican al proyecto y su alias legible.
- **Estado**: puede ser pending, ready o failed, acompañado de un icono y un color distintivo.
- **Base URL y Base Path**: ruta principal de la API bajo prueba.
- **Indicadores (KPIs)**: número de endpoints configurados, cantidad de casos de prueba asociados y fecha o tiempo desde la última ejecución.

El listado permite además aplicar filtros y búsquedas textuales, facilitando la localización de proyectos activos o en espera de configuración según su estado.

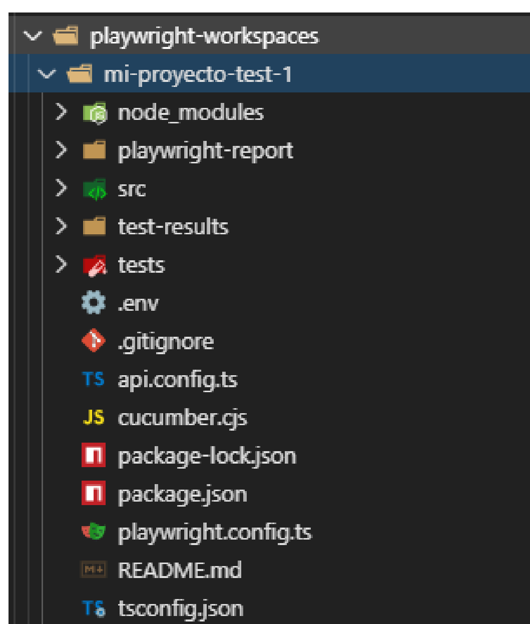
Acciones disponibles

El módulo incluye varias acciones para la gestión completa de los proyectos:

- **Create Project**: abre el diálogo de registro (ProjectRegisterDialog) para crear un nuevo entorno. El formulario solicita los campos name (único), displayName (opcional), baseUrl, basePath (opcional) y type (con valores posibles playwright-bdd o api-only).

  Cuando se crea un proyecto se inicializan todos los servicios del backend involucrados y se empieza a crear de manera silenciosa un proyecto de testing de playwright. Los logs con cada paso de la operación se muestran en la consola. En la siguiente figura se puede visualizar un proyecto de ejemplo generado.

Figura 22. Estructura del workspace generado para un proyecto en entorno local



Nota. Elaborado por el autor.

- **Edit Project**: accesible desde el menú contextual de cada tarjeta. Permite actualizar displayName, baseUrl o basePath. Los cambios se reflejan inmediatamente en la interfaz y en la base de datos SQLite del sistema.
- **Delete Project**: requiere confirmación escribiendo el projectId para evitar eliminaciones accidentales. Elimina tanto el registro como los artefactos asociados, siguiendo las reglas del backend.
- **Run Tests:** ejecuta de forma orquestada todas las pruebas vinculadas al proyecto. El progreso y los resultados se consultan en el módulo Test Executions, donde se registran los escenarios completados y el tiempo de ejecución promedio.

- **Open in Editor:** habilitada cuando el workspace está generado localmente. Abre la carpeta del proyecto en el entorno de desarrollo (por ejemplo, Visual Studio Code), permitiendo inspeccionar archivos feature, steps, schemas y configuraciones generadas automáticamente.

Ciclo de vida de un proyecto

Los estados del proyecto reflejan las etapas de su inicialización y disponibilidad:

- **Pending**: el proyecto ha sido creado y se encuentra en proceso de configuración.
- **Ready**: el workspace fue generado correctamente y está preparado para ejecutar pruebas o sincronizar artefactos.
- **Failed**: la generación presentó errores; el proyecto queda disponible para revisión o eliminación manual.

Feedback y validaciones

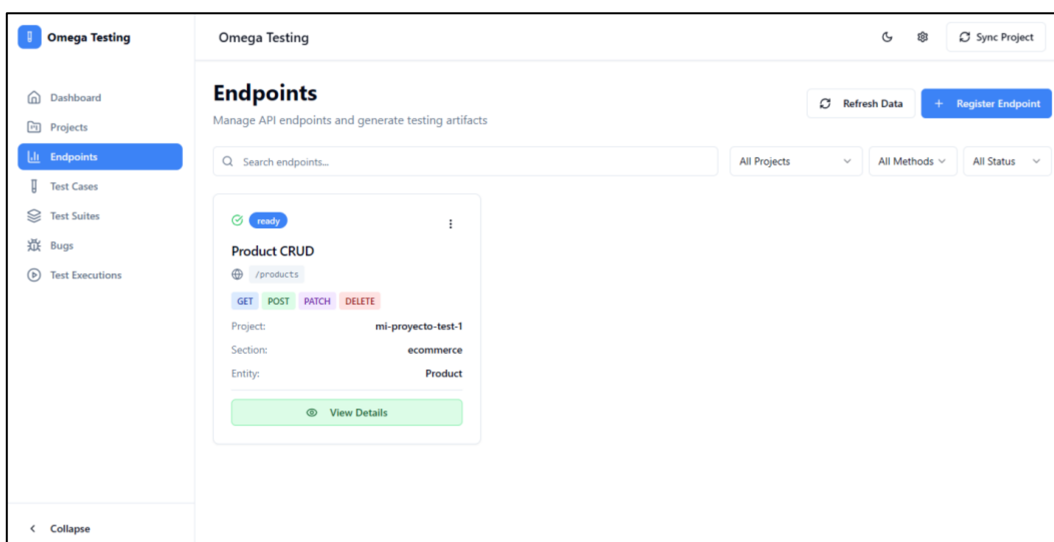
Durante las operaciones, la interfaz presenta mensajes de confirmación o error mediante toasts, registrando cada acción de forma trazable. Los formularios aplican validaciones estrictas para garantizar la integridad de los datos (por ejemplo, formato correcto de URLs y campos obligatorios).

5.3.2.3. Endpoints

El módulo Endpoints permite administrar los recursos de tipo API registrados dentro de cada proyecto, sirviendo como punto de enlace entre la capa de backend y la generación automatizada de artefactos de prueba. Su propósito es centralizar la definición de rutas, métodos y entidades bajo prueba, integrando los datos requeridos para la creación posterior de casos de prueba, esquemas y clientes.

En la siguiente figura se observa la interfaz del módulo, donde cada endpoint se representa mediante una tarjeta con información clave y acciones disponibles.

Figura 23. Vista general del módulo Endpoints de Omega Testing



Nota. Elaborado por el autor.

Visualización general

Cada tarjeta de endpoint muestra los siguientes elementos:

- **Estado**: indica el progreso del análisis o generación del artefacto (*pending*, *analyzing*, *generating*, *ready* o *failed*).
- **Nombre y ruta (path)**: identifican el recurso de la API.
- **Métodos HTTP**: se visualizan como etiquetas de color (chips) correspondientes a *GET*, *POST*, *PUT*, *PATCH* o *DELETE*.
- **Contexto del endpoint**: muestra el nombre del proyecto, la sección funcional y la entidad asociada.

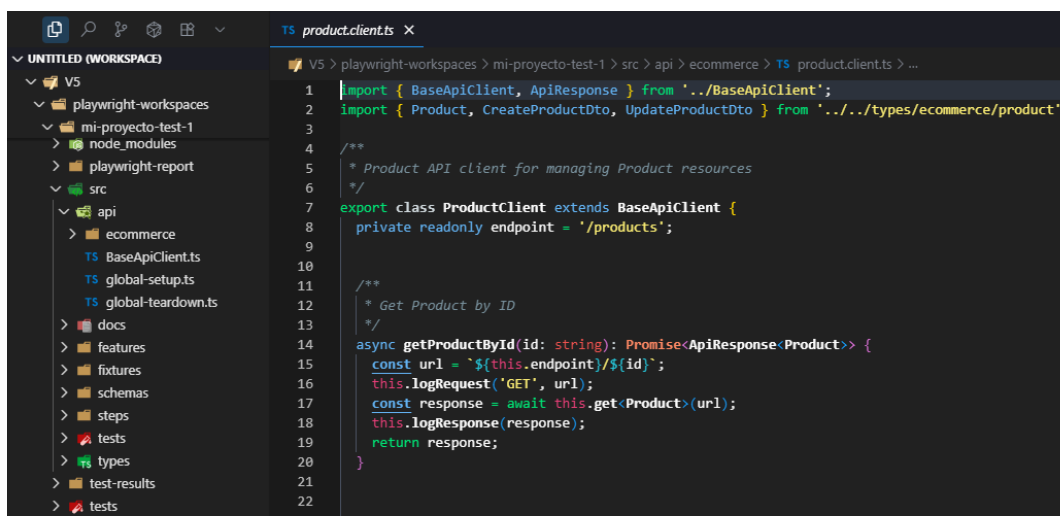
Los filtros permiten buscar endpoints por nombre o ruta, así como filtrar por proyecto, método HTTP o estado actual, facilitando la gestión de múltiples recursos dentro de un mismo entorno.

Registro de endpoints

La acción **Register Endpoint** abre el diálogo de registro donde se definen los parámetros esenciales del recurso.

- **Campos principales**:
    - *projectId*: proyecto asociado.
    - *section*: área funcional o módulo.
    - *entityName*: entidad principal del recurso.
    - *path*: ruta del endpoint.
    - *name*: nombre descriptivo.
    - *description*: campo opcional.

Figura 24. Ejemplo de proyecto de testing generado con artefactos



Nota. Elaborado por el autor.

- **Configuración de métodos**:
    - Se seleccionan los métodos HTTP requeridos (*GET*, *POST*, *PUT*, *PATCH*, *DELETE*).
    - Se indica si requieren autenticación (*requiresAuth*).

  - Para los métodos con cuerpo de solicitud (*POST*, *PUT*, *PATCH*), se define la estructura de campos mediante *requestBodyDefinition*, especificando: *name*, *type* (*string*, *number*, *boolean*, *object*, *array*), *required*, *example* y *validations*.
  - Las validaciones soportadas incluyen: Para *string*: *minLength* y *maxLength*. Para *number*: *minimum* y *maximum*. Para *boolean*: *default*.
  - El sistema permite agregar o eliminar campos individualmente, o bien, realizar inserciones masivas mediante JSON.
- Al guardar, el endpoint queda vinculado al proyecto seleccionado y su estado inicial pasa a *pending* o *analyzing* según el flujo interno de análisis del backend.

Edición y visualización de detalles

Cada endpoint puede inspeccionarse mediante el diálogo **View & Edit**, que organiza la información en pestañas:

- **Basic Info**: muestra el proyecto, la URL base completa (baseUrl + basePath + path), la sección y la entidad asociada.
- **Methods**: lista de métodos registrados junto con su definición de cuerpo de solicitud y validaciones.
- **Generated Artifacts**: referencia los archivos generados a partir del endpoint (feature, steps, fixture, schema, types y client).
- **Analysis Results**: detalla, por método, los resultados del análisis de respuesta, incluyendo códigos de estado, tipo de contenido y campos esperados en la respuesta (*responseFields[]*).
- **Timestamps**: registra las fechas de creación y actualización del endpoint.

Durante la edición, el usuario puede modificar rutas, métodos, configuraciones de autenticación y validaciones, con la posibilidad de revertir cambios antes de guardar.

Acciones complementarias

- **Re-analyze**: recalcula los resultados de análisis y actualiza los artefactos generados, utilizando la lógica de validación y parsing del backend.
- **Generate Tests**: genera automáticamente los casos de prueba asociados al endpoint, redirigiendo al módulo *Test Cases* para su edición o ejecución.

- **Delete Endpoint**: elimina el recurso mediante la API, retirando sus artefactos y referencias del proyecto.

Estados y retroalimentación

Los endpoints atraviesan distintas fases durante su análisis y generación:

- *Pending*: registro inicial sin procesamiento.
- *Analyzing*: en revisión por la capa de IA y el backend.
- *Generating*: en proceso de creación de artefactos.
- *Ready*: endpoint completo y funcional.
- *Failed*: error durante el análisis o generación.

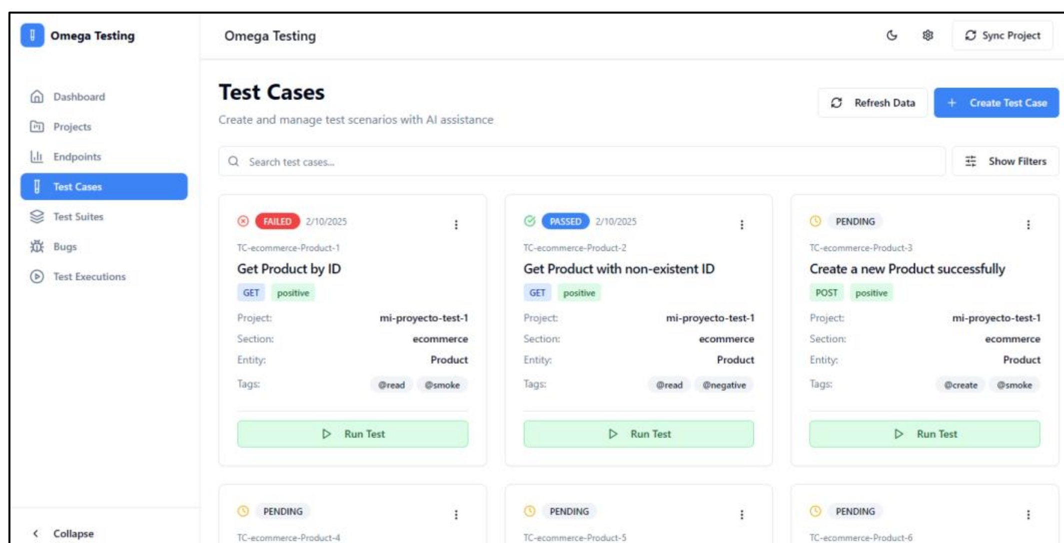
El sistema ofrece retroalimentación constante mediante *toasts* y mensajes visuales coherentes con los códigos de respuesta del backend, además de validaciones que garantizan la consistencia de los datos ingresados.

5.3.2.4. Test Cases

El módulo *Test Cases* es el núcleo operativo donde se definen, editan, ejecutan y gestionan los escenarios de prueba generados por el sistema o creados manualmente. Cada caso está vinculado a un endpoint y se estructura siguiendo el formato BDD (Behavior Driven Development), integrando lógica de negocio, pasos de prueba y resultados esperados.

En la siguiente figura se observa la vista principal del módulo, donde cada caso se muestra como una tarjeta independiente con su estado, tipo de prueba y contexto asociado.

Figura 25. Vista general del módulo Test Cases de Omega Testing



Nota. Elaborado por el autor.

Visualización y estructura de la interfaz

Cada tarjeta de caso de prueba muestra:

- **Identificadores**: *testCaseId* y nombre descriptivo.
- **Contexto**: *section*, *entityName*, método HTTP y tipo de prueba (*testType*).
- **Estado de ejecución**: *pending*, *passed* o *failed*, acompañado de icono e indicador de color.
- **Última ejecución**: fecha, resultado y botón *Run Test*.
- **Etiquetas (tags)**: como *@read*, *@create*, *@negative* o *@smoke*, que agrupan pruebas según su propósito o naturaleza.

Los filtros disponibles permiten buscar casos por nombre o identificador, así como aplicar filtros por proyecto, sección, entidad, método HTTP o estado.

Modos de creación de casos de prueba

La creación de nuevos casos se realiza desde el botón **Create Test Case**, que abre un diálogo con tres modos complementarios:

a) Modo Predefined

Permite crear un caso con pasos ya existentes, reutilizando componentes definidos previamente en el backend.

- Se seleccionan los campos principales: *Project*, *Section*, *Entity*, *Method* y *Test Type*.
- En la pestaña *Scenario*, se eligen los pasos a partir de una lista de *steps* registrados.
- Se añaden la descripción y las etiquetas, y al guardar se genera un nuevo registro vinculado al proyecto.

b) Modo AI

Este modo permite generar escenarios y pasos de prueba automáticamente mediante el módulo de inteligencia artificial del sistema.

- El usuario completa los campos *Project*, *Section* y *Entity*, y describe brevemente el requerimiento funcional en el campo *Requirements*.
- Al presionar **Generate with AI**, el sistema interpreta la descripción y genera automáticamente un escenario Gherkin coherente, con pasos estructurados según el patrón *Given–When–Then*.
- El resultado se muestra en la pestaña *Generated*, desde donde puede revisarse, editarse y guardarse como nuevo caso.
- La lógica de generación se apoya en los servicios internos del backend, que envían la solicitud a OpenAI y procesan la respuesta para transformarla en código y artefactos compatibles.

Figura 26. Vista del modo AI para casos de prueba de Omega Testing

Create Test Case

Predefined | AI | Suggested

Create scenarios and new steps with AI

Project: mi-proyecto-test-1
Section: ecommerce
Entity: Product

Requirements

Create Test Cases with extreme values

Generate with AI

Requirements | Generated

Enter your requirements above and click "Generate with AI" to create test scenarios.

Cancel

Nota. Elaborado por el autor.

c) Modo Suggested

Utiliza el mismo motor de IA, orientado a la generación de *sugerencias* de casos basadas en los endpoints existentes.

- El usuario define *Project*, *Section* y *Entity*, y describe el tipo de pruebas deseadas.
- Al presionar **Generate Suggestions**, el sistema propone posibles escenarios de prueba agrupados por comportamiento esperado o borde de validación.
- Estas sugerencias se guardan y tienen un botón para crear con IA el caso de prueba sugerido.

Edición y mantenimiento

Los casos existentes pueden abrirse desde su tarjeta o mediante el diálogo integral de detalles. Allí se permite:

- Modificar el nombre, descripción, tipo de prueba o etiquetas.
- Editar el escenario completo dentro del *ScenarioEditor*.
- Agregar, reordenar o eliminar pasos manteniendo la estructura lógica del caso.
- Actualizar el método HTTP y conservar las asociaciones con las ejecuciones previas.

Eliminar Test Case: requiere confirmación mediante el *testCaseId*; elimina el registro y desvincula las evidencias asociadas.

Ejecución y seguimiento

Al ejecutar un caso mediante la acción **Run Test**, se crea una nueva instancia de ejecución que se monitorea en el módulo *Test Executions*.

- El progreso se actualiza en tiempo real mediante *Server-Sent Events (SSE)*.
- Al finalizar, se registran métricas como tiempo promedio, estado final y resultados de validación.

Persistencia y trazabilidad

Los casos de prueba se escriben en el archivo feature de la entidad seleccionada, en formato BDD como se muestra en la siguiente figura:

Figura 27. Ejemplo de escenarios BDD en el archivo feature generado

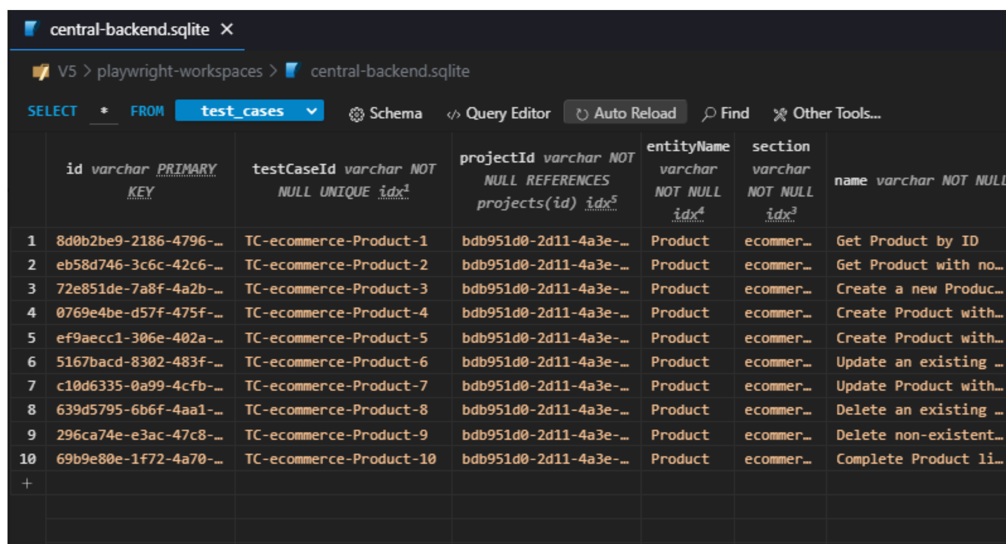
```
@ecommerce @Product @smoke
Feature: Product API
  As a user of the mi-proyecto-test-1 API
  I want to manage Products
  So that I can perform Product operations

  Background:
    Given the API is available

  @TC-ecommerce-Product-1
  @read @smoke
  Scenario: Get Product by ID
    Given a Product exists in the system
    When I get the Product by ID
    Then I should get the Product details
    And I should receive a 202 status code
    And the response should contain valid Product data

  @TC-ecommerce-Product-2
  @read @negative
  Scenario: Get Product with non-existent ID
    When I get a Product with ID "non-existent-id"
    Then I should receive a not found error
    And I should receive a 404 status code
```

Nota. Elaborado por el autor.

Además cada caso queda almacenado en la base de datos SQLite central, donde se conservan sus metadatos esenciales:

- *id*, *testCaseId*, *projectId*, *section*, *entityName*, *name*, *description*, y *estado*. Esta información permite mantener coherencia entre los casos visibles en la interfaz y los artefactos generados dentro del workspace Playwright.

