

INGENIERÍA EN SISTEMAS COMPUTACIONALES

Semestre

9º "A"

Nombre de la materia

Residencias Profesionales

Nombre del alumno

Angel David Mariscal Soto

Proyecto

Irakani Builder

Empresa

ITERADAPTA (Irakani)

Fecha

19 de diciembre del 2025

*Agradecimientos

Quiero expresar mi más sincero agradecimiento a todas las personas que hicieron posible la realización de este proyecto y que me acompañaron durante mi formación profesional.

A **ITERADAPTA (Irakani)**, por brindarme la oportunidad de desarrollar este proyecto en un ambiente profesional, por la confianza depositada en mí y por permitirme crecer tanto personal como profesionalmente durante mi residencia.

A mi **mamá**, por su apoyo incondicional, su amor y su constante motivación. Gracias por creer en mí y por impulsarme a seguir adelante en cada momento de mi carrera.

A mi **papá**, por su ejemplo, sus consejos y por enseñarme el valor del esfuerzo y la dedicación.

A mis **hermanos**, por su compañía, su apoyo y por hacer más llevaderos los momentos difíciles durante esta etapa.

A mis **amigos**, por su amistad sincera, por los momentos compartidos y por estar presentes en cada paso de este camino universitario.

A mi **pareja**, por su amor, paciencia y comprensión. Gracias por acompañarme en esta etapa de mi vida y por ser mi apoyo en los momentos más importantes.

A todos ustedes, mi más profundo agradecimiento.

Angel David Mariscal Soto

*Resumen

El presente documento describe el desarrollo del proyecto **Irakani Builder**, realizado durante el periodo de residencias profesionales en la empresa ITERADAPTA (Irakani).

Irakani Builder es una plataforma innovadora que permite la creación de aplicaciones de manera visual e intuitiva, sin necesidad de conocimientos avanzados de programación. El sistema integra un editor visual, gestión de bases de datos, generación automática de código y múltiples herramientas que facilitan el desarrollo de aplicaciones empresariales.

El proyecto aborda la problemática de la complejidad en el desarrollo de software, proporcionando una solución que democratiza la creación de aplicaciones y reduce significativamente los tiempos de desarrollo. A través de una arquitectura modular y escalable, Irakani Builder permite a usuarios con diferentes niveles de experiencia técnica crear aplicaciones funcionales y profesionales.

Durante el desarrollo se implementaron tecnologías modernas como React, Node.js y diversas herramientas de inteligencia artificial, logrando un sistema robusto, eficiente y fácil de usar.

Palabras clave: Low-code, No-code, Desarrollo visual, Constructor de aplicaciones, Automatización, Inteligencia artificial.

Índice

Agradecimientos	1
Resumen	2
1. Generalidades del proyecto	6
1.1. Antecedentes del Problema(s)	6
1.2. Planteamiento del Problema(s)	6
1.3. Objetivo General	7
1.4. Objetivos Específicos	7
1.5. Justificación	8
1.6. Limitaciones	8
1.7. Delimitaciones	9
2. Marco contextual	10
2.1. Nombre de la empresa	10
2.2. Giro	10
2.3. Dirección	10
2.4. Misión	10
3. Marco teórico	10
3.1. Marco Contextual y Problema de Negocio	11
3.1.1. La Evolución del Desarrollo de Software: Del Código Manual a la Asistencia Inteligente	11
3.1.2. El Contexto Operativo Interno: La Plataforma app.irakani.com como Deuda Técnica	11
3.2. Marco Teórico y Tecnológico	12
3.2.1. Paradigmas de Desarrollo de Software	12
3.2.2. Arquitecturas de Software Modernas	13
3.2.3. Tecnologías Clave del Stack de Desarrollo	14
3.2.4. Integración de Inteligencia Artificial Generativa	15
3.3. Marco Metodológico	15
3.3.1. Fundamento Metodológico: Del Control Predictivo al Empírico	16
3.3.2. Adopción del Framework Scrum	16
3.3.3. Fases del Proceso de Desarrollo Metodológico	18
3.3.4. Herramientas de Soporte a la Metodología	19
4. Desarrollo de la Solución	19
4.1. Metodología Scrum Aplicada	19
4.1.1. Roles del Equipo	20
4.1.2. Duración y Estructura de Sprints	20
4.1.3. Ceremonias Implementadas	20
4.2. Conformación del Product Backlog	20
4.2.1. Épicas del Proyecto	20
4.2.2. Identificación de Historias de Usuario (Requerimientos funcionales)	22
4.2.3. Priorización de tareas en Jira	29
4.3. Diseño de la Arquitectura y Flujo de Datos	30
4.3.1. Arquitectura de Microservicios y Comunicación	30
4.3.2. Estrategia de Integración de Datos (API Wrapper)	31
4.3.3. Diseño de Gestión de Sesiones y Caché (Valkey)	37
4.3.4. Diseño de la Integración con IA (AWS Bedrock)	41
4.3.5. Diseño de la Interfaz de Usuario (UI/UX)	48
4.3.6. Diagramas de Flujo y Casos de Uso	64
4.4. Implementación de la Solución	71
4.4.1. Configuración del Entorno y CI/CD	71
4.4.2. Desarrollo del Backend (BFF)	77

4.4.3. Integración con AWS Bedrock	86
4.4.4. Desarrollo del Frontend	94
4.5. Planificación de Sprints	105
4.5.1. Sprint 0: Concepción y Diseño (14-27 julio 2025)	106
4.5.2. Sprint 1: Autenticación y Seguridad (28 julio - 10 agosto 2025)	106
4.5.3. Sprint 2: Núcleo de IA (11-24 agosto 2025)	107
4.5.4. Sprint 3: Interfaz Base (25 agosto - 7 septiembre 2025)	108
4.5.5. Sprint 4: Editor Visual (8-21 septiembre 2025)	108
4.5.6. Sprint 5: Renderizado y Bases de Datos (22 septiembre - 5 octubre 2025)	109
4.5.7. Sprint 6: Gestión de Datos Avanzada (6-19 octubre 2025)	109
4.5.8. Sprint 7: Asistente de Chat y Sesiones (20 octubre - 2 noviembre 2025)	110
4.5.9. Sprint 8: Gestión de Componentes (3-16 noviembre 2025)	111
4.5.10. Sprint 9: Personalización (17-30 noviembre 2025)	111
4.5.11. Sprint 10: Despliegue y Validación Final (1-14 diciembre 2025)	112
4.5.12. Sprint 11: Buffer y Cierre (15-19 diciembre 2025)	113
4.6. Pruebas y Validación	114
4.6.1. Estrategia de Testing	114
4.6.2. Proceso de Pruebas Unitarias Manuales	114
4.6.3. Verificación de Funcionalidades de IA	115
4.6.4. Validación con Usuario del Área de Ventas	117
4.6.5. Resumen del Proceso de Validación	118
4.7. Despliegue y Puesta en Producción	119
4.7.1. Pipeline CI/CD	119
4.7.2. Infraestructura AWS	123
4.7.3. Monitoreo y Observabilidad	124
4.7.4. Manejo de Incidentes y Corrección de Errores	129
5. Análisis de Resultados	131
5.1. Resultados Funcionales	131
5.1.1. Módulos finalizados	131
5.1.2. Funcionamiento del Editor Visual y Chat con IA	132
5.2. Análisis de Rendimiento y Tiempos	133
5.2.1. Comparativa de tiempos de desarrollo	133
5.2.2. Métricas de eficiencia operativa	134
5.3. Resultados de la Integración de IA	135
5.3.1. Precisión del código generado	135
5.3.2. Análisis de costos y consumo de tokens	136
5.4. Resultados de las Pruebas de Usuario	136
5.4.1. Percepción de eficiencia y productividad	136
5.4.2. Evaluación de calidad de la IA y Usabilidad	137
5.4.3. Interpretación global de la encuesta de satisfacción	137
6. Conclusiones y Recomendaciones	139
6.1. Conclusiones	140
6.1.1. Cumplimiento del objetivo general	140
6.1.2. Impacto en la deuda técnica	141
6.2. Recomendaciones	142
6.2.1. Trabajo a futuro y mejoras potenciales	142

7. Competencias Desarrolladas	142
7.1. Competencias Técnicas (Hard Skills)	143
7.1.1. Desarrollo Frontend	143
7.1.2. Desarrollo Backend	143
7.1.3. Inteligencia Artificial	143
7.1.4. Arquitectura de Software	144
7.1.5. DevOps y Despliegue	144
7.2. Competencias Blandas (Soft Skills)	144
7.2.1. Gestión de Proyectos	144
7.2.2. Resolución de Problemas	145
7.2.3. Comunicación	145
7.2.4. Aprendizaje Continuo	145
7.2.5. Pensamiento Crítico	146
7.2.6. Adaptabilidad	146
7.2.7. Atención al Detalle	146
7.2.8. Orientación a Resultados	146
7.2.9. Trabajo en Equipo	146
7.3. Integración de Competencias y Reflexión	147
7.4. Impacto en el Desarrollo Profesional	147
8. Anexos	148
8.1. Anexo A: Glosario de Términos	148
8.1.1. Términos Técnicos	148
8.2. Anexo B: Encuesta de Satisfacción de Usuarios	153
8.2.1. Descripción de la Encuesta	153
8.2.2. Instrumento de Evaluación	153

1. Generalidades del proyecto

1.1. Antecedentes del Problema(s)

La estructura operativa de Irakani (ITERADAPTA) se organiza en tres áreas técnicas principales: Proyectos, Soporte Técnico y Plataforma. El área de Proyectos es la responsable directa de desarrollar y entregar soluciones personalizadas (aplicaciones móviles y web) a los clientes finales. Para lograrlo, su flujo de trabajo depende fundamentalmente de un ecosistema de herramientas internas gestionado por el área de Plataforma, en la cual se enmarca esta investigación.

El componente central de este ecosistema es la plataforma de desarrollo low-code app.irakani.com, una aplicación web construida con frameworks obsoletos. Esta herramienta fue concebida para que el equipo de Proyectos pudiera construir y desplegar aplicaciones de manera ágil. Sin embargo, con el paso del tiempo, esta plataforma interna se ha convertido en el principal cuello de botella para la innovación y la eficiencia de la empresa, dando origen a la necesidad de desarrollar su sucesor: Irakani Builder.

Los problemas de app.irakani.com son de naturaleza técnica y estratégica, y se pueden desglosar de la siguiente manera:

Tecnología Obsoleta: El uso de un framework desactualizado es la causa raíz de múltiples inconvenientes, entre los que se incluyen:

- Incompatibilidad con navegadores web modernos, afectando la experiencia del usuario.
- Limitaciones severas para implementar nuevas funcionalidades demandadas por el mercado.
- Dificultades crecientes en el mantenimiento y la actualización del código base.

Limitaciones de Rendimiento: La arquitectura actual es ineficiente y no puede gestionar de manera óptima las demandas operativas, lo que se manifiesta en:

- Incapacidad para manejar múltiples usuarios concurrentes sin degradación del servicio.
- Lentitud en el procesamiento de aplicaciones complejas.
- Imposibilidad de una integración fluida con servicios de inteligencia artificial.
- Falta de una gestión optimizada de los recursos del servidor.

Falta de Integración con Inteligencia Artificial (IA): La ausencia de capacidades de IA es una de las mayores debilidades, limitando el potencial de la plataforma al impedir:

- La automatización de los procesos de desarrollo.
- La provisión de asistencia inteligente a los usuarios para facilitar la creación de aplicaciones.
- La optimización automática del código generado.
- La personalización de la experiencia del usuario a través de IA.

Estos problemas internos se ven agravados por las necesidades actuales del mercado, que demanda soluciones que ofrezcan desarrollo rápido, asistentes de IA que guíen el proceso, accesibilidad para usuarios sin conocimientos técnicos avanzados y una alta escalabilidad. La plataforma existente no puede satisfacer estas demandas, justificando así el desarrollo de una nueva solución desde cero.

1.2. Planteamiento del Problema(s)

La plataforma actual, app.irakani.com, representa un problema multifactorial que va más allá de la simple obsolescencia tecnológica. Su arquitectura y stack tecnológico anticuados se han convertido en un freno directo para la eficiencia operativa y la rentabilidad de la empresa, comprometiendo su competitividad. El problema se manifiesta en tres dimensiones críticas y cuantificables:

- 1. Problema de Tiempo y Rendimiento:** El ciclo de desarrollo actual utiliza un enfoque tradicional en el que un desarrollador ejecuta las fases de una metodología de desarrollo de software (análisis, diseño, desarrollo, pruebas y retroalimentación), lo que se traduce en tiempos de entrega prolongados y un bajo rendimiento de las aplicaciones finales. Tareas fundamentales que actualmente consumen jornadas completas, como la generación de una aplicación base (aproximadamente 12 horas) o la construcción de formularios complejos (4 horas), son relativamente lentas. Esta lentitud no solo retrasa los proyectos, sino que impide que la empresa asuma un mayor volumen de trabajo, limitando su capacidad de crecimiento.
- 2. Problema de Costos Operativos:** La ineficiencia en el tiempo se traduce directamente en altos costos de mano de obra. Un análisis de proyectos internos, como el caso de estudio “Ojo Zarco”, revela que una tarea que requería 6 horas-persona (dos desarrolladores durante tres horas) podría ejecutarse con la mitad del personal en el mismo tiempo. Esto indica que la plataforma actual no solo duplica el esfuerzo humano necesario, sino que eleva los costos de proyecto en aproximadamente un 33% en comparación con una solución optimizada. La incapacidad de la plataforma para escalar eficientemente también genera un sobrecosto en la infraestructura de servidores.
- 3. Problema de Innovación y Brecha Competitiva (IA):** La ausencia total de integración con Inteligencia Artificial (IA) es la deficiencia estratégica más grave. Mientras el mercado avanza hacia herramientas que automatizan y asisten el desarrollo, app.irakani.com mantiene a los desarrolladores en un paradigma de trabajo manual. Esto impide aprovechar las mejoras exponenciales de productividad que ofrece la IA, como la generación de código a partir de lenguaje natural, la optimización automática de consultas y la creación de recursos gráficos en minutos en lugar de horas. Esta carencia posiciona a la empresa en una clara desventaja competitiva, impidiéndole ofrecer soluciones más rápidas, económicas e innovadoras.

1.3. Objetivo General

Diseñar e implementar “Irakani Builder”, una plataforma web moderna que solucione la ineficiencia y el alto costo de desarrollo de la empresa. Esto se logrará mediante la construcción de una arquitectura basada en tecnologías actuales (React 18, Node.js) y, fundamentalmente, con la integración de un modelo de Inteligencia Artificial a través de AWS Bedrock. Dicha IA permitirá la generación automática de aplicaciones, código y componentes a partir de lenguaje natural, con el objetivo medible de reducir los tiempos de ciclo de desarrollo en un 30-50% y disminuir los costos de mano de obra asociados hasta en un 33%.

1.4. Objetivos Específicos

- Asimilar el ecosistema de Irakani.
- Analizar los casos de uso y flujo de trabajo del equipo de Proyectos para optimizar el nuevo diseño.
- Interpretar la estructura de las aplicaciones y bases de datos del sistema de Irakani.
- Comprender la interacción entre las áreas de Proyectos, Soporte Técnico y Plataforma de Irakani.
- Implementar una interfaz de usuario moderna y responsive usando React 18 y TypeScript.
- Crear componentes reutilizables para el editor visual de aplicaciones.
- Desarrollar un sistema de temas personalizable con más de 20 opciones predefinidas para la edición de código.
- Integrar Monaco Editor para una edición de código avanzada con resaltado de sintaxis.
- Desarrollar servicios de autenticación y autorización seguros.
- Crear endpoints específicos para la gestión de bases de datos y la generación.
- Integrar AWS Bedrock para la generación automática de código, aplicaciones, listas, perfiles y entidades.
- Integración de gestión de base de datos.
- Desarrollar y optimizar prompts para diferentes tipos de contextos.

- Implementar un sistema de chat inteligente que brinde asistencia durante el desarrollo.
- Crear servicios para el análisis y la optimización automática del código generado.
- Desarrollar un administrador de bases de datos asistido por IA.
- Implementar soporte para múltiples motores de bases de datos, incluyendo MySQL y SQL Server.
- Crear herramientas para la consulta inteligente y la optimización de bases de datos.
- Reducir en un 60% el tiempo necesario para la creación de aplicaciones básicas.
- Lograr una curva de aprendizaje optimizada en un 40% para los nuevos usuarios.

1.5. Justificación

El desarrollo del proyecto Irakani Builder se justifica como una transformación estratégica indispensable, diseñada para resolver la ineficiencia operativa y los altos costos impuestos por la plataforma app.irakani.com. Esta iniciativa no es una simple actualización tecnológica, sino una reinvención de la capacidad productiva de la empresa, anclada en métricas proyectadas y casos de estudio internos que demuestran un impacto directo en la eficiencia, la rentabilidad y la competitividad.

La justificación se cimenta en los siguientes resultados cuantificables esperados:

Reducción Radical de Tiempos y Costos: La principal justificación del proyecto es su impacto financiero y operativo. La integración de Inteligencia Artificial a través de AWS Bedrock y una arquitectura moderna automatizará tareas clave que actualmente consumen una cantidad significativa de recursos.

Métricas de Eficiencia: El resultado más tangible será una drástica reducción del tiempo de desarrollo, con ahorros proyectados del 50% en la generación de aplicaciones completas, 40% en la construcción de formularios y hasta un 60% en tareas de diseño como la creación de iconos.

Impacto en Costos: Esta ganancia de eficiencia se traduce en una reducción directa de los costos de mano de obra. El caso de estudio interno “Ojo Zarco” demostró que una tarea que tradicionalmente requería 6 horas-persona (dos desarrolladores) se completó con solo 3 horas-persona (un desarrollador), logrando una reducción del 33% en el costo total del proyecto y requiriendo un 50% menos de personal.

Mejora de la Capacidad de Negocio y Satisfacción del Cliente: Al acelerar los ciclos de desarrollo, la empresa podrá asumir un mayor volumen de proyectos y de mayor complejidad.

Mejora del Rendimiento y la Calidad: Es fundamental subrayar que Irakani Builder no se concibe como un sustituto del desarrollador, sino como un potente aliado para aumentar su capacidad. La plataforma está diseñada para ser una herramienta de apoyo que agiliza drásticamente las fases iniciales del desarrollo, como la creación de prototipos funcionales, y reduce errores comunes al automatizar tareas repetitivas. Al liberar a los desarrolladores de estas labores, pueden enfocar su talento en la lógica de negocio compleja, la optimización y la obtención de ideas y requerimientos de mayor valor, mejorando significativamente la calidad final del producto. Esta mejora en la calidad y fiabilidad de los entregables impactará directamente en una mayor satisfacción y retención de clientes.

Innovación y Ventaja Competitiva: El mercado exige herramientas que no solo sean rápidas, sino inteligentes. Irakani Builder responde a esta demanda al posicionar la IA como el núcleo de su propuesta de valor. Esto representa una ventaja competitiva decisiva, alineando a la empresa con estudios de mercado, como el de GitHub Copilot, que demuestran aumentos de productividad de más del 55% para desarrolladores que utilizan asistentes de IA.

En resumen, la ejecución de Irakani Builder se justifica plenamente porque cada una de sus características se traduce en un beneficio medible. Resuelve problemas técnicos para generar un valor de negocio tangible: desarrollar mejores productos, más rápido y a un menor costo, asegurando así el crecimiento sostenible y el liderazgo de la empresa en la era de la inteligencia artificial.

1.6. Limitaciones

1. Dependencia de Servicios de Terceros (AWS Bedrock):

- **Disponibilidad y Rendimiento:** La funcionalidad central de generación de código y asistencia inteligente de Irakani Builder depende directamente de la API de AWS Bedrock. Cualquier interrupción, cambio en los términos de

servicio, latencia o degradación del rendimiento del servicio de AWS impactará de forma directa en la operatividad y experiencia de usuario de la plataforma.

- **Costos Operativos:** El modelo de negocio y la rentabilidad de Irakani Builder estarán intrínsecamente ligados a la estructura de precios de AWS Bedrock. Cambios inesperados o incrementos en las tarifas de uso del modelo de IA podrían afectar la viabilidad económica del proyecto a largo plazo.
- **Limitaciones del Modelo de IA:** La calidad, precisión y creatividad del código y los componentes generados están limitadas por las capacidades del modelo de IA subyacente de AWS Bedrock. El proyecto no tiene control sobre el entrenamiento, los posibles sesgos o las “alucinaciones” (respuestas incorrectas pero verosímiles) del modelo, lo que requerirá una supervisión y validación constante por parte del usuario.

2. Complejidad de la Generación de Código para Casos de Uso Avanzados:

- **Personalización y Lógica de Negocio Compleja:** Aunque la plataforma está diseñada para acelerar el desarrollo, la generación automática de código para aplicaciones con lógica de negocio altamente especializada, algoritmos complejos o requisitos de rendimiento muy específicos puede ser limitada. En estos escenarios, la intervención manual de un desarrollador experimentado seguirá siendo indispensable.
- **Optimización del Código Generado:** La IA generará código funcional, pero no necesariamente el más optimizado en términos de rendimiento o buenas prácticas. Se requerirá un proceso de revisión y refactorización por parte de los desarrolladores para asegurar la calidad y escalabilidad de las aplicaciones finales, especialmente en proyectos de gran envergadura.

3. Alcance de la Plataforma como Herramienta Low-Code:

- **Flexibilidad vs. Simplicidad:** Como plataforma low-code, Irakani Builder busca un equilibrio entre la facilidad de uso y la flexibilidad. Inevitablemente, habrá un compromiso: las abstracciones que simplifican el desarrollo para usuarios novatos pueden limitar el control granular que los desarrolladores avanzados podrían desear.
- **Curva de Aprendizaje para Funcionalidades Avanzadas:** Si bien se proyecta reducir la curva de aprendizaje general, dominar las funcionalidades más avanzadas, como la optimización de prompts para la IA o la configuración de integraciones complejas, requerirá un proceso de aprendizaje y capacitación específico para los usuarios.

4. Dependencia Tecnológica y Mantenimiento:

- **Ciclo de Vida de las Tecnologías:** El proyecto se basa en tecnologías modernas como React 18 y Node.js. Sin embargo, el rápido ritmo de evolución de estos frameworks implica la necesidad de un mantenimiento y actualización continuos para evitar la obsolescencia tecnológica que precisamente se busca superar.
- **Gestión de Dependencias:** El proyecto dependerá de librerías y paquetes de terceros del ecosistema de JavaScript. Esto introduce riesgos potenciales de seguridad si no se gestionan y actualizan adecuadamente, además de posibles conflictos de compatibilidad.

5. Limitaciones de los Recursos y del Entorno del Proyecto:

- a) **Recursos Humanos y de Tiempo:** El desarrollo de la plataforma está sujeto a las limitaciones del equipo de desarrollo y los plazos establecidos. El alcance final de las funcionalidades implementadas en la primera versión estará condicionado por estos recursos.
- b) **Enfoque Inicial en Motores de BD Específicos:** Aunque se planea el soporte para múltiples motores de bases de datos, la fase inicial del desarrollo se centrará en MySQL y SQL Server. La integración con otras bases de datos (NoSQL, etc.) quedará como una mejora para futuras iteraciones del proyecto.

1.7. Delimitaciones

Para asegurar la viabilidad del proyecto y enfocar los esfuerzos de desarrollo en los objetivos prioritarios, se establecen las siguientes delimitaciones que definen el alcance de “Irakani Builder”:

Público Objetivo:

El proyecto Irakani Builder está concebido y dirigido exclusivamente al equipo de desarrollo interno de la empresa ITE-RADAPTA (Irakani), con un enfoque principal en el área de Proyectos. Aunque busca reducir la curva de aprendizaje, la herramienta está diseñada para ser utilizada por personal técnico (desarrolladores), y no se contempla en esta fase su uso por parte de usuarios finales sin conocimientos de programación o clientes externos.

Alcance Tecnológico (Stack):

El desarrollo del proyecto se acotará al siguiente stack tecnológico principal:

- **Frontend:** Se utilizará la librería React 18 en conjunto con TypeScript para garantizar un desarrollo robusto y escalable.
La edición de código se implementará a través de Monaco Editor.
- **Backend:** La lógica del servidor y la API se construirán sobre el entorno de ejecución Node.js.
- **Inteligencia Artificial:** La integración de capacidades de IA Generativa se realizará exclusivamente a través de los servicios de AWS Bedrock, sin contemplar el entrenamiento o desarrollo de modelos de lenguaje propios.
- **Bases de Datos:** El soporte inicial se centrará en los motores de bases de datos relacionales MySQL y SQL Server. La compatibilidad con otras bases de datos, como sistemas NoSQL, queda fuera del alcance de la versión inicial.

Tipo y Naturaleza del Proyecto:

Irakani Builder se define como una plataforma web de desarrollo asistido por IA (AI-assisted low-code platform) de uso interno. Su naturaleza es la de una herramienta de productividad y no la de un producto de software comercializable para el público general. El proyecto se enfoca en la creación del constructor de aplicaciones y no en las aplicaciones específicas que se generen con él. No pretende reemplazar por completo un Entorno de Desarrollo Integrado como VS Code, sino especializarse y acelerar los flujos de trabajo y los tipos de aplicaciones que son recurrentes en la empresa.

2. Marco contextual

2.1. Nombre de la empresa

ITERADAPTA (Irakani)

2.2. Giro

Desarrollo de Software y Servicios Tecnológicos

2.3. Dirección

Romanos #367, col. Altamira, C.P. 45160, Zapopan, Jalisco, México.

2.4. Misión

“Ser el Aliado Tecnológico de nuestros clientes, que les facilitará el alcance de sus objetivos de negocio, impactando directamente en su desarrollo operativo y comercial, ya sea a corto, mediano o largo plazo. Buscamos apoyar el crecimiento de nuestros clientes y darles sostenibilidad y replicabilidad a sus operaciones y procesos a través de software.”

3. Marco teórico

Este capítulo presenta los fundamentos conceptuales, contextuales y tecnológicos que sustentan el desarrollo del sistema “Irakani Builder”. Se estructura a partir del análisis del entorno de la industria del desarrollo de software y los desafíos operativos específicos de la empresa Irakani, que definen la problemática y justifican la necesidad del proyecto.

3.1. Marco Contextual y Problema de Negocio

3.1.1. La Evolución del Desarrollo de Software: Del Código Manual a la Asistencia Inteligente

La industria del desarrollo de software se encuentra en una fase de transformación acelerada, impulsada por la creciente demanda de digitalización. Esta evolución ha redefinido las herramientas y procesos, marcando dos hitos clave: la abstracción del código mediante plataformas Low-Code/No-Code y la generación de código asistida por Inteligencia Artificial.

La Abstracción del Código con Plataformas Low-Code/No-Code (LC/NC)

Las plataformas LC/NC surgieron como una respuesta directa a la necesidad de acelerar la entrega de software y ampliar el número de personas capaces de crear aplicaciones.

Las plataformas Low-Code/No-Code son entornos que abstraen la complejidad de la programación. En lugar de escribir código línea por línea, los usuarios emplean interfaces visuales con componentes pre-construidos que pueden usar para diseñar la interfaz y la lógica de una aplicación. La plataforma se encarga de interpretar este diseño visual y generar automáticamente el código subyacente necesario para que la aplicación funcione. Este enfoque no solo acelera el desarrollo, sino que lo democratiza, permitiendo que perfiles no estrictamente técnicos participen en la creación de soluciones digitales [?].

El impacto de esta democratización es profundo. Según un análisis de Gartner, Inc. (2022), se proyecta que para 2025, el 70% de las nuevas aplicaciones desarrolladas por empresas utilizarán tecnologías low-code o no-code, en un mercado que sigue creciendo a un ritmo superior al 20% anual.

La Generación de Código con Inteligencia Artificial (IA) Generativa

Más recientemente, la industria ha dado un paso más allá de la abstracción hacia la generación activa de código, gracias a la IA Generativa.

La IA Generativa en el desarrollo de software utiliza Modelos de Lenguaje Grandes (LLMs) que han sido entrenados con vastas cantidades de código fuente y texto técnico. A diferencia de las plataformas LC/NC que ocultan el código, estas herramientas funcionan como un “asistente de desarrollador” que genera código directamente a partir de instrucciones en lenguaje natural (prompts). Por ejemplo, un desarrollador puede solicitar “crea una función en Python para conectar a una API REST y obtener datos de usuarios”, y el modelo producirá un bloque de código funcional y contextualizado. Herramientas como GitHub Copilot se integran en el entorno de desarrollo para ofrecer sugerencias en tiempo real, desde completar una sola línea hasta escribir funciones enteras [?].

Este salto cuantitativo en la productividad está redefiniendo los estándares de eficiencia. Un estudio controlado referenciado por GitHub demostró que los desarrolladores que utilizaron GitHub Copilot completaron sus tareas un 55% más rápido [?].

La adopción de estas tecnologías ya no es una opción, sino un imperativo estratégico. Como señalan McKinsey & Company, el potencial económico de la IA generativa es inmenso, y las empresas que no integran estas capacidades en sus flujos de trabajo se arriesgan a una pérdida de competitividad sistémica [?, ?].

3.1.2. El Contexto Operativo Interno: La Plataforma app.irakani.com como Deuda Técnica

Dentro del panorama industrial descrito, la plataforma interna app.irakani.com fue la respuesta inicial de Irakani a la necesidad de agilizar el desarrollo. Sin embargo, con el tiempo, ha dejado de ser una solución para convertirse en un claro ejemplo de deuda técnica.

El término “deuda técnica”, acuñado por Ward Cunningham, describe una metáfora que enmarca las consecuencias a largo plazo de decisiones de diseño o desarrollo tomadas para acelerar la entrega a corto plazo. Al igual que una deuda financiera, esta deuda técnica acumula “intereses” en forma de un mayor esfuerzo requerido para el mantenimiento, la corrección de errores y la adición de nuevas funcionalidades [?]. Si no se “paga” (refactorizando el código, actualizando la arquitectura), el costo de la inefficiencia puede llegar a superar el valor que aporta la aplicación.

La plataforma actual de Irakani encarna esta problemática. Las decisiones arquitectónicas del pasado, si bien funcionales en su momento, ahora imponen una carga operativa tan alta que el esfuerzo requerido para mantenerla y adaptarla supera con creces los beneficios que ofrece.

El Problema de Negocio: La Obsolescencia como Freno a la Rentabilidad y la Innovación

El problema de negocio que aborda el proyecto Irakani Builder se deriva directamente de esta deuda técnica acumulada. Se manifiesta en dos dimensiones críticas:

- 1. Ineficiencia Operativa y Altos Costos:** La deuda técnica de app.irakani.com se traduce directamente en ineficiencia y costos elevados. Esto se debe a una arquitectura obsoleta y monolítica, donde los componentes están fuertemente

acoplados, dificultando las modificaciones y actualizaciones. La ineficiencia del ciclo de desarrollo, evidenciada por las 12 horas necesarias para generar una aplicación base, es un síntoma directo de este problema. Esta lentitud impacta la estructura de costos, tal como se documenta en el caso de estudio interno “Ojo Zarco”, donde los costos de mano de obra se elevaron aproximadamente un 33% en comparación con una solución optimizada, simplemente por el tiempo extra que los desarrolladores invirtieron “luchando” contra la plataforma.

Esta situación es consistente con análisis de la industria, como el de McKinsey & Company (2020), que encontró que entre el 10% y el 20% del presupuesto tecnológico destinado a nuevos productos es, en realidad, desviado para resolver problemas causados por la deuda técnica existente.

2. **Brecha Estratégica por Ausencia de Inteligencia Artificial:** Esta es la deficiencia más grave y una consecuencia directa de la deuda técnica. La arquitectura rígida y las tecnologías desactualizadas de app.irakani.com hacen que la integración de servicios modernos de IA Generativa sea técnicamente inviable o prohibitivamente costosa. Esta incapacidad de evolucionar posiciona a Irakani en una clara desventaja competitiva.

La magnitud de la oportunidad perdida queda clara en análisis macroeconómicos. Un informe de Goldman Sachs (2023) estima que la IA generativa podría aumentar el PIB mundial en un 7% (casi \$7 billones de dólares) a lo largo de una década [?]. La incapacidad de Irakani para aprovechar esta ola de innovación no es solo una oportunidad perdida, sino una amenaza existencial a largo plazo, ya que la IA se está convirtiendo en un motor clave de la productividad económica [?].

3.2. Marco Teórico y Tecnológico

Esta sección profundiza en los conceptos, paradigmas y herramientas tecnológicas que constituyen la base fundamental para el diseño y desarrollo de “Irakani Builder”. El objetivo es establecer el sustento teórico que justifica cada una de las decisiones de arquitectura y la selección del stack tecnológico del proyecto.

3.2.1. Paradigmas de Desarrollo de Software

La concepción de Irakani Builder responde a la fusión de dos paradigmas de desarrollo que, si bien fueron introducidos en el marco contextual, operan bajo principios técnicos fundamentalmente distintos: el enfoque declarativo de las plataformas Low-Code/No-Code y el enfoque generativo del desarrollo asistido por IA.

Paradigma Declarativo: Plataformas Low-Code/No-Code (LC/NC)

El principio fundamental de las plataformas LC/NC es abstraer la complejidad del desarrollo a través de un modelo declarativo y dirigido por metadatos (metadata-driven).

En lugar de escribir código imperativo (que le dice a la computadora cómo hacer algo paso a paso), el usuario trabaja de forma declarativa (define qué quiere lograr). Al usar un editor visual para arrastrar un formulario o conectar una base de datos, el usuario no está escribiendo código, sino que está creando metadatos: un modelo de datos que describe la estructura, la apariencia y el comportamiento de la aplicación. La plataforma posee un motor de renderizado y ejecución que interpreta estos metadatos en tiempo real para generar y presentar la aplicación funcional. Este desacoplamiento entre el modelo (los metadatos) y la vista (la aplicación generada) es lo que permite una agilidad radical [?].

Este mercado está en auge precisamente porque responde a la necesidad crítica de las empresas de acelerar la entrega de soluciones digitales, como lo confirma el análisis de Gartner [?]. Irakani Builder adopta este principio para ofrecer un entorno visual y estructurado.

Paradigma Generativo: Desarrollo Asistido por Inteligencia Artificial (AI-Assisted Development)

Este es el paradigma evolutivo que Irakani Builder busca liderar. A diferencia del enfoque determinista de LC/NC, el desarrollo asistido por IA introduce un modelo generativo y probabilístico.

Este paradigma no se basa en un modelo de metadatos predefinido, sino en la capacidad de los LLMs para generar código nuevo a partir de patrones aprendidos. Cuando un desarrollador le pide a una herramienta como GitHub Copilot que cree una función, el modelo no busca en una biblioteca de componentes; predice la secuencia de código más probable que satisface la petición, basándose en los miles de millones de ejemplos con los que fue entrenado. Actúa como un “programador en par” (pair programmer), que no solo acelera la escritura de código repetitivo, sino que también ayuda a los desarrolladores a mantener el flujo de trabajo y a gastar menos energía mental en tareas triviales [?].

Síntesis en Irakani Builder:

Irakani Builder no se limita a adoptar uno de estos paradigmas, sino que los sintetiza estratégicamente. Utiliza la IA Generativa para la fase inicial de creación, traduciendo una descripción en lenguaje natural a un modelo de metadatos estructurado. Posteriormente, presenta este modelo al usuario a través de un entorno visual declarativo (LC/NC), donde puede ser validado, modificado y gestionado de forma segura y eficiente. De esta manera, combina la flexibilidad del lenguaje natural con la robustez de un sistema basado en modelos.

3.2.2. Arquitecturas de Software Modernas

Para evitar la obsolescencia y los cuellos de botella del sistema anterior, Irakani Builder se fundamentará en principios de arquitectura moderna que garantizan escalabilidad, mantenibilidad y resiliencia. Esto implica un alejamiento deliberado de la arquitectura monolítica tradicional en favor de un enfoque distribuido basado en microservicios y comunicado a través de APIs RESTful.

El Problema: La Arquitectura Monolítica

La plataforma app.irakani.com sigue un patrón monolítico, donde todos los componentes de la aplicación (autenticación, gestión de usuarios, generación de aplicaciones, etc.) están empaquetados en un único proceso fuertemente acoplado. Este enfoque, si bien es simple de iniciar, conduce inevitablemente a los problemas de deuda técnica ya mencionados: cualquier pequeño cambio requiere volver a desplegar toda la aplicación, un error en un módulo puede colapsar el sistema completo, y la base de código se vuelve demasiado compleja para mantener o escalar eficientemente.

La Solución: Arquitectura de Microservicios

En contraposición, un enfoque de microservicios estructura una aplicación como una colección de servicios pequeños, autónomos y débilmente acoplados.

Cada servicio es como una mini-aplicación independiente con una única responsabilidad de negocio (por ejemplo, un servicio para la autenticación, otro para la generación de código con IA, y un tercero para la gestión de bases de datos). Cada uno de estos servicios:

- **Se ejecuta en su propio proceso:** Esto garantiza el aislamiento. Un fallo en el servicio de generación de código no afectará al servicio de autenticación.
- **Se comunica a través de APIs bien definidas:** Los servicios no comparten memoria ni código directamente; se comunican a través de protocolos ligeros, típicamente sobre HTTP con APIs RESTful.
- **Puede ser desarrollado, desplegado y escalado de forma independiente:** Si el servicio de generación con IA requiere más potencia de cómputo, solo ese servicio se escala, sin afectar al resto de la aplicación. Esto permite un uso mucho más eficiente de los recursos [?].

Esta modularidad, como explica Red Hat, mejora drásticamente la tolerancia a fallos y permite a los equipos utilizar el stack tecnológico más adecuado para cada tarea específica, optimizando la herramienta para el trabajo a realizar [?].

El Estándar de Comunicación: APIs RESTful

Para que los microservicios y el frontend se comuniquen de manera eficiente y estandarizada, se requiere un protocolo de comunicación robusto. Las APIs RESTful (Representational State Transfer) son el estándar de facto para la web.

REST no es una tecnología, sino un conjunto de principios o restricciones arquitectónicas que, si se siguen, producen un sistema escalable, desacoplado y fácil de mantener. Los principios clave son:

- **Interfaz Uniforme:** Los recursos (como un usuario o una aplicación) se identifican con URLs únicas (ej. /api/applications/app-123). Se manipulan a través de una representación estándar (JSON) usando los verbos HTTP (GET para leer, POST para crear, PUT para actualizar, DELETE para borrar).
- **Sin Estado (Stateless):** Cada solicitud del cliente al servidor debe contener toda la información necesaria para ser procesada. El servidor no almacena ningún estado de la sesión del cliente. Esta restricción es fundamental para la escalabilidad, ya que cualquier servidor puede atender cualquier solicitud, facilitando el balanceo de cargas.
- **Separación Cliente-Servidor:** El cliente (frontend de Irakani Builder) y el servidor (los microservicios) están completamente desacoplados. Solo se conocen a través de la API, lo que permite que ambos evolucionen de forma independiente.

Como define Postman, este conjunto de principios garantiza la interoperabilidad entre los distintos componentes de Irakani Builder [?].

3.2.3. Tecnologías Clave del Stack de Desarrollo

La elección de cada tecnología en el stack de Irakani Builder responde a criterios de rendimiento, escalabilidad, soporte de la comunidad y, fundamentalmente, su capacidad para implementar las características deseadas de una plataforma de desarrollo moderna.

Frontend - React 18 y el Paradigma Declarativo

React es la biblioteca de JavaScript elegida para construir la interfaz de usuario de Irakani Builder. Su principal ventaja radica en su paradigma declarativo, que simplifica la creación de UIs complejas e interactivas.

El enfoque declarativo de React se materializa a través de su DOM Virtual. En lugar de manipular directamente el DOM del navegador (un proceso lento y costoso), React mantiene una representación del DOM en memoria. Cuando el estado de un componente cambia (por ejemplo, el usuario edita el nombre de una aplicación), React crea un nuevo DOM Virtual, lo compara con la versión anterior mediante un eficiente algoritmo de “diferenciación” (diffing), y calcula el conjunto mínimo de cambios necesarios. Finalmente, aplica únicamente esas modificaciones al DOM real en un proceso optimizado.

Para un editor visual de aplicaciones, donde la interfaz es altamente dinámica y sufre constantes actualizaciones (arrastrar componentes, modificar propiedades, etc.), este mecanismo es crucial. Asegura que la interfaz se mantenga fluida y responsive sin importar la complejidad, proporcionando una experiencia de usuario de alta calidad. Su arquitectura basada en componentes reutilizables es ideal para construir la paleta de elementos visuales que los usuarios utilizarán para ensamblar sus aplicaciones [?].

Frontend - TypeScript para Escalabilidad y Robustez

TypeScript es un superconjunto de JavaScript que añade tipado estático opcional, una característica indispensable para un proyecto de esta envergadura.

TypeScript introduce un proceso de compilación que analiza el código antes de que se ejecute. Durante esta fase, verifica que los tipos de datos sean consistentes (ej. que no se intente usar un string como si fuera un number). Si detecta una incoherencia, arroja un error de compilación, impidiendo que el bug llegue a producción. Esto crea un “contrato” de datos que define la estructura de objetos complejos.

La aplicación gestionará una estructura de datos JSON compleja que representa las aplicaciones creadas por los usuarios. El uso de TypeScript para definir interfaces (interface) que modelen esta estructura es vital. Garantiza que tanto el código que interactúa con la IA como el que renderiza el editor visual traten los datos de manera consistente, lo que reduce drásticamente los errores en tiempo de ejecución, facilita la refactorización segura y mejora la colaboración entre los miembros del equipo [?].

Backend - Node.js y el Entorno de Ejecución Asíncrono

Node.js es el entorno de ejecución de JavaScript del lado del servidor elegido para construir los microservicios del backend.

La principal ventaja de Node.js es su modelo de E/S (Entrada/Salida) asíncrono y sin bloqueo, gestionado por un mecanismo conocido como el Event Loop. Cuando el servidor recibe una petición que requiere una operación de E/S lenta (como una consulta a la base de datos o una llamada a la API de AWS Bedrock), en lugar de esperar a que termine (bloqueando el hilo de ejecución), delega la tarea al sistema operativo y pasa inmediatamente a atender la siguiente petición. Una vez que la operación de E/S finaliza, el Event Loop ejecuta la función de callback asociada con el resultado.

Este modelo es extremadamente eficiente para manejar una gran cantidad de conexiones concurrentes con un bajo consumo de memoria. En el contexto de Irakani Builder, que debe gestionar simultáneamente las interacciones de los usuarios, las llamadas a la IA y las operaciones de base de datos, Node.js garantiza una alta escalabilidad y un rendimiento óptimo para la API [?].

Editor de Código - Monaco Editor

Para ofrecer una experiencia de desarrollo de primer nivel dentro de la plataforma, se integra Monaco Editor.

Monaco Editor es el componente de edición de código que impulsa a Visual Studio Code, extraído como una biblioteca independiente para ser utilizado en aplicaciones web. Proporciona, directamente en el navegador, las características avanzadas que un desarrollador espera de un IDE de escritorio: resaltado de sintaxis para múltiples lenguajes, autocompletado inteligente (IntelliSense), validación de errores en tiempo real y un rico API para personalización.

La integración de Monaco es fundamental para cumplir la promesa de ser una herramienta para desarrolladores. Cuando la IA genera código, el desarrollador necesita un entorno potente y familiar para revisarlo, refinarlo y personalizarlo. Monaco Editor cierra la brecha entre la generación de código asistida y la edición manual avanzada, garantizando una experiencia de desarrollo fluida y profesional [?].

3.2.4. Integración de Inteligencia Artificial Generativa

El componente más innovador de Irakani Builder es la integración nativa de IA para la generación de código. Esta capacidad no se basa en una única tecnología, sino en la orquestación de tres pilares fundamentales: los Modelos de Lenguaje Grandes (LLMs) como motor, AWS Bedrock como plataforma de acceso, y la Ingeniería de Prompts como disciplina de control.

1. Modelos de Lenguaje Grandes (LLMs) como Motor de Generación

Los LLMs son el “cerebro” de la funcionalidad generativa de Irakani Builder. Son modelos de aprendizaje profundo basados en la arquitectura Transformer, entrenados con enormes cantidades de datos de texto y código.

La arquitectura Transformer permite a los modelos procesar texto de manera no secuencial y ponderar la importancia de diferentes palabras en una instrucción a través de un mecanismo llamado “atención”. Esto les permite comprender el contexto, la gramática y las estructuras lógicas complejas. Cuando reciben un prompt, no buscan una respuesta predefinida; en su lugar, predicen probabilísticamente la secuencia de “tokens” (palabras o fragmentos de código) más probable que debería seguir a la instrucción, basándose en los patrones aprendidos durante su entrenamiento [?].

Relevancia para Irakani Builder: En el contexto del proyecto, el LLM no solo genera código, sino que primero debe traducir una instrucción en lenguaje natural (ej. “crea un formulario de registro con campos para nombre, email y contraseña”) a una representación de datos estructurada en formato JSON que la plataforma pueda interpretar. Su capacidad para entender la intención semántica del usuario es lo que permite esta traducción de lo no estructurado a lo estructurado [?].

2. AWS Bedrock como Plataforma de Acceso a Modelos Fundacionales (PaaS)

Entrenar y alojar un LLM de vanguardia es computacionalmente prohibitivo y requiere una infraestructura especializada. AWS Bedrock resuelve este problema al funcionar como una Plataforma como Servicio (PaaS) para IA Generativa.

Bedrock abstrae toda la complejidad de la infraestructura. Proporciona una única API unificada para acceder a una variedad de modelos fundacionales de alto rendimiento (de empresas como AI21 Labs, Anthropic, Cohere y Amazon). Esto significa que Irakani Builder puede enviar una solicitud a un único endpoint de AWS, y Bedrock se encarga de dirigirla al modelo seleccionado, gestionar el escalado de los servidores GPU, garantizar la seguridad y mantener los modelos actualizados.

La elección de Bedrock es una decisión estratégica clave. Permite al proyecto integrar capacidades de IA de clase mundial sin la enorme inversión y el riesgo de gestionar la infraestructura subyacente. Además, proporciona flexibilidad: si en el futuro surge un modelo más eficiente para la generación de código, Irakani Builder puede cambiar de proveedor de modelo con modificaciones mínimas en su código, gracias a la API unificada [?].

3. Ingeniería de Prompts para la Generación de Código Preciso

La calidad del resultado de un LLM depende directamente de la calidad de la instrucción (prompt). La Ingeniería de Prompts es la disciplina de diseñar estas instrucciones para guiar al modelo hacia la respuesta deseada.

Dado que los LLMs son probabilísticos, un prompt vago puede generar resultados impredecibles. Para asegurar que la salida sea siempre un JSON válido y con la estructura que Irakani Builder espera, se utilizan técnicas avanzadas:

- **Few-shot Learning:** Se proporcionan al modelo ejemplos concretos dentro del prompt. Se le muestra un par de ejemplos de una petición de usuario y el JSON exacto que se generó. Esto le da al modelo un patrón claro que debe seguir.
- **Chain-of-Thought:** Se le pide al modelo que “piense” paso a paso. El prompt le instruye a descomponer la petición del usuario en partes lógicas (identificar formularios, luego campos, luego propiedades) antes de generar el JSON final. Esto mejora drásticamente la precisión en tareas complejas.

Esta disciplina es fundamental para la fiabilidad de la plataforma. La solicitud del usuario se envuelve en un “meta-prompt” cuidadosamente diseñado que contiene estas técnicas, asegurando que el código generado se ajuste a los requisitos específicos, las convenciones de estilo y la arquitectura de la plataforma [?].

3.3. Marco Metodológico

Para el desarrollo del proyecto “Irakani Builder”, se adoptó un marco de trabajo Ágil, en contraposición a las metodologías predictivas tradicionales como el Modelo en Cascada. La selección de la metodología se basa en una evaluación rigurosa del contexto del proyecto, el cual se caracteriza por la alta incertidumbre tecnológica inherente a la integración de sistemas de Inteligencia Artificial Generativa y la necesidad de gestionar una evolución constante de requisitos funcionales.

El agilismo, con su énfasis en la flexibilidad, la entrega de valor incremental y la adaptación continua, se presentó como la elección natural para gestionar la complejidad y maximizar la eficiencia del equipo de desarrollo. Dentro de este paradigma, se seleccionó Scrum como el marco de trabajo específico. Scrum proporciona una estructura definida a través de roles, eventos y

artefactos, permitiendo una gestión ordenada del proyecto sin sacrificar la capacidad de respuesta ante el cambio. Este enfoque iterativo e incremental es fundamental para abordar el desarrollo de una plataforma asistida por IA [?].

3.3.1. Fundamento Metodológico: Del Control Predictivo al Empírico

El éxito en el desarrollo de plataformas Low-Code asistidas por IA como Irakani Builder depende fundamentalmente de la aplicación del Control Empírico. El empirismo, base filosófica de Scrum, postula que el conocimiento se genera a partir de la experiencia y que las decisiones se toman a partir de la observación de los resultados obtenidos.

La Inviabilidad del Enfoque Predictivo (Modelo en Cascada)

Las metodologías predictivas, exemplificadas por el modelo en cascada, se basan en la premisa de que los requisitos del proyecto deben ser fijos y completamente documentados desde el inicio. Este modelo asume que el camino hacia el resultado final es conocido y estable [?].

El proyecto Irakani Builder no cumple con las condiciones para un enfoque predictivo, debido a dos factores críticos:

1. **Volatilidad de Requisitos y Complejidad Tecnológica:** El proyecto busca resolver la deuda técnica acumulada e integrar capacidades de IA Generativa. El desarrollo implica la integración de múltiples tecnologías modernas (React 18, Node.js) y la orquestación de microservicios, lo que genera una complejidad elevada. Los proyectos que involucran “Tecnologías nuevas, inestables o con muchos elementos diferentes a integrar,” así como aquellos con “Requisitos poco definidos, ambiguos, incompletos, poco maduros o cambiantes,” son inviables bajo un modelo predictivo, requiriendo en su lugar un control empírico.
2. **Riesgo Estratégico de Terceros (AWS Bedrock):** La funcionalidad central de generación de código y asistencia inteligente del Builder depende de la API de AWS Bedrock. La calidad, latencia, y las posibles “alucinaciones” (respuestas incorrectas pero verosímiles) de los modelos de IA son factores no determinísticos. Un plan fijo intentaría predecir el comportamiento de un sistema probabilístico, lo cual es inviable. El control predictivo llevaría inevitablemente a un sobrecosto y un fracaso operativo al intentar corregir desviaciones en fases tardías [?].

3.3.2. Adopción del Framework Scrum

Dentro del paradigma Ágil, se seleccionó Scrum como el marco de trabajo específico. Scrum proporciona una estructura prescriptiva pero flexible a través de roles, eventos y artefactos definidos, lo que permite una gestión de proyectos ordenada sin sacrificar la capacidad de respuesta al cambio. Según Scrum.org, Scrum es un marco ligero que ayuda a las personas, equipos y organizaciones a generar valor a través de soluciones adaptativas para problemas complejos. Su enfoque iterativo e incremental fue fundamental para abordar el desarrollo de Irakani Builder [?].

Los Pilares de Scrum: Transparencia, Inspección y Adaptación

Scrum se basa en el empirismo, sostenido por tres pilares interdependientes que permiten al equipo gestionar de manera proactiva la incertidumbre de la IA y el riesgo de la deuda técnica [?].

1. **Transparencia (Transparency):** La transparencia en Scrum garantiza que el estado real del proceso, el código base, y el progreso del producto sean visibles para todos los miembros del equipo y los stakeholders. En Irakani Builder, esto se logra mediante el uso de artefactos claros como el Product Backlog y el Sprint Backlog, la gestión del control de versiones mediante Bitbucket, y el seguimiento de tareas en Jira. En el contexto específico del desarrollo asistido por IA, la transparencia implica documentar el objetivo del modelo, su nivel de riesgo y la información del proveedor (AWS Bedrock), asegurando que el LLM no sea una “caja negra” inauditable [?].
2. **Inspección (Inspection):** El equipo inspecciona frecuentemente el progreso hacia el Objetivo del Producto y los artefactos de Scrum para detectar desviaciones inaceptables. La inspección se realiza en dos niveles clave: el Scrum Diario y la Revisión del Sprint (para inspeccionar el Incremento funcional). En Irakani Builder, esto implica la inspección temprana de si las salidas de la IA (como la estructura JSON para la “Encuesta de Satisfacción de Visita”) son funcionalmente correctas y si el rendimiento de los endpoints RESTful del backend es óptimo.
3. **Adaptación (Adaptation):** Si la inspección revela que el proceso o el Incremento se desvían de manera inaceptable, el equipo realiza ajustes inmediatos. La adaptación se manifiesta en la repriorización inmediata del Product Backlog o, por parte del Development Team, mediante el ajuste de las técnicas de Ingeniería de Prompts en la Retrospectiva del Sprint.

Por ejemplo, si la inspección revela que la IA produce inconsistencias, el equipo adapta los meta-prompts para mejorar la precisión [?, ?].

Estructura Prescriptiva de Scrum: Roles, Eventos y Artefactos

Para operar efectivamente bajo el Control Empírico, Scrum define una estructura mínima de tres Roles, cinco Eventos y tres Artefactos, esenciales para la transparencia y la cadencia del proyecto.

Roles del Equipo Scrum en “Irakani Builder”

El Equipo Scrum es auto-organizado y multifuncional, siendo idealmente un equipo pequeño para garantizar la eficiencia y minimizar la complejidad de coordinación.

Mapeo de Roles en el Proyecto Irakani Builder

Rol organizacional	Rol formal de Scrum	Responsabilidad principal en el proyecto
Luis Flores (Gerente General/CTO)	Product Owner (PO)	Maximización del valor del producto. Responsable de gestionar y priorizar el Product Backlog en función de la visión de negocio, que incluye la reducción de costos operativos y la ventaja competitiva mediante la IA.
Helmer Omar Tapia Reyes (Líder de Plataforma)	Scrum Master (SM)	Liderazgo de servicio. Responsable de garantizar que el equipo comprenda y aplique Scrum. Elimina impedimentos y protege al Development Team de interrupciones, asegurando la efectividad del proceso.
Ángel David Mariscal Soto (Desarrollador) y Equipo de Plataforma	Development Team (Developers)	Responsable de crear un Incremento de software utilizable que cumpla con la Definición de Terminado en cada Sprint. Realiza todo el trabajo técnico (frontend React/Monaco Editor y backend Node.js/AWS Bedrock).

[?, ?, ?, ?]

Eventos (Ceremonias) de Scrum

El proyecto se opera bajo una cadencia de Sprints de dos semanas, lo que facilita la inspección y adaptación constante, clave para manejar la integración compleja de la IA.

Evento de Scrum	Propósito Central	Duración (Time-box)	Función en el Desarrollo de Irakani Builder
El Sprint	Generar un Incremento de valor funcional que cumpla con la Definición de Terminado.	2 semanas	Garantiza la entrega incremental y funcional de módulos clave, como el Editor de Código Personalizado o la Gestión de Entidades.
Planificación del Sprint	Definir el Objetivo del Sprint y seleccionar elementos del Product Backlog para crear el Sprint Backlog.	Máx. 8 horas (para 4 semanas)	El Development Team (Ángel David) descompone las historias de usuario (ej. “Integrar AWS Bedrock”) en tareas técnicas detalladas y medibles.
Scrum Diario	Inspección del progreso y sincronización del plan para las próximas 24 horas.	Máx. 15 minutos	Identificación diaria de impedimentos, como pueden ser aquellos derivados de la integración de servicios de terceros (latencia de AWS) o conflictos en el código base.

Evento de Scrum	Propósito Central	Duración (Time-box)	Función en el Desarrollo de Irakani Builder
Revisión del Sprint	Presentación del Incremento funcional a los stakeholders para obtener retroalimentación y adaptar el Product Backlog.	Máx. 4 horas (para 4 semanas)	Inspección de Valor: Se valida la usabilidad de la interfaz (React 18) y la precisión de la funcionalidad de la IA.
Retrospectiva del Sprint	Inspección del proceso, las herramientas, la colaboración y el equipo para planificar mejoras de calidad y eficiencia.	Máx. 3 horas (para 4 semanas)	Adaptación del Proceso: Refinamiento de la estrategia de Prompt Engineering y ajuste de las definiciones de calidad.

Artefactos de Scrum para la Transparencia

Los Artefactos aseguran que la información esencial del proyecto sea visible y entendida por todos:

- **Product Backlog (PB):** La lista única y priorizada de todo el trabajo necesario para Irakani Builder. Mantenido por el Product Owner, prioriza elementos que tienen el mayor impacto en la reducción de costos y la mitigación de la deuda técnica.
- **Sprint Backlog (SB):** El subconjunto de elementos del PB seleccionados para el Sprint, junto con el plan para entregar el Incremento. El Development Team (Ángel David) gestiona este plan, adaptándolo dinámicamente.
- **Incremento:** El software funcional producido al final del Sprint, cumpliendo con la Definition of Done. Es crucial que este Incremento esté en un estado potencialmente liberable.

3.3.3. Fases del Proceso de Desarrollo Metodológico

El desarrollo de Irakani Builder se articula a través de fases iterativas, respaldadas por un conjunto de herramientas de ingeniería que garantizan la calidad y la continuidad, un proceso conocido como AI-Assisted Agile [?].

El desarrollo se estructura siguiendo el ciclo de vida de Scrum:

Fase	Descripción y Énfasis
Fase I: Concepción y Diseño (Sprint 0)	Establecimiento de la visión, la arquitectura tecnológica preliminar y desarrollo de prototipos de Interfaz de Usuario (UI) y Experiencia de Usuario (UX). Se configura el Product Backlog inicial, priorizando funcionalidades según el mayor impacto en la reducción de costos.
Fase II: Desarrollo Iterativo e Incremental (Sprints 1-N)	Fase central donde se aplica rigurosamente el ciclo de Scrum (Planificación, Ejecución, Revisión, Retrospectiva) en una cadencia de Sprints de dos semanas. Se entregan Incrementos funcionales (ej., módulo de autenticación, esqueleto del editor visual, integración de AWS Bedrock) que construyen valor sobre la base existente.
Fase III: Pruebas y Despliegue Continuo	Enfoque en la integración y pruebas continuas. La práctica de CI/CD (Integración Continua/Despliegue Continuo) automatiza la compilación, pruebas y despliegue del código, manteniendo la calidad y asegurando que el software esté en un estado potencialmente desplegable al final de cada Sprint.

Casos de Uso Metodológicos Clave: La Ingeniería de Prompts

La metodología Scrum gestiona la interacción con la IA, utilizando sus eventos para inspeccionar el valor y adaptar la precisión de los modelos. Por ejemplo, el caso de uso de “Generar aplicación con IA” ejemplifica el flujo empírico:

- Entrada y Generación:** El desarrollador (Ángel David) ingresa el prompt (“crea una Encuesta de Satisfacción”). La IA (AWS Bedrock) genera una representación de datos estructurada en JSON.
- Inspección y Fallo:** Durante la Ejecución del Sprint, el equipo inspecciona la salida JSON. Si hay errores de estructura (fallos del LLM), se documenta la desviación.
- Adaptación:** El fallo se lleva a la Retrospectiva del Sprint. El equipo adapta los meta-prompts para mejorar la precisión y evitar futuras inconsistencias en la generación de JSON. Este ciclo transforma la incertidumbre de la IA en conocimiento estructurado [?].

Integración de la Ingeniería: La Definition of Done (DoD)

Para mitigar el riesgo de deuda técnica, especialmente porque la IA generará código que podría no ser “el más optimizado”, el Development Team implementa una Definition of Done (DoD) estricta:

- **Revisión y Versión:** El código ha sido revisado por pares mediante Pull Requests en Bitbucket.
- **Pruebas Automatizadas:** Todas las pruebas unitarias y de integración son exitosas.
- **Conformidad del Modelo:** El JSON de metadatos generado por la IA es válido y cumple con el esquema de datos interno de la plataforma.
- **Optimización:** El código generado por la IA debe someterse a una refactorización mínima garantizada para asegurar las mejores prácticas de rendimiento y legibilidad.
- **Despliegue Continuo:** La funcionalidad ha sido integrada y desplegada con éxito en un entorno de pruebas (CI/CD) [?].

3.3.4. Herramientas de Soporte a la Metodología

La implementación exitosa de Scrum se apoya en un conjunto de herramientas tecnológicas diseñadas para facilitar la colaboración, la transparencia y la automatización.

- **Jira (Atlassian): Gestión de Proyectos y Seguimiento de Tareas.** Plataforma diseñada específicamente para la gestión ágil. Permite al equipo (incluido el Product Owner) gestionar el Product Backlog, planificar Sprints y visualizar el progreso en tableros Scrum. Es ideal para desglosar proyectos complejos en tareas gestionables [?].
- **Git y Bitbucket (Atlassian): Control de Versiones y Colaboración de Código.** Git gestiona el código fuente, y Bitbucket aloja el repositorio centralizado, ofreciendo una integración nativa con Jira. Facilita la colaboración profesional mediante Pull Requests y la automatización de flujos de trabajo de CI/CD, esencial para el trabajo en equipo en un proyecto ágil [?].
- **Integración Continua / Despliegue Continuo (CI/CD):** Práctica que automatiza la compilación, las pruebas y el despliegue del software. El CI/CD es vital para mantener la calidad y la velocidad de entrega en un sistema complejo, validando automáticamente cualquier cambio en el stack React/Node.js o la integración de AWS Bedrock [?].

4. Desarrollo de la Solución

4.1. Metodología Scrum Aplicada

El desarrollo de Irakani Builder se ejecutó siguiendo la metodología ágil Scrum, con sprints de 2 semanas de duración cada uno. Esta metodología permitió iteraciones rápidas, retroalimentación continua y adaptación a los cambios en los requerimientos del negocio.