Figura 28. Base de datos SQLite con los casos de prueba



	id varchar PRIMARY KEY	testCaseId varchar NOT NULL UNIQUE idx1	projectId varchar NOT NULL REFERENCES projects(id) idx5	entityName varchar NOT NULL idx4	section varchar NOT NULL idx3	name varchar NOT NULL
1	8d0b2be9-2186-4796-…	TC-ecommerce-Product-1	bdb951d0-2d11-4a3e-…	Product	ecommer…	Get Product by ID
2	eb58d746-3c6c-42c6-…	TC-ecommerce-Product-2	bdb951d0-2d11-4a3e-…	Product	ecommer…	Get Product with no…
3	72e851de-7a8f-4a2b-…	TC-ecommerce-Product-3	bdb951d0-2d11-4a3e-…	Product	ecommer…	Create a new Produc…
4	0769e4be-d57f-475f-…	TC-ecommerce-Product-4	bdb951d0-2d11-4a3e-…	Product	ecommer…	Create Product with…
5	ef9aecc1-306e-402a-…	TC-ecommerce-Product-5	bdb951d0-2d11-4a3e-…	Product	ecommer…	Create Product with…
6	5167bacd-8302-483f-…	TC-ecommerce-Product-6	bdb951d0-2d11-4a3e-…	Product	ecommer…	Update an existing …
7	c10d6335-0a99-4cfb-…	TC-ecommerce-Product-7	bdb951d0-2d11-4a3e-…	Product	ecommer…	Update Product with…
8	639d5795-6b6f-4aa1-…	TC-ecommerce-Product-8	bdb951d0-2d11-4a3e-…	Product	ecommer…	Delete an existing …
9	296ca74e-e3ac-47c8-…	TC-ecommerce-Product-9	bdb951d0-2d11-4a3e-…	Product	ecommer…	Delete non-existent…
10	69b9e80e-1f72-4a70-…	TC-ecommerce-Product-10	bdb951d0-2d11-4a3e-…	Product	ecommer…	Complete Product li…

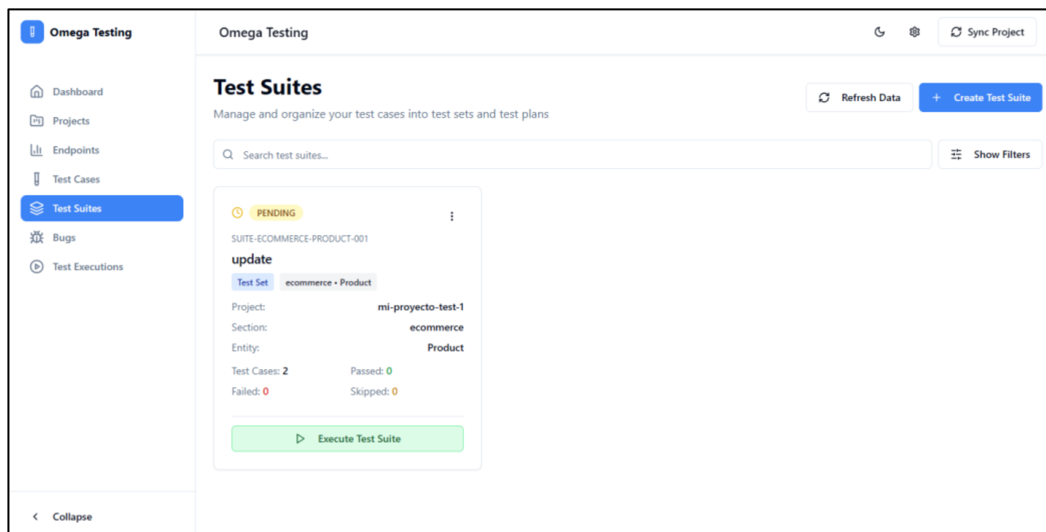
Nota. Elaborado por el autor.

5.3.2.5. Test Suites

El módulo *Test Suites* permite agrupar casos de prueba individuales en conjuntos o planes de ejecución más amplios, facilitando la organización, ejecución y seguimiento de pruebas integradas. Su función principal es consolidar múltiples escenarios en una estructura jerárquica que refleje el flujo completo de validaciones funcionales dentro de un proyecto.

En la siguiente figura se observa la vista principal del módulo, donde cada *suite* aparece representada por una tarjeta con sus indicadores clave y opciones de gestión.

Figura 29. Vista general del módulo Test Suites de Omega Testing



Nota. Elaborado por el autor.

Visualización general

Cada tarjeta de suite incluye los siguientes elementos:

- **Identificadores**: *suiteId* y *name*, que permiten identificar de forma única la suite.
- **Contexto**: *projectName*, *section* y *entity* asociados.
- **KPIs principales**: número total de casos (*totalTestCases*), casos aprobados, fallidos y omitidos, junto con el tiempo total de ejecución.
- **Estado de ejecución**: indicador visual (*pending*, *running*, *passed*, *failed* o *skipped*) con icono y color.

- **Acciones rápidas**: *View*, *Edit*, *Run* y *Delete*, accesibles desde el menú contextual de cada tarjeta.

La interfaz permite además realizar búsquedas textuales y aplicar filtros por proyecto, tipo de suite (*test_set* o *test_plan*), estado, sección, entidad o etiquetas.

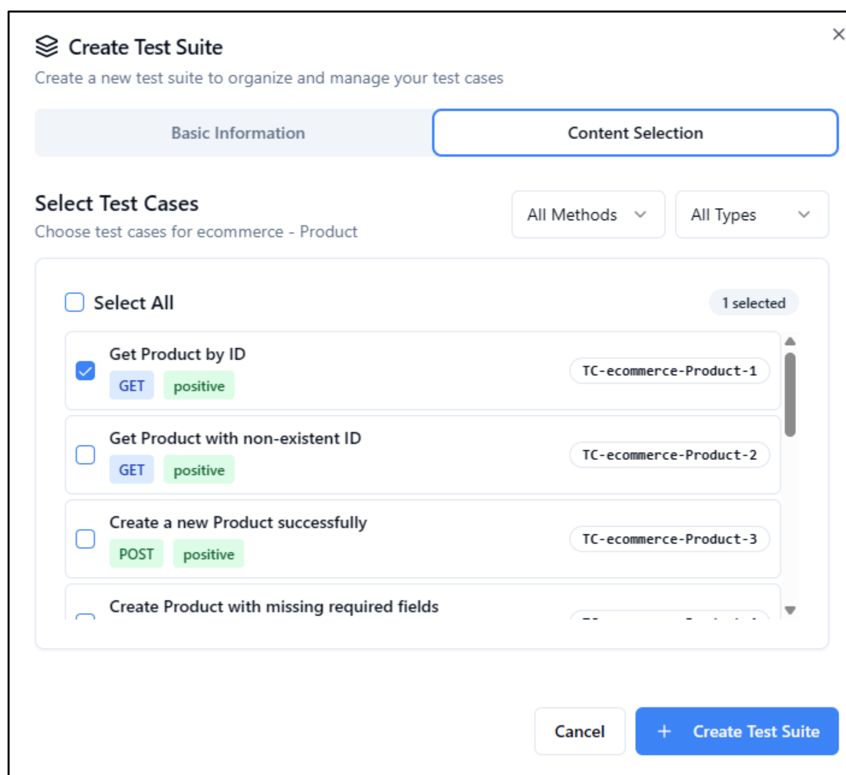
Creación de suites de prueba

Desde el botón **Create Test Suite**, el usuario puede generar nuevas agrupaciones mediante un diálogo estructurado en dos pestañas: *Basic Information* y *Content Selection*.

- **Basic Information**:
    - *Project*: selección del proyecto al cual pertenecerá la suite.
    - *Name* y *Description*: definen el propósito y alcance del conjunto.
    - *Type*: permite especificar si la suite es un test_set (agrupación básica de test cases) o un test_plan (nivel superior que puede contener múltiples test sets).
    - Campos adicionales: *section*, *entity*, *environment* y *tags*.
- **Content Selection**:
    - Muestra la lista de casos de prueba asociados al proyecto, filtrables por método o tipo de prueba.
    - Permite seleccionar casos individuales o marcar *Select All*.
    - Se incluyen contadores dinámicos que muestran el número de casos seleccionados.

Al confirmar la creación, la suite se persiste mediante el servicio *testSuiteService.createTestSuite*, que registra su composición y estado inicial (*pending*).

Figura 30. Interfaz del diálogo Create Test Suite de Omega Testing



Nota. Elaborado por el autor.

Edición y mantenimiento

Las suites existentes pueden modificarse desde la tarjeta o el detalle integral (*TestSuiteDetailsDialog*).

- **Edición básica**: permite cambiar nombre, descripción, tipo y etiquetas.
- **Edición de composición**: mediante el diálogo *TestSuiteEditDialog*, el usuario puede añadir o eliminar casos de prueba, aplicar filtros por sección o entidad y guardar los cambios.
- Los datos se actualizan en la interfaz de manera inmediata tras guardar.

Ejecución de suites

La acción **Execute Test Suite** lanza la ejecución del conjunto seleccionado, coordinando la ejecución de los casos incluidos.

- Los parámetros de ejecución son configurables, admitiendo opciones como:
    - *method* (filtrado por tipo de método HTTP).

  - *testType* (*positive*, *negative* o *all*).
  - *parallel* (ejecución en paralelo).
  - *timeout*, *retries* y *environment* (*local*, *staging* o *production*).
  - *verbose* (nivel de detalle del log).
- Al iniciar la ejecución, se genera un *executionId* que puede consultarse en el módulo *Test Executions*.
- La tarjeta de la suite se actualiza automáticamente con los resultados finales, reflejando el número de pruebas aprobadas, fallidas u omitidas.

Eliminación e integración con ejecuciones

El sistema permite eliminar una suite previa confirmación mediante su *suiteId*. La eliminación no afecta los casos de prueba individuales, que permanecen en el sistema. Cada tarjeta incluye además un acceso directo a la última ejecución, mostrando los KPIs consolidados y el enlace al registro completo dentro de *Test Executions*.

Estados y retroalimentación

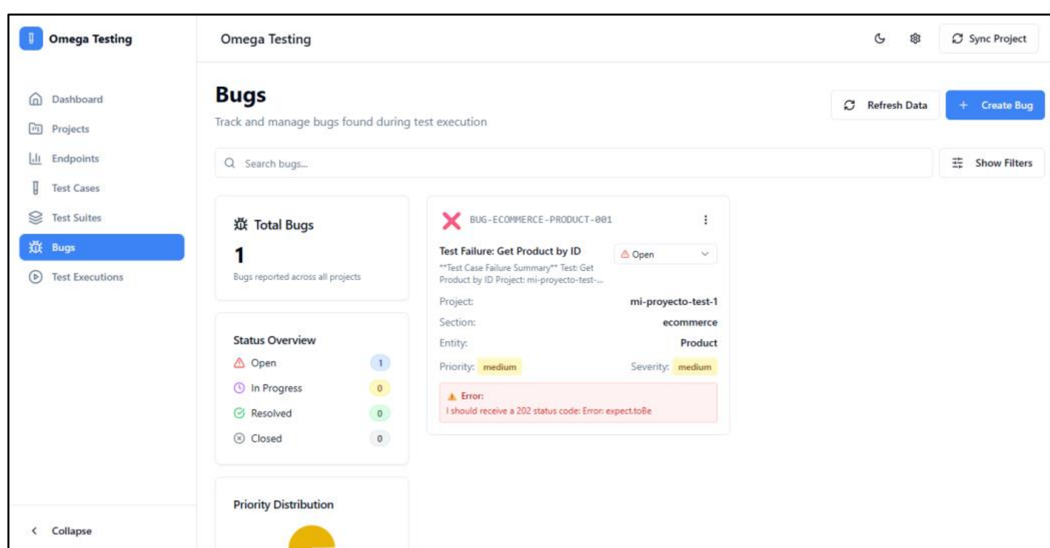
Los procesos de creación, edición y ejecución están acompañados por mensajes visuales (*toasts*) que confirman cada acción o notifican errores. Los formularios validan los campos obligatorios y exigen una selección mínima de casos antes de permitir la creación.

5.3.2.6. Bugs

El módulo *Bugs* centraliza la gestión de incidencias detectadas durante la ejecución de pruebas, permitiendo registrar, clasificar y dar seguimiento a los errores de manera estructurada. Su diseño replica el flujo de herramientas de seguimiento profesional como Jira, aunque integrado directamente en el ecosistema de Omega Testing, garantizando trazabilidad entre casos, ejecuciones y resultados.

En la siguiente figura se aprecia la vista principal del módulo, donde se listan los errores reportados junto con sus indicadores de estado, prioridad y severidad.

Figura 31. Vista general del módulo Bugs de Omega Testing



Nota. Elaborado por el autor.

Visualización y estructura

Cada tarjeta o registro de bug muestra:

- **Identificadores**: *bugId* (formato BUG–{section}–{entity}–{número}) y título descriptivo del error.
- **Estado**: *Open*, *In Progress*, *Resolved* o *Closed*, acompañado de un indicador visual.
- **Contexto de prueba**: proyecto, sección, entidad, método HTTP y endpoint involucrado.
- **Vínculos directos**: *testCaseId* y *executionId*, que conectan el bug con la prueba y la ejecución donde ocurrió la falla.

- **Severidad y prioridad**: ambas clasificadas en niveles (*low*, *medium*, *high*), visibles mediante etiquetas de color.
- **Detalles del error**: resumen del fallo obtenido (por ejemplo, “Expected 202 status code, received 200”).

Además, la interfaz incluye paneles de resumen estadístico:

- *Total Bugs*: número total de incidencias.
- *Status Overview*: distribución por estado operativo.
- *Priority, Severity y Type Distribution*: gráficos circulares que ilustran la composición de errores activos según criticidad y naturaleza.

Registro y creación de errores

La creación de un nuevo bug se realiza mediante el diálogo **Create Bug**, el cual puede iniciarse de forma manual o automáticamente a partir de una ejecución fallida.

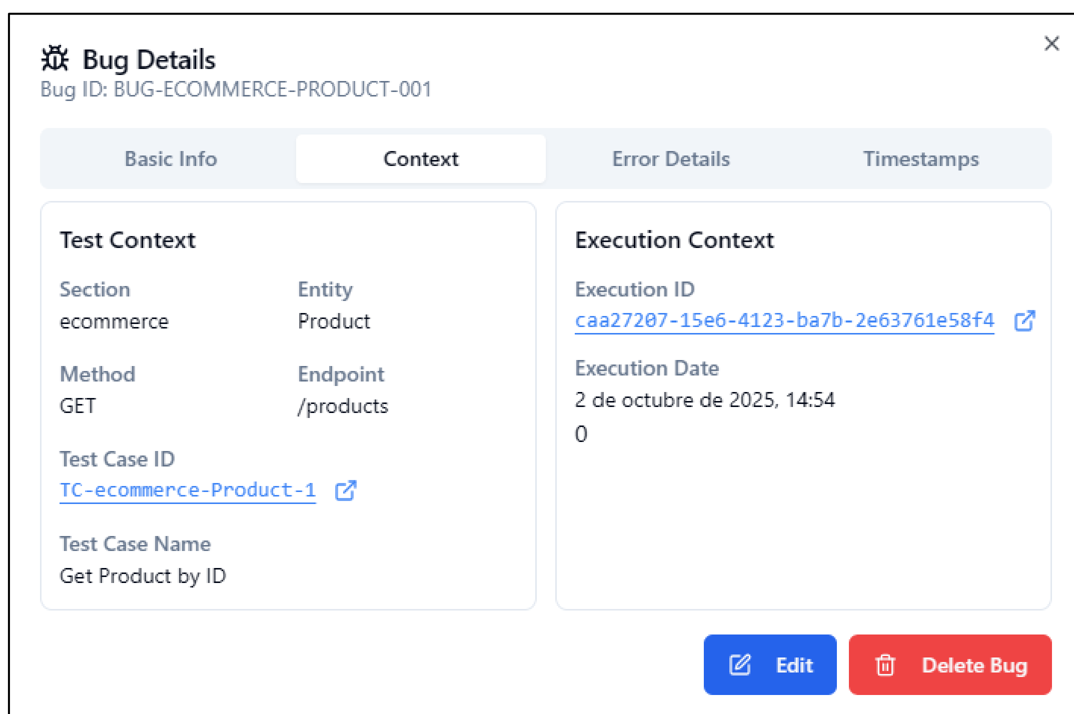
- **Campos principales**:
    - *Title* y *Description*: resumen y detalle del problema.
    - *Type*: clasificación del error (por ejemplo, funcional, validación, comunicación, entorno).
    - *Severity* y *Priority*: impacto y urgencia asignados al defecto.
    - *Environment*: entorno en que se produjo (local, staging o production).
    - *Contexto*: *projectId*, *section*, *entity*, *method* y *endpoint*.
- **Vinculación opcional**:
    - Se puede asociar el bug a un *testCaseId* o *executionId*, generando un enlace directo con la ejecución o caso de prueba responsable del hallazgo.
- **Prefill automático**:
    - Si el bug se genera desde una ejecución fallida, el sistema precarga el contexto del test y los mensajes de error devueltos por el backend, optimizando la trazabilidad del reporte.

Detalle y seguimiento del error

El diálogo **Bug Details** organiza la información en pestañas:

- **Basic Info**: título, descripción, tipo, prioridad, severidad y estado.
- **Context**: muestra la sección, entidad, método y endpoint asociados, así como el *testCaseId* y *executionId* vinculados.
- **Error Details**: detalla el mensaje de error, el resultado esperado y la respuesta real capturada por el sistema.
- **Timestamps**: fechas de creación, actualización y resolución.

Figura 32. Detalle contextual de un bug de Omega Testing



Nota. Elaborado por el autor.

Edición, actualización y control de estado

Los errores pueden modificarse en cualquier momento desde la tarjeta o el detalle:

- Actualización de *title*, *description*, *type*, *priority*, *severity* o *status*.
- Cambio de estado directo mediante un selector contextual (por ejemplo, de *Open* a *Resolved*).
- Eliminación controlada con confirmación por *bugId* (acción irreversible).

El sistema también soporta ordenamiento y filtrado por texto, proyecto, tipo, severidad, prioridad, estado o entorno, además de paginación dinámica para la navegación entre registros.

Integración con otros módulos

Cada bug mantiene una relación directa con los módulos *Test Cases* y *Test Executions*:

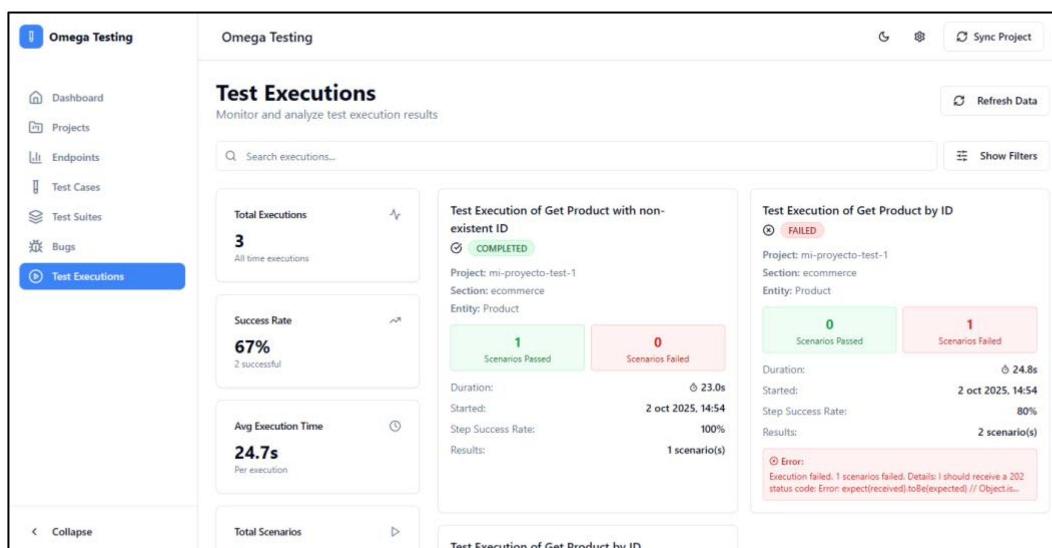
- Desde un *test case* o una ejecución fallida puede generarse un bug de forma inmediata.
- Los enlaces bidireccionales permiten abrir la ejecución asociada para analizar la causa del fallo o reejecutar la prueba tras su corrección.

5.3.2.7. Test Executions

El módulo *Test Executions* constituye el componente encargado de monitorear y analizar los resultados de las ejecuciones de prueba en tiempo real. Desde esta vista, el usuario puede revisar métricas de rendimiento, tasas de éxito, duración promedio y resultados individuales de cada escenario, centralizando la trazabilidad entre casos, suites y bugs.

En la siguiente figura se observa la interfaz principal del módulo, donde se listan las ejecuciones realizadas con sus métricas y estados correspondientes.

Figura 33. Vista general del módulo Test Executions de Omega Testing



Nota. Elaborado por el autor.

Visualización general

Cada tarjeta de ejecución presenta información clave para el análisis de resultados:

- **Identificadores y contexto**: *executionId*, *projectName*, *section* y *entityName*.
- **Estado**: *pending*, *running*, *completed*, *failed* o *cancelled*, representado mediante badges de color.
- **Métricas principales (KPIs)**:
  - *Total Scenarios*, *Scenarios Passed*, *Scenarios Failed*.
  - *Execution Time* y *Step Success Rate*.

- **Timestamps**: *startedAt* y *completedAt*, para medir la duración total.
- **Relaciones**: vínculo directo con el *testCaseId* o *testSuiteId* que originó la ejecución.
- **Feedback de error**: resumen del fallo en caso de resultado negativo, mostrando el mensaje y el código devuelto por el servidor.

La parte superior del módulo muestra indicadores globales:

- *Total Executions*: cantidad acumulada de ejecuciones.
- *Success Rate*: porcentaje de pruebas exitosas.
- *Average Execution Time*: duración promedio por ejecución.
- *Total Scenarios*: número consolidado de escenarios validados.

Detalle de ejecución

Al seleccionar una tarjeta, se abre el diálogo **Test Execution Details**, que organiza la información en tres pestañas principales:

1. **Execution Info**
    - Muestra el resumen de la ejecución, su estado, entidad, sección y escenario ejecutado.
    - Incluye estadísticas consolidadas: *Total Scenarios*, *Passed*, *Failed*, *Total Steps*, *Steps Passed*, *Steps Failed* y *Success Rate*.
    - Se detallan las marcas de tiempo (*Started At* y *Completed At*) y los metadatos de la ejecución (entorno, parámetros y configuración).
2. **Test Results**
    - Lista los escenarios ejecutados con su estado (*Completed* o *Failed*), duración y método asociado.
    - En ejecuciones con *Scenario Outlines*, muestra los *examples* individuales con sus resultados.
    - Incluye enlaces de navegación directa al *Test Case* asociado.
3. **Steps Details**

- Presenta la tabla de pasos ejecutados, mostrando: *Step Name*, *Type* (*Hook* o *Step*), *Status*, *Duration*, *Timestamp* y *Error Message*.
- Permite expandir los errores largos para inspeccionar el detalle de las excepciones o respuestas HTTP.
- Cada fila corresponde a un paso dentro del escenario, registrando su resultado y tiempo exacto de ejecución.

Figura 34. Vista detallada de una ejecución de pruebas de Omega Testing

Nota. Elaborado por el autor.

Actualización en tiempo real

El módulo está conectado a un servicio *Server-Sent Events (SSE)* que permite observar el progreso de las ejecuciones en vivo.

- La aplicación recibe eventos con los estados *started*, *progress*, *completed* o *failed*.

- La interfaz se actualiza automáticamente sin recarga, reflejando los resultados y estadísticas a medida que los pasos se completan.
- En caso de error, los eventos incluyen el mensaje devuelto por el runner o el assertion handler.

Acciones disponibles

- **View Details**: abre el diálogo *Test Execution Details* para inspeccionar resultados, pasos y metadatos.
- **Delete Execution**: acción disponible mediante confirmación por *executionId*, sin afectar los casos o suites asociados.

Filtros y orden

La vista admite filtrado y orden dinámico:

- **Filtros**: *status*, *entityName*, *method*, *testType*, *projectId*, *section*, *dateFrom* y *dateTo*.
- **Búsqueda textual**: por *executionId* o nombre del escenario.
- **Ordenación**: por *startedAt*, *completedAt* o *executionTime* (ASC/DESC).
- **Paginación**: facilita la exploración de grandes volúmenes de resultados históricos.

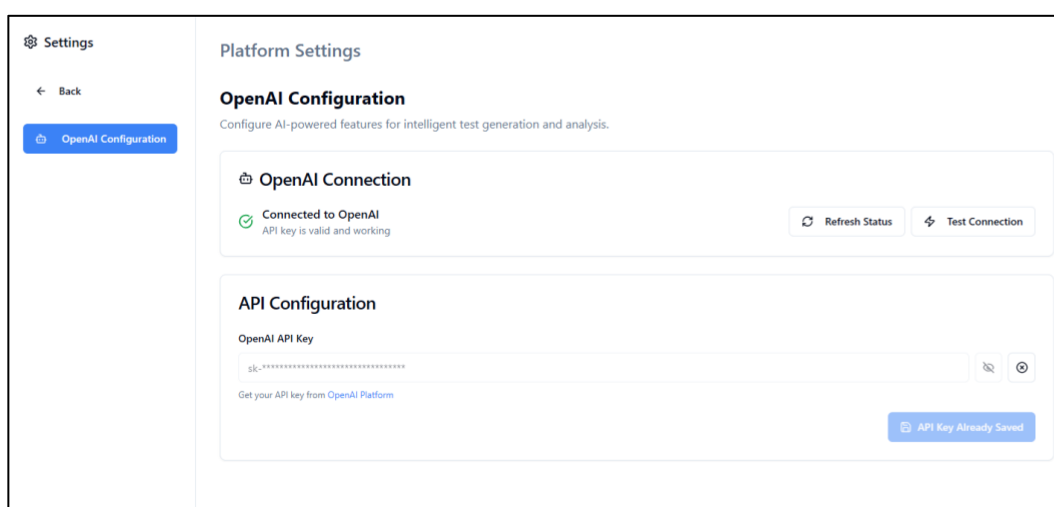
El módulo Test Executions cierra el ciclo de control de calidad dentro de Omega Testing, proporcionando una vista integral del rendimiento de las pruebas, la estabilidad de las APIs y la eficacia de los scripts generados. Su integración con SSE y su trazabilidad completa permiten un monitoreo continuo y preciso de la salud del sistema de pruebas automatizadas.

5.3.2.8. Settings — OpenAI Configuration

El módulo Settings – OpenAI Configuration proporciona el panel de administración donde se configuran las credenciales y la conexión con la API de OpenAI, necesarias para habilitar las funciones inteligentes de generación y análisis de pruebas dentro de Omega Testing.

En la figura siguiente se muestra la interfaz principal, en la que se observan las secciones de conexión y configuración del API Key.

Figura 35. Vista del panel de configuración de OpenAI de Omega Testing



Nota. Elaborado por el autor.

Visualización general

El módulo se compone de dos bloques principales:

- **OpenAI Connection**: Muestra el estado actual de la conexión con la API.
  - *Connected to OpenAI*: indica que la clave configurada es válida y la comunicación con el servicio está activa.
  - *Not Configured / Connection Failed*: aparece cuando no se ha ingresado una clave o cuando la validación falla.
  - Incluye acciones rápidas: **Refresh Status**: actualiza el estado de conexión en tiempo real. **Test Connection**: ejecuta una prueba directa al backend para confirmar la validez de la API Key guardada.

- **API Configuration**: Contiene el campo de ingreso para la clave de acceso al servicio.
    - **OpenAI API Key**: campo de texto protegido donde se introduce la clave con formato estándar (*sk*-***********...*).
    - **Save API Key**: botón que envía la clave al backend para su almacenamiento seguro. La clave no se guarda en el cliente, cumpliendo con buenas prácticas de seguridad.
    - **Delete / Clear Key**: permite eliminar la clave almacenada para restablecer la configuración.
    - Enlace directo a la página de obtención de claves: *Get your API key from OpenAI Platform*.

Estados y retroalimentación

La interfaz utiliza *toasts* informativos para notificar los resultados de las acciones:

- Éxito en la conexión o guardado de clave.
- Error al validar la API Key o al comunicarse con el backend.
- Estado de sincronización actualizado después de ejecutar *Refresh Status*.

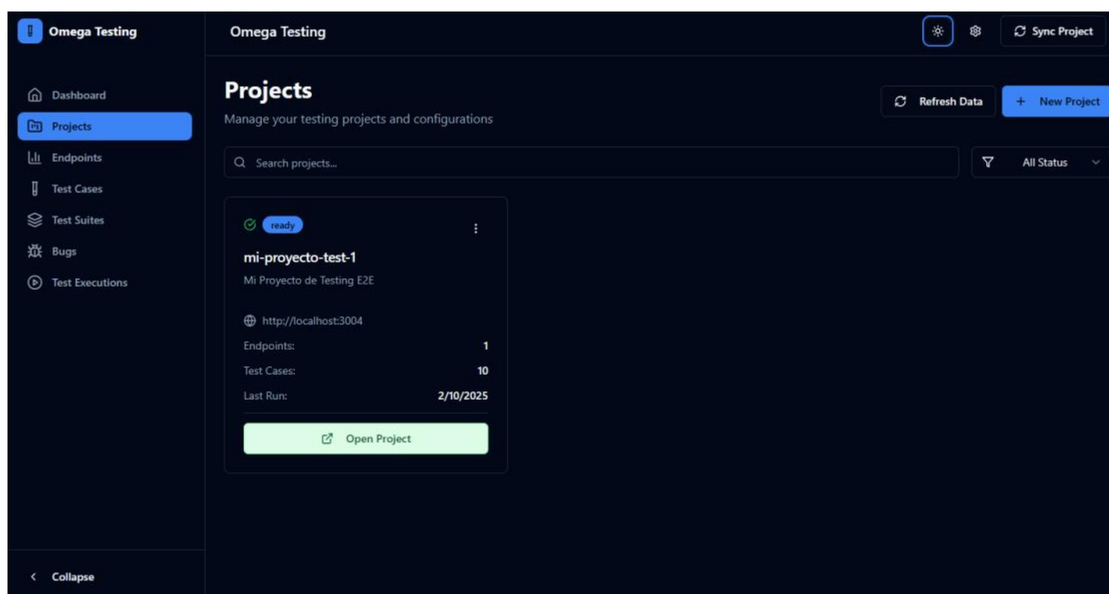
El indicador visual de conexión se actualiza dinámicamente, garantizando transparencia al usuario sobre la disponibilidad de los servicios de inteligencia artificial.

5.3.3. Accesibilidad y experiencia de usuario (UX/UI)

Modo oscuro y contraste adaptable

El cliente de Omega Testing fue diseñado bajo criterios sólidos de accesibilidad, usabilidad y consistencia visual, priorizando la interacción fluida y la compatibilidad con diversas preferencias del usuario y dispositivos. Estas prácticas garantizan que la interfaz sea perceptible, operable y comprensible para la mayor cantidad posible de usuarios, incluidos aquellos que utilizan ayudas técnicas.

Figura 36. Vista del módulo Projects en modo oscuro de de Omega Testing



Nota. Elaborado por el autor.

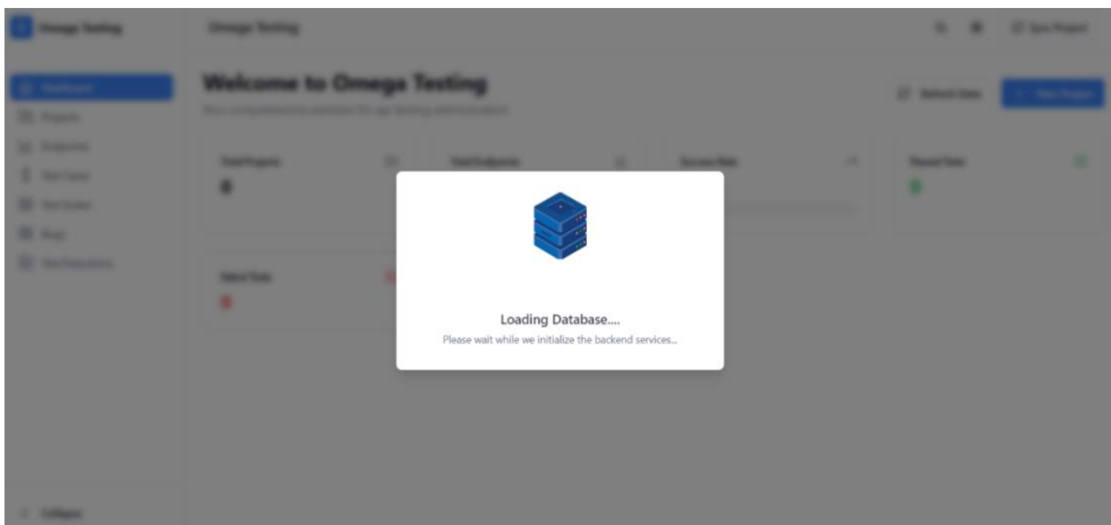
Preferencias del sistema y control de temas

La aplicación detecta automáticamente la preferencia del sistema operativo mediante la directiva prefers-color-scheme y permite alternar manualmente entre los modos oscuro y claro desde la barra superior. Esta preferencia se guarda en el almacenamiento local del navegador, de modo que el entorno visual elegido persiste entre sesiones. El uso de un sistema de **tokens de color (CSS variables)** asegura contraste suficiente entre texto, fondo, bordes e iconografía, cumpliendo con las recomendaciones WCAG para legibilidad en ambos temas.

Diálogos y modales accesibles

Los cuadros de diálogo (como los de creación de proyectos, configuración o detalle de prueba) están implementados con **componentes accesibles tipo Radix**, que gestionan automáticamente el foco al abrirse, permiten cierre con la tecla *Escape* y bloquean la interacción con el fondo mediante un overlay modal. Cada modal define **roles y atributos ARIA** correctos, mientras que el botón de cierre incluye texto oculto (*sr-only*) para lectores de pantalla y un estilo de foco visible.

Figura 37. Pantalla de carga inicial de Omega Testing



Nota. Elaborado por el autor.

Retroalimentación visual y estados de carga

Cuando el sistema espera la conexión al backend, se muestra un overlay de mensajes de carga a pantalla completa, con fondo desenfocado y alto contraste, bloqueando la interacción hasta que el servicio esté disponible. Este patrón previene errores de interacción temprana y orienta al usuario mediante mensajes descriptivos y temporales. Además, los **skeletons** (animaciones tipo “pulse”) indican contenido en tránsito, mientras que los **toasts** proveen mensajes de confirmación o error sin bloquear la navegación.

Así mismo hay animaciones de carga cuando se ejecutan acciones asíncronas como al crear casos de prueba con IA, debido a que este proceso demora, se muestra la animación de carga y al tener la respuesta se notifica al usuario mediante un toast y el cambio de enfoque hacia el contenedor que presenta la respuesta de la IA. Así mismo, al ejecutar

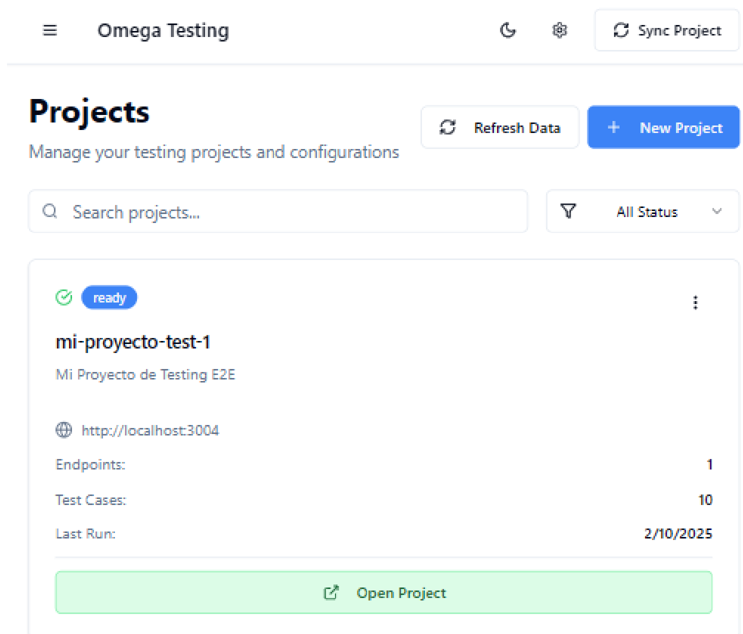
casos de prueba se muestra una animación de carga con espera dinámica que usa el canal de SSE para saber el status de la ejecución y mostrarlo en tiempo real.

Navegación y diseño responsivo

La interfaz está construida sobre una **grilla responsiva** que adapta la disposición de elementos a distintos tamaños de pantalla.

- En escritorio, la barra lateral es fija y permite colapsar o expandir el menú, conservando el estado entre sesiones.
- En dispositivos móviles, se transforma en un panel deslizante con overlay para garantizar accesibilidad táctil.
- La barra superior se mantiene en modo "sticky" para mantener accesibles los controles de tema, sincronización y ajustes durante el desplazamiento.

Figura 38. Diseño responsivo de Omega Testing en otro ancho de pantalla



Nota. Elaborado por el autor.

Interacción mediante teclado

Todos los componentes interactivos —menús, pestañas, selectores, diálogos— admiten **navegación por teclado** (tabulación y flechas) y visibilidad de foco (*focus ring*). Esto

facilita la interacción sin necesidad de ratón, manteniendo la orientación visual mediante resaltado activo.

Iconografía y semántica visual

La iconografía se apoya en símbolos reconocibles y en **badges semánticos** para estados (success, warning, error, info), garantizando legibilidad tanto en modo claro como oscuro. En modo colapsado, los íconos del menú conservan títulos emergentes para usuarios que dependen exclusivamente de ayudas visuales o lectores de pantalla.

Mensajería y ayudas contextuales

Los *empty states* incluyen textos explicativos y acciones sugeridas (“Create Project”, “Register Endpoint”) para orientar al usuario cuando no existen datos disponibles. Asimismo, se integró un proveedor global de *tooltips* para mostrar descripciones breves sobre botones o campos con funciones específicas, sin interferir con la navegación.

5.3.4. Cumplimiento de Requisitos no funcionales

El cumplimiento de los **Requisitos No Funcionales (RFN)** de Omega Testing se demuestra mediante una serie de evidencias técnicas representativas que demuestran la aplicación de buenas prácticas de ingeniería de software tanto en el backend como en el frontend. Estas evidencias reflejan la madurez del sistema en cuanto a eficiencia, observabilidad, confiabilidad, seguridad, mantenibilidad y usabilidad, consolidando una arquitectura sólida y escalable orientada a la automatización inteligente de pruebas.

Eficiencia en el uso de IA (RFN-01)

La eficiencia constituye un eje fundamental dentro del sistema, especialmente en los procesos asistidos por inteligencia artificial, donde el consumo de tokens y la gestión del contexto determinan el rendimiento y los costos de ejecución. En Omega Testing, el servicio test-case-generation.service.ts aplica una **estrategia de truncamiento y reciclaje de contexto**, basada en la propiedad:

truncation_strategy: { type: 'auto' }

Esto permite recortar automáticamente los fragmentos de conversación antiguos y mantener solo la información relevante para cada nueva generación. Esta técnica asegura que las ejecuciones de IA se mantengan ligeras y precisas, evitando la acumulación innecesaria de tokens.

A nivel de monitoreo, el servicio registra métricas detalladas de uso mediante un **sistema de logging** que mide *prompt tokens*, *completion tokens* y *total tokens used*, emitiendo advertencias cuando se superan umbrales definidos (por ejemplo, más de 3000 tokens de prompt o 2000 tokens totales). Esto permite identificar desvíos en tiempo real y optimizar el comportamiento de los prompts.

Finalmente, la eficiencia también se logra mediante la **reutilización de artefactos generados** (features, steps y fixtures). Antes de invocar una nueva generación, el sistema busca los artefactos existentes asociados al mismo projectId, section y entityName, evitando duplicaciones y aprovechando el conocimiento acumulado. Fragmentos del archivo test-case-generation.service.ts confirman la existencia de estas rutinas de truncamiento automático, logging de tokens y carga selectiva de artefactos previos.

Observabilidad del sistema (RFN-02)

La observabilidad permite que el sistema sea transparente, auditable y fácilmente diagnosticable, tanto para los desarrolladores como para los usuarios técnicos.

En el backend, Omega Testing implementa una **documentación dinámica y versionada mediante Swagger**, configurada en el archivo main.ts. Dicha configuración incluye etiquetas (tags) por dominio funcional, persistencia de autorización entre sesiones, ordenamiento alfabético de operaciones y visualización de la duración de cada request. Estas características facilitan la exploración y comprensión de la API central desde una interfaz web autodescriptiva, con control de versiones y endpoints categorizados.

Complementariamente, el sistema integra un canal de eventos en tiempo real mediante Server-Sent Events (SSE)**,** definido en el controlador test-execution.controller.ts. Esta funcionalidad transmite el progreso de las ejecuciones de prueba de forma continua, permitiendo que la interfaz del cliente actualice métricas, logs y estados de ejecución en vivo sin requerir recargas manuales.

Configuraciones registradas en los archivos main.ts (Swagger) y test-execution.controller.ts (SSE) demuestran la implementación efectiva de trazabilidad y monitoreo activo.

Confiabilidad operativa (RFN-03)

La confiabilidad se garantiza mediante políticas de control de errores, reintentos limitados y mecanismos de recuperación. El servicio de colas (queue.service.ts) establece **reintentos automáticos con límites definidos** (maxRetries = 2) y **timeouts configurables** de hasta 10 minutos por tarea. Si una generación no se completa dentro del tiempo estipulado o falla de forma irrecuperable, el sistema actualiza el estado del proyecto a *FAILED* dentro de una transacción controlada, asegurando la coherencia de los registros. Los mensajes de error se capturan mediante logging estructurado, lo que permite auditar los eventos de falla y depurar las causas.

En el frontend, la confiabilidad se refuerza con una **reconexión automática de SSE** implementada en el hook useExecutionEvents.ts. Si la conexión se interrumpe, el cliente inicia un temporizador de reconexión que restablece la comunicación cada cinco

segundos, manteniendo el seguimiento continuo del flujo de ejecución sin intervención del usuario.

Los fragmentos de queue.service.ts y useExecutionEvents.ts muestran las funciones Promise.race para controlar timeouts, actualización de estados a *FAILED* y reconexión progresiva de SSE en caso de interrupciones.

Seguridad y gestión de secretos (RFN-04)

En materia de seguridad, el backend emplea un conjunto de **middlewares de endurecimiento HTTP** como *Helmet* y *Compression*, junto con políticas de **CORS restringido**, que definen explícitamente los orígenes y cabeceras permitidos. Esta configuración evita la exposición no autorizada de la API a entornos externos y mitiga vulnerabilidades comunes relacionadas con encabezados y transporte de datos.

La validación de datos se gestiona a través de un **pipe global de validación (ValidationPipe)** con parámetros whitelist, forbidNonWhitelisted y transformOptions, lo que impide la inclusión de campos no declarados en los DTOs y garantiza la conversión segura de tipos.

La configuración de seguridad se encuentra documentada en main.ts, donde se aplican los middlewares, restricciones de CORS y validadores de DTOs con parámetros de endurecimiento activo.

Mantenibilidad y escalabilidad (RFN-05)

El diseño modular del sistema responde a una **arquitectura por dominios** que agrupa lógicamente los servicios, entidades y repositorios bajo el marco de *NestJS*. En el backend, los servicios de manipulación de código (como FeatureFilesManipulationService o CodeParsingService) se encuentran desacoplados en un módulo común reutilizable, lo que permite su inyección en distintos contextos sin duplicar lógica. Esta estructura refuerza el principio de *Single Responsibility* y facilita la escalabilidad futura de nuevas funcionalidades.

A nivel de persistencia, cada entidad (por ejemplo, Project y Endpoint) se gestiona mediante **repositorios inyectados** con @InjectRepository, lo que aporta independencia entre capas y facilita las pruebas unitarias y de integración. En el frontend, la mantenibilidad se refleja en la **organización modular de componentes y hooks**,

agrupados por dominio funcional (Projects, Endpoints, TestCases, etc.), con tipado explícito y contratos compartidos, garantizando consistencia en los flujos de datos.

Usabilidad y consistencia visual (RFN-06)

La usabilidad se centra en ofrecer **retroalimentación inmediata y accesible** al usuario, garantizando coherencia visual y fluidez en la interacción. El frontend implementa **feedback en tiempo real** a través de SSE, mostrando la ejecución de pruebas y su avance sin interrupciones. Además, la aplicación incluye un **overlay de arranque (BackendLoaderOverlay.tsx)** que bloquea la interfaz hasta confirmar la disponibilidad del backend, evitando interacciones prematuras y comunicando el estado del sistema con mensajes de carga y desenfoque del fondo.

En cuanto a la accesibilidad, todos los diálogos, *toasts* y skeletons fueron diseñados con primitives accesibles (Radix UI), asegurando navegación por teclado, foco visible y mensajes compatibles con lectores de pantalla. El **modo oscuro persistente** complementa esta experiencia, adaptándose a la preferencia del sistema y manteniendo la selección del usuario entre sesiones.

Los componentes BackendLoaderOverlay.tsx, Layout.tsx y el hook useExecutionEvents.ts reflejan la integración de SSE, overlay de inicio y control de tema, consolidando la experiencia visual y funcional del usuario.

En conjunto, las implementaciones descritas en estos seis requerimientos no funcionales ayudan a demostrar que Omega Testing no solo responde a las necesidades técnicas de un entorno de automatización inteligente, sino que también incorpora principios de eficiencia, seguridad, observabilidad y accesibilidad que fortalecen su sostenibilidad y usabilidad en escenarios de producción reales.

5.4. Mantenimiento del Sistema

El mantenimiento del sistema comprende las acciones destinadas a preservar la operatividad, corregir errores y facilitar la evolución de futuras versiones de Omega Testing.