Nota sobre el Cronograma: El proyecto se planificó para ejecutarse desde julio hasta diciembre de 2025, con un período total de 6 meses distribuidos en 11 sprints. La planificación detallada se realizó durante la fase de concepción del proyecto, estableciendo fechas objetivo para cada entregable y permitiendo ajustes según la velocidad real del equipo durante la ejecución.

4.1.1. Roles del Equipo

El equipo Scrum estuvo conformado por los siguientes roles:

- **Product Owner:** Luis Flores - Responsable de definir y priorizar el Product Backlog según el valor de negocio
- **Scrum Master / Desarrollador:** Angel Mariscal - Facilitador del proceso Scrum y desarrollador principal
- **Stakeholders:** Equipo de Proyectos de Irakani - Usuarios finales que proporcionaron retroalimentación continua

4.1.2. Duración y Estructura de Sprints

El proyecto se planificó en 11 sprints con una duración de 2 semanas cada uno (excepto el Sprint 11 de cierre con 5 días), abarcando un período total de 6 meses desde julio hasta diciembre de 2025. La Tabla 4 muestra la planificación completa de sprints:

Cuadro 4: Planificación de Sprints del Proyecto Irakani Builder

Sprint	Épica Principal	Inicio	Fin
Sprint 0	Concepción y Diseño	14/07/2025	27/07/2025
Sprint 1	Autenticación y Seguridad	28/07/2025	10/08/2025
Sprint 2	Núcleo de IA	11/08/2025	24/08/2025
Sprint 3	Interfaz Base	25/08/2025	07/09/2025
Sprint 4	Editor Visual	08/09/2025	21/09/2025
Sprint 5	Renderizado y Bases de Datos	22/09/2025	05/10/2025
Sprint 6	Gestión de Datos Avanzada	06/10/2025	19/10/2025
Sprint 7	Asistente de Chat y Sesiones	20/10/2025	02/11/2025
Sprint 8	Gestión de Componentes	03/11/2025	16/11/2025
Sprint 9	Personalización	17/11/2025	30/11/2025
Sprint 10	Despliegue y Validación Final	01/12/2025	14/12/2025
Sprint 11	Buffer y Cierre	15/12/2025	19/12/2025

4.1.3. Ceremonias Implementadas

Durante cada sprint se realizaron las siguientes ceremonias Scrum:

- **Sprint Planning:** Reunión al inicio de cada sprint para seleccionar historias de usuario del Product Backlog y definir el Sprint Goal
- **Daily Standup:** Reuniones diarias de 15 minutos para sincronización del equipo (adaptadas al contexto de equipo reducido)
- **Sprint Review:** Demostración de funcionalidades completadas al Product Owner y stakeholders al final de cada sprint
- **Sprint Retrospective:** Reflexión sobre el proceso y definición de mejoras para el siguiente sprint
- **Backlog Refinement:** Sesiones continuas para refinar y estimar historias de usuario futuras

4.2. Conformación del Product Backlog

4.2.1. Épicas del Proyecto

El Product Backlog se organizó en 11 épicas principales que agrupan las historias de usuario por funcionalidad y alineadas con cada sprint. Cada épica representa un conjunto cohesivo de funcionalidades que aportan valor incremental al producto:

1. Épica 1 - Sprint 0: Concepción y Diseño

- Configuración del entorno de desarrollo

- Definición de arquitectura de microservicios
- Establecimiento de estándares de código y documentación

2. Épica 2 - Sprint 1: Autenticación y Seguridad

- Sistema de autenticación con JWT
- Integración con AWS Cognito
- Gestión de sesiones seguras

3. Épica 3 - Sprint 2: Núcleo de IA

- Integración con AWS Bedrock
- Ingeniería de prompts para generación de componentes
- Optimización de llamadas a modelos de IA

4. Épica 4 - Sprint 3: Interfaz Base

- Desarrollo de interfaz con React 18
- Sistema de navegación y routing
- Componentes base reutilizables

5. Épica 5 - Sprint 4: Editor Visual

- Editor visual de formularios
- Integración de Monaco Editor
- Panel de propiedades contextual

6. Épica 6 - Sprint 5: Renderizado y Bases de Datos

- Transformación JSON a componentes visuales
- Conexión multi-motor de bases de datos
- Pool de conexiones optimizado

7. Épica 7 - Sprint 6: Gestión de Datos Avanzada

- Administrador de base de datos con IA
- Generación de queries desde lenguaje natural
- Explorador de esquemas de base de datos

8. Épica 8 - Sprint 7: Asistente de Chat y Sesiones

- Chat de asistencia con IA
- Sistema de guardado y recuperación de sesiones
- Gestión de múltiples sesiones simultáneas

9. Épica 9 - Sprint 8: Gestión de Componentes

- Listado y organización de aplicaciones
- Operaciones CRUD sobre componentes
- Versionamiento de aplicaciones

10. Épica 10 - Sprint 9: Personalización

- Sistema de temas personalizable

- Configuración de preferencias de usuario
- Más de 20 temas predefinidos

11. Épica 11 - Sprint 10: Despliegue y Validación Final

- Pipeline CI/CD completo
- Despliegue en producción
- Pruebas de integración final
- Monitoreo y observabilidad

12. Épica 12 - Sprint 11: Buffer y Cierre

- Resolución de deuda técnica
- Documentación final
- Preparación de entrega

4.2.2. Identificación de Historias de Usuario (Requerimientos funcionales)

Las siguientes historias de usuario fueron identificadas y estructuradas siguiendo el formato estándar de Scrum: “Como [rol], quiero [funcionalidad], para [beneficio]”. Cada historia incluye criterios de aceptación medibles que definen cuándo se considera completada según la Definition of Done (DoD).

Las historias se organizan por épica, alineadas con la planificación de sprints del proyecto.

Épica 1: Sprint 0 - Concepción y Diseño

HU-001: Configuración del Entorno de Desarrollo

- **Como** desarrollador del equipo de Plataforma
- **Quiero** tener un repositorio Git configurado con flujos CI/CD automatizados
- **Para** garantizar la integración continua y el despliegue automatizado del código

Criterios de Aceptación:

- Repositorio creado en Bitbucket con estructura de carpetas definida
- Pipeline CI/CD configurado para compilación automática
- Pruebas automatizadas ejecutándose en cada commit
- Documentación del flujo de trabajo en README.md

Prioridad: Crítica | **Story Points:** 5 | **Sprint:** Sprint 0

HU-002: Definición de Arquitectura de Microservicios

- **Como** arquitecto de software
- **Quiero** diseñar una arquitectura basada en microservicios con APIs RESTful
- **Para** evitar la deuda técnica y garantizar escalabilidad

Criterios de Aceptación:

- Diagrama de arquitectura documentado (frontend, backend, servicios)
- Definición de endpoints RESTful principales
- Especificación de comunicación entre servicios

- Stack tecnológico validado (React 18, Node.js, AWS Bedrock)

Prioridad: Crítica | **Story Points:** 8 | **Sprint:** Sprint 0

Épica 2: Sprint 1 - Autenticación y Seguridad

HU-003: Sistema de Autenticación con JWT

- **Como** usuario del equipo de Proyectos
- **Quiero** poder iniciar sesión de forma segura en la plataforma
- **Para** acceder a las funcionalidades del Builder de manera protegida

Criterios de Aceptación:

- Endpoint /api/auth/login funcional con validación de credenciales
- Generación de tokens JWT con expiración configurable
- Middleware de autenticación implementado en Node.js
- Manejo de errores (credenciales inválidas, token expirado)
- Pruebas unitarias con cobertura >80 %

Prioridad: Alta | **Story Points:** 8 | **Sprint:** Sprint 1

Épica 3: Sprint 2 - Núcleo de IA

HU-004: Integración con AWS Bedrock

- **Como** desarrollador
- **Quiero** conectar la plataforma con AWS Bedrock
- **Para** habilitar la generación de código asistida por IA

Criterios de Aceptación:

- Cliente de AWS Bedrock configurado en Node.js
- Endpoint /api/ai/generate funcional
- Manejo de errores de API (timeout, rate limit, errores del modelo)
- Logs de uso y latencia de llamadas a Bedrock
- Pruebas de integración exitosas

Prioridad: Crítica | **Story Points:** 13 | **Sprint:** Sprint 2

HU-005: Ingeniería de Prompts para Generación y Edición de Componentes

- **Como** Product Owner
- **Quiero** que la IA genere estructuras JSON válidas a partir de descripciones en lenguaje natural
- **Para** automatizar la creación y edición de componentes

Criterios de Aceptación:

- Meta-prompts diseñados con técnicas Few-shot y Chain-of-Thought
- Validación de esquema JSON de salida

- Tasa de éxito >85 % en generación de estructuras válidas
- Documentación de patrones de prompts efectivos
- Casos de prueba: formulario de registro, encuesta, visita

Prioridad: Crítica | **Story Points:** 13 | **Sprint:** Sprint 2

Épica 4: Sprint 3 - Interfaz Base

HU-006: Interfaz de Usuario con React 18

- **Como** usuario
- **Quiero** una interfaz moderna y responsive
- **Para** trabajar cómodamente en cualquier dispositivo

Criterios de Aceptación:

- Componentes React con TypeScript implementados
- Diseño responsive
- Sistema de temas personalizable (>20 temas)
- Navegación fluida entre secciones

Prioridad: Alta | **Story Points:** 13 | **Sprint:** Sprint 3

Épica 5: Sprint 4 - Editor Visual

HU-007: Editor Visual

- **Como** desarrollador
- **Quiero** seleccionar componentes visuales para diseñar componentes
- **Para** acelerar el proceso de prototipado

Criterios de Aceptación:

- Paleta de componentes (formularios, botones, campos, etc.)
- Funcionalidad de selección operativa
- Panel de propiedades para configurar componentes
- Actualización en tiempo real de la vista previa
- Persistencia del diseño en formato JSON

Prioridad: Crítica | **Story Points:** 21 | **Sprint:** Sprint 4

HU-008: Integración de Monaco Editor

- **Como** desarrollador avanzado
- **Quiero** editar código manualmente con resaltado de sintaxis y autocompletado
- **Para** refinar el código generado por la IA

Criterios de Aceptación:

- Monaco Editor integrado en la interfaz

- Soporte para JSON, JavaScript, TypeScript, SQL, HTML
- Autocompletado funcional

Prioridad: Alta | **Story Points:** 13 | **Sprint:** Sprint 4

Épica 6: Sprint 5 - Renderizado y Bases de Datos

HU-009: Transformación JSON a Componentes Visuales

- **Como** sistema
- **Quiero** interpretar estructuras JSON y renderizar componentes visuales correspondientes
- **Para** mostrar la aplicación generada por la IA

Criterios de Aceptación:

- Algoritmo de parsing de JSON a componentes React
- Mapeo de tipos de datos a controles UI (text → input, select → dropdown)
- Renderizado dinámico sin recarga de página
- Manejo de errores en JSON malformado
- Pruebas con 10+ estructuras de aplicación diferentes

Prioridad: Crítica | **Story Points:** 13 | **Sprint:** Sprint 5

HU-010: Conexión Multi-Motor de Base de Datos

- **Como** desarrollador
- **Quiero** conectar la plataforma a MySQL y SQL Server
- **Para** gestionar datos de aplicaciones en diferentes espacios

Criterios de Aceptación:

- Servicio de conexión a MySQL implementado
- Servicio de conexión a SQL Server implementado
- Pool de conexiones configurado para optimizar rendimiento
- Endpoint /api/database/connect con validación de credenciales
- Manejo de errores de conexión y timeout

Prioridad: Alta | **Story Points:** 8 | **Sprint:** Sprint 5

Épica 7: Sprint 6 - Gestión de Datos Avanzada

HU-011: Administrador de Base de Datos Asistido por IA

- **Como** desarrollador
- **Quiero** realizar query SQL mediante lenguaje natural
- **Para** optimizar el tiempo de análisis de datos

Criterios de Aceptación:

- Endpoint /api/database/query-ai que traduce lenguaje natural a SQL

Prioridad: Media | **Story Points:** 13 | **Sprint:** Sprint 6

Épica 8: Sprint 7 - Asistente de Chat y Sesiones

HU-012: Chat de Asistencia con IA

- **Como** usuario
- **Quiero** interactuar con un asistente de IA mediante chat
- **Para** recibir ayuda contextual durante el desarrollo

Criterios de Aceptación:

- Interfaz de chat integrada en la plataforma
- Conexión con AWS Bedrock para respuestas inteligentes
- Contexto de conversación mantenido
- Sugerencias y mejores prácticas

Prioridad: Media | **Story Points:** 13 | **Sprint:** Sprint 7

HU-013: Guardado y Recuperación de Sesiones

- **Como** usuario
- **Quiero** guardar mi progreso y recuperarlo en sesiones futuras
- **Para** no perder mi trabajo al cerrar la plataforma

Criterios de Aceptación:

- Endpoint /api/sessions/save para persistir estado
- Endpoint /api/sessions/load para recuperar sesión
- Interfaz para listar sesiones guardadas

Prioridad: Alta | **Story Points:** 8 | **Sprint:** Sprint 7

Épica 9: Sprint 8 - Gestión de Componentes

HU-014: Gestión de componentes de Usuario

- **Como** usuario
- **Quiero** ver un listado de todos mis componentes creados
- **Para** organizarlos y acceder rápidamente a ellos

Criterios de Aceptación:

- Endpoint /api/apps/list con paginación
- Interfaz de tarjetas con descripción de componentes
- Acciones: editar, duplicar, eliminar
- Confirmación antes de eliminar

Prioridad: Media | **Story Points:** 5 | **Sprint:** Sprint 8

Épica 10: Sprint 9 - Personalización

HU-015: Sistema de Temas Personalizable

- **Como** usuario
- **Quiero** personalizar la apariencia del editor de código
- **Para** trabajar con un tema visual que me resulte cómodo

Criterios de Aceptación:

- Selector de temas en configuración de usuario
- Mínimo 20 temas predefinidos (Monokai, Dracula, GitHub, etc.)
- Persistencia de preferencia de tema
- Aplicación inmediata sin recarga
- Vista previa de temas antes de aplicar

Prioridad: Baja | **Story Points:** 5 | **Sprint:** Sprint 9

Épica 11: Sprint 10 - Despliegue y Validación Final

HU-016: Despliegue en Producción

- **Como** Product Owner
- **Quiero** desplegar la plataforma en un entorno de producción
- **Para** que el equipo de Proyectos pueda comenzar a utilizarla

Criterios de Aceptación:

- Pipeline CI/CD desplegando automáticamente a producción
- CodePipeline con 6 etapas configuradas (Source, Build, Deploy, Approval, CreateAMI, UpdateASG)
- CodeBuild projects operativos (Frontend, Backend, Deploy, AMI, ASG)
- ECR creados y funcionales
- Auto Scaling Group con Target Groups
- S3 para artefactos
- Monitoreo y logs configurados (CloudWatch)
- Plan de rollback de versionado

Prioridad: Crítica | **Story Points:** 13 | **Sprint:** Sprint 10

HU-017: Pruebas de Integración Final

- **Como** Product Owner
- **Quiero** ejecutar pruebas de integración completas del sistema
- **Para** validar que todos los componentes funcionan correctamente en conjunto

Criterios de Aceptación:

- Suite de pruebas de integración ejecutándose exitosamente
- Pruebas end-to-end de flujos críticos (generación, edición, guardado)
- Validación de integración con servicios AWS (Bedrock, S3, Cognito)
- Pruebas de carga y rendimiento
- Documentación de casos de prueba y resultados
- Cobertura de pruebas >80%

Prioridad: Crítica | **Story Points:** 8 | **Sprint:** Sprint 10

Épica 12: Sprint 11 - Buffer y Cierre

HU-018: Resolver Deuda Técnica y Preparar Entrega Final

- **Como** equipo de desarrollo
- **Quiero** resolver la deuda técnica acumulada y preparar la documentación final
- **Para** entregar un producto de calidad con documentación completa

Criterios de Aceptación:

- Refactorización de código con deuda técnica identificada
- Optimización de rendimiento en áreas críticas
- Documentación técnica completa (README, guías de instalación, API docs)
- Documentación de usuario final
- Video demo de funcionalidades principales
- Preparación de presentación final
- Entrega de código fuente y artefactos

Prioridad: Alta | **Story Points:** 8 | **Sprint:** Sprint 11

Resumen del Product Backlog:

El Product Backlog completo consta de:

- **Total de Épicas:** 12 épicas principales
- **Total de Historias de Usuario:** 18 historias de usuario
- **Total de Story Points:** 159 story points
- **Duración Total:** 6 meses (11 sprints de 2 semanas + 1 sprint de 5 días)
- **Velocidad Promedio Estimada:** 14.5 story points por sprint

La distribución de story points por sprint se muestra en la Tabla 5:

Cuadro 5: Distribución de Story Points por Sprint

Sprint	Story Points	Historias de Usuario
Sprint 0	13	HU-001, HU-002
Sprint 1	8	HU-003
Sprint 2	26	HU-004, HU-005
Sprint 3	13	HU-006
Sprint 4	34	HU-007, HU-008
Sprint 5	21	HU-009, HU-010
Sprint 6	13	HU-011
Sprint 7	21	HU-012, HU-013
Sprint 8	5	HU-014
Sprint 9	5	HU-015
Sprint 10	21	HU-016, HU-017
Sprint 11	8	HU-018
Total	159	18 HU

4.2.3. Priorización de tareas en Jira

La gestión y priorización del Product Backlog se realizó utilizando Jira, la plataforma de gestión ágil de Atlassian. Jira permite visualizar, organizar y priorizar las historias de usuario de manera eficiente, facilitando la planificación de Sprints y el seguimiento del progreso del proyecto.

En Jira, el proyecto se estructuró de la siguiente manera:

- **Épicas:** Cada sprint se representó como una épica en Jira (IR-3 a IR-14), agrupando las historias de usuario correspondientes
- **Historias de Usuario:** Cada HU se creó como una historia vinculada a su épica padre (IR-15 a IR-32)
- **Estados:** Las historias transicionaron por los estados: Tareas por hacer → En curso → Finalizada
- **Campos Personalizados:** Se configuraron campos para Story Points, Sprint, fechas de inicio y vencimiento

La Figura 1 muestra el tablero de Jira con las épicas e historias de usuario organizadas cronológicamente. Las historias críticas relacionadas con la arquitectura base (Sprint 0) y la integración de AWS Bedrock (Sprint 2) fueron priorizadas en los primeros Sprints, mientras que las funcionalidades de personalización y optimización se programaron para Sprints posteriores.

Nota: El tablero completo de Jira puede consultarse en: [\[ENLACE INTERNO CENSURADO\]](#)

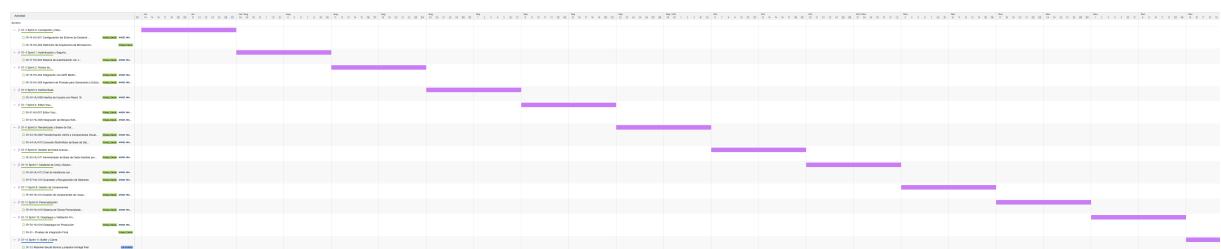


Figura 1: Priorización de tareas en Jira para el proyecto Irakani Builder

La priorización se basó en los siguientes criterios:

- **Valor de Negocio:** Impacto directo en la reducción de costos operativos y mejora de la productividad del equipo de Proyectos.
- **Dependencias Técnicas:** Funcionalidades que son prerequisitos para otras historias de usuario (ej. autenticación antes de gestión de sesiones).

- **Riesgo Técnico:** Historias con mayor incertidumbre o complejidad se priorizaron temprano para mitigar riesgos.

- **Esfuerzo Estimado:** Medido en Story Points utilizando la secuencia de Fibonacci (1, 2, 3, 5, 8, 13, 21).

El Product Owner (Luis Flores) fue responsable de mantener y ajustar continuamente estas prioridades en función de la retroalimentación del equipo y los cambios en las necesidades del negocio, siguiendo los principios de Scrum de inspección y adaptación.

4.3. Diseño de la Arquitectura y Flujo de Datos

4.3.1. Arquitectura de Microservicios y Comunicación

La arquitectura de Irakani Builder se diseñó siguiendo el patrón de microservicios, donde cada componente del sistema opera como un servicio independiente y autónomo. Esta decisión arquitectónica responde directamente a la necesidad de evitar la deuda técnica que caracterizaba a la plataforma anterior (app.irakani.com) y garantizar la escalabilidad, mantenibilidad y resiliencia del sistema.

La Figura 2 ilustra la arquitectura completa del sistema, mostrando la interacción entre los diferentes componentes y servicios.

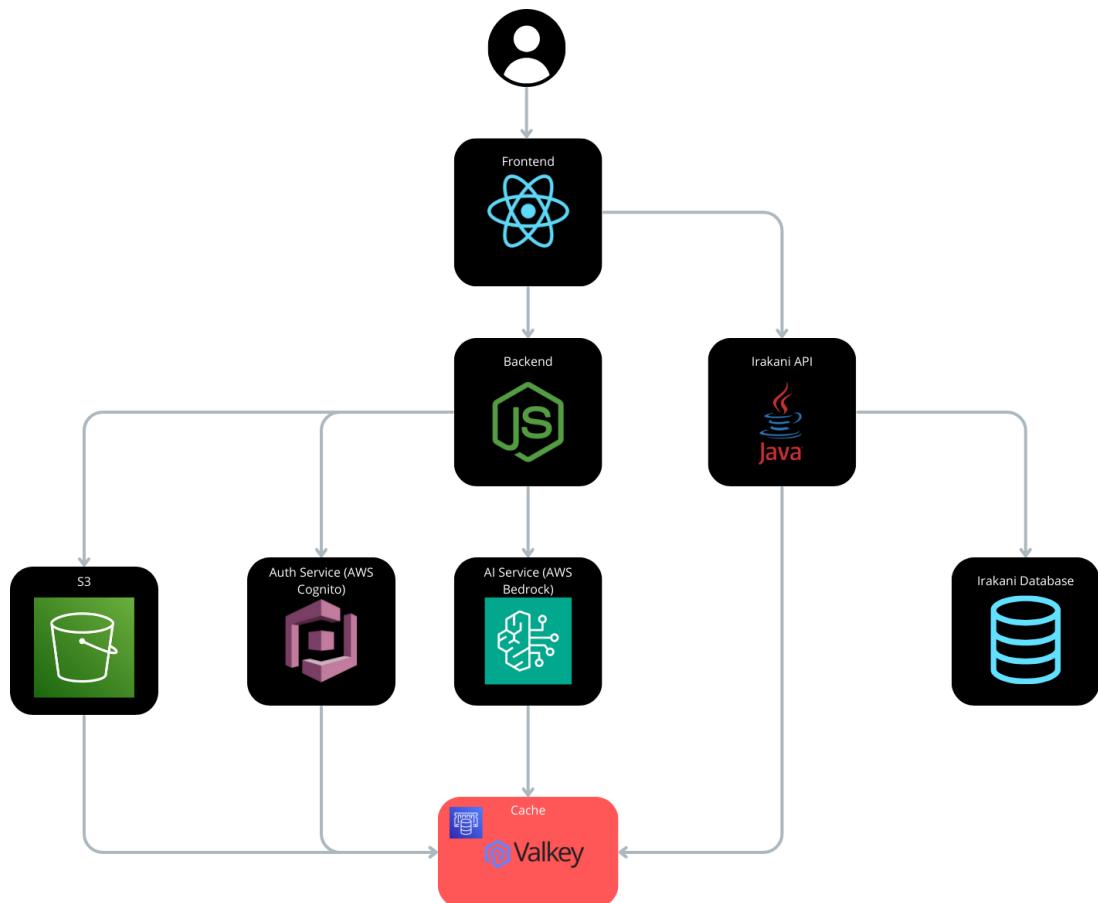


Figura 2: Arquitectura de microservicios de Irakani Builder

Componentes Principales de la Arquitectura:

1. Capa de Presentación - Frontend (React 18):

- Interfaz de usuario construida con React 18 y TypeScript
- Componentes reutilizables para el editor visual
- Integración de Monaco Editor para edición de código

- Sistema de temas personalizable
- Comunicación con el backend mediante APIs RESTful

2. Capa de Aplicación - Backend (Node.js):

- Servidor Node.js con arquitectura basada en Express.js
- Gestión de autenticación y autorización mediante JWT
- Orquestación de llamadas a microservicios especializados
- Manejo de sesiones de usuario
- Validación de datos y lógica de negocio

3. Microservicios Especializados:

- **Auth Service (AWS Cognito):** Gestión de autenticación y autorización de usuarios, generación y validación de tokens JWT
- **AI Service (AWS Bedrock):** Servicio de generación de código asistida por IA, procesamiento de prompts y generación de estructuras JSON
- **Database Service:** Gestión de conexiones a múltiples motores de bases de datos (MySQL, SQL Server), ejecución de queries y optimización de consultas

4. Capa de Datos:

- **S3 (Amazon S3):** Almacenamiento de artefactos, archivos estáticos y backups
- **Irakani Database:** Base de datos principal para almacenar metadatos de aplicaciones, sesiones de usuario y configuraciones

5. Capa de Caché (Valkey):

- Sistema de caché distribuido para mejorar el rendimiento
- Almacenamiento temporal de respuestas de IA frecuentes
- Gestión de sesiones activas
- Reducción de latencia en operaciones repetitivas

Ventajas de esta Arquitectura:

- **Escalabilidad Independiente:** Cada microservicio puede escalarse de forma independiente según la demanda
- **Tolerancia a Fallos:** Un fallo en un servicio no compromete la operación de todo el sistema
- **Flexibilidad Tecnológica:** Cada servicio puede utilizar el stack tecnológico más adecuado para su función
- **Facilidad de Mantenimiento:** Los servicios pueden actualizarse y desplegarse de forma independiente
- **Desarrollo Paralelo:** Diferentes equipos pueden trabajar en servicios distintos simultáneamente

4.3.2. Estrategia de Integración de Datos (API Wrapper)

El sistema Irakani Builder implementa una arquitectura de Backend for Frontend (BFF) que actúa como capa de abstracción entre el frontend React y los servicios externos de AWS. Esta estrategia permite centralizar la lógica de integración, gestionar la autenticación, y optimizar las peticiones hacia los servicios de almacenamiento y procesamiento.

A) Arquitectura de Integración:

El sistema se conecta directamente a una API de Irakani legado, además implementa su propio backend (Node.js/Express) que orquesta las operaciones con servicios de AWS:

1. Frontend (React + TypeScript) → Backend BFF (Node.js/Express) → Servicios AWS (S3, DynamoDB, Bedrock, Step Functions)

2. Servicios de Integración Implementados:

- `irakaniApi.ts`: Cliente para operaciones de almacenamiento en la base de datos del espacio
- `databaseService.ts`: Cliente para gestión de conexiones a bases de datos (MySQL, SQL Server)
- `authService.ts`: Gestión de autenticación con AWS Cognito
- `bedrockChatService.ts`: Integración con modelos de IA de AWS Bedrock

B) Endpoints del Backend BFF y sus Interfaces JSON:

1. OPERACIONES DE ALMACENAMIENTO (S3):

Endpoint: POST /api/s3/upload

Propósito: Guardar estado de aplicaciones generadas con versionamiento

Request Body:

```
1 {
2   "bucket": "string",           // Nombre del bucket S3
3   "key": "string",             // Ruta del archivo
4   "data": "object",            // Datos JSON de la aplicación
5   "userId": "string",          // ID del usuario
6   "appName": "string"          // Nombre de la aplicación
7 }
```

Listing 1: Request Body para guardar en S3

Response:

```
1 {
2   "success": true,
3   "versionId": "string",        // ID de versión de S3
4   "appId": "string",            // ID único de la aplicación
5   "message": "string"
6 }
```

Listing 2: Response de S3 Upload

Endpoint: POST /api/s3/download

Propósito: Recuperar estado de aplicaciones desde S3

Request Body:

```
1 {
2   "bucket": "string",
3   "key": "string"
4 }
```

Listing 3: Request Body para S3 Download

Response: JSON con el contenido del archivo almacenado

2. OPERACIONES DE REGISTRO DE APLICACIONES (DynamoDB):

Endpoint: GET /api/user/:userId/apps

Propósito: Listar todas las aplicaciones de un usuario

Response:

```
1 {
2   "apps": [
3     {
4       "appId": "string",
5       "appName": "string",
6       "s3Key": "string",
7       "s3Bucket": "string",
8       "versionId": "string",
9       "createdAt": "ISO8601",
10      "updatedAt": "ISO8601"
11    }
12  ]
13 }
```

```
11     }
12   ]
13 }
```

Listing 4: Response de listado de aplicaciones

Endpoint: DELETE /api/user/:userId/apps/:appId

Propósito: Eliminar registro de aplicación

Response:

```
1 {
2   "success": true,
3   "message": "string"
4 }
```

Listing 5: Response de eliminacion

3. OPERACIONES DE INTELIGENCIA ARTIFICIAL (AWS Bedrock):

Endpoint: POST /api/chat/bedrock

Propósito: Chat con IA para asistencia en desarrollo

Request Body:

```
1 {
2   "messages": [
3     {
4       "role": "user o assistant",
5       "content": [{"text": "string"}]
6     }
7   ],
8   "systemPrompt": "string"      // Opcional
9 }
```

Listing 6: Request Body para chat con Bedrock

Response:

```
1 {
2   "message": "string"          // Respuesta del modelo
3 }
```

Listing 7: Response del chat

Endpoint: POST /api/chat/bedrock/stream

Propósito: Chat con streaming para respuestas en tiempo real

Request Body: Igual que /api/chat/bedrock

Response: Server-Sent Events (SSE) con chunks de texto

4. OPERACIONES DE GENERACIÓN (Step Functions):

Endpoint: POST /api/analyze

Propósito: Analizar requerimientos para generación de aplicaciones

Endpoint: POST /api/generate

Propósito: Generar código de aplicación

Endpoint: POST /api/stepfunctions/status

Propósito: Consultar estado de ejecución de Step Functions

Request Body:

```
1 {
2   "executionArn": "string"
3 }
```

Listing 8: Request Body para consultar estado

Response: Estado de la ejecución (RUNNING, SUCCEEDED, FAILED, etc.)