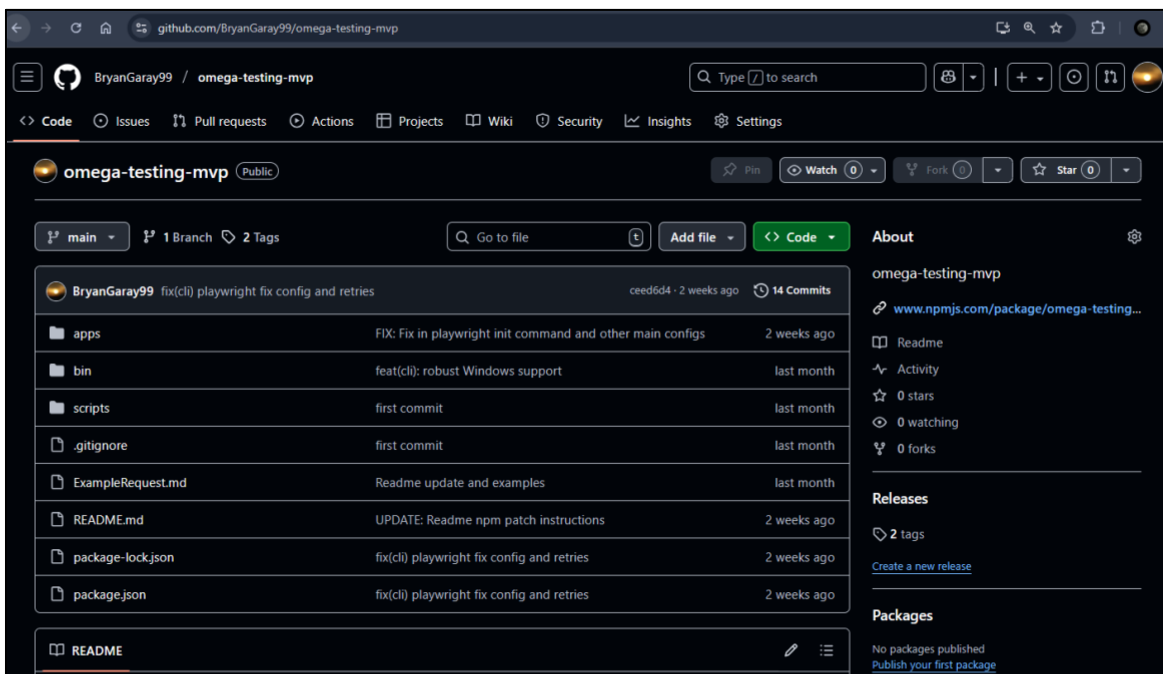
Este proceso abarca la gestión del código fuente mediante control de versiones, la publicación de módulos en el ecosistema NPM, la depuración a través de registros del sistema y el monitoreo del uso de la inteligencia artificial integrada.

Estas prácticas aseguran la continuidad operativa y establecen una base confiable para la mejora continua de la plataforma.

Repositorio y control de versiones

El sistema se mantiene bajo control de versiones en la plataforma GitHub, donde se aloja el repositorio oficial del proyecto. Esta herramienta permite conservar el historial completo de commits, ramas y versiones publicadas, garantizando trazabilidad y colaboración entre desarrolladores.

Figura 39. Vista general del repositorio de Omega Testing en GitHub



Nota. Elaborado por el autor.

Repositorio oficial: https://github.com/BryanGaray99/omega-testing-mvp

La arquitectura del proyecto está organizada bajo un esquema monorepo, que agrupa el backend (NestJS) y el frontend (React + TypeScript) en una misma estructura. Este enfoque

facilita el mantenimiento conjunto de ambas capas, reduciendo redundancias y permitiendo sincronizar dependencias y scripts comunes.

Cada nueva versión del sistema se desarrolla en ramas separadas y se integra posteriormente en la rama main, previa revisión de código. Esta metodología asegura estabilidad y control durante el ciclo de despliegue continuo.

Publicación de versiones en NPM

El programa Omega Testing compilado como una solución inntegral se distribuye como paquete dentro del registro NPM (Node Package Manager). Esta mecanismo permite el uso de la herramienta en cualquier entorno que cuente con Node.js.

El uso de NPM permite aplicar versionamiento semántico (por ejemplo 1.0.0, 1.1.0) y mantener una trazabilidad clara entre las actualizaciones del código fuente y las versiones publicadas.

Para actualizar el aplicativo ya sea por motivos de mejoras, cambios o arreglo por presencia de defectos se puede hacer mediante una serie de comandos que habilitarán una nueva versión del paquete en el registro de NPM. Estos comandos que permitirán el mantenimiento son:

```
1) npm version patch -m "fix(cli) message"
2) git push & git push --tags
3) npm install
4) npm run build:dist
5) npm publish --access public
```

Estos comandos siempre y cuando el usuario administrador se encuentre logeado con las credenciales de usuario y contraseña de la cuenta desde donde se publicó el paquete NPM.

Paquete disponible en: https://www.npmjs.com/package/omega-testing-mvp

Figura 40. Página del paquete Omega Testing MVP en NPM

Nota. Elaborado por el autor.

Depuración y registros del sistema

El backend implementa un sistema de logging estructurado basado en el LoggerService de NestJS, que genera trazas con distintos niveles de severidad (log, warn, error) y prefijos de contexto (por ejemplo, [AI Service], [Queue Service]).

Figura 41. Registros de logs en consola de Omega Testing

Nota. Elaborado por el autor.

Estos registros permiten identificar con precisión la secuencia de eventos durante la ejecución de pruebas o la generación de artefactos, facilitando la detección de errores y el mantenimiento correctivo.

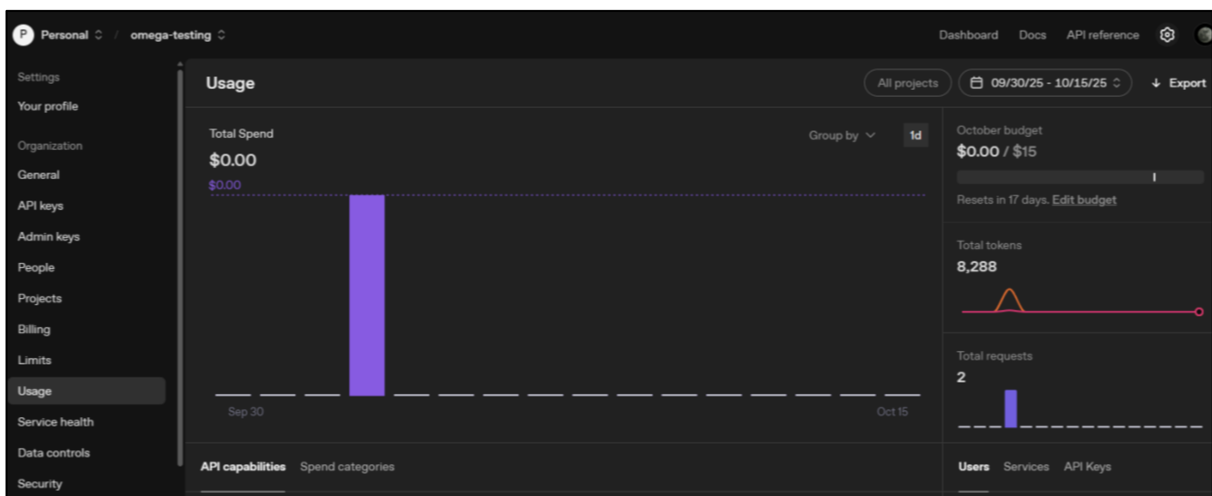
Durante la operación, el sistema emite advertencias automáticas cuando se superan umbrales de consumo de tokens o se producen tiempos de espera prolongados, lo que permite intervenir tempranamente para optimizar los procesos de generación asistida por IA.

Monitoreo del uso de la inteligencia artificial

El consumo y rendimiento de la inteligencia artificial integrada se supervisan desde el panel de OpenAI, asociado a la API Key configurada en el sistema. Este panel permite consultar métricas de uso diario, número total de tokens procesados y costos acumulados por modelo, ofreciendo una visión precisa del comportamiento del agente generador de casos de prueba.

Dicha información complementa los registros internos del sistema, proporcionando un nivel adicional de control sobre la eficiencia del modelo y ayudando a planificar ajustes en los parámetros de truncamiento o reutilización de contexto definidos en el RFN-01.

Figura 42. Panel de OpenAI con métricas de consumo de tokens



Nota. Elaborado por el autor.

En conjunto, estas prácticas de mantenimiento garantizan la trazabilidad del código, la distribución controlada de versiones, la capacidad de diagnóstico mediante logs y la observación continua del desempeño de la IA. Este enfoque integral asegura que Omega Testing permanezca estable, escalable y preparado para incorporar futuras mejoras sin comprometer la fiabilidad de las versiones en producción.

CAPITULO VI

6. EVALUACIÓN Y RESULTADOS

6.1. Pruebas Funcionales del Sistema

Con el objetivo de comprobar la estabilidad, coherencia y cumplimiento de los requerimientos definidos, se realizaron pruebas funcionales en todos los módulos del sistema. Cada conjunto de pruebas incluyó escenarios positivos y negativos, abarcando tanto la interacción entre frontend y backend como la correcta respuesta de la interfaz ante operaciones inválidas o interrupciones del servicio. La siguiente tabla resume los resultados obtenidos, junto con una breve descripción de los aspectos evaluados en cada módulo.

Tabla 23. Resumen general de pruebas funcionales por módulo

Módulo	Descripción General de las Pruebas	Pruebas Positivas	Pruebas Negativas	Total de Pruebas	Requerimientos Funcionales Validados
Dashboard	Validación de la carga inicial del tablero, visualización de KPIs, actualización de métricas en tiempo real y respuesta ante falta de datos o conexión con backend.	3	3	6	RF-01, RF-06
Projects	Pruebas sobre el ciclo completo de proyectos: creación, filtrado, edición, ejecución, eliminación y manejo de estados *pending*, *ready* y *failed*.	6	4	10	RF-02, RF-09
Endpoints	Verificación del registro, edición, reanálisis y eliminación de endpoints, con control de métodos HTTP, validaciones de entrada y generación de artefactos.	7	2	9	RF-03, RF-08
Test Cases	Evaluación de creación manual, generación asistida por IA y sugerencias automáticas; edición, ejecución y eliminación de casos, junto con manejo de errores.	8	4	12	RF-04, RF-08
Test Suites	Validación de la creación de *Test Sets* y *Test Plans*, edición, ejecución y visualización de resultados consolidados, asegurando coherencia entre casos y suites.	7	3	10	RF-05, RF-06
Bugs	Comprobación del registro automático y manual de defectos, cambios de estado, edición, vinculación con ejecuciones y eliminación controlada.	10	3	13	RF-07, RF-06
Test Executions	Verificación de listados, filtros, visualización de KPIs y detalle de ejecución; validación del flujo SSE y control de errores en la eliminación.	6	2	8	RF-06

Sync Project	Validación de sincronización de artefactos por proyecto y global, gestión de mensajes de éxito/error y restricciones según estado del proyecto.	2	2	4	RF-09
Settings (OpenAI)	Pruebas de configuración de la API Key, validación de conexión, y manejo de errores con claves inválidas o formatos incorrectos.	2	2	4	RF-10
Totales	—	51	25	76	Cobertura completa de RF-01 a RF-10

Nota. Elaborado por el autor.

Los resultados globales reflejan una cobertura total de los diez requerimientos funcionales del sistema. Se ejecutaron 76 pruebas distribuidas en nueve módulos, de las cuales 51 correspondieron a escenarios positivos y 25 a negativos, confirmando la robustez y estabilidad del sistema ante diferentes condiciones operativas.

El módulo Test Cases concentró la mayor carga de validación, dada su relevancia en la automatización de pruebas y su dependencia del asistente de IA. Los módulos Projects, Bugs y Test Suites también registraron un alto número de pruebas por su papel en la gestión del flujo de trabajo de testing. En contraste, Sync Project y Settings tuvieron menor cantidad por su carácter auxiliar, aunque su funcionamiento resultó esencial para la integridad del sistema y la configuración de la IA.

Se observó el cumplimiento integral de los requerimientos funcionales: desde la visualización y gestión de artefactos hasta la ejecución, sincronización y monitoreo de resultados. Las pruebas negativas confirmaron la presencia de mecanismos de validación, manejo de errores y restricciones de acceso coherentes con las políticas de seguridad y consistencia del sistema.

Bajo este escenario, los resultados funcionales verificaron que Omega Testing opera conforme a los objetivos planteados en su diseño: integrar generación, ejecución y trazabilidad de pruebas automatizadas en APIs REST con soporte inteligente, garantizando estabilidad técnica y cumplimiento de estándares de calidad de software.

6.2. Resultados de Encuesta de Usabilidad y Desempeño

La evaluación de usabilidad y desempeño tuvo como propósito analizar la experiencia de interacción de los usuarios con el sistema Omega Testing, valorando tanto su facilidad de uso como el impacto operativo en los procesos de prueba funcional. El estudio buscó determinar el grado en que la herramienta resulta intuitiva, eficiente y confiable para los profesionales del área de QA, así como su capacidad para optimizar tiempos y mejorar la calidad del testing asistido por inteligencia artificial.

Para ello, se diseñó una encuesta estructurada en dos secciones complementarias. La primera, basada en las 10 heurísticas de Nielsen, evaluó dimensiones clave de usabilidad como visibilidad del estado del sistema, consistencia visual, control del usuario, flexibilidad, eficiencia de uso y prevención de errores. La segunda sección incluyó dos preguntas específicas de desempeño, orientadas a medir la reducción de tiempo en la generación y ejecución de pruebas, así como la mejora percibida en cobertura y calidad del proceso automatizado.

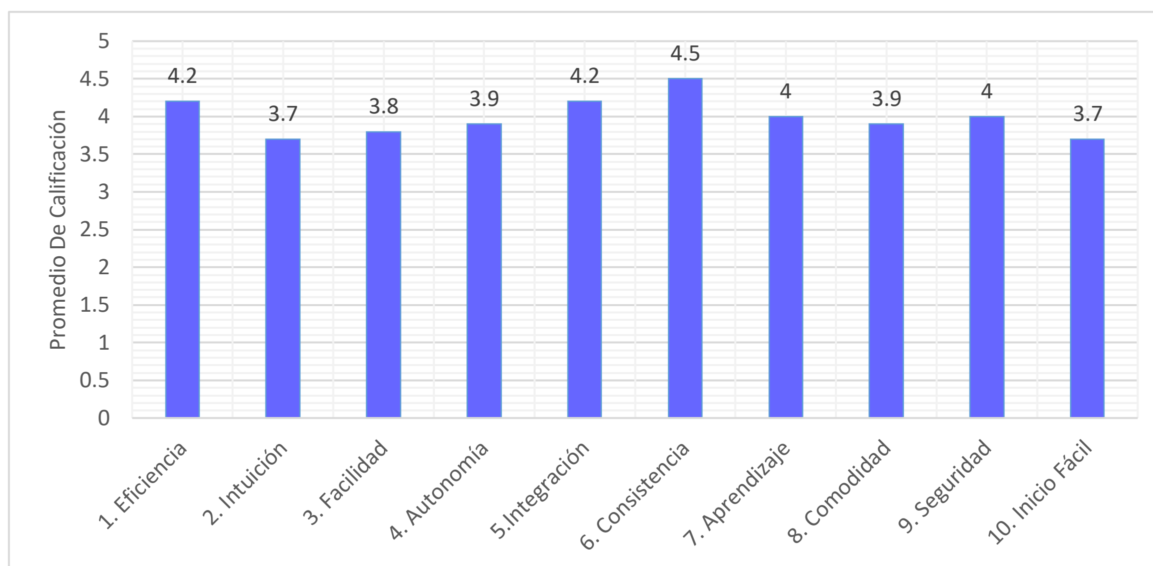
El instrumento utilizó una escala Likert de 5 puntos para las heurísticas de usabilidad, donde 1 representó una percepción muy baja y 5 una percepción excelente. En las preguntas de desempeño, los participantes seleccionaron rangos porcentuales de mejora (0–10 %, 11–30 %, 31–50 %, 51–70 % y más del 70 %), lo que permitió cuantificar la magnitud del impacto percibido.

La encuesta fue aplicada a un grupo de 10 profesionales con experiencia en aseguramiento de la calidad de software, testing funcional y automatización de APIs. Los resultados obtenidos se consolidaron en gráficos comparativos que reflejan tanto la aceptación del sistema en términos de diseño y usabilidad, como su contribución al ahorro de tiempo y mejora en la calidad del proceso de pruebas funcionales.

Evaluación de Usabilidad

En la Figura 43, se muestra el gráfico de columnas que resume las respuestas obtenidas en torno a las heurísticas de Nielsen, distribuidas por nivel de valoración. Este gráfico permite identificar los aspectos del sistema con mejor percepción de usabilidad y aquellos con oportunidades de mejora.

Figura 43. Promedio de calificación de las heurísticas de usabilidad según Nielsen.



Nota. Elaborado por el autor.

El análisis de los resultados muestra valores homogéneos entre 3.7 y 4.5, con una media global de 4.0 puntos, lo que evidencia una percepción positiva y consistente respecto a la usabilidad general. Las heurísticas mejor valoradas fueron Consistencia (4.5), Eficiencia (4.2) e Integración (4.2), destacando la estabilidad del diseño, la coherencia visual y la integración entre módulos como los aspectos más apreciados.

Por otra parte, las calificaciones más bajas correspondieron a Intuición (3.7) e Inicio fácil (3.7), lo que sugiere oportunidades de mejora en la curva inicial de aprendizaje y en la simplificación de algunos procesos de configuración, como la carga de la API Key o el reconocimiento de flujos entre módulos. Aun así, los resultados reflejan que la interfaz mantiene una alta aceptabilidad entre usuarios expertos, cumpliendo con los objetivos del requerimiento funcional RF-06 (usabilidad y consistencia visual).

Evaluación del Impacto en la Eficiencia Operativa

La Figura 44 presenta los resultados de la pregunta 11, relacionada con la reducción del tiempo requerido para la generación y ejecución de pruebas funcionales gracias al agente inteligente. La pregunta fue: **¿En qué medida considera que el uso del agente inteligente reduce el tiempo total requerido para la generación y ejecución de pruebas funcionales en APIs REST, comparado con los métodos tradicionales?**

Figura 44. Percepción de reducción del tiempo de generación y ejecución de pruebas

Nota. Elaborado por el autor.

El 40 % de los encuestados consideró que la reducción fue significativa (31–50 %), mientras que el 30 % la calificó como alta (51–70 %). Solo el 10 % reportó reducción muy alta y un 20 % la definió como moderada (11–30 %).

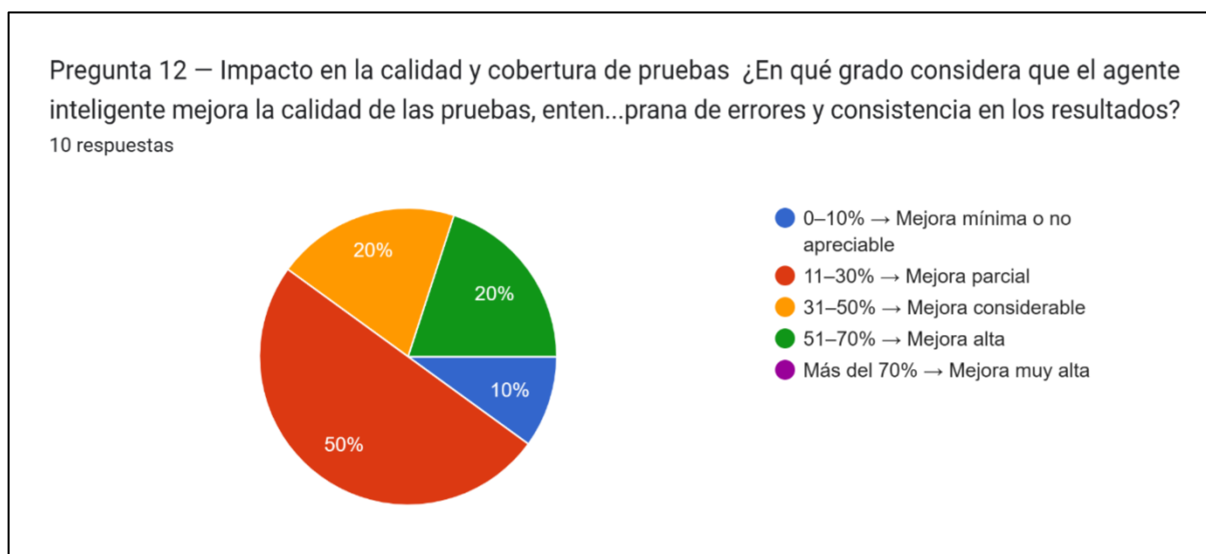
Estos resultados evidencian que la mayoría de los usuarios perciben un ahorro sustancial de tiempo, lo que confirma el cumplimiento del objetivo de agilización planteado en los requerimientos funcionales RF-04 (generación de casos de prueba asistida por IA) y RF-06 (orquestación y seguimiento de ejecuciones en tiempo real), así como la pregunta de investigación planteada que cuestionada la valía de la herramienta como medio para agilizar la automatización de casos de prueba o pruebas funcionales de API testing.

Evaluación del Impacto en la Calidad y Cobertura de Pruebas

La Figura 45 muestra los resultados de la pregunta 12: **¿En qué grado considera que el agente inteligente mejora la calidad de las pruebas, entendida como mayor cobertura, detección temprana de errores y consistencia en los resultados?**

Como se puede observar, esta se enfoca en el grado de mejora percibida en la calidad del proceso de pruebas, considerando cobertura, detección temprana de errores y consistencia en los resultados.

Figura 45. Percepción de calidad y cobertura de Omega Testing.



Nota. Elaborado por el autor.

El 50 % de los participantes percibió una mejora parcial (11–30 %), mientras que un 20 % identificó una mejora considerable (31–50 %) y otro 20 % una mejora alta (51–70 %). Solo el 10 % la clasificó como mínima o no apreciable.

Estos valores reflejan que, si bien el impacto en calidad es más moderado que en eficiencia, existe una clara tendencia positiva, atribuible a la generación automática de casos consistentes, la detección de errores recurrentes y la disminución de pasos manuales propensos a fallo.

Retroalimentación cualitativa

Entre los comentarios adicionales proporcionados por los encuestados, destacan observaciones que complementan el análisis cuantitativo. De Sowmya Male, QA con 10 años de experiencia en Calidad de Software, se recibió el comentario: “This tool could

certainly help reduce the time spent creating test cases, and the automatic bug reporting feature is innovative." Ella destaca la eficiencia del sistema y la novedad de su funcionalidad para el registro automático de errores. La observación coincide con los resultados de la encuesta sobre desempeño, donde la mayoría de los participantes señaló reducciones significativas en el tiempo de creación y ejecución de pruebas. Este comentario valida que el agente inteligente cumple su propósito principal de optimizar el trabajo de los testers, automatizando tareas que normalmente son repetitivas y consumen mucho tiempo.

El segundo comentario, de Nicole Toala, Test Automation Engineer con 4 años de experiencia en el sector, expresa: "Los tiempos de creación de pruebas son muy rápidos. El uso de la IA es bastante útil, me gustó que el código generado se inserta directamente al feature file", resalta la agilidad del proceso de generación y la integración fluida entre la inteligencia artificial y el entorno de pruebas. La mención al archivo feature confirma que el flujo técnico entre backend y frontend fue correctamente implementado, permitiendo al usuario generar código funcional sin intervención manual. Este punto se relaciona directamente con las valoraciones altas obtenidas en las heurísticas de eficiencia y consistencia, lo que refleja una experiencia de uso coherente y estable.

Por último el comentario de Valeria Muner, QA Lead de Argentina, comparte que "La herramienta está bien, hay espacios de mejora como funcionalidades para trabajo colaborativo en futuras versiones", introduce una sugerencia valiosa para el crecimiento del sistema. Aunque la colaboración no formaba parte del alcance del MVP, esta retroalimentación revela una necesidad real en equipos de QA que trabajan de forma distribuida. El comentario se complementa con los valores moderados en autonomía e inicio fácil observados en la encuesta, que sugieren que algunos usuarios valoran una experiencia más conectada entre varios testers.

En conjunto, los comentarios y resultados de las encuestas reflejan una percepción positiva de la herramienta y evidencian que las funciones principales de automatización, generación de casos y gestión de errores fueron comprendidas y valoradas por los expertos en QA. Las observaciones no apuntan a fallas críticas, sino a mejoras evolutivas que fortalecerían el trabajo en equipo y la escalabilidad. En general, la retroalimentación confirma que Omega Testing logra su objetivo de agilizar la automatización de pruebas funcionales y mejorar la eficiencia de los procesos de calidad en entornos ágiles.

6.3. Beneficios del Sistema y Oportunidades de Mejora

La implementación del sistema Omega Testing permitió comprobar mejoras tangibles en el proceso de automatización de pruebas funcionales en APIs REST, tanto en eficiencia operativa como en consistencia de resultados. El uso de un agente inteligente integrado en el flujo de trabajo redujo significativamente los tiempos asociados a la generación y ejecución de pruebas, optimizando la productividad de los equipos de QA.

Uno de los principales beneficios observados fue la automatización completa en la creación de casos de prueba a partir de descripciones funcionales, lo que redujo la dependencia de la programación manual y permitió una mayor cobertura en menos tiempo. Este avance se tradujo en una disminución de tareas repetitivas y una mejor trazabilidad entre los módulos de proyectos, endpoints, casos y ejecuciones. Además, la incorporación del registro automático de errores aportó una capa adicional de control, asegurando que las incidencias detectadas durante las pruebas fueran documentadas sin intervención del usuario.

El sistema también demostró mejoras en la visibilidad del proceso de testing. Los reportes y métricas consolidados en el tablero permitieron a los analistas supervisar el estado de ejecución en tiempo real, evaluar tasas de éxito, y detectar desviaciones en etapas tempranas. Este nivel de observabilidad fortaleció la toma de decisiones, reduciendo tiempos de diagnóstico y mejorando la calidad final del producto evaluado.

En términos de usabilidad, los resultados de la encuesta mostraron una alta aceptación del diseño y del flujo de interacción. Los usuarios percibieron un entorno intuitivo y coherente, con componentes visuales consistentes y una experiencia alineada a las heurísticas de Nielsen. El modo oscuro persistente, las notificaciones en tiempo real y el soporte accesible de teclado aportaron una sensación de fluidez y control, reforzando la satisfacción general.

Entre las oportunidades de mejora identificadas se destaca la incorporación de funcionalidades de trabajo colaborativo, que permitirían a varios testers interactuar sobre un mismo proyecto de forma simultánea, esto mediante integración de sistemas de versionamiento de código como GitHub, GitLab entre otros. Asimismo, se sugiere fortalecer la documentación de ayuda y expandir los reportes automáticos con métricas comparativas por proyecto o por versión del sistema. Estas sugerencias coinciden con las observaciones de los evaluadores, que ven en Omega Testing una herramienta sólida, con alto potencial de crecimiento hacia entornos reales más complejos que van más allá del MVP propuesto como solución para este trabajo de investigación.

En un contexto de adopción del sistema durante sprints reales, Omega Testing no solo fortalecería la automatización de pruebas funcionales, sino que también serviría como base para el análisis de indicadores clave revisados en un Sprint Retrospective. La información generada por el sistema a lo largo de las ejecuciones permitiría evaluar la evolución de la calidad y del flujo de trabajo de forma objetiva y continua.

A partir de esta información, podrían analizarse los siguientes KPIs:

- **Cobertura de cambio**: permitiría identificar qué funcionalidades o endpoints modificados durante el sprint contarían con casos de prueba automatizados ejecutados, aportando evidencia sobre el nivel de validación alcanzado antes de cada entrega.
- **Defectos por periodo**: facilitaría el seguimiento de la cantidad de fallos detectados en cada sprint, lo que permitiría observar tendencias de calidad, comparar periodos y detectar áreas del sistema con mayor recurrencia de errores.
- **Lead Time de validación**: posibilitaría medir el tiempo transcurrido desde que un cambio estaría listo para ser validado hasta su verificación mediante pruebas funcionales automatizadas, ofreciendo una visión clara de la eficiencia del proceso de testing.
- **Fugas a producción**: permitiría identificar defectos que alcanzarían entornos productivos y analizar si estos contasen con cobertura de pruebas previa, fortaleciendo la retroalimentación y las acciones preventivas en sprints futuros.

En conjunto, estos indicadores reforzarían el valor de Omega Testing como una herramienta que, además de optimizar la automatización de pruebas, tendría el potencial de evolucionar hacia un soporte estratégico para el análisis de calidad y rendimiento en equipos ágiles, contribuyendo a retrospectivas más informadas y a un proceso de mejora continua sostenido dentro del marco de trabajo Scrum.

CAPITULO VII

7. DISCUSIÓN

7.1. Conclusiones

- La implementación del sistema Omega Testing demostró ser técnicamente viable y funcional como una solución de automatización de pruebas funcionales en APIs REST, integrando generación, ejecución y análisis de resultados dentro de un flujo unificado orientado a equipos de QA.
- El uso de un agente inteligente permitió reducir de forma significativa el esfuerzo manual asociado a la creación de casos de prueba, facilitando la generación automática de escenarios positivos y negativos a partir de descripciones funcionales, lo que contribuyó a una mayor cobertura de pruebas en menor tiempo.
- El sistema logró centralizar la trazabilidad entre proyectos, endpoints, casos de prueba, ejecuciones y defectos, mejorando la visibilidad del proceso de testing y permitiendo un seguimiento más estructurado del estado de las pruebas durante el ciclo de desarrollo.
- Los resultados de la evaluación funcional confirmaron que el sistema cumple con los requerimientos definidos, ejecutando correctamente los distintos flujos de prueba y registrando de forma automática los errores detectados durante las ejecuciones.
- La evaluación de usabilidad evidenció una alta aceptación por parte de los usuarios, destacando la coherencia del diseño, la claridad del flujo de interacción y la alineación con las heurísticas de Nielsen, lo que favorece la adopción del sistema en contextos reales de trabajo.
- En conjunto, los resultados obtenidos confirman que Omega Testing constituye una herramienta sólida que aporta eficiencia, consistencia y control al proceso de pruebas funcionales automatizadas, cumpliendo con los objetivos planteados en esta investigación.

7.2. Recomendaciones

- Se recomienda la adopción de Omega Testing en proyectos reales de desarrollo de software que utilicen APIs REST, con el fin de evaluar su desempeño de forma continua a lo largo de múltiples sprints y contextos operativos.

- Es aconsejable extender la evaluación del sistema mediante estudios con un mayor número de participantes y equipos de QA, lo que permitiría obtener resultados más representativos y fortalecer la validez externa de los hallazgos.
- Se sugiere incorporar funcionalidades de trabajo colaborativo, que permitan la interacción simultánea de varios testers sobre un mismo proyecto, mejorando la coordinación y el intercambio de información dentro del equipo.
- Es recomendable integrar el sistema con herramientas de gestión de proyectos y control de versiones, lo que permitiría una trazabilidad más completa entre cambios funcionales, pruebas automatizadas y defectos detectados.
- Finalmente, se sugiere explorar la evolución del sistema hacia la incorporación de métricas de calidad y flujo, como cobertura de cambio, defectos por periodo, Lead Time de validación y fugas a producción, consolidando a Omega Testing como una plataforma de soporte estratégico para la toma de decisiones en equipos Scrum.

BIBLIOGRAFÍA

Aizprúa Alfonso, S., Ortega, A., & Von Chong, L. (2019). Calidad del Software una Perspectiva Continua. *Centros: Revista Científica Universitaria, ISSN-e 2953-3007, Vol. 8, Nº. 2, 2019, págs. 120-134*, *8*(2).

Alharbi, S. J., & Moulahi, T. (2023). API Security Testing: The Challenges of Security Testing for Restful APIs. *International Journal of Innovative Science and Research Technology*, *8*(5).

Ariel Menéndez-Verdecia, J. I., & Noralma Aguilar-Moncayo III, L. (2021). Mejores prácticas de calidad en el desarrollo de software integradas al conocimiento de la ingeniería. *Polo del Conocimiento*, *6*(1).

Atoum, I., Baklizi, M. K., Alsmadi, I., Otoom, A. A., Alhersh, T., Ababneh, J., Almalki, J., & Alshahrani, S. M. (2021). Challenges of Software Requirements Quality Assurance and Validation: A Systematic Literature Review. En *IEEE Access* (Vol. 9). https://doi.org/10.1109/ACCESS.2021.3117989

Bertolino, A., Hong, S., & Mathur, A. P. (2023). Introduction to the special issue on automation of software test and test code quality. En *Journal of Software: Evolution and Process* (Vol. 35, Número 4). https://doi.org/10.1002/smr.2453

Bhanushali, A. (2023). Challenges and Solutions in Implementing Continuous Integration and Continuous Testing for Agile Quality Assurance. *International Journal of Science and Research (IJSR)*, *12*(10). https://doi.org/10.21275/sr231021114758

Chang, T. S. (2023). Evaluation of an artificial intelligence project in the software industry based on fuzzy analytic hierarchy process and complex adaptive systems. *Journal of Enterprise Information Management*, *36*(4). https://doi.org/10.1108/JEIM-02-2022-0056

de Carvalho Souza, M., & Weigang, L. (2025). *Grok, Gemini, ChatGPT and DeepSeek: Comparison and Applications in Conversational Artificial Intelligence*. *1*. https://doi.org/10.5281/zenodo.14885243

Duraisamy, S. K., Bass, B., & Mukkavilli, S. (2021). Embedding Performance Testing in Agile Software Model. *International Journal of Software Engineering & Applications*, *12*(06). https://doi.org/10.5121/ijsea.2021.12601

Ehsan, A., Abuhaliqa, M. A. M. E., Catal, C., & Mishra, D. (2022). RESTful API Testing Methodologies: Rationale, Challenges, and Solution Directions. En *Applied Sciences (Switzerland)* (Vol. 12, Número 9). https://doi.org/10.3390/app12094369

Fang, Y. (2021). The Theory and Application in Agile Project Management (APM) with Scrum. *Proceedings of the 2021 6th International Conference on Modern Management and Education Technology （MMET 2021）, 582*. https://doi.org/10.2991/assehr.k.211011.117

Golmohammadi, A., Zhang, M., & Arcuri, A. (2023). Testing RESTful APIs: A Survey. *ACM Transactions on Software Engineering and Methodology*, *33*(1). https://doi.org/10.1145/3617175

Gudavalli, A., & JayaLakshmi, G. (2022). Implementation of Test Automation with Selenium Webdriver. *13th International Conference on Advances in Computing, Control, and Telecommunication Technologies, ACT 2022*, *8*.

Gutierrez Sullca, E. M. (2023). Impact of software testing automation on the development cycle. *Revista de Investigación Científica Huamachuco*, *1*(1). https://doi.org/10.61709/huamachuco.v1i1.3

Haldorai, A. (2022). Analysis on Intelligent Agent based Approach for Software Engineering. *Journal of Computing and Natural Science*. https://doi.org/10.53759/181x/jcns202202020

Hayat, A., Islam, S., & Hossain, F. (2024). The Evolving Role of Artificial Intelligence in Software Testing: Prospects and Challenges. *International Journal For Multidisciplinary Research*, *6*(2). https://doi.org/10.36948/ijfmr.2024.v06i02.14783

Hernández-Sampieri, R., Fernández, C., & Baptista, P. (2016). Metodología de la investigación. 6ta Edición Sampieri. *Guía para realizar investigaciones sociales. Plaza y Valdés.*

Hossain, Md. S. (2018). CHALLENGES OF SOFTWARE QUALITY ASSURANCE AND TESTING. *International Journal of Software Engineering and Computer Systems*, *4*(1). https://doi.org/10.15282/ijsecs.4.1.2018.11.0044

Job, M. A. (2021). Automating and Optimizing Software Testing using Artificial Intelligence Techniques. *International Journal of Advanced Computer Science and Applications*, *12*(5). https://doi.org/10.14569/IJACSA.2021.0120571

Jonsson, M., Qvarnström, E., Lindell, R., & Gustafsson, J. (2022). A Performance Comparison On Rest-Apis In Express.Js, Flask And Asp.Net Core. *Digitala Vetenskapliga Arkivet*.

Juviler, J. (2023). *REST APIs: How They Work and What You Need to Know*. Hubspot.

Kalech, M., & Stern, R. (2020). Ai for software quality assurance blue sky ideas talk. *AAAI 2020 - 34th AAAI Conference on Artificial Intelligence*. https://doi.org/10.1609/aaai.v34i09.7076

Kassie, N. B., & Singh, J. (2020). A study on software quality factors and metrics to enhance software quality assurance. *International Journal of Productivity and Quality Management*, *29*(1). https://doi.org/10.1504/IJPQM.2020.104547

Khan, S. A., Oshin, N. T., Nizam, M., Ahmed, I., Musfique, M. M., & Hasan, M. (2024). AI-Based Software Testing. *Lecture Notes in Networks and Systems*, *833*. https://doi.org/10.1007/978-981-99-8346-9_28

Khankhoje, R. (2024a). AI in Test Automation: Overcoming Challenges, Embracing Imperatives. *International Journal on Soft Computing, Artificial Intelligence and Applications*, *13*(1). https://doi.org/10.5121/ijscai.2024.13101

Khankhoje, R. (2024b). An Intelligent Apitesting: Unleashing the Power of AI. *International Journal of Software Engineering & Applications*, *15*(1). https://doi.org/10.5121/ijsea.2024.15101

Kore, P. P., Lohar, M. J., Surve, M. T., & Jadhav, S. (2022). API Testing Using Postman Tool. *International Journal for Research in Applied Science and Engineering Technology*, *10*(12). https://doi.org/10.22214/ijraset.2022.48030

Lee, M.-C. (2014). Software Quality Factors and Software Quality Metrics to Enhance Software Quality Assurance. *British Journal of Applied Science & Technology*, *4*(21). https://doi.org/10.9734/bjast/2014/10548

López-Rodríguez, S. A., & García-Peña, V. R. (2021). Metodologías de desarrollo de software seguro con propiedades agiles. *Polo del Conocimiento*, *5*(10).

Lucas, M. (2023). *What are LLMs, and how are they used in generative AI?* ComputerWorld.

Martin-Lopez, A., & Alonso, J. C. (2023). Testing of RESTful Web APIs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *13821 LNCS*. https://doi.org/10.1007/978-3-031-26507-5_43

Martin-Lopez, A., Segura, S., & Ruiz-Cortés, A. (2022). Online testing of RESTful APIs: promises and challenges. *ESEC/FSE 2022 - Proceedings of the 30th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. https://doi.org/10.1145/3540250.3549144

Minhas, N. M., Petersen, K., Börstler, J., & Wnuk, K. (2020). Regression testing for large-scale embedded software development – Exploring the state of practice. *Information and Software Technology*, *120*. https://doi.org/10.1016/j.infsof.2019.106254

Mohamad Yusop, N. S., Mohd Fikri, A. S., & Yahaya, N. (2023). STEM: A WEB-BASED SYSTEM FOR MANAGING TEST ARTIFACTS IN SOFTWARE TESTING COURSE. *Malaysian Journal of Sport Science and Recreation*, *19*(1). https://doi.org/10.24191/mjssr.v19i1.21766

Mudassir, M., & Mushtaq, M. (2024). The role of APIs in modern software development. *World Journal of Advanced Engineering Technology and Sciences*, *13*(1), 1045–1047. https://doi.org/10.30574/wjaets.2024.13.1.0515

Najihi, S., Elhadi, S., Abdelouahid, R. A., & Marzak, A. (2022). Software Testing from an Agile and Traditional view. *Procedia Computer Science*, *203*. https://doi.org/10.1016/j.procs.2022.07.116

Nass, M., Alégroth, E., & Feldt, R. (2021). Why many challenges with GUI test automation (will) remain. *Information and Software Technology*, *138*. https://doi.org/10.1016/j.infsof.2021.106625

Nembhard, F. D., Slhoub, K. A., & Carvalho, M. M. (2023). An Agent-Based Approach Toward Smart Software Testing. *Lecture Notes in Networks and Systems*, *814 LNNS*. https://doi.org/10.1007/978-3-031-47451-4_21

Palani, N. (2021). Cypress-based API Testing. En *Automated Software Testing with Cypress*. https://doi.org/10.1201/9781003145110-16

Papakitsos, E. C. (2022). Robust Software Quality Assurance. *Bulletin of the Georgian National Academy of Sciences*, *16*(2).

Pardo, M. A. P., Erazo, H. A. O., & Lozada, C. A. C. (2021). Documenting and implementing DevOps good practices with test automation and continuous deployment tools through

software refinement. *Periodicals of Engineering and Natural Sciences*, *9*(4). https://doi.org/10.21533/pen.v9i4.2239

Pargaonkar, S. (2023). A Comprehensive Research Analysis of Software Development Life Cycle (SDLC) Agile & Waterfall Model Advantages, Disadvantages, and Application Suitability in Software Quality Engineering. *International Journal of Scientific and Research Publications*, *13*(8). https://doi.org/10.29322/ijsrp.13.08.2023.p14015

Patilla, H., Gómez, E., Pulache, J., Lozano, J., Solórzano, E., & Meneses, Y. (2021). Modelo de Gestión de Desarrollo de Software Ágil mediante Scrum y Kanban sobre la Programación Extrema. *Revista Ibérica de Sistemas e Tecnologías de Informação*.

Patni, S. (2023). Fundamentals of RESTful APIs. En *Pro RESTful APIs with Micronaut*. https://doi.org/10.1007/978-1-4842-9200-6_1

Polkhovskaya, A. (2025). *Improving QA Processes andImplementing Playwright for APIand E2E Testing Automation*. https://www.theseus.fi/bitstream/handle/10024/882858/Polkhovskaya_Alevtina.pdf?sequence=2&isAllowed=y

Popov, A., Momot, M., & Yelizieva, A. (2022). CHOOSING THE TEST AUTOMATION SYSTEM ACCORDING TO CUSTOMER REQUIREMENTS. *Innovative Technologies and Scientific Solutions for Industries*, (1 (19)). https://doi.org/10.30837/itssi.2022.19.040

Qasim, M., Bibi, A., Hussain, S. J., Jhanjhi, N. Z., Humayun, M., & Sama, N. U. (2021). Test case prioritization techniques in software regression testing: An overview. *International Journal of Advanced and Applied Sciences*, *8*(5). https://doi.org/10.21833/ijaas.2021.05.012

Qazi, S., Memon, M. S., Ali, A., & Nizamani, S. (2022). ROLE OF ARTIFICIAL INTELLIGENCE (AI) TOOLS FOR ASSURING QUALITY IN SOFTWARE. *Journal of Southwest Jiaotong University*, *57*(2). https://doi.org/10.35741/issn.0258-2724.57.2.5

Rohit Khankhoje. (2023). An In-Depth Review of Test Automation Frameworks: Types and Trade-offs. *International Journal of Advanced Research in Science, Communication and Technology*. https://doi.org/10.48175/ijarsct-13108

Sauvola, J., Tarkoma, S., Klemettinen, M., Riekki, J., & Doermann, D. (2024). Future of software development with generative AI. *Automated Software Engineering*, *31*(1). https://doi.org/10.1007/s10515-024-00426-z

Spillner, Andreas, Linz, & Tilo. (2021). Software Testing Foundations: A Study Guide for the Certified Tester Exam-Foundation Level-ISTQB. En *Dpunkt. Verlag*.

Tetteh, S. G. (2024). Empirical Study of Agile Software Development Methodologies: A Comparative Analysis. *Asian Journal of Research in Computer Science*, *17*(5). https://doi.org/10.9734/ajrcos/2024/v17i5436

Thayer, K., Chasins, S. E., & Ko, A. J. (2021). A Theory of Robust API Knowledge. *ACM Transactions on Computing Education*, *21*(1). https://doi.org/10.1145/3444945

Thekkan Othayoth, J., & Anuar, S. (2022). Modern Web Automation with Cypress.Io. *Open International Journal of Informatics (OIJI)*, *10*(2).

Thooriqoh, H. A., Annisa, T. N., & Yuhana, U. L. (2021). SELENIUM FRAMEWORK FOR WEB AUTOMATION TESTING: A SYSTEMATIC LITERATURE REVIEW. *JUTI: Jurnal Ilmiah Teknologi Informasi*. https://doi.org/10.12962/j24068535.v19i2.a1021

Umar, M. A., & Zhanfang, C. (2019). A Study of Automated Software Testing : Automation Tools and Frameworks. *International Journal of Computer Science Engineering (IJCSE)*, *8*(06).

van Driel, W. D., Bikker, J. W., Tijink, M., & Di Bucchianico, A. (2020). Software reliability for agile testing. *Mathematics*, *8*(5). https://doi.org/10.3390/MATH8050791

Verwijs, C., & Russo, D. (2023). A Theory of Scrum Team Effectiveness. *ACM Transactions on Software Engineering and Methodology*, *32*(3). https://doi.org/10.1145/3571849

Vinueza, D. (2009). Estudio de las bases conceptuales, para la implementación de las mejores prácticas para el control y aseguramiento de la calidad software. *Universidad Internacional SEK*.