5. OPERACIONES DE BASE DE DATOS:

Endpoint: POST /api/database/connect

Propósito: Establecer conexión a base de datos externa

Request Body:

```
1 {
2     "dbType": "mysql o sqlserver o cloudflare",
3     "config": {
4         "host": "string",           // Para MySQL
5         "server": "string",        // Para SQL Server
6         "user": "string",
7         "password": "string",
8         "database": "string",
9         "port": "number",          // Opcional
10        "encrypt": "boolean",      // SQL Server
11        "trustServerCertificate": "boolean",
12        "ssl": "boolean"           // MySQL
13    }
14 }
```

Listing 9: Request Body para conexión a BD

Response:

```
1 {
2     "connectionId": "string",      // ID único de conexión
3     "token": "string",            // Token de autenticación
4     "status": "connected",
5     "dbType": "string",
6     "databaseName": "string"
7 }
```

Listing 10: Response de conexión a BD

Endpoint: POST /api/database/query

Propósito: Ejecutar consultas SQL

Headers:

```
1 Authorization: Bearer {token}
2 X-Connection-Id: {connectionId}
```

Listing 11: Headers requeridos

Request Body:

```
1 {
2     "query": "string",
3     "params": ["any"]           // Parámetros preparados
4 }
```

Listing 12: Request Body para ejecutar query

Response:

```
1 {
2     "rows": ["object"],          // Resultados de la consulta
3     "fields": ["object"],        // Metadatos de columnas
4     "rowCount": "number"
5 }
```

Listing 13: Response de query SQL

Endpoint: GET /api/database/tables
Propósito: Obtener lista de tablas de la base de datos
Headers: Authorization, X-Connection-Id, X-User-Id
Response:

```

1 {
2   "tables": [
3     {
4       "name": "string",
5       "schema": "string",           // Opcional
6       "type": "TABLE o VIEW"
7     }
8   ]
9 }
```

Listing 14: Response de listado de tablas

Endpoint: GET /api/database/tables/:tableName/structure
Propósito: Obtener estructura de una tabla
Response:

```

1 {
2   "columns": [
3     {
4       "name": "string",
5       "type": "string",
6       "nullable": "boolean",
7       "key": "PRI o UNI o vacio",
8       "default": "any",
9       "extra": "string"
10    }
11  ]
12 }
```

Listing 15: Response de estructura de tabla

Endpoint: POST /api/database/disconnect

Propósito: Cerrar conexión a base de datos

Headers: Authorization, X-Connection-Id

6. OPERACIONES DE GENERACIÓN DE ICONOS:

Endpoint: POST /api/generate-icon

Propósito: Generar iconos usando IA (Titan Image Generator)

Request Body:

```

1 {
2   "prompt": "string",           // Descripcion del icono
3   "bucket": "string",          // Opcional
4   "userId": "string"
5 }
```

Listing 16: Request Body para generar icono

Response:

```

1 {
2   "imageUrl": "string",         // URL local del icono
3   "s3Key": "string",
4   "s3Bucket": "string",
5   "originalPrompt": "string",
6   "translatedPrompt": "string", // Traduccion al ingles
7   "customColor": "string",      // Color hex
8   "tokenUsage": {
```

```

9   "haiku": {
10    "inputTokens": "number",
11    "outputTokens": "number",
12    "model": "string"
13  },
14  "titan": {
15    "imagesGenerated": "number",
16    "model": "string"
17  }
18}
19

```

Listing 17: Response de generacion de icono

Endpoint: GET /api/imagen/:imagenKey

Propósito: Servir imágenes almacenadas en S3

Response: Imagen binaria (image/png o image/jpeg)

C) Gestión de Sesiones y Autenticación:

1. Autenticación con AWS Cognito:

- El authService.ts gestiona el flujo de autenticación
- Las credenciales se almacenan en Valkey para persistencia
- Cada petición al backend incluye headers de identificación:
 - X-User-Id: Username de Cognito
 - X-Context: Contexto de la operación
 - X-Schema: Esquema de datos utilizado

2. Tokens de Conexión a Base de Datos:

- Cada conexión genera un token único
- Los tokens se validan en cada petición
- Las conexiones se almacenan en Valkey por usuario

D) Patrón de Wrapper Implementado:

El servicio irakaniApi.ts implementa un patrón de wrapper que:

1. Centraliza la configuración de endpoints (apiConfig.ts)
2. Abstacta la complejidad de las peticiones HTTP
3. Maneja errores de forma consistente
4. Proporciona una interfaz TypeScript tipada
5. Permite mock de servicios para desarrollo (mockServices.ts)

Ejemplo de implementación del wrapper:

```

1 // src/services/irakaniApi.ts
2 export const irakaniApi = {
3   async saveToS3(data: any): Promise<any> {
4     const backendUrl = `${apiConfig.baseUrl}${apiConfig.endpoints.s3Upload}`;
5     const response = await fetch(backendUrl, {
6       method: 'POST',
7       headers: { 'Content-Type': 'application/json' },
8       body: JSON.stringify({
9         bucket: apiConfig.bucket,
10        key: data.fileName || 'improved/app-state.json',
11      })
12    });
13    return response;
14  }
15}
16

```

```

11     data: data.jsonData || data,
12     userId: data.userId,
13     appName: data.appName
14   })
15 );
16
17 if (!response.ok) {
18   throw new Error(`Error ${response.status}: ${await response.text()}`);
19 }
20
21 return await response.json();
22
23 }

```

Listing 18: Implementación del wrapper en TypeScript

E) Trazabilidad y Monitoreo:

El backend implementa trazabilidad completa de operaciones:

1. Logging de tokens consumidos por modelo de IA
2. Cálculo de costos por operación
3. Registro en DynamoDB (tabla UsageLogService) de:
 - Tokens de entrada/salida
 - Modelo utilizado
 - Usuario y contexto
 - Timestamp de operación
 - Costo estimado

Esta arquitectura permite una integración escalable, mantenible y observable con los servicios de AWS, sin depender de una API REST legado de Irakani.

4.3.3. Diseño de Gestión de Sesiones y Caché (Valkey)

Valkey es un almacén de datos en memoria de código abierto, compatible con Redis, que proporciona almacenamiento de estructuras de datos de alta velocidad. En Irakani Builder, Valkey se utiliza como capa de caché distribuida y sistema de gestión de sesiones, reemplazando el uso tradicional de localStorage del navegador con una solución centralizada y escalable.

A) Justificación del Uso de Valkey:

1. Rendimiento y Baja Latencia:

- Almacenamiento en memoria RAM: Valkey opera completamente en memoria, proporcionando tiempos de respuesta en microsegundos (sub-milisegundo) para operaciones de lectura/escritura
- Operaciones atómicas: Las operaciones SET/GET son atómicas, garantizando consistencia sin necesidad de bloqueos complejos
- Throughput elevado: Capaz de manejar cientos de miles de operaciones por segundo, ideal para aplicaciones con múltiples usuarios concurrentes

2. Persistencia de Sesiones Multi-Dispositivo:

- A diferencia de localStorage (limitado al navegador local), Valkey permite que un usuario acceda a su sesión desde cualquier dispositivo
- Las sesiones persisten incluso si el usuario cierra el navegador o cambia de máquina
- Soporte para múltiples sesiones simultáneas por usuario (desarrollo paralelo de diferentes aplicaciones)

3. Escalabilidad Horizontal:

- Valkey Cluster (implementado en AWS ElastiCache) permite distribución automática de datos entre múltiples nodos
- Capacidad de escalar agregando más nodos sin tiempo de inactividad
- Replicación automática para alta disponibilidad

4. Gestión Centralizada de Estado:

- Todos los estados temporales del editor (tema, conexiones DB, tokens, configuraciones) se centralizan en un único sistema
- Facilita la sincronización entre diferentes componentes de la aplicación
- Simplifica el debugging y monitoreo de sesiones activas

5. Expiración Automática (TTL):

- Soporte nativo para Time-To-Live (TTL) en claves
- Limpieza automática de datos obsoletos sin intervención manual
- Gestión eficiente de memoria al eliminar sesiones inactivas

6. Fallback a localStorage:

- La implementación incluye un mecanismo de fallback automático a localStorage si Valkey no está disponible
- Garantiza que la aplicación siga funcionando incluso con problemas de conectividad

B) Arquitectura de Integración con Valkey:

La integración se implementa en tres capas:

1. Capa de Infraestructura (AWS ElastiCache):

- Cluster: [REDACTED]
- Puerto: [REDACTED] (protocolo Redis)
- Modo: Cluster con replicación automática
- Región: [REGIÓN AWS CENSURADA]

2. Capa de Backend (Node.js):

- Cliente: redis (npm package compatible con Valkey)
- Rutas: /api/valkey/* (backend/routes/valkey.js)
- Gestión de conexión con retry automático
- Middleware de verificación de disponibilidad

3. Capa de Frontend (TypeScript):

- valkeyService.ts: Cliente base para operaciones CRUD
- userValkeyService.ts: Servicio con contexto de usuario y sesiones
- Fallback automático a localStorage

C) Estructura de Datos en Caché (Key-Value Pairs):

El sistema implementa una jerarquía de claves estructurada para organizar los datos por usuario y sesión:

1. Claves Globales (sin usuario):

- Patrón: global:{key}
- Uso: Configuraciones compartidas o datos temporales sin autenticación

- Ejemplo: `global:app-config`

2. Claves de Usuario (sin sesión específica):

- Patrón: `user:{userId}:{key}`
- Uso: Datos persistentes del usuario que no dependen de una sesión
- Ejemplos:
 - `user:johndoe:sessions:list` → ["session_1234", "session_5678"]
 - `user:johndoe:sessions:active` → "session_1234"
 - `user:johndoe:cognito-session` → {username:"johndoe",...}

3. Claves de Sesión de Usuario (contexto completo):

- Patrón: `user:{userId}:session:{sessionId}:{key}`
- Uso: Datos específicos de una sesión de trabajo (aplicación en desarrollo)
- Ejemplos:
 - `user:johndoe:session:session_1234:editor-theme` → "vs-dark"
 - `user:johndoe:session:session_1234:irakani-selected-space` → "12345"
 - `user:johndoe:session:session_1234:db_connection_ids` → [conn_001]
 - `user:johndoe:session:session_1234:github_token` → "ghp_..."

D) Tipos de Datos Almacenados:

1. Sesiones de Autenticación:

- Clave: `cognito-session`
- TTL: Sin expiración (se elimina en logout)

2. Configuraciones del Editor:

- Clave: `editor-theme`
- Valores: "vs-dark" | "vs-light" | "hc-black" | "monokai" | ... (20+ temas)
- TTL: Sin expiración

3. Conexiones a Bases de Datos:

- Clave: `db_connection_{connectionId}`
- TTL: 24 horas (renovable con actividad)

4. Tokens de Integración:

- Claves: `github_token`, `irakani-app-token`
- Valores: Strings de tokens JWT o API keys
- TTL: Según política de cada servicio

5. Estado de Aplicación en Desarrollo:

- Clave: `irakani-app-storage`
- Estructura: JSON completo del estado de la aplicación (puede ser varios MB)
- TTL: 7 días de inactividad

6. Contexto de Espacio Irakani:

- Clave: `irakani-selected-space`

- Valor: ID del espacio seleccionado
- TTL: Sin expiración

7. Gestión de Sesiones:

- Clave: sessions:list → ["sessionId1", "sessionId2", ...]
- Clave: sessions:active → "sessionId"(sesión actualmente en uso)

E) Operaciones Implementadas:

1. Operaciones Básicas (valkeyService.ts):

- setItem(key, value, ttl?): Almacena un valor con TTL opcional
- getItem(key): Recupera un valor por clave
- removeItem(key): Elimina una clave
- clear(): Limpia todas las claves (FLUSHALL)
- exists(key): Verifica si una clave existe
- keys(pattern): Busca claves por patrón (ej: user:johndoe:*)
- deleteMultiple(keys): Elimina múltiples claves en una operación

2. Operaciones de Usuario y Sesión (userValkeyService.ts):

- setCurrentUser(userId): Establece el contexto de usuario actual
- setCurrentSession(sessionId): Establece la sesión activa
- getCurrentSession(): Obtiene la sesión activa del usuario
- createSession(sessionName?): Crea una nueva sesión de trabajo
- listSessions(): Lista todas las sesiones del usuario
- deleteSession(sessionId): Elimina una sesión y todas sus claves asociadas
- setItem/getItem/removeItem/exists: Con contexto automático de usuario y sesión

F) Endpoints del Backend para Valkey:

```

1 POST /api/valkey/set
2 Request: { "key": "string", "value": "string", "ttl": number }
3 Response: { "success": true }

4
5 POST /api/valkey/get
6 Request: { "key": "string" }
7 Response: { "value": string | null }

8
9 POST /api/valkey/del
10 Request: { "key": "string" }
11 Response: { "success": true }

12
13 POST /api/valkey/clear
14 Request: {}
15 Response: { "success": true }

16
17 POST /api/valkey/exists
18 Request: { "key": "string" }
19 Response: { "exists": boolean }
```

Listing 19: Endpoints de Valkey

G) Estrategia de Migración y Compatibilidad:

1. Migración Automática de Datos:

- Al crear la primera sesión, el sistema migra automáticamente datos existentes del formato antiguo (`user:{userId}:{key}`) al nuevo formato con sesión
- Claves migradas: `editor-theme`, `db_connection_ids`, `github_token`, `irakani-selected-space`, `cognito-session`, etc.

2. Fallback a localStorage:

- Todas las operaciones incluyen try-catch con fallback a localStorage
- Garantiza funcionamiento offline o con problemas de red
- Transparente para el código cliente

3. Gestión de Errores:

- Retry automático con backoff exponencial (hasta 10 intentos)
- Timeout de 1 hora para reintentos totales
- Logging detallado de errores de conexión

H) Beneficios de la Arquitectura Implementada:

1. **Aislamiento de Sesiones:** Cada sesión de trabajo es independiente, permitiendo desarrollo paralelo de múltiples aplicaciones
2. **Persistencia Confiable:** Los datos sobreviven a reinicios del navegador y cambios de dispositivo
3. **Rendimiento Optimizado:** Operaciones en memoria con latencia sub-milisegundo
4. **Escalabilidad:** Arquitectura preparada para miles de usuarios concurrentes
5. **Mantenibilidad:** Estructura de claves clara y jerárquica facilita debugging
6. **Resiliencia:** Fallback automático garantiza disponibilidad continua

Esta arquitectura de caché con Valkey es fundamental para proporcionar una experiencia de usuario fluida y profesional, eliminando las limitaciones de localStorage y preparando el sistema para escalar a nivel empresarial.

4.3.4. Diseño de la Integración con IA (AWS Bedrock)

Irakani Builder implementa una integración completa con AWS Bedrock para proporcionar capacidades de IA generativa en múltiples contextos. La arquitectura se basa en un sistema de comunicación bidireccional entre el frontend (React) y el backend (Node.js), que actúa como intermediario con los servicios de AWS.

A) Arquitectura de la Integración:

1. Componentes Principales:

Frontend (bedrockChatService.ts):

- Servicio centralizado para todas las interacciones con IA
- Gestión de historial de conversaciones por aplicación
- Manejo de contextos dinámicos y especializados
- Soporte para streaming de respuestas en tiempo real
- Comandos especiales (/icon, /code, /ticket)
- Caché de prompts contextuales para optimización

Backend (server.js):

- Cliente AWS Bedrock Runtime configurado con SDK v3
- Endpoints REST para chat estándar y streaming
- Registro de uso de tokens y costos por usuario
- Integración con DynamoDB para persistencia de métricas
- Soporte para múltiples modelos de IA

Ejemplo de Configuración del Cliente AWS Bedrock:

```
1 const { BedrockRuntimeClient, ConverseCommand,
2         ConverseStreamCommand } = require('@aws-sdk/client-bedrock-runtime');
3
4 // Configuración con soporte para rol IAM o credenciales explícitas
5 const awsConfig = {
6   region: process.env.AWS_REGION || '[REDACTED]'
7 };
8
9 if (process.env.AWS_ACCESS_KEY_ID && process.env.AWS_SECRET_ACCESS_KEY) {
10   awsConfig.credentials = {
11     accessKeyId: process.env.AWS_ACCESS_KEY_ID,
12     secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY,
13     sessionToken: process.env.AWS_SESSION_TOKEN
14   };
15 }
16
17 const bedrockClient = new BedrockRuntimeClient(awsConfig);
18
19 // Precios por modelo (por 1000 tokens) - [REDACTED]
20 const MODEL_PRICING = {
21   'claude-sonnet-4': { input: [REDACTED], output: [REDACTED] },
22   'claude-3-5-haiku': { input: [REDACTED], output: [REDACTED] },
23   'titan-image-generator-v2': { perImage: [REDACTED] }
24 };
25
26 function calculateCost(model, inputTokens = 0, outputTokens = 0) {
27   const pricing = MODEL_PRICING[model];
28   if (!pricing) return 0;
29   const inputCost = (inputTokens / 1000) * pricing.input;
30   const outputCost = (outputTokens / 1000) * pricing.output;
31   return inputCost + outputCost;
32 }
```

Listing 20: Configuración del cliente AWS Bedrock en backend

2. Modelos de IA Utilizados:

Claude Sonnet 4 (anthropic.claude-sonnet-4-20250514-v1:0):

- Modelo principal para generación de código y asistencia
- Ventana de contexto: 200K tokens
- Costo: [PRECIO CENSURADO] por 1K tokens
- Uso: Chat contextual, generación de aplicaciones, modificación de JSON

Claude 3.5 Haiku (anthropic.claude-3-5-haiku-20241022-v1:0):

- Modelo alternativo para tareas más simples
- Costo: [PRECIO CENSURADO] por 1K tokens

- Uso: Respuestas rápidas, análisis ligeros

Titan Image Generator V2 (amazon.titan-image-generator-v2:0):

- Generación de iconos y recursos gráficos
- Costo: [PRECIO CENSURADO] por imagen
- Uso: Comando /icon para crear iconos de aplicaciones

Llama 4 Maverick 17B (meta.llama4-maverick-17b-instruct-v1:0):

- Modelo experimental para casos específicos
- Costo: [PRECIO CENSURADO] por 1K tokens

B) Flujos de Comunicación:

1. Flujo de Chat Estándar (Sin Streaming):

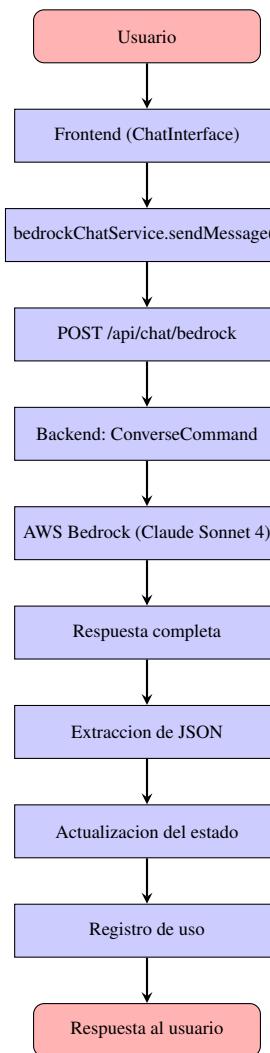


Figura 3: Diagrama de flujo: Chat estándar con AWS Bedrock

2. Flujo de Chat con Streaming:

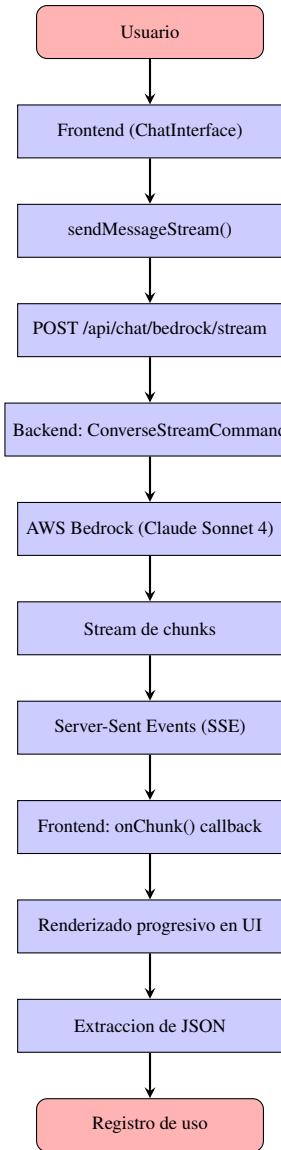


Figura 4: Diagrama de flujo: Chat con streaming en tiempo real

C) Sistema de Contextos Dinámicos:

El servicio implementa un sistema sofisticado de contextos que permite a la IA entender y modificar partes específicas de la aplicación:

1. Contextos Globales:

- 'aplicacion': Toda la estructura de la aplicación
- 'listas': Todas las listas de datos
- 'perfils': Todos los perfiles de usuario
- 'entidades': Todas las entidades de datos
- 'workflows': Todos los flujos de trabajo

2. Contextos Específicos:

- 'lista-{id}': Una lista específica
- 'lista-{id}-campos': Solo los campos de una lista

- 'perfil-{id}': Un perfil específico
- 'entidad-{id}': Una entidad específica
- 'workflows-{id}': Un workflow específico
- 'formulario-{index}': Un formulario específico
- 'elemento-{formIndex}-{elementId}': Un elemento específico
- 'formulario-{index}-crear-elementos': Agregar elementos a un formulario

3. Gestión de Contextos:

- Caché de prompts por contexto para evitar regeneración
- Limpieza automática de historial al cambiar contexto
- Preservación de scripts personalizados al actualizar JSON
- Merge inteligente de cambios parciales

D) Comandos Especiales:

1. Comando /icon:

- Sintaxis: /icon descripción del ícono [#color]
- Detecta colores hexadecimales en el prompt
- Traduce automáticamente al inglés
- Genera 4 versiones de la imagen: Original, Fondo blanco, Transparente, Color personalizado
- Descarga automática de todas las versiones
- Actualiza JSON con imagenKey para referencia

2. Comando /code:

- Sintaxis: /code descripción del código
- Genera código JavaScript o SQL según contexto
- Extrae bloques de código de la respuesta
- Callback automático para insertar en editor
- Soporta scripts de elementos, listas, entidades

3. Comando /ticket:

- Sintaxis: /ticket descripción del ticket
- Genera código de impresión de tickets
- Soporta múltiples formatos (TSPL, ESCPOS, ZPL, etc.)
- Extrae y envía código al editor de tickets

E) Registro de Uso y Costos:

El sistema implementa un registro completo de uso de IA para análisis y facturación:

1. Métricas Registradas:

- Usuario (username de Cognito)
- Modelo utilizado

- Tokens de entrada y salida
- Costo calculado en USD
- Timestamp de la operación
- Contexto de la operación
- Esquema de trabajo (espacio de Irakani)

2. Almacenamiento:

- DynamoDB: Tabla 'irakani-builder-usage-logs'
- Partition Key: userId
- Sort Key: timestamp
- TTL: 90 días (configurable)

3. Cálculo de Costos:

```

1 inputCost = (inputTokens / 1000) * precioInput
2 outputCost = (outputTokens / 1000) * precioOutput
3 costoTotal = inputCost + outputCost

```

Listing 21: Cálculo de costos de uso de IA

4. Dashboard de Uso:

- Componente UsageDashboard en SpaceMenu
- Visualización de tokens consumidos
- Costos por modelo y período
- Gráficos de tendencias
- Exportación de reportes

F) Optimizaciones Implementadas:

1. Caché de Prompts:

- Los system prompts se cachean por contexto
- Evita regeneración en mensajes subsecuentes
- Reduce latencia y tokens consumidos
- Se limpia al cambiar de contexto

2. Historial Limitado:

- Máximo 20 mensajes por conversación
- Previene crecimiento excesivo de contexto
- Reduce costos de tokens de entrada
- Mantiene relevancia de la conversación

3. Contextos Resumidos:

- Función createResumedContext() para JSON grandes
- Incluye solo información esencial
- Reduce tokens de entrada significativamente

- Mantiene funcionalidad completa

4. Streaming Optimizado:

- Server-Sent Events para respuestas en tiempo real
- Renderizado progresivo en UI
- Mejor experiencia de usuario
- Permite cancelación de operaciones

G) Manejo de Errores y Resiliencia:

1. Errores de Red:

- Reintentos automáticos con backoff exponencial
- Mensajes de error descriptivos al usuario
- Fallback a modo sin streaming si falla

2. Errores de Parsing:

- Validación de JSON antes de aplicar
- Preservación del estado anterior si falla
- Mensajes claros sobre qué falló

3. Tokens Expirados:

- Detección automática de sesiones expiradas
- Redirección a login cuando es necesario
- Preservación del trabajo en progreso

4. Límites de Rate:

- Manejo de errores 429 (Too Many Requests)
- Mensajes informativos al usuario
- Sugerencias de espera

H) Seguridad y Privacidad:

1. Autenticación:

- Todos los endpoints requieren usuario autenticado
- Header 'x-user-id' en todas las peticiones
- Validación de sesión en backend

2. Aislamiento de Datos:

- Cada usuario tiene su propio historial
- Contextos separados por aplicación
- Esquemas de trabajo aislados (espacios de Irakani)

3. Sanitización:

- Limpieza de prompts antes de enviar a Bedrock
- Validación de JSON recibido

- Protección contra inyección de prompts

4. Auditoría:

- Registro completo de todas las operaciones
- Trazabilidad por usuario y timestamp
- Análisis de patrones de uso

Esta arquitectura proporciona una integración robusta, escalable y eficiente con AWS Bedrock, permitiendo a Irakani Builder ofrecer capacidades de IA generativa de clase empresarial mientras mantiene control sobre costos, seguridad y experiencia de usuario.

4.3.5. Diseño de la Interfaz de Usuario (UI/UX)

Irakani Builder implementa una interfaz de usuario moderna y profesional basada en React 18 con TypeScript, diseñada para maximizar la productividad del desarrollador mediante una arquitectura de paneles modulares y redimensionables. La UI sigue principios de diseño centrado en el usuario, con énfasis en la eficiencia del flujo de trabajo y la accesibilidad de las herramientas de IA.

A) Arquitectura de Componentes y Flujo de Navegación:

El sistema implementa un flujo de autenticación y navegación multi-nivel que guía al usuario desde el login hasta el entorno de desarrollo completo:

1. Flujo de Autenticación (AuthComponent):

La Figura 5 muestra la interfaz de autenticación implementada con AWS Cognito, que incluye:

- Login/Registro con validación de credenciales
- Verificación de código 2FA para seguridad adicional
- Recuperación de contraseña mediante email
- Gestión de sesiones persistentes con Valkey

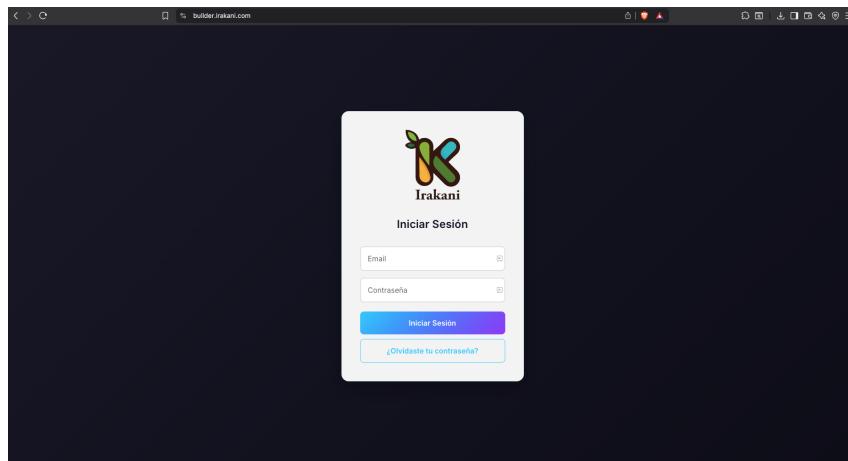


Figura 5: Interfaz de autenticación de Irakani Builder

2. Selección de Modo de Acceso (AccessSelection):

Después de autenticarse, el usuario puede elegir entre dos modos de trabajo, como se muestra en la Figura 6:

- **Builder Mode:** Acceso directo al constructor de aplicaciones sin dependencias externas
- **Irakani Spaces:** Integración con espacios de trabajo de la plataforma Irakani para sincronización de datos

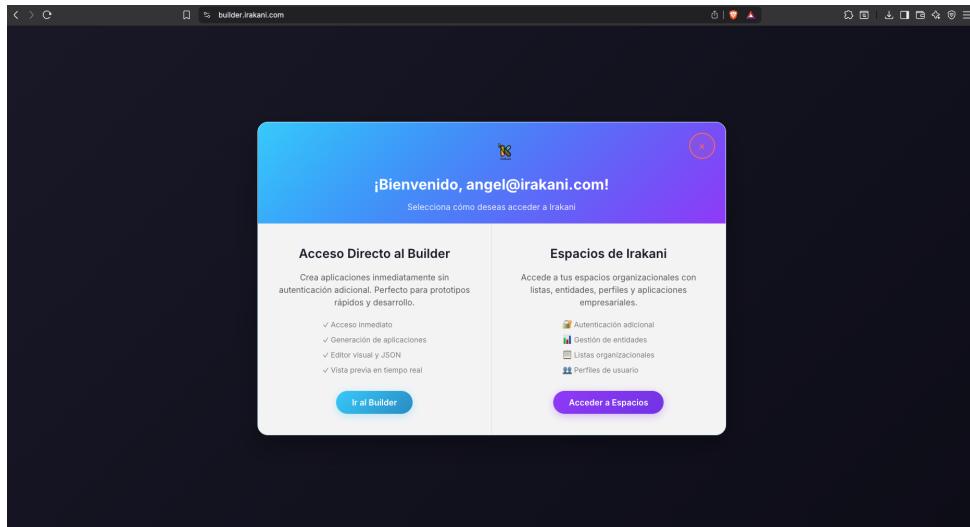


Figura 6: Selección de modo de acceso al Builder

3. Selección de Espacio (SpaceSelection):

Para usuarios que eligen el modo Irakani Spaces, la Figura 7 muestra la interfaz de selección de espacios de trabajo:

- Lista de espacios disponibles del usuario
- Información de cada espacio (nombre, descripción, última modificación)
- Selección de espacio de trabajo activo
- Creación de nuevos espacios

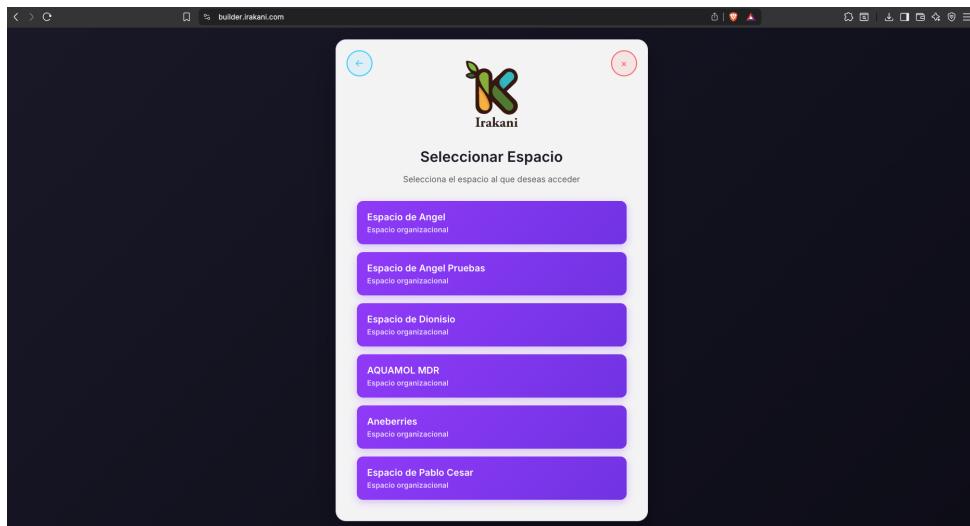


Figura 7: Selección de espacio de trabajo en Irakani

4. Menú de Navegación de Espacios:

Una vez seleccionado un espacio de trabajo, el usuario tiene acceso a un menú de navegación que organiza las diferentes secciones del espacio Irakani. Este menú proporciona acceso rápido a las funcionalidades principales del espacio:

a) Menú de Aplicaciones:

La Figura 8 muestra el menú de aplicaciones disponibles en el espacio:

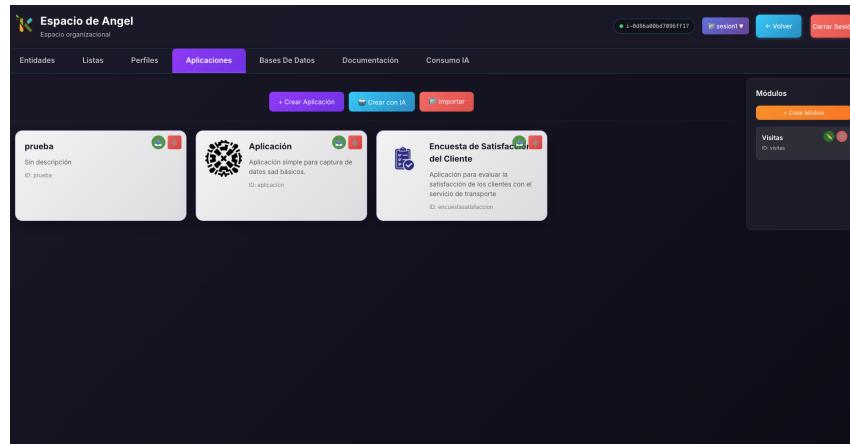


Figura 8: Menú de aplicaciones del espacio Irakani

b) Menú de Base de Datos:

La Figura 9 muestra el menú de gestión de bases de datos:

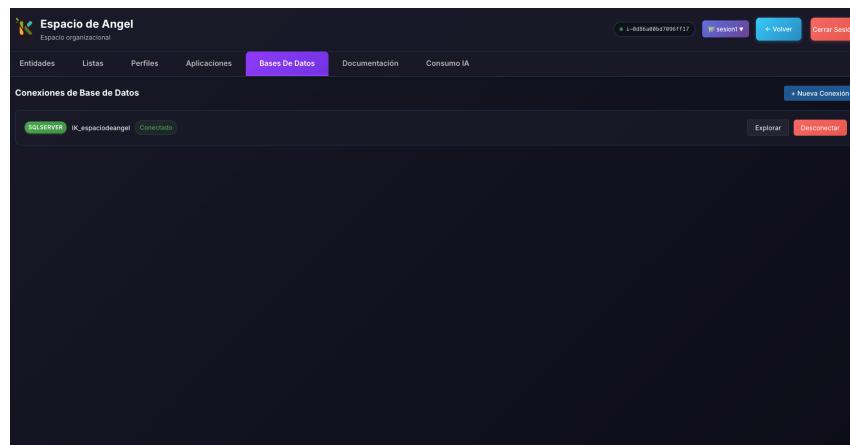


Figura 9: Menú de base de datos del espacio Irakani

c) Menú de Listas:

La Figura 10 muestra el menú de gestión de listas de datos:

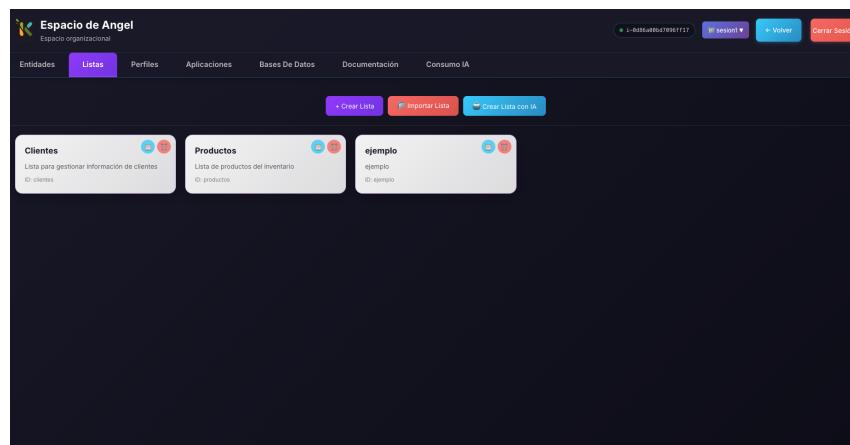


Figura 10: Menú de listas del espacio Irakani

d) Menú de Perfiles:

La Figura 11 muestra el menú de gestión de perfiles de usuario:

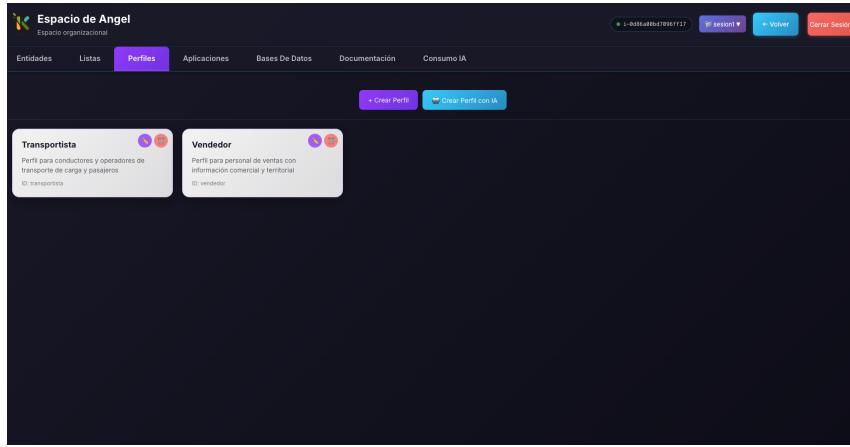


Figura 11: Menú de perfiles del espacio Irakani

e) Menú de Entidades:

La Figura 12 muestra el menú de gestión de entidades de datos:

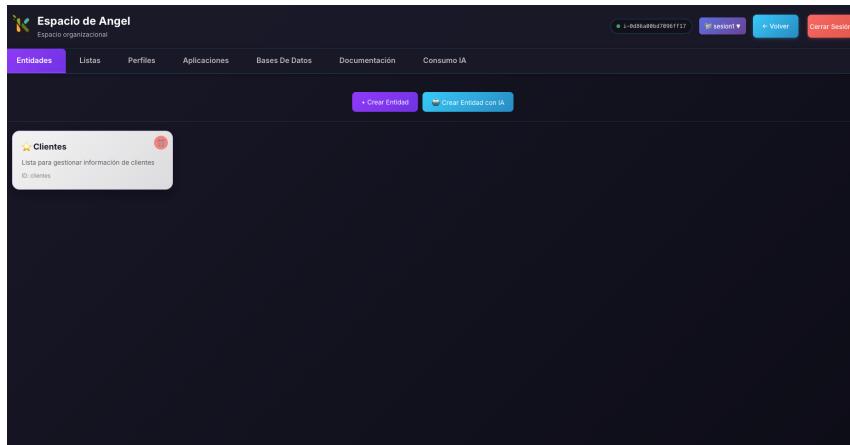


Figura 12: Menú de entidades del espacio Irakani

Estos menús permiten al usuario navegar entre las diferentes secciones del espacio de trabajo, facilitando el acceso a las funcionalidades de gestión de aplicaciones, bases de datos, listas, perfiles y entidades. La integración con el espacio Irakani permite sincronizar automáticamente los datos entre el Builder y la plataforma principal.

5. Gestión de Sesiones:

El sistema permite trabajar con múltiples sesiones simultáneas, como se observa en la Figura 13. Cada sesión representa un proyecto o aplicación independiente:

- Creación de nuevas sesiones de trabajo
- Cambio entre sesiones existentes
- Eliminación de sesiones con confirmación
- Aislamiento completo de datos por sesión
- Persistencia automática en Valkey

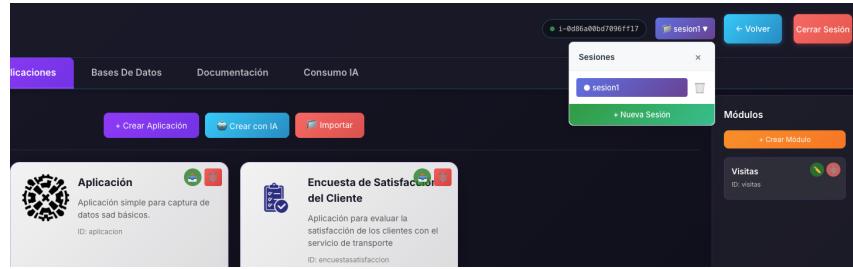


Figura 13: Gestión de sesiones de trabajo

B) Diseño del Builder Principal (Layout de 3 Paneles): Tab Código La interfaz del Builder se organiza en una arquitectura de tres paneles principales redimensionables, optimizada para el flujo de trabajo del desarrollador. La Figura 14 muestra la vista completa del Builder:

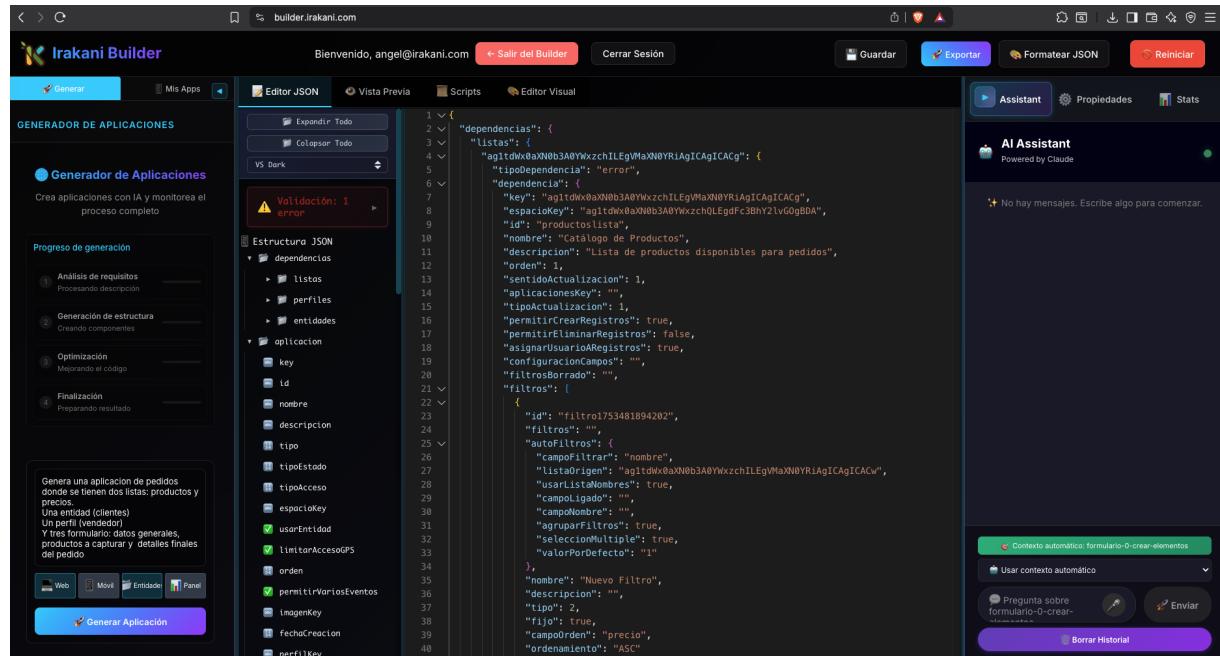


Figura 14: Vista completa del Builder con arquitectura de 3 paneles

C) Panel Izquierdo (Left Panel) - 350px (redimensionable 300-450px):

El panel izquierdo proporciona acceso a las funcionalidades de generación y gestión de aplicaciones, como se muestra en la Figura 15:



Figura 15: Panel izquierdo con generador de aplicaciones

1. Tab Generador (AppGenerator):

Componentes principales:

- Campo de texto multi-línea para descripción de la aplicación en lenguaje natural
- Selector de tipo de aplicación (WEB/MÓVIL)
- Botón "Generar Aplicación" con indicador de carga
- Visualización de estado del workflow (Step Functions)
- Indicadores de progreso de generación en tiempo real

Funcionalidad:

- Generación de aplicaciones mediante IA (AWS Bedrock)
- Integración con Step Functions para procesamiento asíncrono
- Guardado automático del prompt en Valkey
- Validación de entrada antes de enviar a la IA

La Figura 16 ilustra el flujo completo de generación de aplicaciones:



Figura 16: Flujo de generación de aplicaciones con IA

2. Características del Panel Izquierdo:

- Colapsible con botón toggle para maximizar espacio de trabajo
- Redimensionable con drag handle entre 300-450px
- Zoom con Ctrl+Scroll para ajustar tamaño de fuente
- Persistencia de estado (tamaño, tab activo) en Valkey

D) Panel Central (Main Editor) - Ancho flexible:

El panel central es el área principal de trabajo, dividida en dos secciones verticales:

1. Tree View (JsonTreeView) - 250px (redimensionable 200-500px):

La Figura 17 muestra el árbol jerárquico de navegación de la estructura JSON:

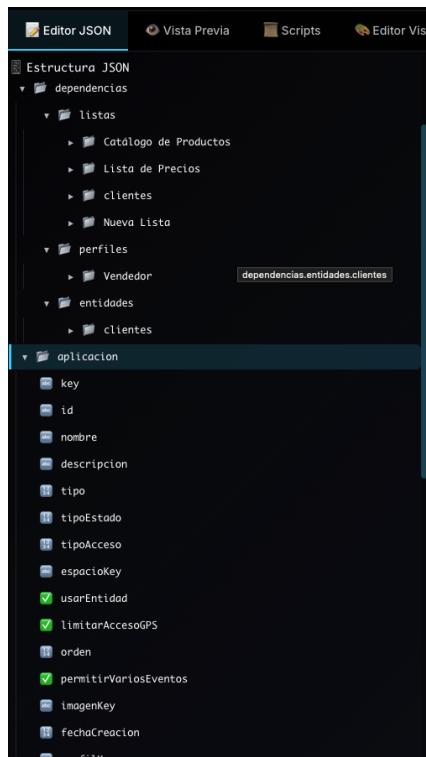


Figura 17: Tree View para navegación de estructura JSON

Componentes del Tree View:

- Árbol jerárquico del JSON de la aplicación
- Iconos diferenciados por tipo de nodo (aplicación, forma, elemento, lista, perfil, entidad)
- Indicadores de expansión/collapse para navegación eficiente
- Resaltado del elemento seleccionado
- Badges con contadores de elementos hijos

Funcionalidad:

- Navegación por la estructura completa de la aplicación
- Selección de elementos para edición en el panel derecho
- Expansión automática de nodos padres al seleccionar un hijo
- Búsqueda de nodos por nombre o tipo
- Sincronización bidireccional con el editor de código

Estructura visualizada en el Tree View:

```
[APP] Aplicacion
+-- [FORM] Formas
    +-- [FORM] Forma 1
        +-- [INPUT] Elemento 1 (input)
        +-- [SELECT] Elemento 2 (select)
        +-- [FORM] Subforma
    +-- [FORM] Forma 2
+-- [DEPS] Dependencias
    +-- [LIST] Listas
        +-- Lista 1
        +-- Lista 2
    +-- [USER] Perfiles
    +-- [DB] Entidades
    +-- [FLOW] Workflows
+-- [GEAR] Configuracion
```

2. Tabs de Edición:

a) Tab Visual (EditorVisual):

La Figura 18 muestra el editor visual para diseño de formularios:

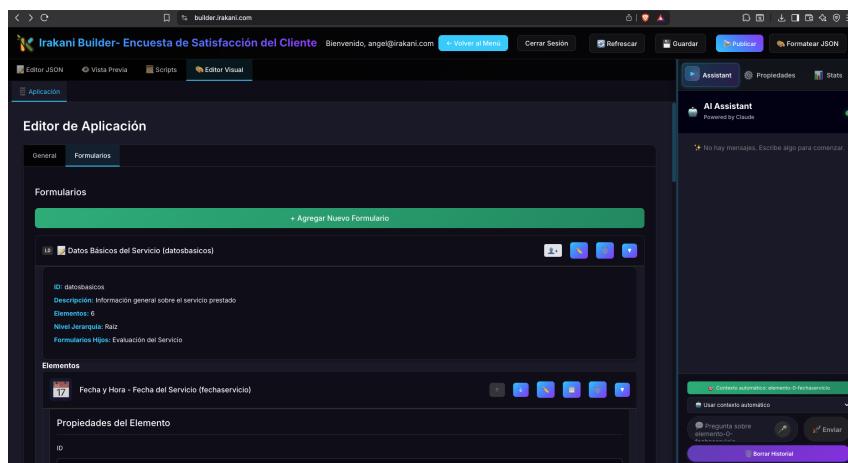


Figura 18: Editor visual de formularios

Características del editor visual:

- Selector de tipo de elemento (input, select, button, checkbox, radio, textarea, etc.)
- Configuración de propiedades generales (id, nombre, etiqueta, placeholder)
- Propiedades específicas por tipo de elemento
- Validaciones configurables (requerido, longitud, formato)
- Preview en tiempo real del formulario

Sub-editores especializados:

- **AplicacionEditor:** Configuración de propiedades globales de la aplicación
- **ListasEditor:** Gestión de listas de datos y campos
- **PerfilesEditor:** Configuración de perfiles de usuario y permisos
- **EntidadesEditor:** Diseño de entidades de datos y relaciones
- **WorkflowsEditor:** Diseño de flujos de trabajo y procesos

b) Tab Código (Monaco Editor):

La Figura 19 muestra la integración del Monaco Editor para edición de código:

```

Template: Validar Fin de Jornada - Al cargar
Temporario (puedes borrarlo, _1803409373)
1 //EsteEditorAlCargar()
2 if(!estaEditorAlCargar()){
3     notificar("No es posible editar el Fin de Jornada", function(){
4         salir();
5     });
6     return -1;
7 }
8 var fecha = new Date().getTime();
9 var obtenerMilisegundosFechaSinHora = function(millis) {
10     var date = new Date(parseInt(millis, 10));
11     return new Date(date.getFullYear(), date.getMonth(), date.getDate()).getTime();
12 };
13 fecha = obtenerMilisegundosFechaSinHora(fecha);
14
15 //Valida que se capture un evento de Inicio de Jornada
16 global.validaInicio = false;
17 //Valida que no se capture mas si ya realizo el Fin de Jornada
18 //global.validaFin = false;
19
20 if(estaEditorAlCargar()){
21     obtenerEventos("findeJornada", function(eventos){
22         var esEventoDeDia = function(evento) {
23             var fechaCreacionMillis = obtenerMilisegundosFechaSinHora(evento.fechaCreacionInicial);
24             var fechaReconexionMillis = obtenerMilisegundosFechaSinHora(evento.fechaReconexion);
25             return fechaReconexionMillis == fecha;
26         };
27         eventos.forEach(function (evento) {
28             if(esEventoDeDia(evento)) {
29                 if(esEventoDeDia(evento)){
30                     global.validaFin = true;
31                 }
32             }
33         });
34     });
35 }

```

Figura 19: Monaco Editor integrado con resaltado de sintaxis

Características del Monaco Editor:

- Editor de código JSON con sintaxis highlighting avanzado
- Autocompletado inteligente basado en el esquema
- Validación en tiempo real con indicadores de errores
- Más de 20 temas personalizables (Monokai, Dracula, GitHub, Nord, etc.)
- Formateo automático con Prettier
- Búsqueda y reemplazo con expresiones regulares
- Minimap de navegación para archivos grandes
- Plegado de código (code folding)

- Múltiples cursos y selección rectangular

La Figura 20 muestra el selector de temas disponibles:

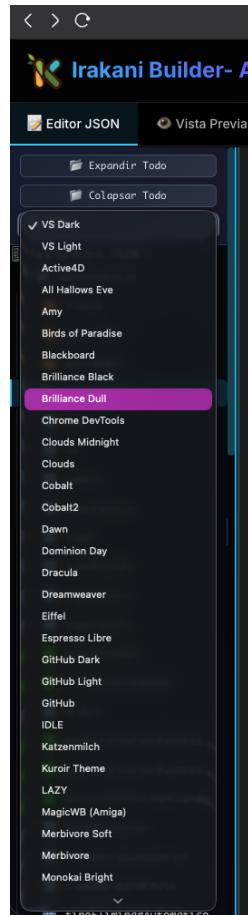


Figura 20: Selector de temas del editor de código

Temas disponibles incluyen:

- Temas claros: vs-light, github-light, solarized-light
- Temas oscuros: vs-dark, monokai, dracula, github-dark, one-dark, nord, cobalt, tomorrow-night
- Temas de alto contraste: hc-black, hc-light
- Temas especializados: material-theme, ayu-dark, ayu-light, y más

c) Tab Chat (ChatInterface):

La Figura 21 muestra la interfaz de chat con el asistente de IA:

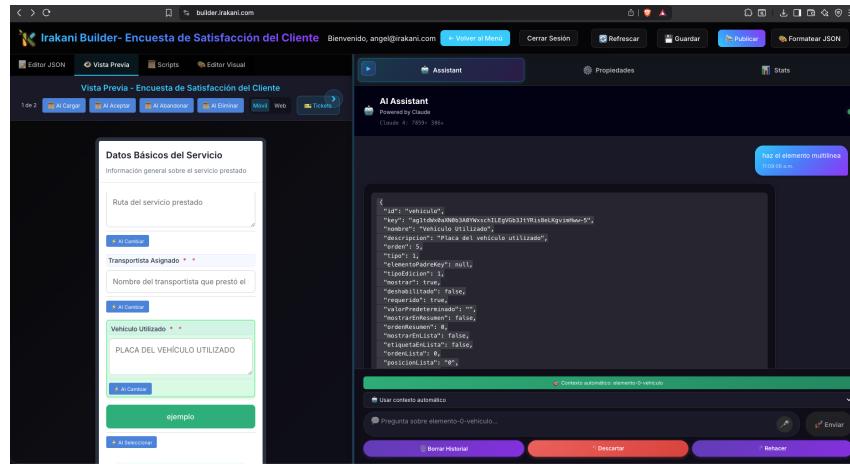


Figura 21: Interfaz de chat con asistente de IA

Características del chat con IA:

- Interfaz de conversación con historial persistente
- Selector de contexto (aplicación completa, listas, entidades, elemento seleccionado)
- Selector de templates de prompts predefinidos
- Streaming de respuestas en tiempo real
- Aplicación automática de cambios sugeridos al JSON
- Indicadores de tokens consumidos y costos estimados
- Soporte para código en las respuestas con syntax highlighting
- Botones de acción rápida (Aplicar, Copiar, Regenerar)

Funcionalidades del Chat:

- Generación de código para nuevos componentes
- Explicación de estructuras existentes
- Sugerencias de mejora y optimización
- Corrección de errores de sintaxis o lógica
- Optimización de consultas y validaciones
- Traducción entre formatos de datos

d) Tab Preview (FormPreview):

La Figura 22 muestra la vista previa de formularios:

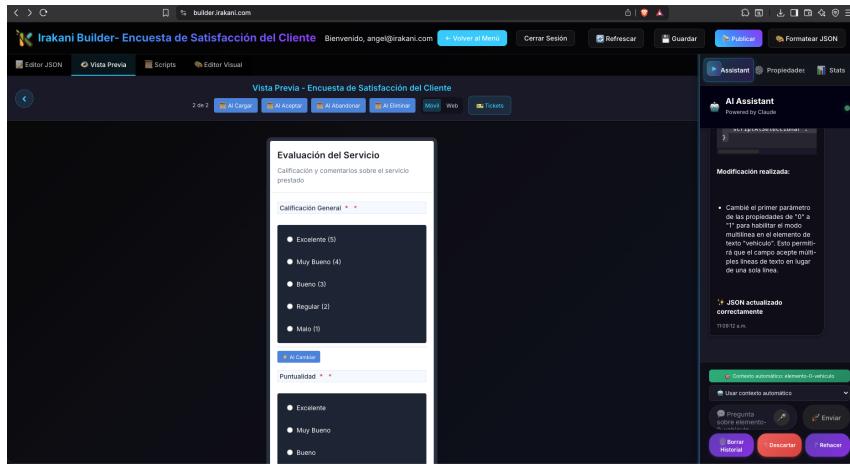


Figura 22: Vista previa de formularios renderizados

Características del Preview:

- Renderizado en tiempo real de formularios
- Simulación de interacciones de usuario
- Validación de campos según reglas definidas
- Visualización de mensajes de error
- Diseño resposivo (web, móvil)
- Modo de prueba interactivo

3. Características del Panel Central:

- Tabs dinámicos según contexto del elemento seleccionado
- Indicador visual de guardado en progreso
- Historial de cambios con undo/redo
- Redimensionamiento del Tree View con drag handle

E) Panel Derecho (Right Panel) - 400px (redimensionable 350-600px):

El panel derecho proporciona acceso contextual a propiedades, asistencia de IA y herramientas de análisis.

1. Tab Propiedades (PropertiesPanel):

La Figura 23 muestra el panel de propiedades contextual:

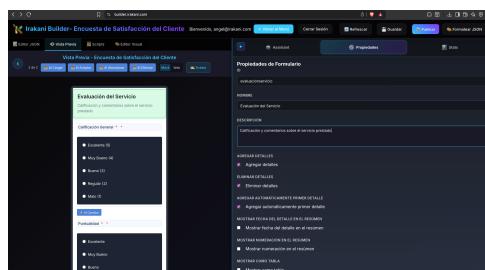


Figura 23: Panel de propiedades contextual

Paneles especializados según elemento seleccionado:

a) AplicacionPropertiesPanel:

- Nombre de la aplicación

- Descripción detallada

b) ElementosPropertiesPanel:

- Propiedades generales (id, nombre, etiqueta, placeholder)
- Propiedades específicas por tipo (opciones de select, formato de input, etc.)
- Validaciones (requerido, longitud mínima/máxima, patrón regex)
- Eventos y acciones (onChange, onClick, onBlur)

c) ListasPropertiesPanel:

- Nombre de la lista
- Campos de la lista con tipos de datos
- Validaciones por campo
- Valores por defecto
- Integración con API Irakani
- Configuración de sincronización

d) EntidadesPropertiesPanel:

- Nombre de la entidad
- Campos de la entidad
- Índices y claves primarias/foráneas
- Permisos de acceso por perfil

2. Tab Estadísticas (JSONStatsPanel):

La Figura 24 muestra el panel de estadísticas y métricas:

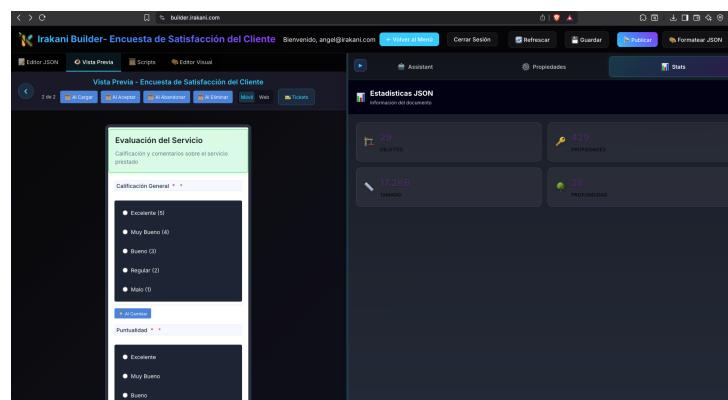


Figura 24: Panel de estadísticas y métricas de uso

Métricas mostradas:

- Tamaño del JSON en bytes y KB
- Número total de elementos por tipo
- Complejidad de la aplicación

- Tiempo de generación promedio

3. Características del Panel Derecho:

- Contenido contextual según elemento seleccionado
- Redimensionable entre 350-600px
- Colapsable para maximizar espacio de edición
- Persistencia de estado en Valkey

F) Componentes Especializados:

1. DB Admin (DBAdmin):

La Figura 25 muestra el administrador de bases de datos integrado:

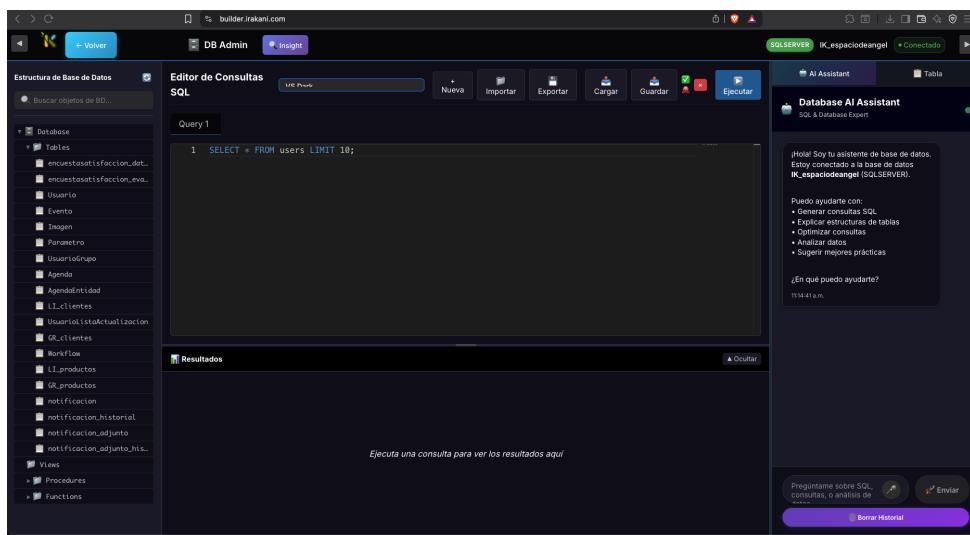


Figura 25: Administrador de bases de datos con asistente de IA

Características del DB Admin:

- Gestión de conexiones a múltiples bases de datos (MySQL, SQL Server)
- Explorador de tablas y esquemas con navegación jerárquica
- Editor de consultas SQL con autocompletado
- Visualización de resultados en tabla con paginación
- Asistente de IA para generación de queries desde lenguaje natural
- Historial de consultas ejecutadas
- Análisis de planes de ejecución

2. Generador de Iconos (Icon Generator):

La Figura 26 muestra el generador de iconos con IA:

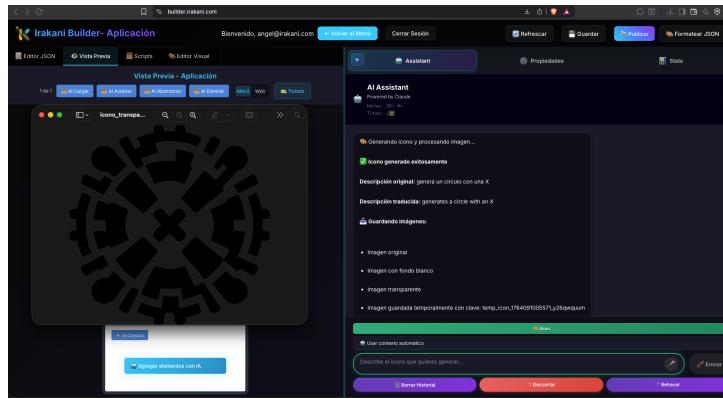


Figura 26: Generador de iconos con Titan Image Generator

Características del generador de iconos:

- Prompt en español con traducción automática al inglés
- Generación con Amazon Titan Image Generator
- Personalización de colores con selector de paleta
- Almacenamiento automático en S3
- Historial de iconos generados
- Ajuste de tamaño y resolución

3. Sistema de Notificaciones (NotificationSystem):

La Figura 27 muestra el sistema de notificaciones toast:

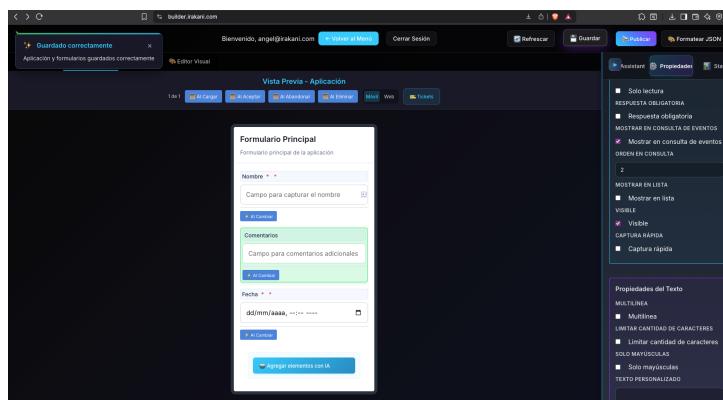


Figura 27: Sistema de notificaciones no intrusivas

Características de las notificaciones:

- Notificaciones toast no intrusivas en esquina superior derecha
- Tipos diferenciados: success, error, warning, info, loading
- Confirmaciones con callbacks para acciones críticas
- Auto-dismiss configurable (2-10 segundos)
- Stack de notificaciones con límite de 5 simultáneas
- Animaciones suaves de entrada/salida

- Iconos contextuales por tipo
- Botón de cierre manual

G) Características de Experiencia de Usuario:

1. Paneles Redimensionables:

La Figura 28 muestra los paneles que pueden ajustarse de tamaño:

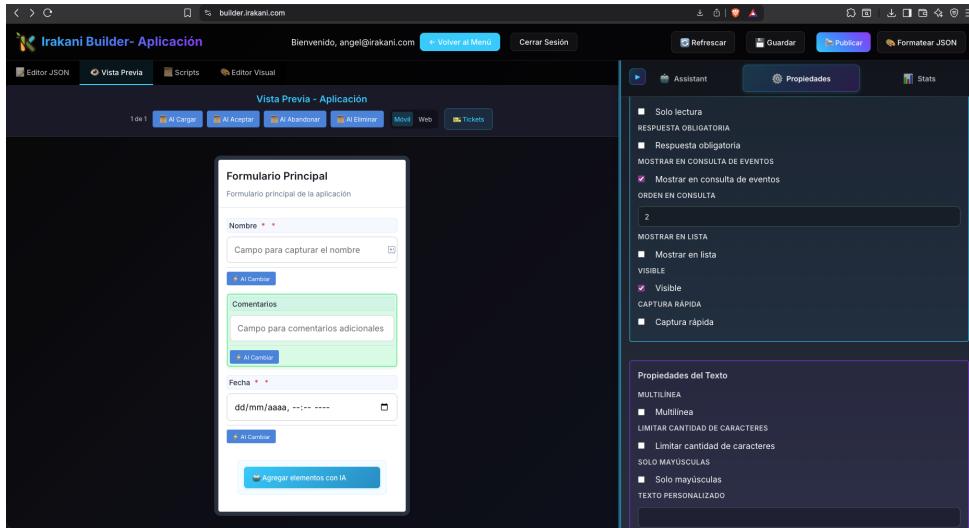


Figura 28: Paneles con tamaño ajustable

Los paneles pueden redimensionarse arrastrando sus bordes, y el tamaño se guarda automáticamente para cada usuario.