Wang, Y., Mäntylä, M. V., Liu, Z., & Markkula, J. (2022). Test automation maturity improves product quality—Quantitative study of open source projects using continuous integration. *Journal of Systems and Software*, *188*. https://doi.org/10.1016/j.jss.2022.111259

Wang, Y., Mäntylä, M. V., Liu, Z., Markkula, J., & Raulamo-jurvanen, P. (2022). Improving test automation maturity: A multivocal literature review. *Software Testing Verification and Reliability*, *32*(3). https://doi.org/10.1002/stvr.1804

Yuda Syahidin, & Randy Ramadhan. (2021). REST API Architecture Design on Multi-Platform Device Development. *Jurnal E-Komtek (Elektro-Komputer-Teknik)*, *5*(2). https://doi.org/10.37339/e-komtek.v5i2.762

Zhang, H., & Shao, H. (2024). Exploring the Latest Applications of OpenAI and ChatGPT: An In-Depth Survey. En *CMES - Computer Modeling in Engineering and Sciences* (Vol. 138, Número 3). https://doi.org/10.32604/cmes.2023.030649

ANEXOS

ANEXO A - Casos de Uso para el Diseño del Sistema Propuesto

A continuación se presenta mediante tablas el detalle de los casos de uso.

Tabla 24. CU-01 — Dashboard de entrada y navegación

Campo	Detalle
Código	CU-01
Nombre	Dashboard de entrada y navegación
Actor principal	Usuario
Objetivo	Visualizar KPIs generales (proyectos, endpoints, tasas de éxito/fallo) y acceder a los módulos del sistema.
Precondición	-
Flujo básico (resumen)	1) Abrir la aplicación. 2) Visualizar KPIs generales. 3) Navegar a Proyectos, Endpoints, Casos, Suites, Ejecuciones o Bugs.
Postcondición / Resultado	Usuario contextualizado con el estado general y ubicado en el módulo elegido.
Canal	UI (local)

Nota. Elaborado por el autor

Tabla 25. CU-02 — Crear y mantener proyectos

Campo	Detalle
Código	CU-02
Nombre	Crear y mantener proyectos
Actor principal	Usuario
Objetivo	Organizar el trabajo por iniciativa de pruebas y habilitar el resto de los componentes.
Precondición	-
Flujo básico (resumen)	1) Crear proyecto con metadatos. 2) Listar y buscar proyectos. 3) Ver detalle con KPIs. 4) Actualizar o eliminar de forma controlada.
Postcondición / Resultado	Proyecto disponible como raíz de endpoints, casos, suites, ejecuciones y bugs.
Canal	UI+API (local)

Nota. Elaborado por el autor

Tabla 26. CU-03 — Registrar y administrar endpoints

Campo	Detalle
Código	CU-03
Nombre	Registrar y administrar endpoints
Actor principal	Usuario
Objetivo	Mantener un catálogo de endpoints por proyecto, sección y entidad.
Precondición	Tener un Proyecto Creado
Flujo básico (resumen)	1) Registrar endpoint (ruta, métodos y definiciones). 2) Listar/filtrar. 3) Editar o eliminar. 4) Consultar detalle integral.
Postcondición / Resultado	Endpoints listos para vincular pruebas y artefactos asociados.
Canal	UI+API (local)

Nota. Elaborado por el autor

Tabla 27. CU-04 — Diseñar casos de prueba manuales (BDD)

Campo	Detalle
Código	CU-04
Nombre	Diseñar casos de prueba manuales (BDD)
Actor principal	Usuario
Objetivo	Definir escenarios Given/When/Then con pasos parametrizados.
Precondición	Entidad, Sección y Proyecto creado.
Flujo básico (resumen)	1) Crear caso en editor BDD. 2) Editar escenario/steps, duplicar o eliminar. 3) Ejecutar caso desde su vista (opcional).
Postcondición / Resultado	Caso válido y listo para ejecución o inclusión en suites.
Canal	UI+API (local)

Nota. Elaborado por el autor

Tabla 28. CU-05 — Generar casos de prueba con IA

Campo	Detalle
Código	CU-05
Nombre	Generar casos de prueba con IA
Actor principal	Usuario
Objetivo	Acelerar el diseño de pruebas a partir de descripciones/criterios del endpoint.
Precondición	Asistente de IA configurado en Settings/OpenAI. Entidad, Sección y Proyecto creado.
Flujo básico (resumen)	1) Solicitar generación por proyecto/sección/entidad. 2) Recibir salida en formato estricto (bloques). 3) Revisar y aplicar inserciones.
Postcondición / Resultado	Casos generados e insertados, con trazabilidad en el proyecto.
Canal	UI+API (local)

Nota. Elaborado por el autor

Tabla 29. CU-06 — Obtener sugerencias de casos con IA

Campo	Detalle
Código	CU-06
Nombre	Obtener sugerencias de casos con IA
Actor principal	Usuario
Objetivo	Ampliar cobertura con propuestas en formato estructurado evitando duplicados.
Precondición	Asistente de IA configurado en Settings/OpenAI. Entidad, Sección y Proyecto creado.
Flujo básico (resumen)	1) Solicitar 5 sugerencias por entidad/sección. 2) Revisar sugerencias. 3) Aplicar las pertinentes o descartar.
Postcondición / Resultado	Sugerencias integradas como casos nuevos o descartadas.
Canal	UI+API (local)

Nota. Elaborado por el autor

Tabla 30. CU-07 — Componer y mantener suites de prueba

Campo	Detalle
Código	CU-07
Nombre	Componer y mantener suites de prueba
Actor principal	Usuario
Objetivo	Agrupar casos (y/o suites) para ejecuciones orientadas a objetivos.
Precondición	Tener casos de prueba creados.
Flujo básico (resumen)	1) Crear suite seleccionando casos/suites. 2) Editar composición. 3) Listar/filtrar o eliminar.
Postcondición / Resultado	Suite lista para ejecución.
Canal	UI+API (local)

Nota. Elaborado por el autor

Tabla 31. CU-08 — Ejecutar pruebas y monitorear en tiempo real

Campo	Detalle
Código	CU-08
Nombre	Ejecutar pruebas y monitorear en tiempo real
Actor principal	Usuario
Objetivo	Correr casos o suites y observar avance con SSE.
Precondición	Tener casos de prueba creados.
Flujo básico (resumen)	1) Iniciar ejecución por proyecto o suite. 2) Recibir eventos de progreso (started/progress/completed/failed). 3) Cancelar/finalizar si aplica.
Postcondición / Resultado	Ejecución registrada con estados por paso y métricas básicas.
Canal	UI+API (local)

Nota. Elaborado por el autor

Tabla 32. CU-09 — Consultar detalle y resumen de ejecuciones

Campo	Detalle
Código	CU-09
Nombre	Consultar detalle y resumen de ejecuciones
Actor principal	Usuario
Objetivo	Analizar resultados y desempeño de pruebas.
Precondición	Tener Ejecuciones de Prueba Creadas
Flujo básico (resumen)	1) Listar/filtrar ejecuciones. 2) Abrir detalle con escenarios, steps, errores y métricas. 3) Consultar resumen global.
Postcondición / Resultado	Evidencia disponible para toma de decisiones.
Canal	UI+API (local)

Nota. Elaborado por el autor

Tabla 33. CU-10 — Registrar y gestionar bugs

Campo	Detalle
Código	CU-10
Nombre	Registrar y gestionar bugs
Actor principal	Usuario
Objetivo	Documentar defectos detectados y su contexto.
Precondición	-
Flujo básico (resumen)	1) Crear bug desde ejecución fallida o manualmente. 2) Editar atributos (tipo/severidad/prioridad/estado). 3) Listar/filtrar; ver estadísticas y vista árbol.
Postcondición / Resultado	Bug trazado a casos/ejecuciones/endpoints con su evolución registrada.
Canal	UI+API (local)

Nota. Elaborado por el autor

Tabla 34. CU-11 — Sincronizar artefactos del proyecto

Campo	Detalle
Código	CU-11
Nombre	Sincronizar artefactos del proyecto
Actor principal	Usuario
Objetivo	Alinear el espacio de trabajo con lo registrado en el sistema.
Precondición	Tener una base de código de dónde actualizar.
Flujo básico (resumen)	1) Ejecutar sincronización por proyecto o masiva. 2) Revisar reporte (conteos, errores, tiempos).
Postcondición / Resultado	Consistencia restablecida entre archivos y registros.
Canal	UI+API (local)

Nota. Elaborado por el autor

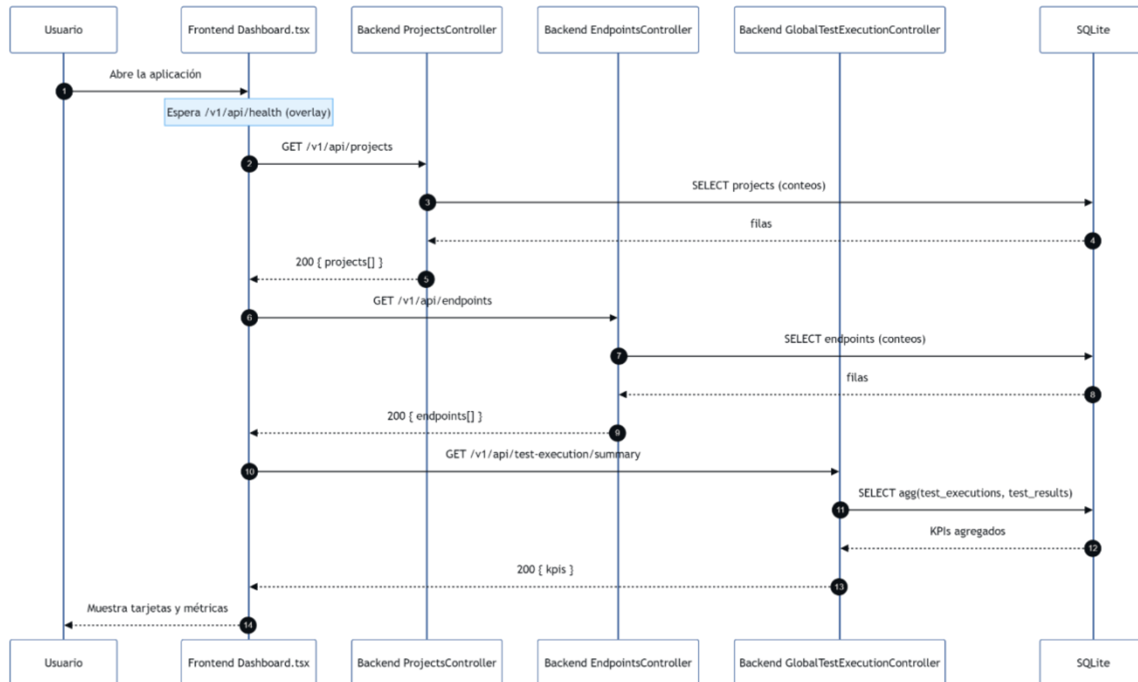
Tabla 35. CU-12 — Configurar proveedor de IA

Campo	Detalle
Código	CU-12
Nombre	Configurar proveedor de IA
Actor principal	Usuario
Objetivo	Habilitar funciones asistidas por IA.
Precondición	-
Flujo básico (resumen)	1) Guardar clave del proveedor de IA. 2) Probar conexión y verificar estado.
Postcondición / Resultado	Proyecto listo para generar/sugerir casos con IA.
Canal	UI (local)

Nota. Elaborado por el autor

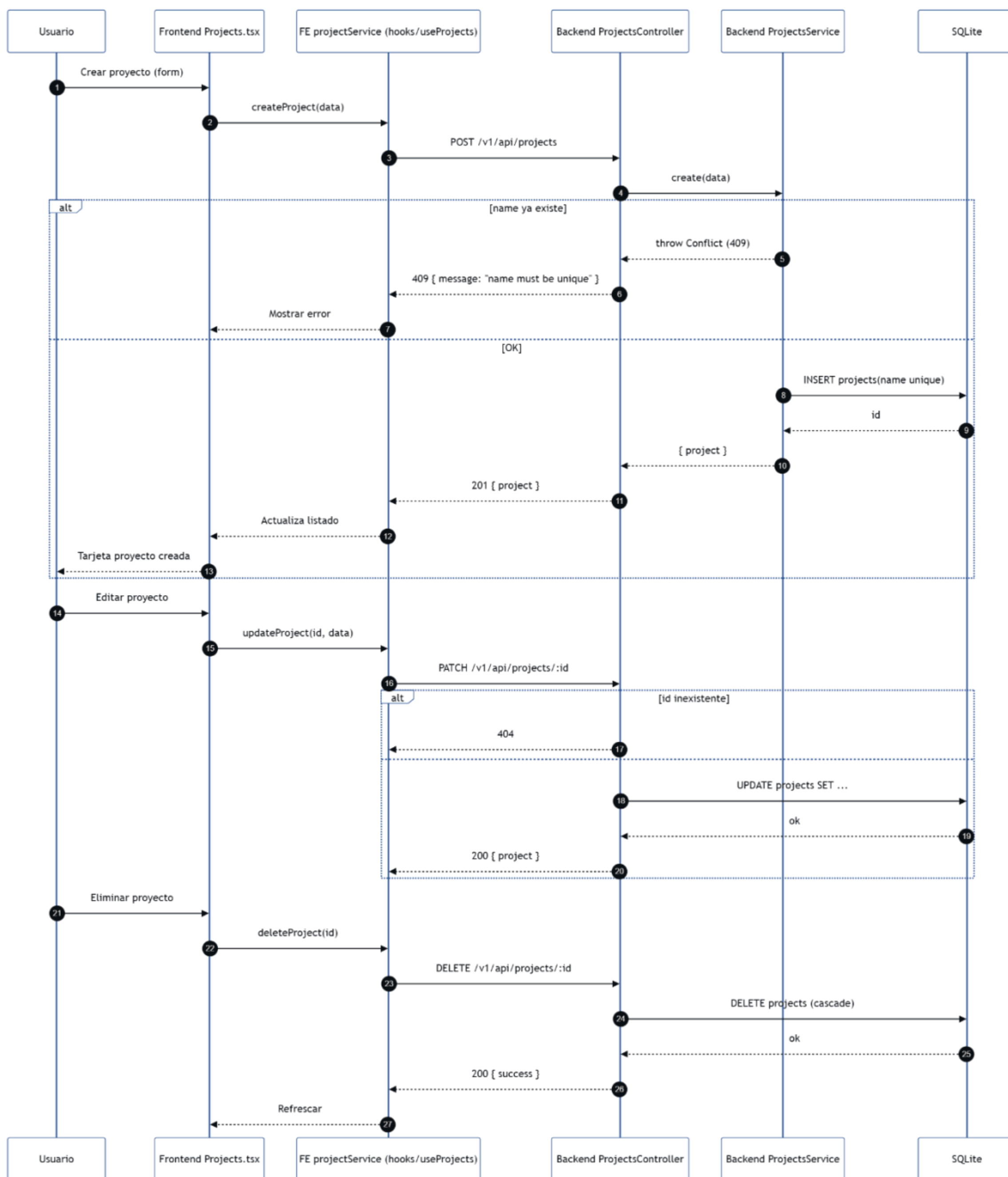
ANEXO B – Diagramas de Secuencia para las Principales Actividades del Sistema

Figura 46. Diagrama de secuencia: RF-01 — Dashboard (KPIs)



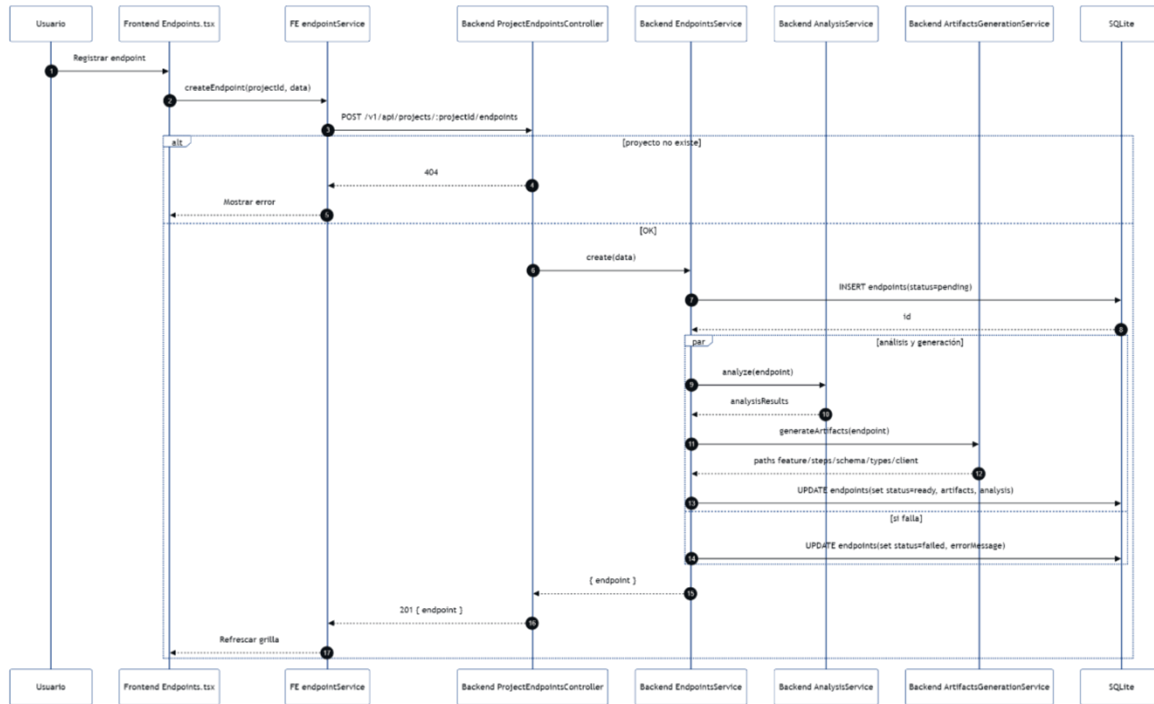
Nota. Elaborado por el autor

Figura 47. Diagrama de secuencia: RF-02 — Proyectos (CRUD)



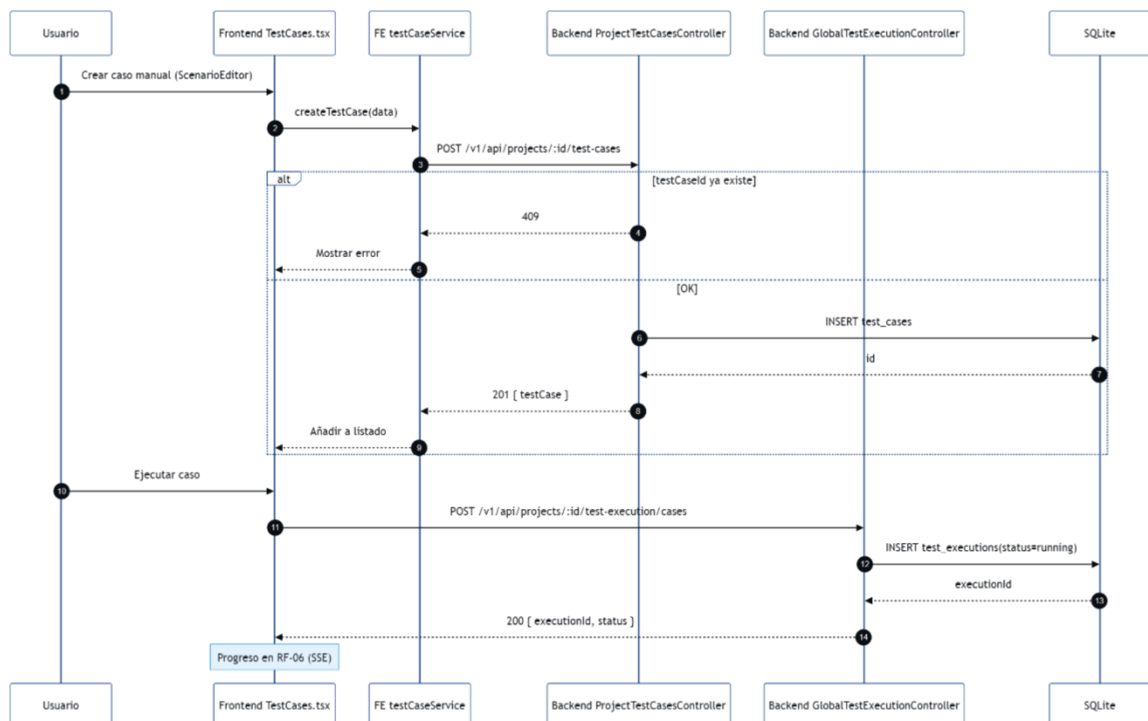
Nota. Elaborado por el autor

Figura 48. Diagrama de secuencia: RF-03 — Endpoints por proyecto (registro, análisis y artefactos)