2. Guardado Automático:

El sistema guarda los cambios automáticamente cada medio segundo. Se muestra un indicador visual del estado (guardando, guardado o error) y se previene la pérdida de datos al cerrar la aplicación.

3. Navegación:

- El elemento seleccionado se resalta en todos los paneles
- Atajos de teclado: Ctrl+Z (deshacer), Ctrl+Y (rehacer), Ctrl+F (buscar)

4. Indicadores Visuales:

- Barras de progreso durante la carga
- Tooltips informativos al pasar el cursor
- Colores diferentes para cada estado (activo, deshabilitado, etc.)

5. Diseño Adaptable:

La interfaz se adapta a diferentes tamaños de pantalla, colapsando paneles automáticamente en pantallas pequeñas para mantener la usabilidad.

6. Accesibilidad:

Se puede navegar completamente usando el teclado (Tab, Enter, Escape) y la interfaz es compatible con lectores de pantalla.

H) Flujos de Trabajo Principales:

Los flujos de trabajo principales del sistema se documentan en detalle en la subsección 4.3.6 “Diagramas de Flujo y Casos de Uso”, donde se presentan tanto los diagramas de flujo como los casos de uso detallados que describen las interacciones del sistema.

I) Principios de Diseño Aplicados:

La interfaz de Irakani Builder se diseñó siguiendo principios fundamentales de usabilidad y experiencia de usuario, implementados de manera práctica en cada componente del sistema:

1. Consistencia Visual: Se estableció un sistema de diseño unificado utilizando una paleta de colores coherente con efectos glassmorphism, tipografía Inter para la interfaz y JetBrains Mono para el código, y un espaciado sistemático basado en múltiplos de 8px. Esto garantiza que todos los elementos visuales mantengan una apariencia profesional y predecible.

2. Jerarquía de Información: Los elementos se organizaron por importancia visual, destacando las funciones principales con mayor tamaño y contraste. Se implementaron paneles colapsables y agrupación lógica de funciones relacionadas para optimizar el espacio disponible y facilitar la navegación.

3. Eficiencia del Flujo de Trabajo: Se minimizó el número de clics necesarios para tareas comunes mediante acciones contextuales y navegación por pestañas (Visual, Código, Chat, Scripts). Las operaciones asíncronas incluyen indicadores de progreso claros para mantener al usuario informado.

4. Prevención de Errores: Se implementó validación en tiempo real de campos de entrada, confirmaciones para acciones destructivas, y mensajes de error descriptivos con sugerencias de solución. El editor Monaco valida la sintaxis JSON automáticamente, y el sistema maneja tokens expirados con redirección automática al login.

5. Retroalimentación Inmediata: Cada interacción del usuario recibe respuesta visual instantánea mediante un sistema de notificaciones toast diferenciado (éxito, error, advertencia, información, carga) y animaciones suaves con Framer Motion. Los estados de carga se diferencian según la operación (guardando, exportando, generando).

J) Objetivos de Métricas de Usabilidad:

- **Tiempo de aprendizaje:** Objetivo de que usuarios nuevos puedan generar su primera aplicación en menos de 5 minutos
- **Eficiencia:** Meta de reducción del 70% en tiempo de desarrollo comparado con codificación manual
- **Tasa de error:** Objetivo de menos del 5% de operaciones resulten en errores gracias a validación en tiempo real
- **Satisfacción:** Meta de score promedio de 4.5/5 en encuestas de usabilidad
- **Retención:** Objetivo de 90% de usuarios continúen usando la plataforma después de la primera semana

Esta arquitectura de UI/UX proporciona una experiencia de desarrollo profesional, intuitiva y eficiente, aprovechando las capacidades de IA para acelerar el proceso de construcción de aplicaciones mientras mantiene el control total del desarrollador sobre el resultado final.

4.3.6. Diagramas de Flujo y Casos de Uso

A continuación se presentan los diagramas de flujo de trabajo principales del sistema, seguidos de casos de uso detallados que describen las interacciones críticas entre el usuario, la interfaz y el motor de IA.

Caso de Uso 1: Generación de Componentes con IA

Actor: Desarrollador

Descripción: El usuario solicita la creación de un elemento de interfaz específico mediante lenguaje natural, aprovechando las capacidades de AWS Bedrock para generar automáticamente la estructura JSON del componente.

Flujo Principal:

1. El usuario selecciona un contenedor existente en el Editor Visual donde desea agregar el nuevo componente
2. Ingrera una descripción en lenguaje natural en el chat de IA (ejemplo: “Botón con icono de impresión”)
3. El sistema captura el contexto actual del contenedor seleccionado, incluyendo:
 - Estructura JSON actual del formulario
 - Componentes hermanos existentes
 - Propiedades del contenedor padre
4. El sistema construye un prompt optimizado combinando:
 - La descripción del usuario
 - El contexto del contenedor
 - Restricciones de validación

- Ejemplos de componentes similares (Few-shot learning)

5. El sistema envía la solicitud a AWS Bedrock utilizando el modelo Claude 4 Sonnet
6. AWS Bedrock procesa el prompt y genera la estructura JSON del componente solicitado
7. El sistema valida el JSON generado contra el esquema definido
8. Si la validación es exitosa, el componente se inserta automáticamente en el contenedor seleccionado
9. El Editor Visual renderiza el nuevo componente en tiempo real
10. El sistema sincroniza la vista de código (Monaco Editor) con la estructura actualizada
11. Se muestra una notificación de éxito al usuario
12. El sistema guarda automáticamente el estado en Valkey (caché) y programa el guardado persistente en S3

Flujos Alternativos:

- **FA1 - Error de Bedrock:** Si AWS Bedrock no responde o devuelve un error, el sistema muestra un mensaje descriptivo y sugiere reformular la solicitud
- **FA2 - Timeout:** Si la generación excede 30 segundos, se cancela la operación y se notifica al usuario

Postcondiciones:

- El componente generado está visible en el Editor Visual
- El JSON actualizado está sincronizado en todas las vistas
- El estado se ha guardado en Valkey para recuperación rápida

La Figura 16 ilustra visualmente este proceso de generación asistida por IA, mientras que la Figura 29 muestra el diagrama de secuencia detallado de las interacciones entre componentes.

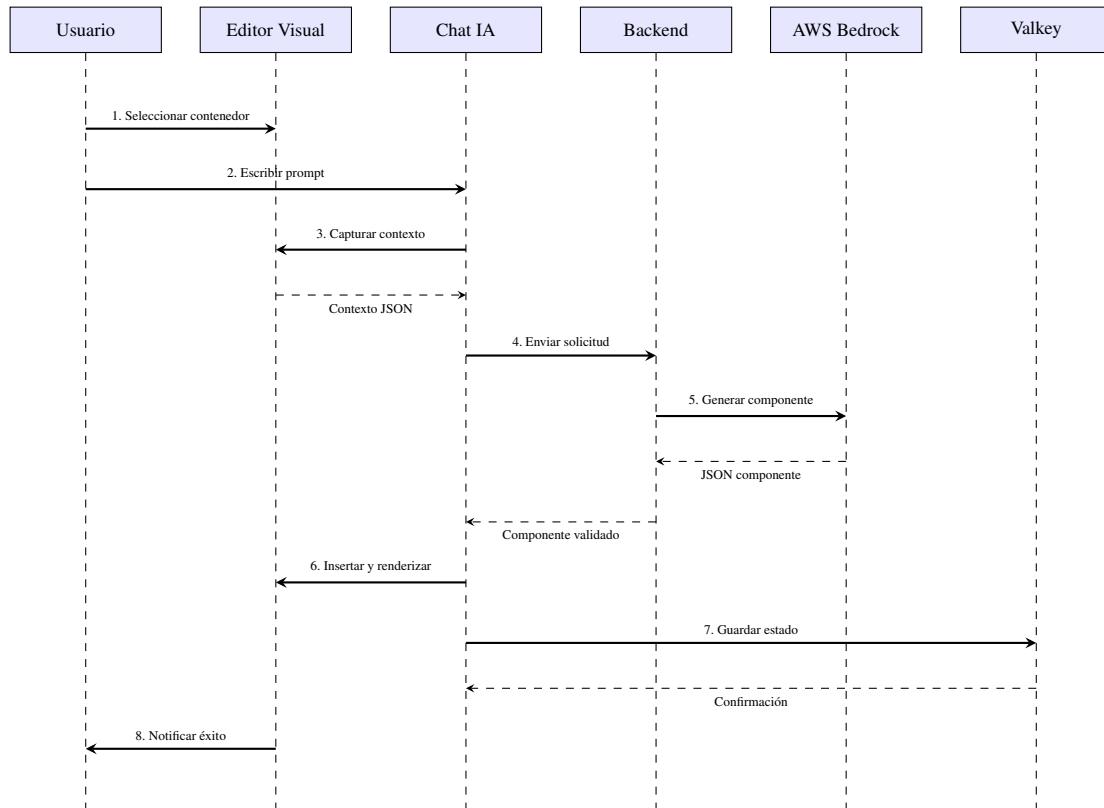


Figura 29: Diagrama de secuencia: Generación de Componentes con IA

Caso de Uso 2: Conexión a Base de Datos Externa

Actor: Desarrollador

Descripción: El usuario establece una conexión segura a una base de datos externa (MySQL o SQL Server) para integrar datos reales en las aplicaciones generadas, permitiendo operaciones de consulta y manipulación de datos.

Precondiciones:

- El usuario tiene credenciales válidas de acceso a la base de datos
- La base de datos es accesible desde la red donde se ejecuta Irakani Builder
- El motor de base de datos es MySQL 5.7+ o SQL Server 2016+

Flujo Principal:

1. El usuario accede al panel “DB Admin” desde el menú de la aplicación
2. El sistema presenta el formulario de configuración de conexión con los siguientes campos:
 - Motor de base de datos (MySQL / SQL Server)
 - Host (dirección IP o nombre de dominio)
 - Puerto ([PUERTO_{CENSURADO}] para MySQL, [PUERTO_{CENSURADO}] para SQL Server por defecto)
 - Nombre de la base de datos
 - Usuario
 - Contraseña (campo enmascarado)
3. El usuario selecciona el motor de base de datos apropiado del dropdown
4. Ingrera las credenciales de conexión en los campos correspondientes
5. Hace clic en el botón “Conectar”
6. El sistema valida que todos los campos requeridos estén completos
7. El backend establece una conexión de prueba utilizando el pool de conexiones configurado
8. Si la conexión es exitosa:
 - Almacena el token de conexión encriptado en la sesión del usuario
 - Consulta el esquema de la base de datos para obtener la lista de tablas
 - Recupera los metadatos de cada tabla (columnas, tipos de datos, claves primarias)
9. El sistema muestra la lista de tablas disponibles en el panel lateral izquierdo
10. Para cada tabla, se presenta:
 - Nombre de la tabla
 - Número de columnas
 - Icono indicador del tipo de tabla
 - Opción de expandir para ver columnas
11. El usuario puede hacer clic en una tabla para:
 - Ver su estructura completa
 - Ejecutar consultas SQL
 - Utilizar el asistente de IA para generar queries en lenguaje natural
12. La conexión permanece activa durante la sesión del usuario

13. El sistema mantiene un heartbeat cada 5 minutos para verificar la conexión

Flujos Alternativos:

- **FA1 - Credenciales Inválidas:** Si las credenciales son incorrectas, el sistema muestra un mensaje de error específico (“Usuario o contraseña incorrectos”) y permite reintentar
- **FA2 - Host Inaccesible:** Si no se puede alcanzar el host, se muestra un error de timeout con sugerencias de verificación de red
- **FA3 - Permisos Insuficientes:** Si el usuario de base de datos no tiene permisos para listar tablas, se muestra un mensaje indicando la necesidad de permisos SELECT en INFORMATION_SCHEMA
- **FA4 - Conexión Perdida:** Si la conexión se pierde durante la sesión, el sistema detecta el fallo en el siguiente heartbeat y solicita reconexión

Postcondiciones:

- La conexión a la base de datos está establecida y activa
- La lista de tablas está disponible para consulta
- El usuario puede ejecutar operaciones SQL sobre la base de datos conectada

Consideraciones de Seguridad:

- Las conexiones utilizan SSL/TLS cuando está disponible
- Los tokens de sesión expiran después de 4 horas de inactividad
- Se implementa rate limiting para prevenir ataques de fuerza bruta
- Todas las queries se ejecutan con prepared statements para prevenir SQL injection

La Figura 25 muestra la interfaz del administrador de bases de datos con una conexión activa y la lista de tablas disponibles, mientras que la Figura 30 presenta el diagrama de secuencia del proceso de conexión.

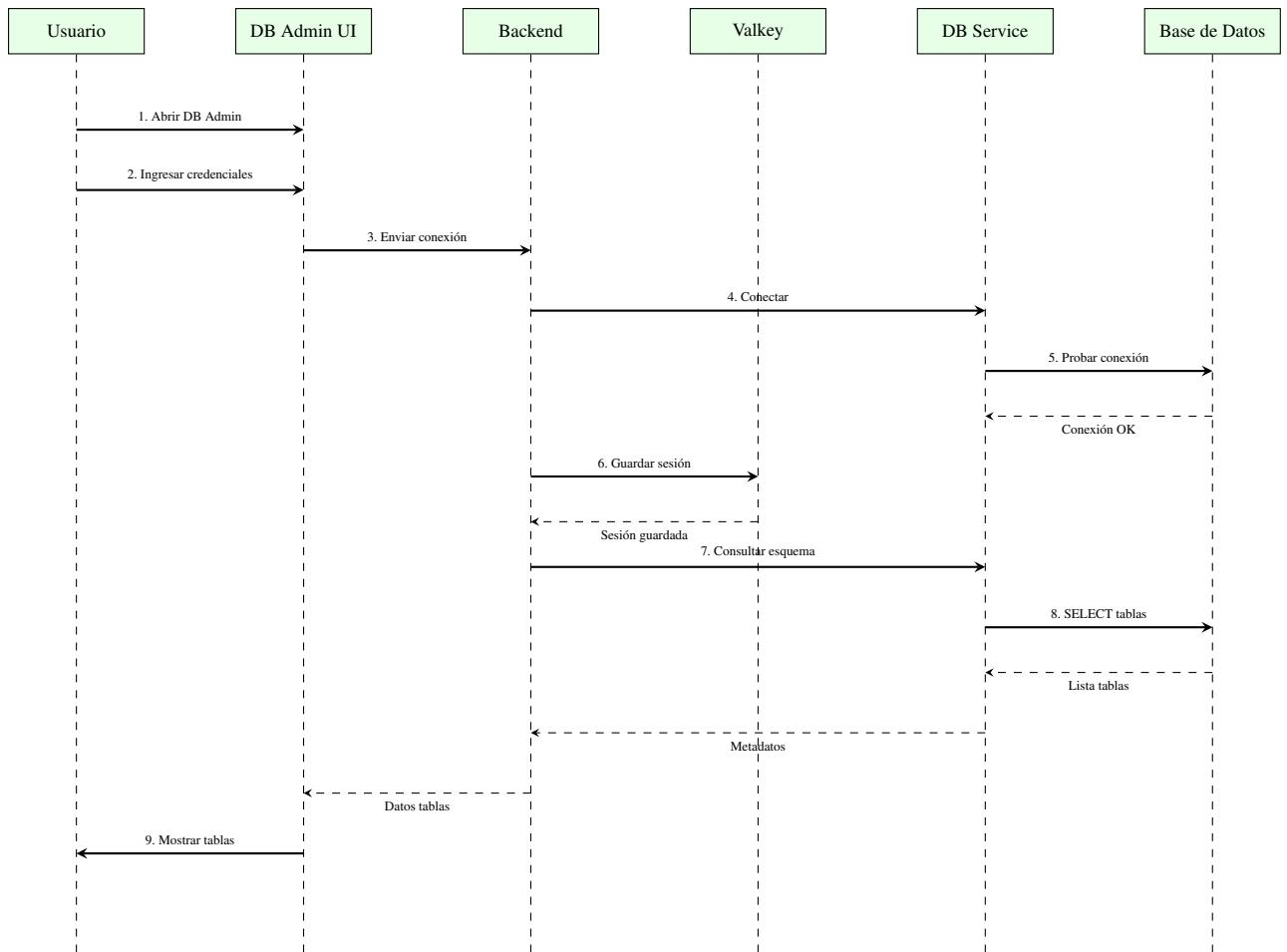


Figura 30: Diagrama de secuencia: Conexión a Base de Datos Externa

Caso de Uso 3: Edición Visual de Componentes

Actor: Desarrollador

Descripción: El usuario modifica las propiedades de un componente existente utilizando el panel de propiedades contextual, permitiendo ajustes precisos sin necesidad de editar código manualmente.

Flujo Principal:

- El usuario hace clic sobre un componente en el Editor Visual
- El sistema resalta el componente seleccionado con un borde distintivo
- El panel de propiedades se actualiza automáticamente mostrando las propiedades editables del componente:
 - Propiedades básicas (label, placeholder, valor por defecto)
 - Propiedades de validación (requerido, patrón, longitud)
 - Propiedades de comportamiento (deshabilitado, solo lectura)
- El usuario modifica una o más propiedades utilizando los controles apropiados (inputs, selects, color pickers, toggles)
- Cada cambio se refleja instantáneamente en la vista previa del componente
- El sistema valida los valores ingresados en tiempo real
- Si un valor es inválido, se muestra un indicador visual y un mensaje de error
- Una vez que el usuario termina de editar, el sistema espera 500ms de inactividad (debounce)

9. El sistema actualiza el JSON subyacente con los nuevos valores
10. Se sincroniza la vista de código (Monaco Editor) con los cambios
11. El estado actualizado se guarda automáticamente en Valkey
12. Se muestra una notificación sutil indicando “Cambios guardados”

Postcondiciones:

- Las propiedades del componente están actualizadas
- La vista previa refleja los cambios realizados
- El JSON está sincronizado en todas las vistas
- Los cambios están persistidos en caché

La Figura 31 ilustra el diagrama de secuencia de la edición visual de componentes con actualización en tiempo real.

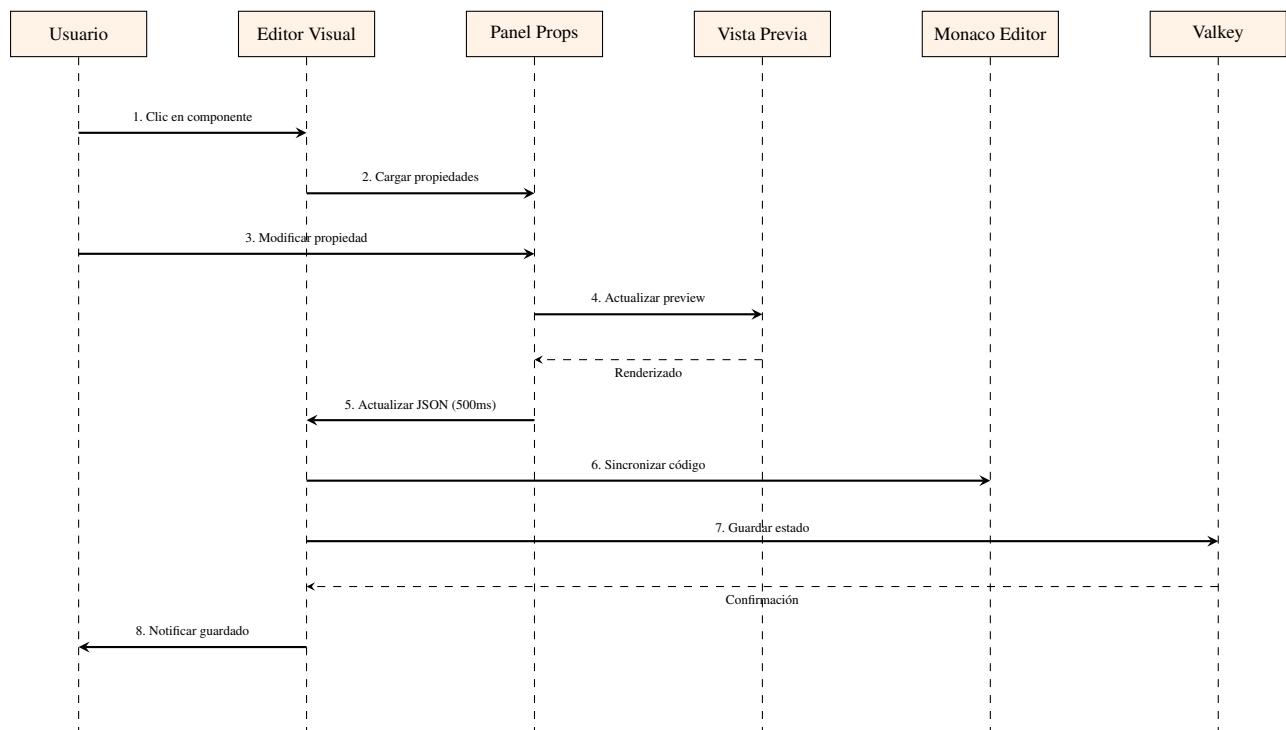


Figura 31: Diagrama de secuencia: Edición Visual de Componentes

Caso de Uso 4: Asistencia Contextual con Chat de IA

Actor: Desarrollador

Descripción: El usuario interactúa con el asistente de IA para obtener ayuda, sugerencias o realizar modificaciones complejas mediante lenguaje natural.

Flujo Principal:

1. El usuario selecciona un componente o sección de la aplicación
2. Abre la pestaña “Chat” en el panel derecho
3. Escribe una pregunta o solicitud en lenguaje natural (ejemplo: “¿Cómo puedo validar que este campo sea un email válido?”)
4. El sistema captura el contexto actual:

- Componente seleccionado y sus propiedades
- Estructura completa del formulario
- Historial de conversación de la sesión

5. El sistema envía el mensaje junto con el contexto a AWS Bedrock
6. La IA analiza la solicitud y genera una respuesta contextualizada
7. El sistema muestra la respuesta en el chat con formato Markdown
8. Si la respuesta incluye código o cambios sugeridos, se presenta un botón “Aplicar cambios”
9. El usuario puede:
 - Aceptar los cambios sugeridos (se aplican automáticamente)
 - Rechazar los cambios (se mantiene el estado actual)
 - Hacer preguntas de seguimiento
10. Si el usuario acepta los cambios, el sistema valida y actualiza el componente
11. Se muestra una notificación de éxito
12. El historial de conversación se mantiene durante toda la sesión

Postcondiciones:

- El usuario ha recibido asistencia contextual
- Los cambios sugeridos (si fueron aceptados) están aplicados
- El historial de conversación está disponible para referencia futura

La Figura 21 muestra la interfaz del chat con el asistente de IA en acción, mientras que la Figura 32 presenta el diagrama de secuencia de la interacción completa.

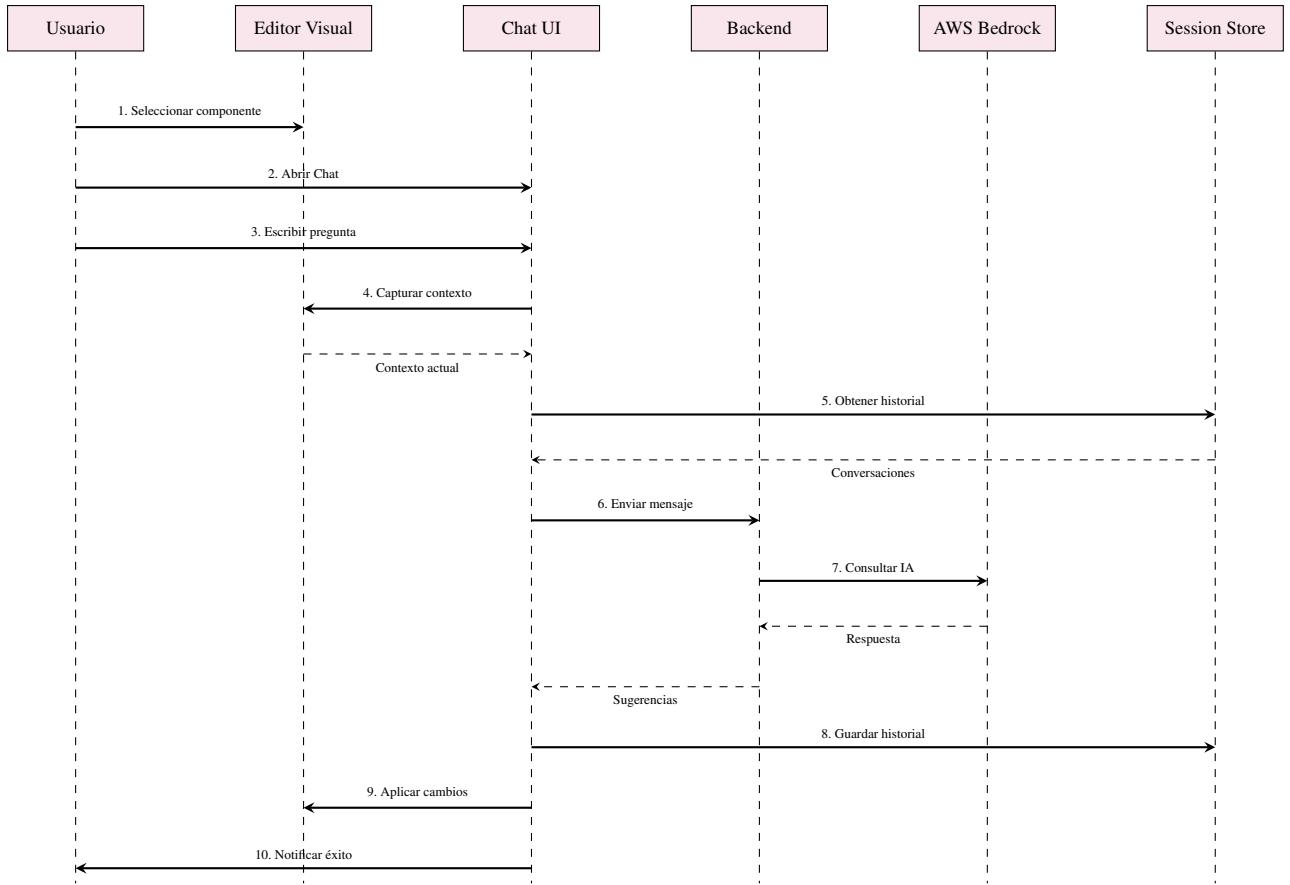


Figura 32: Diagrama de secuencia: Asistencia Contextual con Chat de IA

Estos casos de uso demuestran cómo Irakani Builder integra capacidades de IA de manera fluida en el flujo de trabajo del desarrollador, reduciendo significativamente el tiempo de desarrollo mientras mantiene el control total sobre el resultado final.

4.4. Implementación de la Solución

Esta sección documenta los aspectos técnicos de la implementación de Irakani Builder, incluyendo la configuración del entorno de desarrollo, el pipeline CI/CD, el desarrollo del backend y frontend, y la integración con servicios de AWS.

4.4.1. Configuración del Entorno y CI/CD

Arquitectura de CI/CD El proyecto Irakani Builder implementa un pipeline completo de CI/CD utilizando AWS CodePipeline integrado con Bitbucket como repositorio de código fuente. La arquitectura está diseñada para automatizar el proceso de construcción, pruebas y despliegue de la aplicación.

Componentes del Pipeline 1. Repositorio de Código (Bitbucket)

- **Repositorio:** expertfect/irakani-app-builder
- **Rama principal:** main
- **Conexión AWS:** Configurada mediante AWS CodeStar Connections
- **ARN de conexión:** [ARN_CENSURADO]

2. Infraestructura como Código (AWS CDK)

El proyecto utiliza AWS CDK (Cloud Development Kit) para definir y desplegar toda la infraestructura de manera programática. La estructura del pipeline se encuentra en el directorio `ci-cd/`:

```

ci-cd/
lib/          # Definiciones de stacks CDK
bin/          # Punto de entrada de la aplicación CDK
scripts/      # Scripts de automatización
deploy-buildspec.yml # Especificación de construcción
docker-compose.yml # Orquestación de contenedores

```

3. Etapas del Pipeline

El pipeline se compone de seis etapas principales:

Stage 1: Source (Obtención de Código)

- Se activa automáticamente con cada push a la rama `main`
- Obtiene el código fuente desde Bitbucket
- Genera un artefacto con el código fuente para las siguientes etapas

Stage 2: Build (Construcción)

Ejecuta dos procesos de construcción en paralelo:

Frontend Build:

```

1 version: 0.2
2 phases:
3   pre_build:
4     commands:
5       - echo Logging in to Amazon ECR...
6       - aws ecr get-login-password --region [REGION_AWS] | 
7         docker login --username AWS --password-stdin
8         [ECR_ENDPOINT_CENSURADO]
9   build:
10    commands:
11      - echo Building Docker image for frontend...
12      - docker build -t irakani-builder-frontend-ci-cd:latest
13        -f Dockerfile.frontend .
14      - docker tag irakani-builder-frontend-ci-cd:latest
15        [ECR_ENDPOINT_CENSURADO]/
16        irakani-builder-frontend-ci-cd:latest
17   post_build:
18     commands:
19       - echo Pushing Docker image to ECR...
20       - docker push [ECR_ENDPOINT_CENSURADO]/
21         irakani-builder-frontend-ci-cd:latest

```

Listing 22: Buildspec para frontend

Backend Build: Similar al frontend, pero utilizando el Dockerfile del backend.

Stage 3: Deploy (Despliegue)

Utiliza AWS Systems Manager (SSM) para ejecutar comandos en la instancia EC2. El proceso genera dinámicamente un archivo `docker-compose.yml`, lo sube a S3 y envía comandos a la instancia EC2 para descargar las nuevas imágenes desde ECR e iniciar los contenedores.

Stage 4: Manual Approval (Aprobación Manual)

Este stage requiere aprobación humana antes de proceder a producción:

- Permite verificar el despliegue en la instancia EC2 antes de crear AMI
- Evita crear AMIs con código defectuoso
- Proporciona un punto de control para revisión humana
- Pipeline se pausa automáticamente

- Se envía notificación SNS al equipo
- Revisor verifica la aplicación y aprueba o rechaza desde la consola de CodePipeline

Stage 5: Create AMI (Creación de AMI)

Este stage crea una Amazon Machine Image (AMI) de la instancia EC2 con la aplicación desplegada:

- Espera a que el deployment termine completamente
- Crea AMI con timestamp sin reiniciar la instancia
- Espera a que la AMI esté disponible
- Crea nueva versión del Launch Template con la nueva AMI
- Establece la nueva versión como default

Beneficios de la AMI:

- **Despliegues rápidos:** Nuevas instancias se lanzan en ~2 minutos
- **Consistencia:** Todas las instancias tienen exactamente la misma configuración
- **Rollback rápido:** Posibilidad de volver a una AMI anterior en minutos
- **Escalabilidad:** Auto Scaling puede lanzar instancias automáticamente

Stage 6: Update ASG (Actualización de Auto Scaling Group)

Este stage final actualiza el Auto Scaling Group con la nueva AMI mediante un rolling update:

Características del Rolling Update:

- **MinHealthyPercentage: 50 %:** Mantiene al menos 50 % de instancias saludables durante el update
- **InstanceWarmup: 300s:** Espera 5 minutos para que cada instancia nueva esté lista
- **Zero Downtime:** Los usuarios no experimentan interrupciones
- **Rollback automático:** Si las nuevas instancias fallan health checks, se revierte

Proceso del Rolling Update:

1. ASG lanza una nueva instancia con la nueva AMI
2. Espera 5 minutos (warmup) para que la instancia esté lista
3. Verifica health checks (HTTP en puerto [PUERTO_CENSURADO])
4. Si está saludable, termina una instancia antigua
5. Repite hasta que todas las instancias estén actualizadas
6. Limpia AMIs antiguas para ahorrar costos

Script de Despliegue en EC2 El archivo deploy.sh en la instancia EC2 maneja el proceso de actualización:

```

1 #!/bin/bash
2 set -e  # Salir si hay error
3
4 echo "Iniciando despliegue de Irakani Builder..."
5
6 # Autenticar con ECR
7 echo "Autenticando con ECR..."
8 aws ecr get-login-password --region [REGION_AWS] |
9 docker login --username AWS --password-stdin
[ECR_ENDPOINT_CENSURADO]
10

```

```

11 # Detener contenedores actuales
12 echo "Deteniendo contenedores actuales..."
13 docker-compose down || true
14
15 # Limpiar imágenes antiguas
16 echo "Limpiando imágenes antiguas..."
17 docker image prune -af
18
19 # Descargar nuevas imágenes
20 echo "Descargando nuevas imágenes..."
21 docker-compose pull
22
23 # Iniciar contenedores
24 echo "Iniciando contenedores..."
25 docker-compose up -d
26
27 # Verificar estado
28 echo "Verificando estado de contenedores..."
29 docker-compose ps
30
31 # Health check
32 echo "Verificando salud de la aplicación..."
33 sleep 10
34
35 if curl -f [URL_BACKEND_LOCAL]/api/check > /dev/null 2>&1; then
36   echo "Backend está funcionando"
37 else
38   echo "Backend no responde"
39   exit 1
40 fi
41
42 if curl -f [URL_FRONTEND_LOCAL] > /dev/null 2>&1; then
43   echo "Frontend está funcionando"
44 else
45   echo "Frontend no responde"
46   exit 1
47 fi
48
49 echo "Despliegue completado exitosamente!"

```

Listing 23: Script de despliegue

Infraestructura de Red VPC Existente:

- **VPC ID:** [VPC_ID_CENSURADO]
- Subnet Pública: [SUBNET_ID_CENSURADA]
- Security Groups: Configurados para permitir tráfico HTTP/HTTPS y SSH

Instancia EC2:

- **Tipo:** t3.small
- **AMI:** Amazon Linux 2
- **Rol IAM:** Con permisos para ECR, S3, SSM
- **Software instalado:** Docker, Docker Compose, AWS CLI

Monitoreo y Logs del Pipeline

1. Monitoreo en CodePipeline:

```
1 # Ver estado del pipeline
2 aws codepipeline get-pipeline-state
3   --name irakani-builder-ci-cd-pipeline
4
5 # Ver historial de ejecuciones
6 aws codepipeline list-pipeline-executions
7   --pipeline-name irakani-builder-ci-cd-pipeline --max-items 10
```

2. Logs de CodeBuild:

```
1 # Listar builds recientes
2 aws codebuild list-builds --for-project
3   --project-name irakani-builder-frontend-build
4
5 # Ver logs de un build específico
6 aws logs get-log-events
7   --log-group-name /aws/codebuild/irakani-builder-frontend-build
8   --log-stream-name <log-stream-name>
```

3. Logs de la Aplicación en EC2:

```
1 # Conectarse a EC2 via SSM
2 aws ssm start-session --target <instance-id>
3
4 # Ver logs de Docker Compose
5 cd /opt/irakani-builder
6 docker-compose logs -f
7
8 # Ver logs específicos de un servicio
9 docker-compose logs -f backend
10 docker-compose logs -f frontend
```

Flujo Completo del Pipeline El flujo detallado del pipeline es el siguiente:

1. **Developer Push (t=0s)**: El desarrollador hace push a la rama `main`
2. **Bitbucket Webhook (t=1s)**: Bitbucket detecta el push y notifica a AWS
3. **Stage 1: Source (t=2-5s)**: CodePipeline descarga el código y crea artefacto
4. **Stage 2: Build (t=5s-5min)**: Construcción paralela de frontend y backend
5. **Stage 3: Deploy (t=5min-7min)**: Despliegue en EC2 con health checks
6. **Stage 4: Manual Approval**: Pausa para revisión humana
7. **Stage 5: Create AMI (5-10min)**: Creación de AMI y actualización de Launch Template
8. **Stage 6: Update ASG (5-15min)**: Rolling update del Auto Scaling Group

Tiempo Total del Pipeline:

- Sin AMI/ASG: 6-8 minutos desde push hasta aplicación desplegada
- Con AMI/ASG: 15-25 minutos (incluye aprobación manual y rolling update)

Configuración de Notificaciones El sistema implementa notificaciones mediante SNS y CloudWatch Alarms:

SNS para Alertas del Pipeline:

```
1 # Crear t pico SNS
2 aws sns create-topic --name irakani-builder-pipeline-notifications
3
4 # Suscribir email
5 aws sns subscribe
6   --topic-arn [ARN_SNS_CENSURADO]:
7     irakani-builder-pipeline-notifications
8   --protocol email
9   --notification-endpoint developer@irakani.com
```

CloudWatch Alarms:

```
1 # Alarma para builds fallidos
2 aws cloudwatch put-metric-alarm
3   --alarm-name irakani-builder-build-failures
4   --alarm-description "Alerta cuando fallan builds"
5   --metric-name FailedBuilds
6   --namespace AWS/CodeBuild
7   --statistic Sum
8   --period 300
9   --threshold 1
10  --comparison-operator GreaterThanThreshold
11
12 # Alarma para alta utilizaci n de CPU en EC2
13 aws cloudwatch put-metric-alarm
14   --alarm-name irakani-builder-high-cpu
15   --alarm-description "Alerta cuando CPU > 80%"
16   --metric-name CPUUtilization
17   --namespace AWS/EC2
18   --statistic Average
19   --period 300
20   --threshold 80
21   --comparison-operator GreaterThanThreshold
```

Arquitectura del Pipeline CI/CD

El proyecto implementa un pipeline completo de CI/CD utilizando AWS CodePipeline integrado con Bitbucket como repositorio de código fuente. El pipeline consta de 6 etapas principales:

1. **Stage 1 - Source:** Obtención automática del código desde Bitbucket al detectar cambios en la rama `main`
2. **Stage 2 - Build:** Construcción paralela de imágenes Docker para frontend (React) y backend (Node.js), con push a Amazon ECR
3. **Stage 3 - Deploy:** Despliegue automático en instancia EC2 mediante AWS Systems Manager (SSM)
4. **Stage 4 - Manual Approval:** Punto de control para revisión humana antes de proceder a producción
5. **Stage 5 - Create AMI:** Creación de Amazon Machine Image para escalabilidad
6. **Stage 6 - Update ASG:** Actualización del Auto Scaling Group mediante rolling update

Componentes de Infraestructura:

- **Repositorio:** Bitbucket (`expertfect/irakani-app-builder`)
- **Infraestructura como Código:** AWS CDK (Cloud Development Kit)
- **Contenedores:** Docker con orquestación mediante Docker Compose
- **Registro de Imágenes:** Amazon ECR (Elastic Container Registry)

- **Cómputo:** Instancias EC2 t3.small con Amazon Linux 2
- **Escalabilidad:** Auto Scaling Group con Launch Templates
- **Monitoreo:** CloudWatch Logs, Metrics y Alarms

El tiempo total del pipeline es de 6-8 minutos para despliegue básico, o 15-25 minutos cuando incluye la creación de AMI y actualización del Auto Scaling Group.

4.4.2. Desarrollo del Backend (BFF)

Arquitectura del Backend El backend de Irakani Builder está diseñado como un BFF (Backend for Frontend) que actúa como capa de orquestación entre el frontend y los servicios externos (AWS Bedrock, bases de datos, S3, etc.).

Stack Tecnológico

```

1 // package.json - Dependencias principales
2 {
3   "dependencies": {
4     "@aws-sdk/client-bedrock-runtime": "^3.0.0",      // IA Generativa
5     "@aws-sdk/client-dynamodb": "^3.840.0",           // Base de datos NoSQL
6     "@aws-sdk/client-s3": "^3.0.0",                    // Almacenamiento
7     "express": "^4.18.2",                            // Framework web
8     "redis": "^4.6.0",                             // Cliente Valkey/Redis
9     "axios": "^1.4.0",                            // Cliente HTTP
10    "jsonwebtoken": "^9.0.2",                      // Autenticacion JWT
11    "helmet": "^8.1.0",                           // Seguridad HTTP
12    "cors": "^2.8.5"                            // CORS
13  }
14}
```

Listing 24: Dependencias principales del backend

Implementación del Cliente API Irakani El backend centraliza todas las peticiones hacia el sistema legado de Irakani mediante wrappers de Axios configurados específicamente:

```

1 // backend/server.js - Cliente API Irakani
2 const axios = require('axios');
3
4 // Configuracion base para API de Irakani
5 const IRAKANI_API_BASE =
6   '[ENDPOINT_API_CENSURADO]';
7
8 // Endpoint para analisis de aplicaciones
9 app.post('/api/analyze', async (req, res) => {
10   try {
11     const response = await axios.post(
12       `${IRAKANI_API_BASE}/analyze`,
13       req.body,
14       {
15         headers: { 'Content-Type': 'application/json' },
16         timeout: 30000 // 30 segundos de timeout
17       }
18     );
19     res.json(response.data);
20   } catch (error) {
21     console.error('Error en analisis:', error);
22     res.status(500).json({
23       error: 'Error en analisis',
24       message: error.message,
25       details: error.response?.data || {}
26     });
27   }
28 }
```

```

27     }
28 });
29
30 // Endpoint para generacion de aplicaciones
31 app.post('/api/generate', async (req, res) => {
32   try {
33     const response = await axios.post(
34       `${IRAKANI_API_BASE}/generate`,
35       req.body,
36     {
37       headers: { 'Content-Type': 'application/json' },
38       timeout: 60000 // 60 segundos para generacion
39     }
40   );
41   res.json(response.data);
42 } catch (error) {
43   console.error('Error en generacion:', error);
44   res.status(500).json({
45     error: 'Error en generacion',
46     message: error.message,
47     details: error.response?.data || {}
48   });
49 }
50 });

```

Listing 25: Cliente API Irakani

Características del Cliente:

- **Timeouts configurables:** Diferentes tiempos según la operación
- **Manejo de errores robusto:** Captura y transforma errores de la API
- **Logging detallado:** Registra todas las peticiones para debugging
- **Retry logic:** Implementado en el cliente Axios para reintentos automáticos

Implementación de Valkey (Redis) Valkey es un fork de Redis que se utiliza para el almacenamiento de sesiones y caché de datos. La implementación proporciona una capa de persistencia rápida y eficiente.

Configuración del Cliente Valkey

```

1 // backend/routes/valkey.js
2 const { createClient } = require('redis');
3
4 // Configuracion de conexion
5 const VALKEY_CONFIG = {
6   socket: {
7     host: process.env.VALKEY_HOST ||
8       '[VALKEY_ENDPOINT_CENSURADO]',
9     port: parseInt(process.env.VALKEY_PORT) || [PUERTO_CENSURADO]
10   },
11   retry_strategy: (options) => {
12     if (options.error && options.error.code === 'ECONNREFUSED') {
13       return new Error('El servidor Valkey rechazo la conexion');
14     }
15     if (options.total_retry_time > 1000 * 60 * 60) {
16       return new Error('Tiempo de reintento agotado');
17     }
18     if (options.attempt > 10) {
19       return undefined;
20     }
21     // Backoff exponencial: 100ms, 200ms, 400ms, ...
22     return Math.min(options.attempt * 100, 3000);

```

```

23     }
24   };
25
26 // Inicializacion del cliente
27 let valkeyClient = null;
28
29 const initValkeyClient = async () => {
30   if (!valkeyClient) {
31     try {
32       valkeyClient = createClient(VALKEY_CONFIG);
33
34       valkeyClient.on('error', (err) => {
35         console.error('Error de cliente Valkey:', err);
36       });
37
38       valkeyClient.on('connect', () => {
39         console.log('Conectado a Valkey');
40       });
41
42       valkeyClient.on('reconnecting', () => {
43         console.log('Reconectando a Valkey...');
44       });
45
46       await valkeyClient.connect();
47     } catch (error) {
48       console.error('Error conectando a Valkey:', error);
49       valkeyClient = null;
50     }
51   }
52
53   return valkeyClient;
54 };

```

Listing 26: Configuración del cliente Valkey

Operaciones de Sesión con Valkey 1. SET - Establecer valor con TTL:

```

1 // Middleware para verificar conexión
2 const ensureConnection = async (req, res, next) => {
3   try {
4     const client = await initValkeyClient();
5     if (!client) {
6       return res.status(503).json({
7         error: 'Servicio Valkey no disponible'
8       });
9     }
10    req.valkeyClient = client;
11    next();
12  } catch (error) {
13    res.status(503).json({
14      error: 'Error conectando a Valkey',
15      message: error.message
16    });
17  }
18};
19
20 // Endpoint SET con TTL opcional
21 router.post('/set', ensureConnection, async (req, res) => {
22   try {
23     const { key, value, ttl } = req.body;
24
25     if (!key || value === undefined) {
26       return res.status(400).json({

```

```

27     error: 'key y value son requeridos'
28   });
29 }
30
31 // Si se proporciona TTL, usar setEx (SET con expiracion)
32 if (ttl) {
33   await req.valkeyClient.setEx(key, ttl, value);
34   console.log('SET ${key} con TTL de ${ttl}s');
35 } else {
36   await req.valkeyClient.set(key, value);
37   console.log('SET ${key} sin expiracion');
38 }
39
40 res.json({ success: true });
41 } catch (error) {
42   console.error('Error en SET:', error);
43   res.status(500).json({
44     error: 'Error estableciendo valor',
45     message: error.message
46   });
47 }
48 );

```

Listing 27: Operación SET con TTL

2. GET - Obtener valor:

```

1 // Soporta tanto GET como POST para flexibilidad
2 router.get('/get/:key?', ensureConnection, async (req, res) => {
3   try {
4     const key = req.params.key || req.query.key;
5
6     if (!key) {
7       return res.status(400).json({
8         error: 'key es requerido como parametro o query'
9       });
10   }
11
12   const value = await req.valkeyClient.get(key);
13   console.log(`GET ${key}: ${value ? 'encontrado' : 'no encontrado'}`);
14
15   res.json({ value });
16 } catch (error) {
17   console.error('Error en GET:', error);
18   res.status(500).json({
19     error: 'Error obteniendo valor',
20     message: error.message
21   });
22 }
23 );

```

Listing 28: Operación GET

3. DEL - Eliminar valor:

```

1 router.post('/del', ensureConnection, async (req, res) => {
2   try {
3     const { key } = req.body;
4
5     if (!key) {
6       return res.status(400).json({ error: 'key es requerido' });
7     }
8
9     await req.valkeyClient.del(key);

```

```

10    console.log('DEL ${key}');
11
12    res.json({ success: true });
13 } catch (error) {
14     console.error('Error en DEL:', error);
15     res.status(500).json({
16       error: 'Error eliminando valor',
17       message: error.message
18     });
19   }
20 });

```

Listing 29: Operación DELETE

4. EXISTS - Verificar existencia:

```

1 router.post('/exists', ensureConnection, async (req, res) => {
2   try {
3     const { key } = req.body;
4
5     if (!key) {
6       return res.status(400).json({ error: 'key es requerido' });
7     }
8
9     const exists = await req.valkeyClient.exists(key);
10    console.log('EXISTS ${key}: ${exists === 1}');
11
12    res.json({ exists: exists === 1 });
13  } catch (error) {
14    console.error('Error en EXISTS:', error);
15    res.status(500).json({
16      error: 'Error verificando existencia',
17      message: error.message
18    });
19  }
20 });

```

Listing 30: Operación EXISTS

Casos de Uso de Valkey en Irakani Builder 1. Gestión de Sesiones de Usuario:

```

1 // Almacenar sesion de usuario con TTL de 24 horas
2 const sessionKey = 'session:${userId}';
3 const sessionData = JSON.stringify({
4   userId,
5   email: user.email,
6   role: user.role,
7   loginTime: Date.now()
8 });
9 await valkeyClient.setEx(sessionKey, 86400, sessionData); // 24 horas

```

2. Caché de Conexiones de Base de Datos:

```

1 // Almacenar token de conexion a BD con TTL de 1 hora
2 const connectionKey = 'db:connection:${connectionId}';
3 const connectionData = JSON.stringify({
4   connectionId,
5   dbType: 'mysql',
6   host: 'db.example.com',
7   token: encryptedToken
8 });
9 await valkeyClient.setEx(connectionKey, 3600, connectionData); // 1 hora

```

3. Rate Limiting:

```
1 // Limitar peticiones por usuario
2 const rateLimitKey = `ratelimit:${userId}:${endpoint}`;
3 const count = await valkeyClient.incr(rateLimitKey);
4
5 if (count === 1) {
6     // Primera petición, establecer TTL de 1 minuto
7     await valkeyClient.expire(rateLimitKey, 60);
8 }
9
10 if (count > 100) {
11     return res.status(429).json({ error: 'Demasiadas peticiones' });
12 }
```

Middleware de Autenticación El sistema implementa un middleware de autenticación robusto que valida el acceso usando tokens JWT y mantiene las sesiones activas en Valkey.

Implementación del Middleware de Autenticación

```
1 // backend/middleware/auth.js
2 const SecurityService = require('../services/SecurityService');
3 const DatabaseService = require('../services/DatabaseService');
4
5 /**
6 * Middleware de autenticacion para conexiones de base de datos
7 * Valida el token JWT y verifica la conexión activa
8 */
9 const authenticateDB = (req, res, next) => {
10     // Extraer token del header Authorization
11     const token = req.headers.authorization?.replace('Bearer ', '');
12     const connectionId = req.headers['x-connection-id'];
13
14     // Validar presencia de credenciales
15     if (!token || !connectionId) {
16         return res.status(401).json({
17             error: 'Token y connectionId requeridos',
18             details: 'Debe proporcionar un token de autenticacion y un ID de conexion validos'
19         });
20     }
21
22     try {
23         // Verificar validez del token JWT
24         SecurityService.verifyToken(token);
25
26         // Validar que el token corresponda a la conexión
27         if (!DatabaseService.validateToken(connectionId, token)) {
28             return res.status(401).json({
29                 error: 'Token invalido para esta conexión',
30                 details: 'El token proporcionado no corresponde a la conexión especificada'
31             });
32         }
33
34         // Adjuntar connectionId al request para uso posterior
35         req.connectionId = connectionId;
36         next();
37     } catch (error) {
38         console.error('Error en autenticacion:', error);
39         res.status(401).json({
40             error: 'Token invalido',
41             details: error.message
42         });
43     }
44 }
```

```

43     }
44   };
45
46 module.exports = { authenticateDB };

```

Listing 31: Middleware de autenticación

Servicio de Seguridad

```

1 // backend/services/SecurityService.js
2 const jwt = require('jsonwebtoken');
3 const crypto = require('crypto');
4
5 class SecurityService {
6   constructor() {
7     this.jwtSecret = process.env.JWT_SECRET ||
8       'PYelaMo_GeE1Ki62783uks_&b14T';
9     this.encryptionKey = process.env.DB_SECRET_KEY ||
10      'default_secret_key_2024';
11   }
12
13 /**
14  * Genera un token JWT para una conexión de base de datos
15  * @param {string} connectionId - ID único de la conexión
16  * @param {string} userId - ID del usuario
17  * @param {object} metadata - Metadatos adicionales
18  * @returns {string} Token JWT firmado
19 */
20 generateToken(connectionId, userId, metadata = {}) {
21   const payload = {
22     connectionId,
23     userId,
24     ...metadata,
25     iat: Math.floor(Date.now() / 1000),
26     exp: Math.floor(Date.now() / 1000) + (60 * 60 * 24) // 24 horas
27   };
28   return jwt.sign(payload, this.jwtSecret);
29 }
30
31 /**
32  * Verifica la validez de un token JWT
33  * @param {string} token - Token a verificar
34  * @returns {object} Payload decodificado
35  * @throws {Error} Si el token es inválido o ha expirado
36 */
37 verifyToken(token) {
38   try {
39     return jwt.verify(token, this.jwtSecret);
40   } catch (error) {
41     if (error.name === 'TokenExpiredError') {
42       throw new Error('Token expirado');
43     }
44     throw new Error('Token inválido');
45   }
46 }
47
48 /**
49  * Encripta credenciales de base de datos
50  * @param {object} credentials - Credenciales a encriptar
51  * @returns {string} Credenciales encriptadas en base64
52 */
53 encryptCredentials(credentials) {
54   const algorithm = 'aes-256-cbc';

```

```

55     const key = crypto.scryptSync(this.encryptionKey, 'salt', 32);
56     const iv = crypto.randomBytes(16);
57
58     const cipher = crypto.createCipheriv(algorithm, key, iv);
59     let encrypted = cipher.update(
60       JSON.stringify(credentials), 'utf8', 'hex'
61     );
62     encrypted += cipher.final('hex');
63
64     return Buffer.from(
65       iv.toString('hex') + ':' + encrypted
66     ).toString('base64');
67   }
68
69 /**
70 * Desencripta credenciales de base de datos
71 * @param {string} encryptedData - Datos encriptados en base64
72 * @returns {object} Credenciales desencriptadas
73 */
74 decryptCredentials(encryptedData) {
75   const algorithm = 'aes-256-cbc';
76   const key = crypto.scryptSync(this.encryptionKey, 'salt', 32);
77
78   const data = Buffer.from(encryptedData, 'base64').toString('utf8');
79   const [ivHex, encrypted] = data.split(':');
80   const iv = Buffer.from(ivHex, 'hex');
81
82   const decipher = crypto.createDecipheriv(algorithm, key, iv);
83   let decrypted = decipher.update(encrypted, 'hex', 'utf8');
84   decrypted += decipher.final('utf8');
85
86   return JSON.parse(decrypted);
87 }
88
89 module.exports = new SecurityService();

```

Listing 32: Servicio de seguridad

Flujo de Autenticación 1. Creación de Conexión:

```

1 // Cliente solicita conexión a base de datos
2 POST /api/database/connect
3 {
4   "dbType": "mysql",
5   "config": {
6     "host": "db.example.com",
7     "port": [PUERTO_CENSURADO],
8     "user": "admin",
9     "password": "encrypted_password",
10    "database": "mydb"
11  }
12}
13
14 // Servidor responde con token y connectionId
15 {
16   "connectionId": "conn_abci123xyz",
17   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
18   "status": "connected"
19 }

```

2. Uso del Token en Peticiones Subsecuentes:

```

1 // Cliente incluye token en headers
2 POST /api/database/query
3
4 Headers:
5   Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
6   x-connection-id: conn_abc123xyz
7
8 Body:
9 {
10   "query": "SELECT * FROM users WHERE active = ?",
11   "params": [true]
12 }

```

3. Validación en Middleware:

El middleware `authenticateDB` valida:

1. Presencia de token y connectionId
2. Validez del token JWT (firma y expiración)
3. Correspondencia entre token y connectionId
4. Existencia de la conexión activa

Si todo es válido, la petición continúa. Si falla alguna validación, retorna 401 Unauthorized.

Integración con Valkey para Sesiones

```

1 // Almacenar sesión en Valkey al crear conexión
2 const sessionKey = `session:${connectionId}`;
3 const sessionData = {
4   userId,
5   connectionId,
6   dbType: config.dbType,
7   createdAt: Date.now(),
8   lastActivity: Date.now()
9 };
10
11 await valkeyClient.setEx(
12   sessionKey,
13   86400, // 24 horas
14   JSON.stringify(sessionData)
15 );
16
17 // Actualizar última actividad en cada petición
18 const updateActivity = async (connectionId) => {
19   const sessionKey = `session:${connectionId}`;
20   const session = await valkeyClient.get(sessionKey);
21
22   if (session) {
23     const data = JSON.parse(session);
24     data.lastActivity = Date.now();
25     await valkeyClient.setEx(sessionKey, 86400, JSON.stringify(data));
26   }
27 };
28
29 // Limpiar sesión al cerrar conexión
30 const closeSession = async (connectionId) => {
31   const sessionKey = `session:${connectionId}`;
32   await valkeyClient.del(sessionKey);
33   console.log(`Sesión cerrada: ${connectionId}`);
34 };

```

Listing 33: Gestión de sesiones con Valkey

El backend de Irakani Builder está diseñado como un Backend for Frontend (BFF) que actúa como capa de orquestación entre el frontend React y los servicios externos de AWS.

Stack Tecnológico del Backend:

- **Framework:** Node.js con Express.js
- **SDK AWS:** @aws-sdk/client-bedrock-runtime, @aws-sdk/client-s3, @aws-sdk/client-dynamodb
- **Caché:** Redis (cliente para Valkey)
- **Autenticación:** jsonwebtoken para JWT
- **Seguridad:** helmet para headers HTTP seguros
- **Cliente HTTP:** axios para peticiones a APIs externas

Servicios Implementados:

1. **Cliente API Irakani:** Wrapper de Axios para comunicación con el sistema legado de Irakani, con manejo de timeouts configurables.
2. **Servicio Valkey (Redis):** Implementación de caché distribuido para:
 - Gestión de sesiones de usuario con TTL de 24 horas
 - Caché de conexiones a bases de datos con TTL de 1 hora
 - Rate limiting por usuario y endpoint
 - Operaciones: SET, GET, DEL, EXISTS con soporte para expiración automática
3. **Middleware de Autenticación:** Sistema robusto de validación de tokens JWT que:
 - Verifica la validez y firma de tokens JWT
 - Valida la correspondencia entre token y connectionId
 - Gestiona la expiración de tokens (24 horas)
 - Encripta/desencrypta credenciales de bases de datos usando AES-256-CBC

4.4.3. Integración con AWS Bedrock

AWS Bedrock es el servicio de IA Generativa que potencia las capacidades inteligentes de Irakani Builder. La integración permite la generación automática de código, aplicaciones, y asistencia en tiempo real.