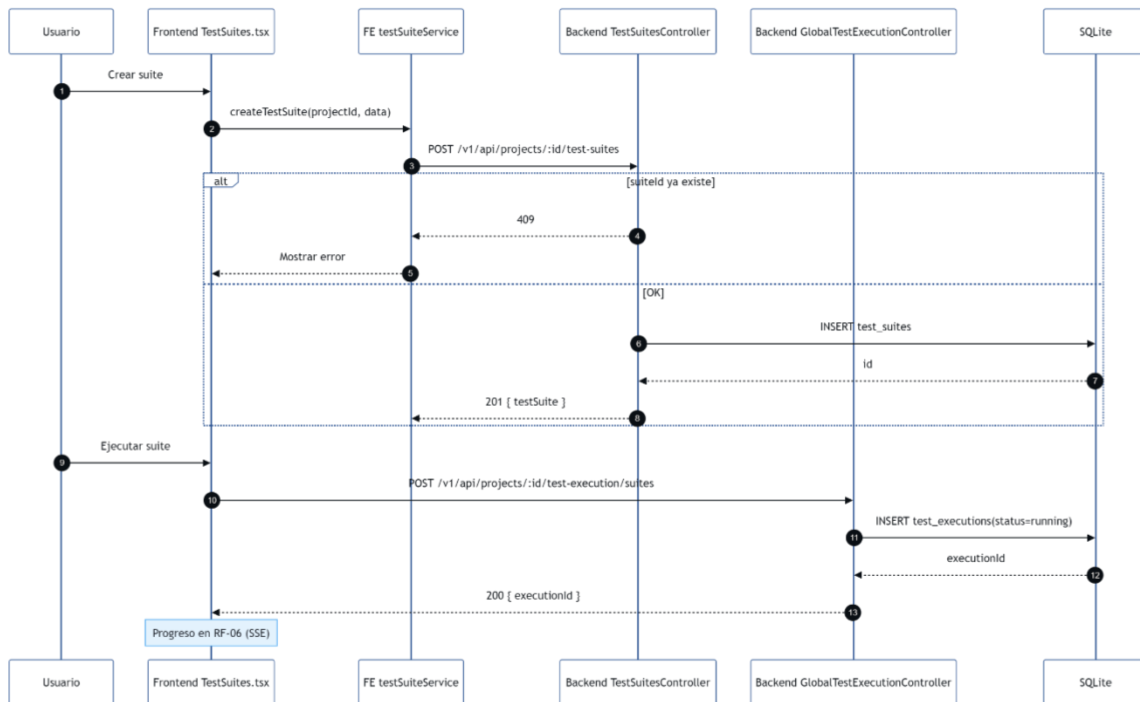
Nota. Elaborado por el autor

Figura 49. Diagrama de secuencia: RF-04 — Casos de prueba (manual BDD y ejecución desde la vista)



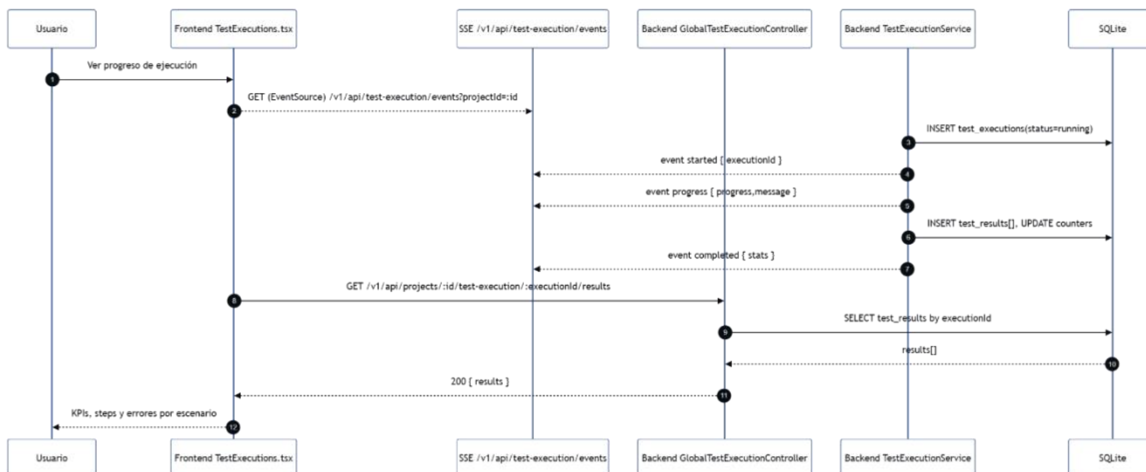
Nota. Elaborado por el autor

Figura 50. Diagrama de secuencia: RF-05 — Suites (crear, editar, listar y ejecutar)



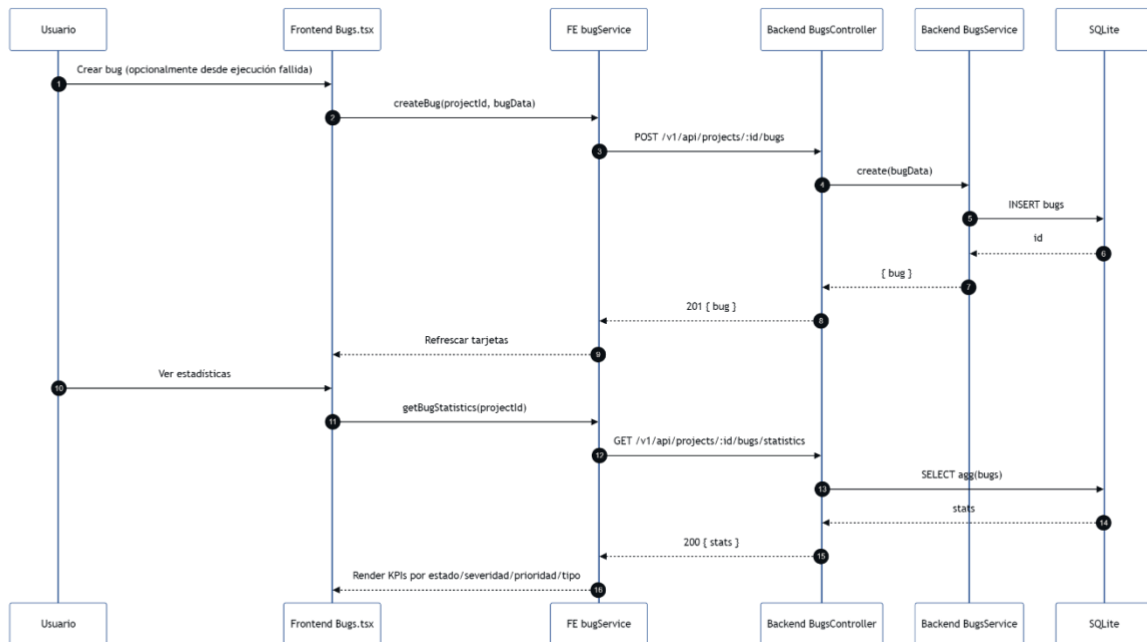
Nota. Elaborado por el autor

Figura 51. Diagrama de secuencia: RF-06 — Ejecuciones con SSE (progreso y resultados)



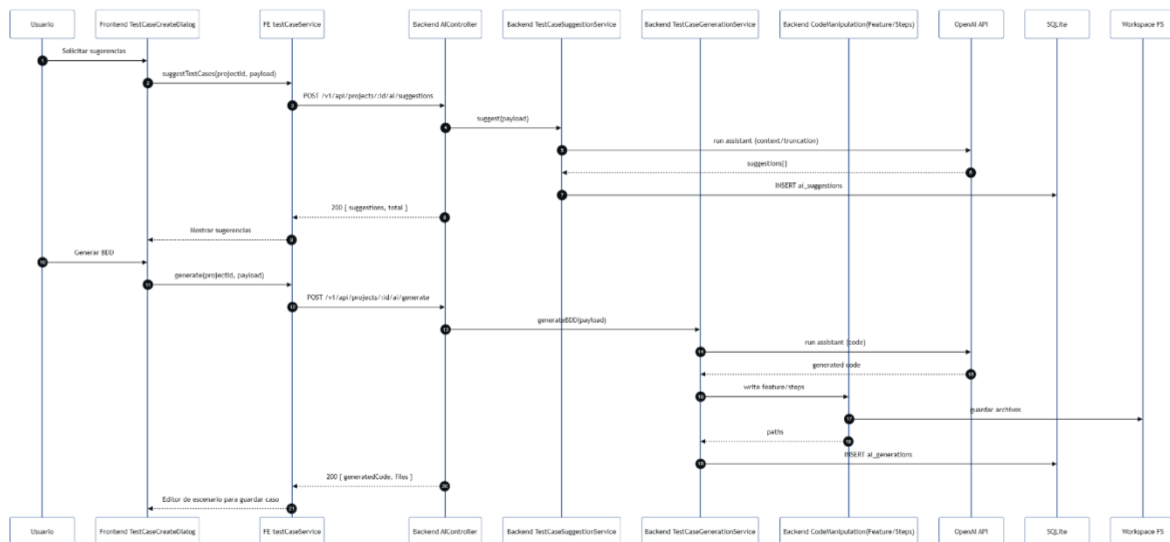
Nota. Elaborado por el autor

Figura 52. Diagrama de secuencia: RF-07 — Bugs/Defectos (crear, filtrar, estadísticas)



Nota. Elaborado por el autor

Figura 53. Diagrama de secuencia: RF-08 — Asistente de IA (sugerir y generar BDD)

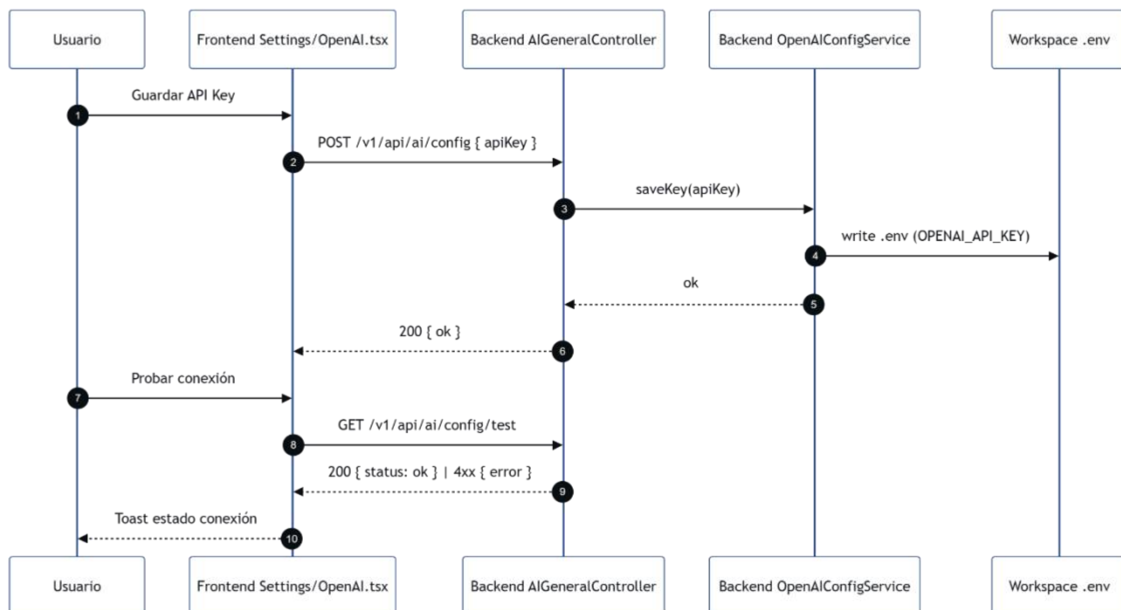


Nota. Elaborado por el autor

Figura 54. Diagrama de secuencia: RF-09 — Sincronización con workspace

Nota. Elaborado por el autor

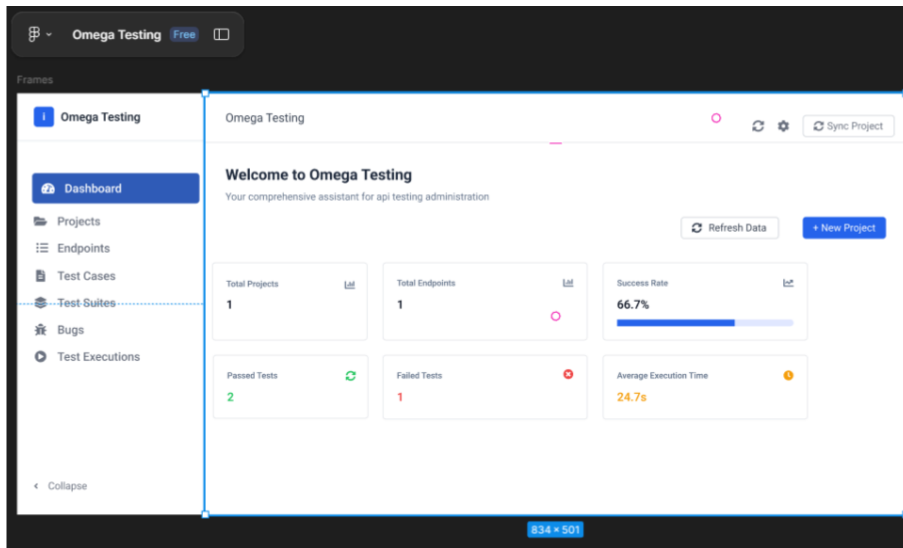
Figura 55. Diagrama de secuencia: RF-10 — Configuración y prueba de OpenAI API Key



Nota. Elaborado por el autor

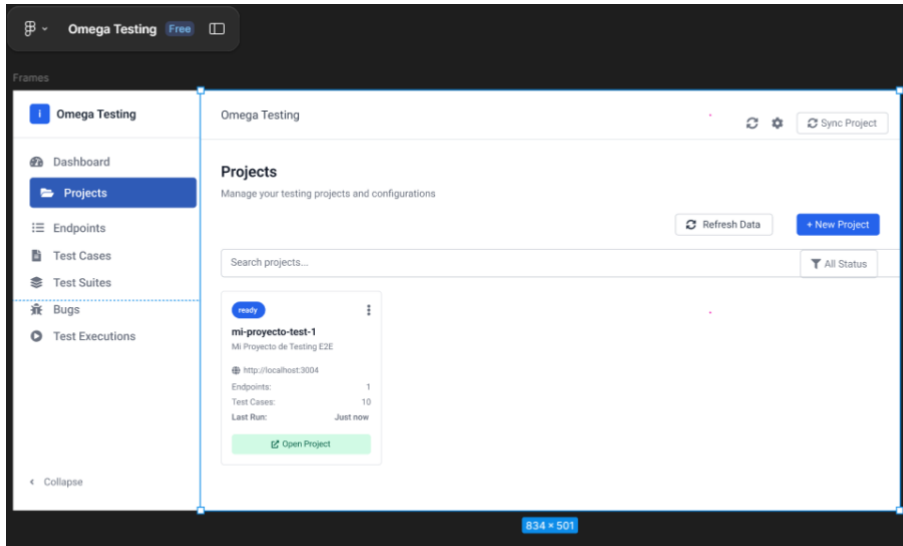
ANEXO C – Diseño de Pantallas de la Interfaz en Figma

Figura 56. Diseño del Dashboard central de KPI's en Figma



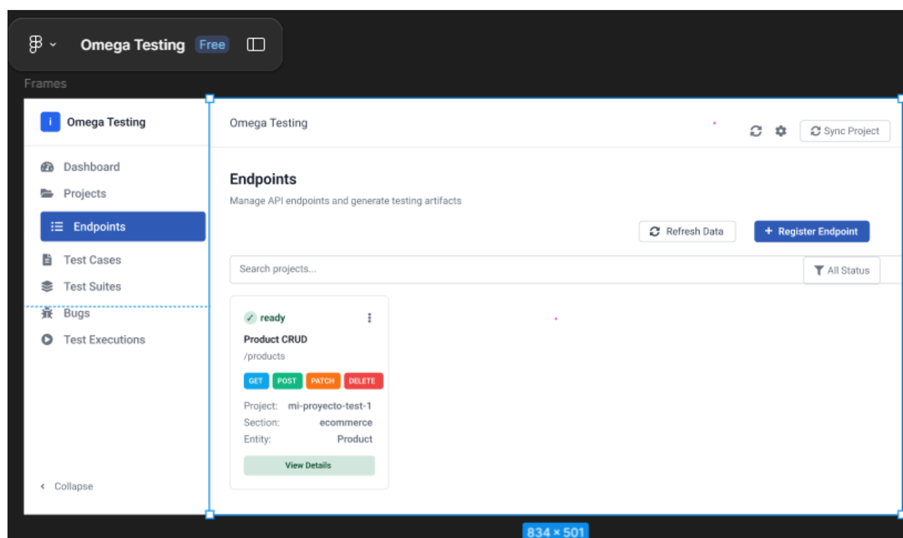
Nota. Elaborado por el autor

Figura 57. Diseño de Página de Projects



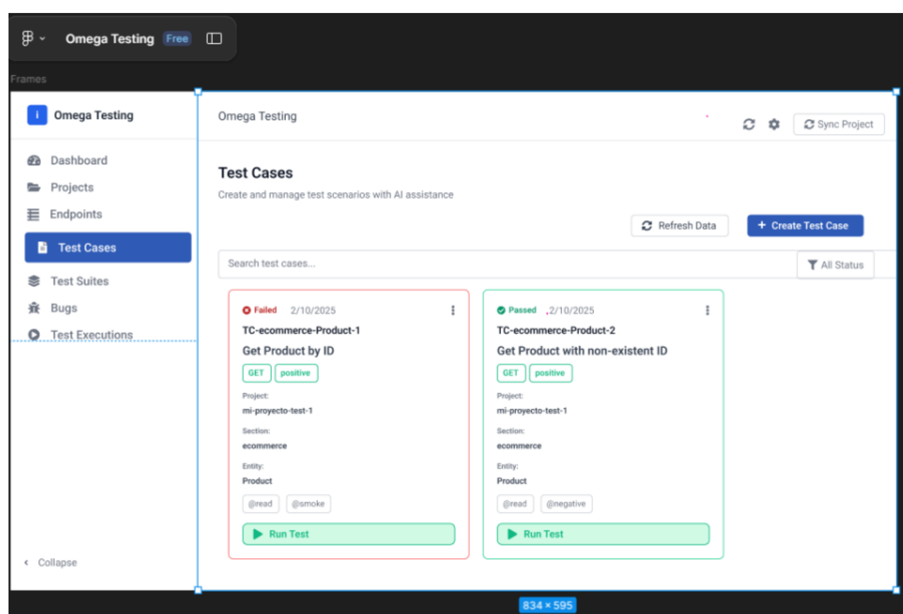
Nota. Elaborado por el autor

Figura 58. Diseño de Página de Endpoints



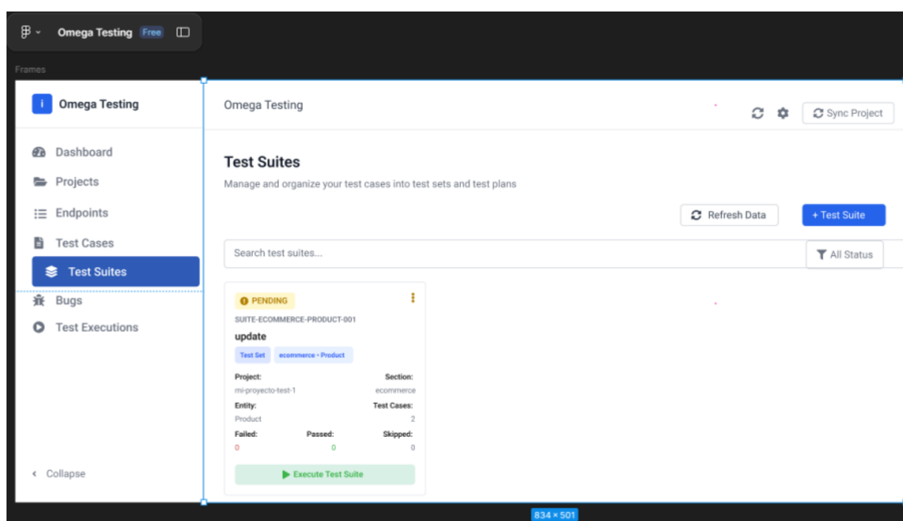
Nota. Elaborado por el autor

Figura 59. Diseño de Página de Test Cases



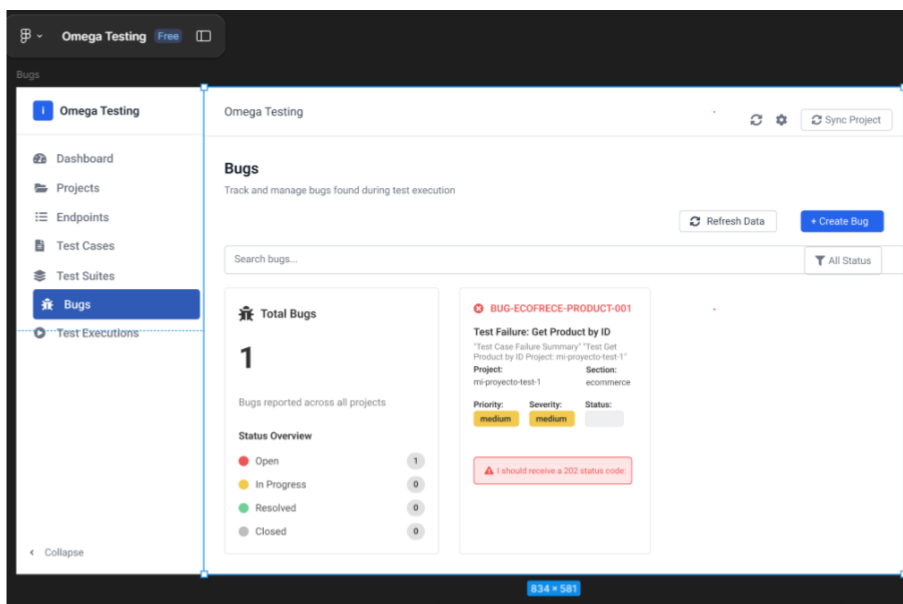
Nota. Elaborado por el autor

Figura 60. Diseño de Página de Test Suites



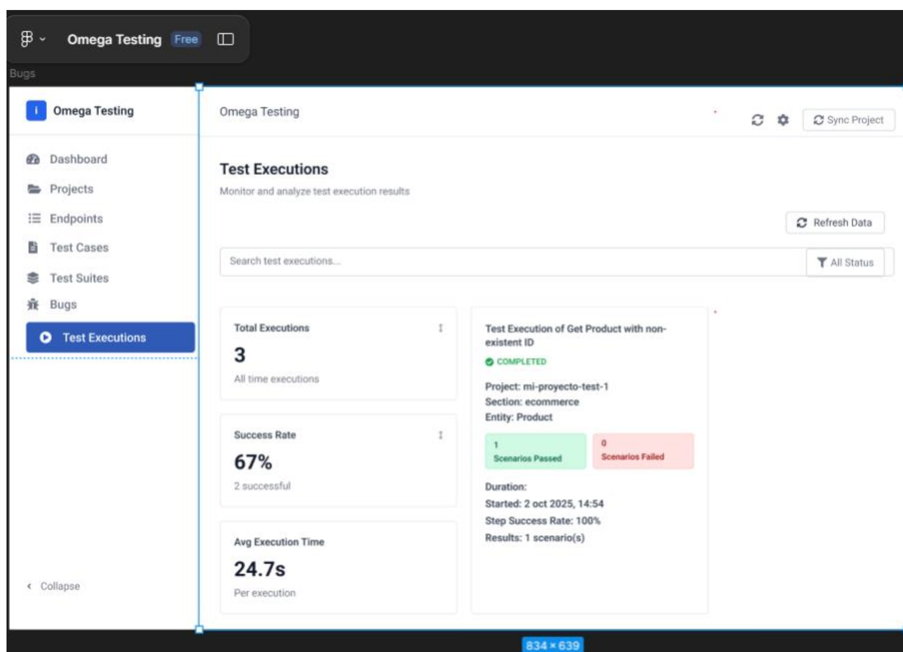
Nota. Elaborado por el autor

Figura 61. Diseño de Página de Bugs



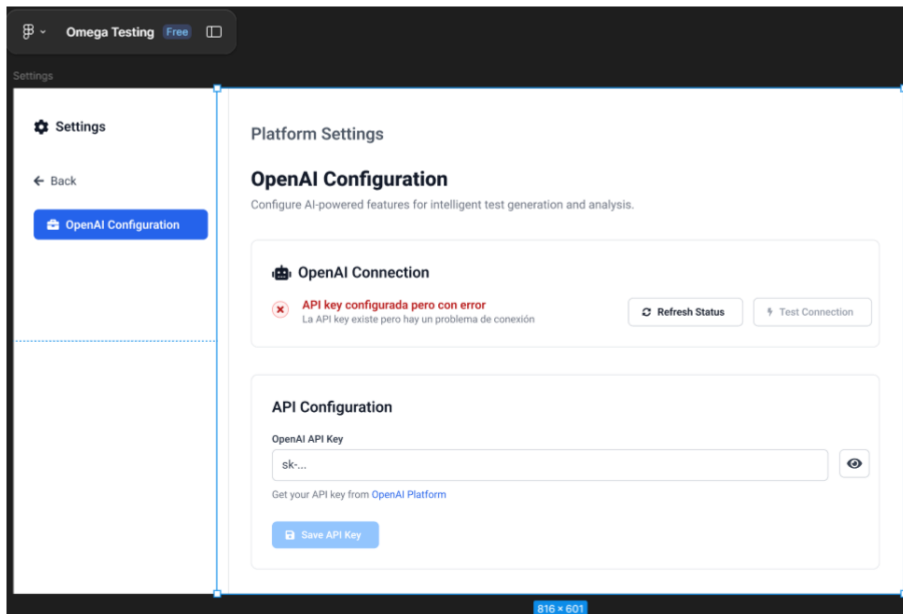
Nota. Elaborado por el autor

Figura 62. Diseño de Página de Test Executions



Nota. Elaborado por el autor

Figura 63. Diseño de Página de Settings



Nota. Elaborado por el autor

ANEXO D – Preguntas de la Encuesta realizada

Preguntas de Usabilidad de escala de puntuación de 1 a 5

1. ¿Crees que te gustaría usar una herramienta de testing con este diseño con frecuencia?

2. ¿Consideras que el diseño de esta herramienta de testing es intuitiva y comprensible?

3. ¿Te resultó sencillo realizar las tareas pedidas en esta herramienta de testing ?

4. ¿Crees que puedes esta herramienta de testing sin necesidad de ayuda?

7. ¿Imaginas que la mayoría de testers podrían aprender a usar esta herramienta de testing fácilmente?

5. ¿Piensas que las funciones disponibles en esta herramienta de testing están bien integradas?

6. ¿Crees que esta herramienta de testing es consistente en su diseño y funcionamiento?

8. ¿Te pareció que esta herramienta de testing es cómoda y fácil de manejar?

9. ¿Te sentiste seguro(a) y confiado(a) al usar esta herramienta de testing?

10. ¿Consideras que fue fácil comenzar a usar esta herramienta de testing sin necesidad de aprender muchas cosas?

Preguntas de Desempeño Con Escala Porcentual

11. Impacto en la eficiencia operativa

¿En qué medida considera que el uso del agente inteligente reduce el tiempo total requerido para la generación y ejecución de pruebas funcionales en APIs REST, comparado con los métodos tradicionales?

0–10% → Reducción mínima o nula

11–30% → Reducción moderada

31–50% → Reducción significativa

51–70% → Reducción alta

Más del 70% → Reducción muy alta

12. Impacto en la calidad y cobertura de pruebas

¿En qué grado considera que el agente inteligente mejora la calidad de las pruebas, entendida como mayor cobertura, detección temprana de errores y consistencia en los resultados?

0–10% → Mejora mínima o no apreciable

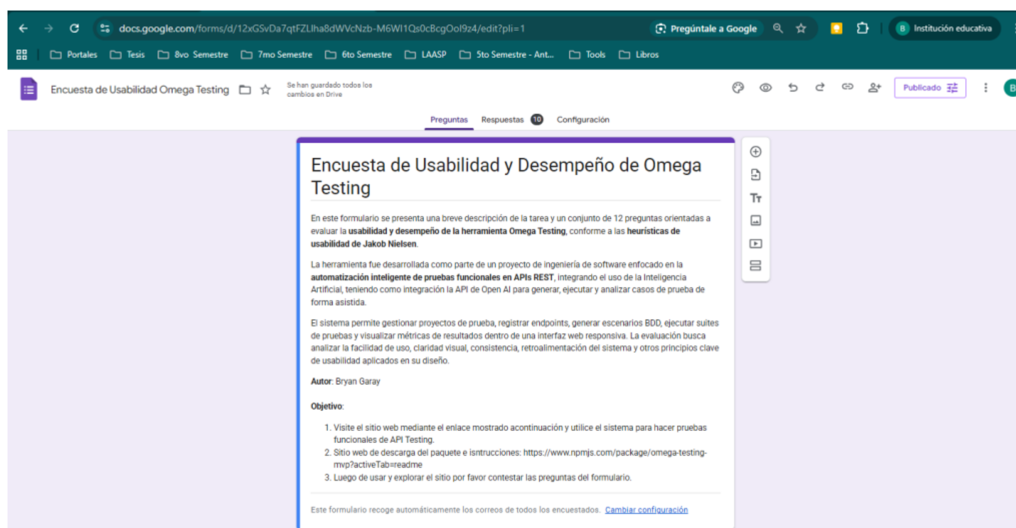
11–30% → Mejora parcial

31–50% → Mejora considerable

51–70% → Mejora alta

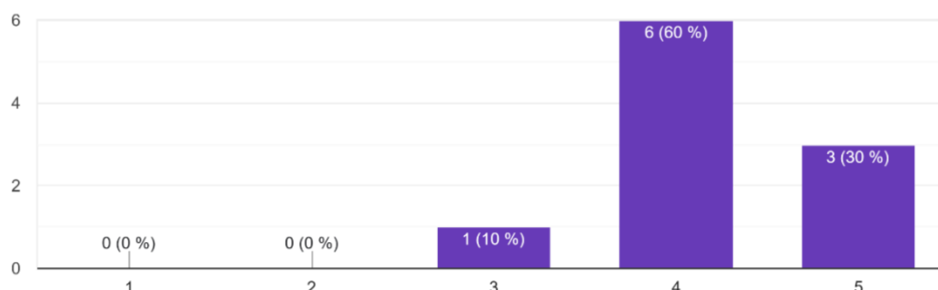
Más del 70% → Mejora muy alta

Figura 64. Capturas de Pantalla de ejemplo como evidencia de encuesta realizada en Google Forms



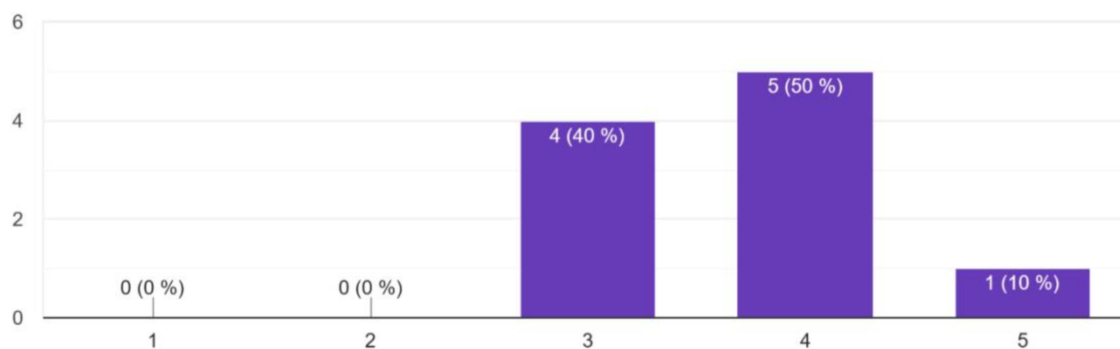
Nota. Elaborado por el autor

Figura 65. Resultado de encuestas – Pregunta 1



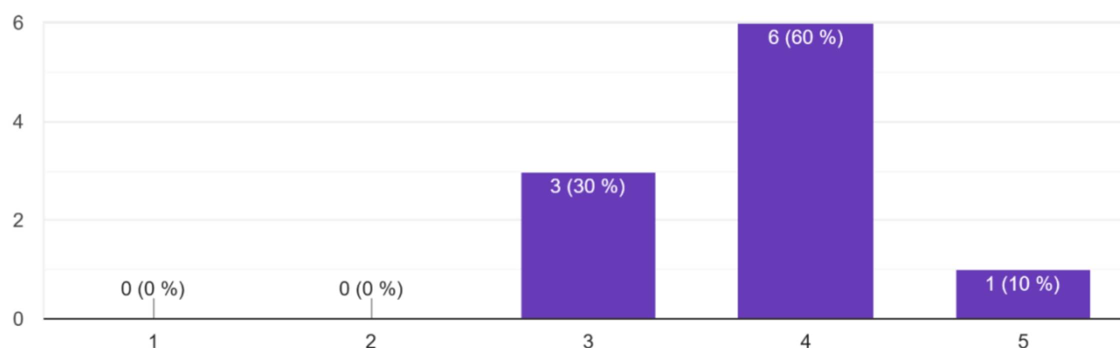
Nota. Elaborado por el autor

Figura 66. Resultado de encuestas – Pregunta 2



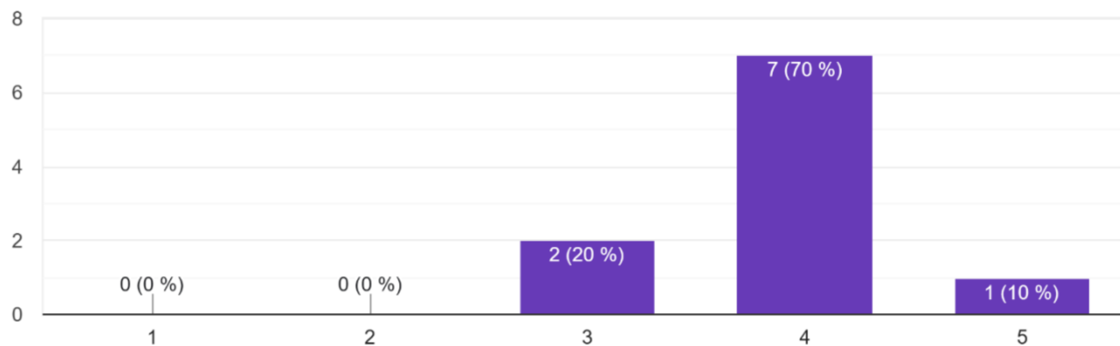
Nota. Elaborado por el autor

Figura 67. Resultado de encuestas – Pregunta 3



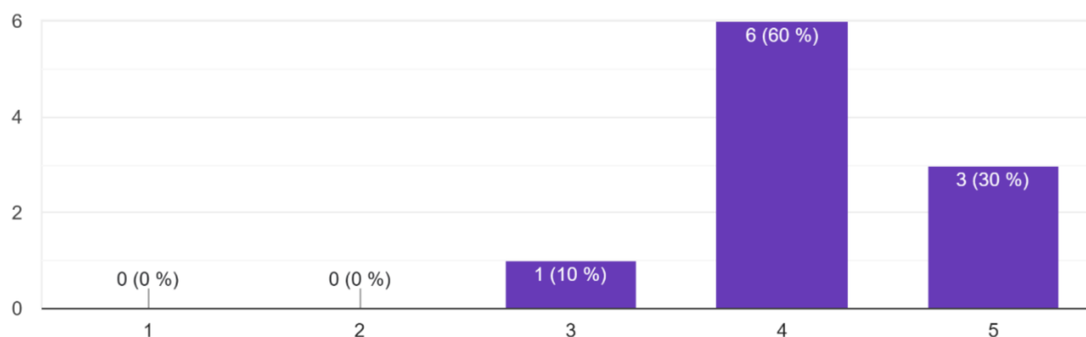
Nota. Elaborado por el autor

Figura 68. Resultado de encuestas – Pregunta 4



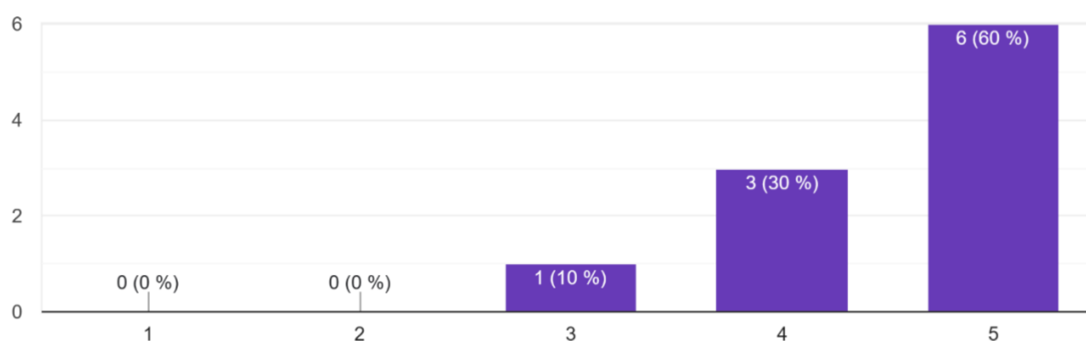
Nota. Elaborado por el autor

Figura 69. Resultado de encuestas – Pregunta 5



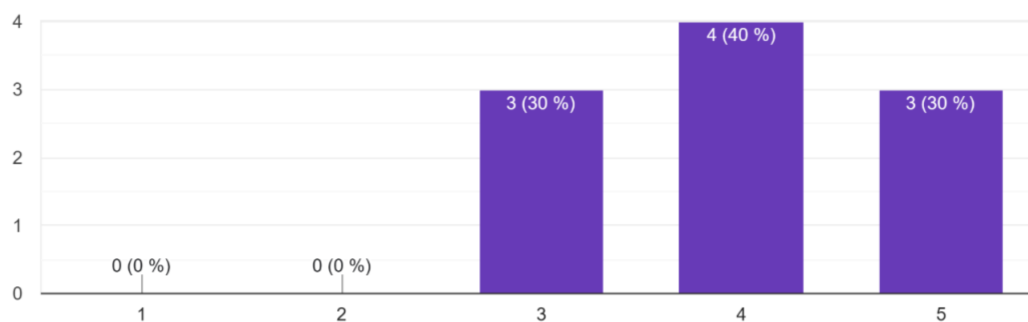
Nota. Elaborado por el autor

Figura 70. Resultado de encuestas – Pregunta 6



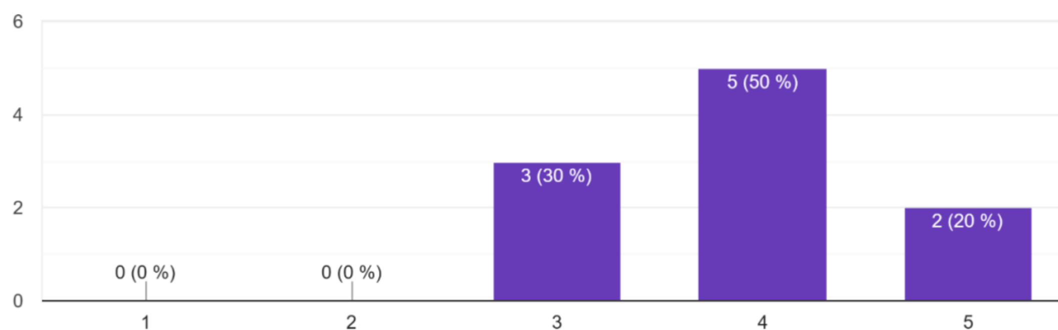
Nota. Elaborado por el autor

Figura 71. Resultado de encuestas – Pregunta 7



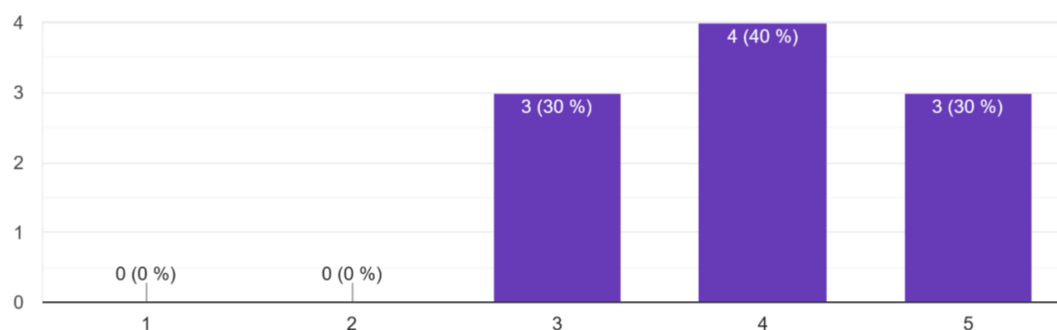
Nota. Elaborado por el autor

Figura 72. Resultado de encuestas – Pregunta 8



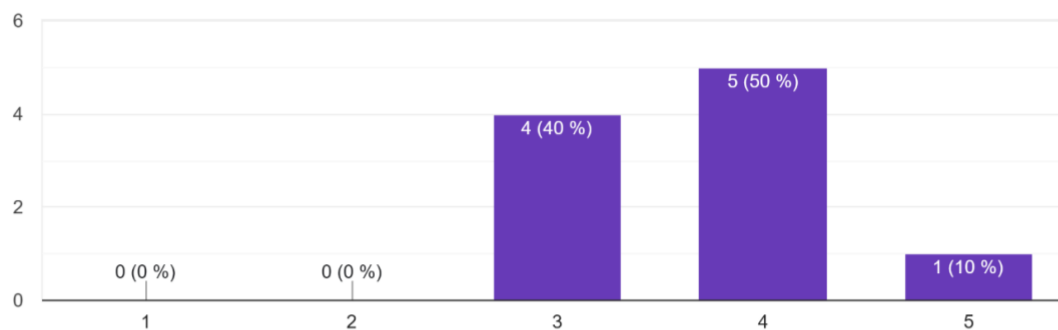
Nota. Elaborado por el autor

Figura 73. Resultado de encuestas – Pregunta 9



Nota. Elaborado por el autor

Figura 74. Resultado de encuestas – Pregunta 10



Nota. Elaborado por el autor

ANEXO E – Instrucciones Dadas Para las Pruebas de Expertos

Instrucciones y pruebas del sistema Omega Testing:

El presente anexo describe las instrucciones proporcionadas a los evaluadores expertos para la instalación, configuración y validación funcional del sistema Omega Testing, con el objetivo de ejecutar un conjunto homogéneo de actividades y recopilar retroalimentación comparable entre participantes.

1. Prerrequisitos

Para la correcta ejecución del sistema Omega Testing se consideraron los siguientes requisitos:

- Sistema operativo Windows 10 o superior.
- Node.js instalado en una versión igual o superior a 18.0.0 (requisito obligatorio).
- Acceso a internet para descarga de dependencias y revisión de documentación.

2. Instalación del sistema

1. Acceder a la página del paquete en npm:
   https://www.npmjs.com/package/omega-testing-mvp
2. Revisar la documentación disponible en el README para comprender el alcance de la herramienta.
3. Crear una carpeta nueva en un directorio local y abrir una terminal en dicha ubicación.
4. Ejecutar los comandos de instalación y arranque local:
   npm i omega-testing-mvp
5. npx omega-testing-mvp start-local
6. Una vez iniciado el servicio, acceder a la interfaz web del sistema desde el navegador.

3. Actividades de prueba solicitadas (gestión por módulos)

Las actividades solicitadas se orientaron a verificar operación básica de módulos, navegación y consistencia del flujo. En cada módulo, el evaluador debía realizar acciones de visualización y gestión (crear, actualizar y eliminar) cuando estuvieran disponibles en la interfaz.

3.1. Dashboard (monitoreo)

- Visualizar el tablero principal.

- Revisar los KPI's y métricas mostradas, verificando que carguen correctamente y reflejen información coherente con el estado del sistema.

3.2. Settings (configuración)

- Visualizar la sección de configuración.
- Actualizar la configuración de token de IA.

3.3. Projects (gestión de proyectos)

- Visualizar el listado de proyectos existentes.
- Crear un nuevo proyecto.
- Actualizar información del proyecto creado (por ejemplo, nombre o atributos disponibles).
- Eliminar el proyecto creado.

3.4. Endpoints (gestión de endpoints)

- Visualizar el listado de endpoints asociados a un proyecto.
- Crear un endpoint (según las opciones habilitadas por el sistema).
- Actualizar información del endpoint.
- Eliminar el endpoint creado.

3.5. Test Cases (gestión de casos de prueba)

- Visualizar los casos de prueba generados y su detalle.
- Crear al menos un caso de prueba (manual o con IA y probar sugerencias de casos de prueba).
- Actualizar el caso creado.
- Eliminar el caso de prueba creado.

3.6. Test Suites (gestión de suites)

- Visualizar el listado de suites existentes.
- Crear una suite de prueba de tipo test set o test plan y asociar casos de prueba a la suite.
- Actualizar la configuración o composición de la suite.
- Eliminar la suite creada.

3.7. Test Executions (gestión de ejecuciones)

- Visualizar el historial de ejecuciones y el detalle de resultados.
- Crear/Lanzar una ejecución de pruebas (por ejemplo, ejecutando una suite o conjunto de casos).
- Revisar el resultado, estados y trazabilidad de la ejecución.
- Eliminar una ejecución registrada para validar consistencia del módulo.

3.8. Bugs (registro y trazabilidad de defectos)

- Visualizar el listado de defectos registrados.
- Revisar el detalle de un defecto, incluyendo su relación con la ejecución o caso de prueba (si aplica).
- Actualizar el estado del defecto (por ejemplo abierto, en proceso, resuelto) o agregar información complementaria.
- Eliminar un registro de defecto para validar gestión básica.

4. Criterio de cierre de la prueba

La prueba se consideró completada cuando el evaluador:

- ejecutó actividades de visualización y gestión en los módulos principales,
- verificó la persistencia de cambios,
- revisó resultados de ejecuciones y defectos,
- y posteriormente respondió la encuesta de evaluación y retroalimentación.