Configuración del Cliente Bedrock La configuración del cliente AWS SDK v3 se realiza en el backend, permitiendo tanto desarrollo local con credenciales explícitas como producción usando roles IAM de EC2:

```
1 // backend/server.js - Configuración AWS SDK v3
2 const { BedrockRuntimeClient, ConverseCommand,
3         ConverseStreamCommand, InvokeModelCommand } =
4     require('@aws-sdk/client-bedrock-runtime');
5
6 // Configuración del cliente AWS
7 const awsConfig = {
8     region: process.env.AWS_REGION || '[REGION_AWS]'
9 };
10
11 // Solo agregar credenciales si están definidas (desarrollo local)
12 // En producción, usa el rol IAM de la instancia EC2
13 if (process.env.AWS_ACCESS_KEY_ID &&
14     process.env.AWS_SECRET_ACCESS_KEY) {
15     awsConfig.credentials = {
16         accessKeyId: process.env.AWS_ACCESS_KEY_ID,
```

```

17     secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY,
18     sessionToken: process.env.AWS_SESSION_TOKEN
19   };
20 }
21
22 const bedrockClient = new BedrockRuntimeClient(awsConfig);

```

Listing 34: Configuración del cliente AWS Bedrock

Modelos Utilizados y Estructura de Precios El sistema utiliza diferentes modelos de AWS Bedrock según la tarea, optimizando costos y rendimiento:

Modelo	Input (por 1K tokens)	Output (por 1K tokens)
Claude Sonnet 4	[PRECIO CENSURADO]	[PRECIO CENSURADO]
Claude 3.5 Haiku	[PRECIO CENSURADO]	[PRECIO CENSURADO]
Llama 4 Maverick 17B	[PRECIO CENSURADO]	[PRECIO CENSURADO]

Cuadro 6: Precios de modelos de texto en AWS Bedrock

Modelo	Precio por Imagen
Titan Image Generator V2	[PRECIO CENSURADO]

Cuadro 7: Precios de modelos de generación de imágenes

La función de cálculo de costos permite trazabilidad financiera de cada operación:

```

1 function calculateCost(model, inputTokens = 0,
2                         outputTokens = 0, imageCount = 0) {
3   const pricing = MODEL_PRICING[model];
4   if (!pricing) return 0;
5
6   // Para modelos de generacion de imagenes
7   if (model.includes('titan-image')) {
8     return imageCount * pricing.perImage;
9   }
10
11  // Para modelos de texto
12  const inputCost = (inputTokens / 1000) * pricing.input;
13  const outputCost = (outputTokens / 1000) * pricing.output;
14  return inputCost + outputCost;
15}

```

Listing 35: Función de cálculo de costos

Servicio de Chat con Bedrock El sistema implementa dos modalidades de chat: respuesta completa y streaming en tiempo real.

Chat Básico (Sin Streaming): Utilizado para operaciones que requieren la respuesta completa antes de procesarla:

```

1 app.post('/api/chat/bedrock', async (req, res) => {
2   try {
3     const { messages, systemPrompt } = req.body;
4
5     // Configurar comando para Bedrock
6     const command = new ConverseCommand({
7       modelId: 'us.anthropic.claude-sonnet-4-20250514-v1:0',
8       messages,
9       system: systemPrompt ? [{ text: systemPrompt }] : [],
10      inferenceConfig: {
11        maxTokens: 8000,
12        temperature: 0.4, // Creatividad moderada
13      }
14    });
15
16    const response = await command.execute();
17
18    res.status(200).json(response);
19  } catch (error) {
20    console.error(error);
21    res.status(500).json({ error: 'Internal Server Error' });
22  }
23}

```

```

13     topP: 0.4,
14     stopSequences: []
15   }
16 });
17
18 const result = await bedrockClient.send(command);
19 const output = result.output.message.content[0].text;
20
21 // Trazabilidad de tokens y costo
22 const usage = result.usage;
23 const cost = calculateCost('claude-sonnet-4',
24                           usage.inputTokens,
25                           usage.outputTokens);
26
27 console.log('Chat Bedrock:', {
28   inputTokens: usage.inputTokens,
29   outputTokens: usage.outputTokens,
30   cost: `$$ ${cost.toFixed(6)}`
31 });
32
33 // Guardar uso en DynamoDB
34 await usageLogService.logUsage(
35   req.headers['x-user-id'] || 'anonymous',
36   usage.inputTokens,
37   usage.outputTokens,
38   'claude-sonnet-4',
39   req.headers['x-schema'] || 'chat',
40   req.headers['x-context'] || 'default'
41 );
42
43 res.json({ message: output });
44 } catch (error) {
45   console.error('Error en chat con Bedrock:', error);
46   res.status(500).json({ error: 'Error en chat con Bedrock' });
47 }
48 });

```

Listing 36: Endpoint de chat básico con Bedrock

Chat con Streaming: Proporciona respuestas en tiempo real usando Server-Sent Events (SSE), mejorando la experiencia de usuario:

```

1 app.post('/api/chat/bedrock/stream', async (req, res) => {
2   try {
3     const { messages, systemPrompt } = req.body;
4
5     // Configurar headers para Server-Sent Events
6     res.writeHead(200, {
7       'Content-Type': 'text/event-stream',
8       'Cache-Control': 'no-cache',
9       'Connection': 'keep-alive'
10    });
11
12    const command = new ConverseStreamCommand({
13      modelId: 'us.anthropic.claude-sonnet-4-20250514-v1:0',
14      messages,
15      system: systemPrompt ? [{ text: systemPrompt }] : [],
16      inferenceConfig: {
17        maxTokens: 8000,
18        temperature: 0.2, // Mas determinista para streaming
19        topP: 0.4
20      }
21    });

```

```

22
23     const response = await bedrockClient.send(command);
24     let usage = null;
25
26     // Procesar stream de respuesta
27     for await (const chunk of response.stream) {
28         if (chunk.contentBlockDelta?.delta?.text) {
29             const text = chunk.contentBlockDelta.delta.text;
30             res.write(`data: ${JSON.stringify({
31                 type: 'chunk',
32                 content: text
33             })}\n\n`);
34         }
35
36         if (chunk.metadata?.usage) {
37             usage = chunk.metadata.usage;
38         }
39     }
40
41     // Enviar metricas finales
42     if (usage) {
43         res.write(`data: ${JSON.stringify({
44             type: 'tokens',
45             usage
46         })}\n\n`);
47     }
48
49     res.write(`data: ${JSON.stringify({ type: 'end' })}\n\n`);
50     res.end();
51 } catch (error) {
52     console.error('Error en chat streaming:', error);
53     res.write(`data: ${JSON.stringify({
54         type: 'error',
55         message: error.message
56     })}\n\n`);
57     res.end();
58 }
59 );

```

Listing 37: Endpoint de chat con streaming

Generación de Iconos con Titan Image Generator El sistema genera iconos personalizados usando IA en un proceso de tres pasos:

1. **Traducción del prompt:** Usa Claude Haiku para traducir descripciones en español a inglés
2. **Generación de imagen:** Titan Image Generator V2 crea el ícono en blanco y negro
3. **Almacenamiento:** La imagen se guarda en S3 para acceso posterior

```

1 app.post('/api/generate-icon', async (req, res) => {
2     try {
3         const { prompt, bucket = 'irakani-app-builder', userId } = req.body;
4
5         // PASO 1: Traducir prompt al inglés usando Claude Haiku
6         const translateCommand = new ConverseCommand({
7             modelId: 'us.anthropic.claude-3-5-haiku-20241022-v1:0',
8             messages: [
9                 {
10                     role: 'user',
11                     content: [
12                         {
13                             text: 'Translate this Spanish text to English for an

```

```

12         icon description. Only return the English
13         translation: ${prompt}'
14     }]
15   ],
16   inferenceConfig: {
17     maxTokens: 100,
18     temperature: 0.1 // Muy determinista para traducción
19   }
20 );
21
22 const translateResult = await bedrockClient.send(translateCommand);
23 const englishPrompt =
24   translateResult.output.message.content[0].text.trim();
25
26 // PASO 2: Generar imagen con Titan
27 const enhancedPrompt = `${englishPrompt} icon, black and white
28   only, solid black icon on pure white background. Monochrome
29   design, minimalist flat style, clean vector illustration,
30   modern app icon, centered composition, high contrast.‘;
31
32 const negativePrompt = ‘colors, gradient, shadows, 3d effects,
33   realistic, text, letters, watermark, blurry, low quality‘;
34
35 // Generar semilla determinística
36 const seed = Math.abs(prompt.split('')).reduce((acc, char) => {
37   return ((acc << 5) - acc) + char.charCodeAt(0);
38 }, 0)) % 2147483647;
39
40 const imageCommand = new InvokeModelCommand({
41   modelId: 'amazon.titan-image-generator-v2:0',
42   body: JSON.stringify({
43     taskType: 'TEXT_IMAGE',
44     textToImageParams: {
45       text: enhancedPrompt,
46       negativeText: negativePrompt
47     },
48     imageGenerationConfig: {
49       number_of_images: 1,
50       height: 512,
51       width: 512,
52       cfg_scale: 8.5,
53       seed: seed,
54       quality: 'premium'
55     }
56   })
57 );
58
59 const imageResult = await bedrockClient.send(imageCommand);
60 const responseBody =
61   JSON.parse(new TextDecoder().decode(imageResult.body));
62
63 // PASO 3: Subir imagen a S3
64 const imageData = responseBody.images[0];
65 const timestamp = Date.now();
66 const iconKey = `icons/${userId || 'anonymous'}/${timestamp}.png`;
67
68 const s3Command = new PutObjectCommand({
69   Bucket: bucket,
70   Key: iconKey,
71   Body: Buffer.from(imageData, 'base64'),
72   ContentType: 'image/png'
73 });

```

```

74     await s3Client.send(s3Command);
75
76     const imageUrl =
77       `/api/image/${bucket}/${userId || 'anonymous'}/${timestamp}.png`;
78
79     res.json({
80       imageUrl,
81       s3Key: iconKey,
82       originalPrompt: prompt,
83       translatedPrompt: englishPrompt
84     });
85   } catch (error) {
86     console.error('Error generando icono:', error);
87     res.status(500).json({ error: 'Error generando icono' });
88   }
89 }
90 );

```

Listing 38: Generación de iconos con IA

Autocompletado Inteligente de Código El sistema proporciona sugerencias contextuales de código usando Claude Haiku, optimizado para respuestas rápidas y económicas:

```

1 app.post('/api/ai/autocomplete', async (req, res) => {
2   try {
3     const { code, scriptType, currentJson } = req.body;
4
5     const systemPrompt = 'GENERADOR DE AUTOCOMPLETADO IRAKANI
Eres un experto en JavaScript ES6+ y la plataforma Irakani.
Analiza el código y sugiere 2-3 autocompletados contextuales.
6
7 FUNCIONES IRAKANI:
8 - datosEspacio: Variables compartidas entre apps
9 - entidad: Info de entidad seleccionada
10 - obtenerEvento(): Obtiene evento actual
11 - colocarValor(id, valor): Establece valor de campo
12 - obtenerValor(id): Obtiene valor de campo
13 - siguienteFormulario(): Avanza al siguiente formulario
14 - notificar(texto): Muestra notificación
15
16 Retorna JSON: [{"label": "nombre", "detail": "descripción",
17   "code": "código"}];
18
19   const contextInfo = currentJson ?
20     createResumedContext(currentJson) : '';
21
22   const userMessage = `CÓDIGO ACTUAL:
23 ${code}
24
25 TIPO DE SCRIPT: ${scriptType}
26 ${contextInfo ? `\nCONTEXTOS:\n${contextInfo}` : ''}
27
28 Analiza el código y sugiere autocompletados relevantes.';
29
30
31   const command = new ConverseCommand({
32     modelId: 'us.anthropic.claude-haiku-4-5-20251001-v1:0',
33     messages: [
34       {
35         role: 'user',
36         content: [{ text: userMessage }]
37     },
38     system: [{ text: systemPrompt }],
39     inferenceConfig: {

```

```

40     maxTokens: 3000,
41     temperature: 0.3 // Baja creatividad para precision
42   }
43 );
44
45 const result = await bedrockClient.send(command);
46 const output = result.output.message[0].text.trim();
47
48 // Extraer JSON del output
49 const jsonMatch = output.match(/[\s*\{[\s\S]*\}\s*\]/);
50 const suggestions = jsonMatch ? JSON.parse(jsonMatch[0]) : [];
51
52 res.json({ suggestions });
53 } catch (error) {
54   console.error('Error en autocomplete:', error);
55   res.json({ suggestions: [] });
56 }
57 );

```

Listing 39: Autocompletado asistido por IA

Servicio de Registro de Uso Para trazabilidad y análisis de costos, se implementó un servicio que registra cada uso de IA en DynamoDB:

```

1 // backend/services/UsageLogService.js
2 const { DynamoDBClient, PutItemCommand } =
3   require('@aws-sdk/client-dynamodb');
4
5 class UsageLogService {
6   constructor() {
7     this.dynamoClient = new DynamoDBClient({
8       region: process.env.AWS_REGION || '[REGION_AWS]'
9     });
10    this.tableName = 'IrakaniBuilderUsageLogs';
11  }
12
13  async logUsage(userId, inputTokens, outputTokens, model,
14    schema, context, sessionId = null, imageCount = 0) {
15    try {
16      const timestamp = new Date().toISOString();
17      const logId = `${userId}_${Date.now()}_${Math.random()
18        .toString(36).substr(2, 9)} `;
19
20      const cost = this.calculateCost(model, inputTokens,
21        outputTokens, imageCount);
22
23      const command = new PutItemCommand({
24        TableName: this.tableName,
25        Item: {
26          logId: { S: logId },
27          userId: { S: userId },
28          timestamp: { S: timestamp },
29          model: { S: model },
30          schema: { S: schema },
31          context: { S: context },
32          inputTokens: { N: inputTokens.toString() },
33          outputTokens: { N: outputTokens.toString() },
34          totalTokens: { N: (inputTokens + outputTokens).toString() },
35          imageCount: { N: imageCount.toString() },
36          cost: { N: cost.toFixed(6) },
37          sessionId: sessionId ? { S: sessionId } : { NULL: true }
38        }
39      );
40
41      await this.dynamoClient.send(command);
42    } catch (error) {
43      console.error(`Error al registrar el uso de IA: ${error}`);
44    }
45  }
46}

```

```

39   });
40
41   await this.dynamoClient.send(command);
42   console.log(`Uso registrado: ${model} - ${cost.toFixed(6)}`);
43 } catch (error) {
44   console.error('Error registrando uso:', error);
45 }
46
47
48 calculateCost(model, inputTokens, outputTokens, imageCount) {
49   const pricing = {
50     'claude-sonnet-4': { input: [PRECIO_CENSURADO], output: [PRECIO_CENSURADO] },
51     'claude-3-5-haiku': { input: [PRECIO_CENSURADO], output: [PRECIO_CENSURADO] },
52     'titan-image-generator-v2': { perImage: [PRECIO_CENSURADO] }
53   };
54
55   const modelPricing = pricing[model];
56   if (!modelPricing) return 0;
57
58   if (model.includes('titan-image')) {
59     return imageCount * modelPricing.perImage;
60   }
61
62   const inputCost = (inputTokens / 1000) * modelPricing.input;
63   const outputCost = (outputTokens / 1000) * modelPricing.output;
64   return inputCost + outputCost;
65 }
66
67
68 module.exports = UsageLogService;

```

Listing 40: Servicio de registro de uso de IA

Este servicio permite:

- Trazabilidad completa de uso de modelos de IA
- Análisis de costos por usuario, sesión y contexto
- Optimización de uso de modelos según patrones detectados
- Auditoría de operaciones de IA para cumplimiento

AWS Bedrock proporciona las capacidades de IA generativa del sistema. La integración incluye:

Modelos Utilizados:

- **Claude Sonnet 4:** Modelo principal para generación de código y asistencia ([PRECIO CENSURADO])
- **Claude 3.5 Haiku:** Modelo alternativo para tareas más simples ([PRECIO CENSURADO])
- **Titan Image Generator V2:** Generación de iconos ([PRECIO CENSURADO])

Servicios de IA Implementados:

1. **Chat Básico:** Endpoint `/api/chat/bedrock` que retorna respuesta completa con trazabilidad de tokens y costos
2. **Chat con Streaming:** Endpoint `/api/chat/bedrock/stream` que utiliza Server-Sent Events (SSE) para respuestas en tiempo real, mejorando la percepción de velocidad
3. **Generación de Iconos:** Endpoint `/api/generate-icon` que:
 - Traduce prompts del español al inglés usando Claude Haiku
 - Genera iconos en blanco y negro con Titan Image Generator

- Almacena resultados en S3
- Retorna métricas de uso y costos por operación

Sistema de Trazabilidad:

Todas las operaciones de IA se registran en DynamoDB (tabla IrakaniBuilderUsageLogs) con:

- Tokens de entrada y salida consumidos
- Modelo utilizado y costo calculado
- Usuario, contexto y timestamp de la operación
- Esquema de trabajo (espacio de Irakani)

4.4.4. Desarrollo del Frontend

El frontend de Irakani Builder está construido con React 18 y TypeScript, proporcionando una experiencia de usuario moderna y responsive con capacidades avanzadas de edición visual y de código.

Stack Tecnológico Frontend El frontend utiliza un conjunto moderno de tecnologías:

Tecnología	Versión	Propósito
React	18.2.0	Framework principal de UI
TypeScript	5.0.0	Tipado estático y seguridad
Monaco Editor	4.7.0	Editor de código profesional
AWS Amplify	6.15.1	Autenticación con Cognito
Zustand	5.0.6	Gestión de estado ligera
ReactFlow	11.11.4	Editor visual de workflows
Fabric.js	6.7.1	Canvas para editor visual
Framer Motion	10.16.0	Animaciones fluidas
Vite	7.0.0	Build tool y dev server

Cuadro 8: Stack tecnológico del frontend

Implementación del Editor Visual El editor visual permite diseñar aplicaciones mediante Fabric.js para manipulación de canvas:

```

1 // src/components/VisualEditor/VisualEditor.tsx
2 import React, { useState, useCallback } from 'react';
3 import { fabric } from 'fabric';
4 import { useEditorStore } from '../../../../../stores/editorStore';
5
6 export const VisualEditor: React.FC = ({
7   applicationData,
8   onUpdate
9 }) => {
10   const [canvas, setCanvas] = useState<fabric.Canvas | null>(null);
11   const { selectedElement, setSelectedElement } = useEditorStore();
12
13   // Inicializar canvas
14   React.useEffect(() => {
15     const fabricCanvas = new fabric.Canvas('visual-editor-canvas', {
16       width: 800,
17       height: 600,
18       backgroundColor: '#f5f5f5'
19     });
20
21     setCanvas(fabricCanvas);
22
23     return () => {
24       fabricCanvas.dispose();
25     };
26   }, []);
27 }
```

```

26 }, []);
27
28 // Agregar elemento al canvas
29 const addElement = useCallback((type: string) => {
30   if (!canvas) return;
31
32   let element: fabric.Object;
33
34   switch (type) {
35     case 'text':
36       element = new fabric.IText('Texto', {
37         left: 100,
38         top: 100,
39         fontSize: 16,
40         fill: '#000000'
41       });
42       break;
43
44     case 'button':
45       const button = new fabric.Rect({
46         left: 100, top: 100,
47         width: 120, height: 40,
48         fill: '#007bff',
49         rx: 5, ry: 5
50       });
51
52       const buttonText = new fabric.Text('Boton', {
53         left: 130, top: 110,
54         fontSize: 14, fill: '#ffffff'
55       });
56
57       element = new fabric.Group([button, buttonText], {
58         left: 100, top: 100
59       });
60       break;
61
62     case 'input':
63       element = new fabric.Rect({
64         left: 100, top: 100,
65         width: 200, height: 35,
66         fill: '#ffffff',
67         stroke: '#cccccc',
68         strokeWidth: 1,
69         rx: 3, ry: 3
70       });
71       break;
72
73     default:
74       return;
75   }
76
77   canvas.add(element);
78   canvas.setActiveObject(element);
79   canvas.renderAll();
80
81   onUpdate(serializedCanvas(canvas));
82 }, [canvas, onUpdate]);
83
84 // Serializar canvas a JSON
85 const serializedCanvas = (canvas: fabric.Canvas) => {
86   return canvas.toJSON(['id', 'name', 'type', 'customData']);
87 };

```

```

88
89     return (
90       <div className="visual-editor">
91         <div className="toolbar">
92           <button onClick={() => addElement('text')}>
93             Agregar Texto
94           </button>
95           <button onClick={() => addElement('button')}>
96             Agregar Boton
97           </button>
98           <button onClick={() => addElement('input')}>
99             Agregar Input
100            </button>
101          </div>
102          <canvas id="visual-editor-canvas" />
103        </div>
104      );
105    };

```

Listing 41: Componente principal del editor visual

Implementación del Editor de Código con Monaco Monaco Editor proporciona una experiencia profesional con resaltado de sintaxis, autocompletado inteligente asistido por IA, y validación en tiempo real:

```

1 // src/components/CodeEditor/CodeEditor.tsx
2 import React, { useRef, useState } from 'react';
3 import Editor, { Monaco } from '@monaco-editor/react';
4 import * as monaco from 'monaco-editor';
5 import { aiService } from '../../../../../services/aiService';
6
7 export const CodeEditor: React.FC = ({
8   value,
9   language,
10  onChange,
11  theme = 'vs-dark',
12  readOnly = false
13}) => {
14  const editorRef = useRef<monaco.editor.IStandaloneCodeEditor>(null);
15  const [isLoadingAutocomplete, setIsLoadingAutocomplete] =
16    useState(false);
17
18  const handleEditorDidMount = (
19    editor: monaco.editor.IStandaloneCodeEditor,
20    monaco: Monaco
21  ) => {
22    editorRef.current = editor;
23
24    // Configurar autocompletado personalizado con IA
25    monaco.languages.registerCompletionItemProvider('javascript', {
26      provideCompletionItems: async (model, position) => {
27        setIsLoadingAutocomplete(true);
28
29        try {
30          // Obtener código hasta la posición actual
31          const textUntilPosition = model.getValueInRange({
32            startLineNumber: 1,
33            startColumn: 1,
34            endLineNumber: position.lineNumber,
35            endColumn: position.column
36          });
37

```

```

38     // Solicitar sugerencias a la IA
39     const suggestions =
40       await aiService.getAutocompleteSuggestions({
41         code: textUntilPosition,
42         scriptType: language,
43         currentJson: null
44       });
45
46     // Convertir sugerencias a formato Monaco
47     return {
48       suggestions: suggestions.map((suggestion: any) => ({
49         label: suggestion.label,
50         kind: monaco.languages.CompletionItemKind.Function,
51         detail: suggestion.detail,
52         insertText: suggestion.code,
53         insertTextRules: monaco.languages
54           .CompletionItemInsertTextRule.InsertAsSnippet,
55         documentation: suggestion.detail
56       }))
57     };
58   } catch (error) {
59     console.error('Error obteniendo autocomplete:', error);
60     return { suggestions: [] };
61   } finally {
62     setIsLoadingAutocomplete(false);
63   }
64 },
65 triggerCharacters: ['.', '(', ')']
66 );
67
68 // Validacion en tiempo real
69 editor.onDidChangeModelContent(() => {
70   const value = editor.getValue();
71   onChange(value);
72 });
73
74 const editorOptions = {
75   readOnly,
76   minimap: { enabled: true },
77   fontSize: 14,
78   lineNumbers: 'on',
79   automaticLayout: true,
80   tabSize: 2,
81   wordWrap: 'on',
82   formatOnPaste: true,
83   formatOnType: true,
84   suggestOnTriggerCharacters: true,
85   quickSuggestions: {
86     other: true,
87     comments: false,
88     strings: true
89   },
90   parameterHints: { enabled: true },
91   folding: true
92 };
93
94 return (
95   <div className="code-editor-container">
96     {isLoadingAutocomplete && (
97       <div className="autocomplete-loading">
98         Obteniendo sugerencias de IA...
99

```

```

100         </div>
101     )}
102     <Editor
103       height="600px"
104       language={language}
105       value={value}
106       theme={theme}
107       options={editorOptions}
108       onMount={handleEditorDidMount}
109     />
110   </div>
111 );
112 }

```

Listing 42: Editor de código con autocompletado IA

Servicio de Autenticación con AWS Cognito La integración con AWS Amplify proporciona autenticación segura:

```

1 // src/services/authService.ts
2 import { Amplify } from 'aws-amplify';
3 import { signIn, signOut, getCurrentUser,
4         fetchAuthSession } from 'aws-amplify/auth';
5
6 // Configurar Amplify con Cognito
7 Amplify.configure({
8   Auth: {
9     Cognito: {
10       userPoolId: import.meta.env.VITE_USER_POOL_ID,
11       userPoolClientId: import.meta.env.VITE_USER_POOL_CLIENT_ID,
12       identityPoolId: import.meta.env.VITE_IDENTITY_POOL_ID,
13       region: import.meta.env.VITE_AWS_REGION || '[REGION_AWS]'
14     }
15   }
16 });
17
18 export class AuthService {
19   async login(username: string, password: string) {
20     try {
21       const user = await signIn({ username, password });
22       console.log('Login exitoso:', user);
23       return user;
24     } catch (error) {
25       console.error('Error en login:', error);
26       throw error;
27     }
28   }
29
30   async logout() {
31     try {
32       await signOut();
33       console.log('Logout exitoso');
34     } catch (error) {
35       console.error('Error en logout:', error);
36       throw error;
37     }
38   }
39
40   async getCurrentUser() {
41     try {
42       const user = await getCurrentUser();
43       return user;
44     } catch (error) {

```

```

45     return null;
46   }
47 }
48
49 async getAuthToken(): Promise<string | null> {
50   try {
51     const session = await fetchAuthSession();
52     return session.tokens?.idToken?.toString() || null;
53   } catch (error) {
54     return null;
55   }
56 }
57
58 async getAWS Credentials() {
59   try {
60     const session = await fetchAuthSession();
61     return session.credentials;
62   } catch (error) {
63     return null;
64   }
65 }
66
67 export const authService = new AuthService();

```

Listing 43: Servicio de autenticación

Servicio de IA para el Frontend El servicio encapsula todas las interacciones con el backend de IA:

```

1 // src/services/aiService.ts
2 import axios from 'axios';
3 import { authService } from './authService';
4
5 const API_BASE_URL = import.meta.env.VITE_API_URL ||
6   '[URL_BACKEND_LOCAL]';
7
8 export class AIService {
9   private axiosInstance;
10
11   constructor() {
12     this.axiosInstance = axios.create({
13       baseURL: API_BASE_URL,
14       timeout: 60000,
15       headers: { 'Content-Type': 'application/json' }
16     });
17
18   // Interceptor para agregar token de autenticacion
19   this.axiosInstance.interceptors.request.use(
20     async (config) => {
21       const token = await authService.getAuthToken();
22       if (token) {
23         config.headers.Authorization = `Bearer ${token}`;
24       }
25
26       const user = await authService.getCurrentUser();
27       if (user) {
28         config.headers['x-user-id'] = user.username;
29       }
30
31       return config;
32     }
33   );

```

```

34     }
35
36     async chat(messages: any[], systemPrompt?: string) {
37       const response = await this.axiosInstance.post(
38         '/api/chat/bedrock',
39         { messages, systemPrompt },
40         { headers: { 'x-context': 'chat', 'x-schema': 'general-chat' } }
41       );
42       return response.data;
43     }
44
45     async chatStream(
46       messages: any[],
47       systemPrompt: string | undefined,
48       onChunk: (chunk: string) => void,
49       onTokens?: (usage: any) => void
50     ) {
51       const user = await authService.getCurrentUser();
52       const token = await authService.getAuthToken();
53
54       const response = await fetch(
55         `${API_BASE_URL}/api/chat/bedrock/stream`,
56         {
57           method: 'POST',
58           headers: {
59             'Content-Type': 'application/json',
60             'Authorization': `Bearer ${token}`,
61             'x-user-id': user?.username || 'anonymous'
62           },
63           body: JSON.stringify({ messages, systemPrompt })
64         }
65       );
66
67       const reader = response.body?.getReader();
68       const decoder = new TextDecoder();
69
70       while (true) {
71         const { done, value } = await reader.read();
72         if (done) break;
73
74         const chunk = decoder.decode(value);
75         const lines = chunk.split('\n');
76
77         for (const line of lines) {
78           if (line.startsWith('data: ')) {
79             const data = JSON.parse(line.slice(6));
80
81             if (data.type === 'chunk') {
82               onChunk(data.content);
83             } else if (data.type === 'tokens' && onTokens) {
84               onTokens(data.usage);
85             } else if (data.type === 'end') {
86               return;
87             }
88           }
89         }
90       }
91     }
92
93     async generateIcon(prompt: string, userId?: string) {
94       const response = await this.axiosInstance.post(
95         '/api/generate-icon',

```

```

96     { prompt, userId, bucket: 'irakani-app-builder' }
97   );
98   return response.data;
99 }
100
101 async getAutocompleteSuggestions(params: {
102   code: string;
103   scriptType: string;
104   currentJson: any;
105 }) {
106   try {
107     const response = await this.axiosInstance.post(
108       '/api/ai/autocomplete',
109       params
110     );
111     return response.data.suggestions || [];
112   } catch (error) {
113     return [];
114   }
115 }
116
117 async generateApplication(prompt: string, options?: any) {
118   const response = await this.axiosInstance.post(
119     '/api/generate',
120     { prompt, ...options }
121   );
122   return response.data;
123 }
124
125
126 export const aiService = new AIService();

```

Listing 44: Servicio de IA del frontend

Gestión de Estado con Zustand Zustand proporciona gestión de estado ligera y eficiente:

```

1 // src/stores/editorStore.ts
2 import { create } from 'zustand';
3
4 interface EditorState {
5   currentApplication: any | null;
6   isModified: boolean;
7   selectedElement: any | null;
8   editorMode: 'visual' | 'code';
9   isChatOpen: boolean;
10  chatMessages: any[];
11  isGenerating: boolean;
12
13  setCurrentApplication: (app: any) => void;
14  setIsModified: (modified: boolean) => void;
15  setSelectedElement: (element: any) => void;
16  setEditorMode: (mode: 'visual' | 'code') => void;
17  toggleChat: () => void;
18  addChatMessage: (message: any) => void;
19  setIsGenerating: (generating: boolean) => void;
20  resetEditor: () => void;
21 }
22
23 export const useEditorStore = create<EditorState>((set) => ({
24   currentApplication: null,
25   isModified: false,
26   selectedElement: null,

```

```

27   editorMode: 'visual',
28   isChatOpen: false,
29   chatMessages: [],
30   isGenerating: false,
31
32   setCurrentApplication: (app) =>
33     set({ currentApplication: app }),
34   setIsModified: (modified) =>
35     set({ isModified: modified }),
36   setSelectedElement: (element) =>
37     set({ selectedElement: element }),
38   setEditorMode: (mode) =>
39     set({ editorMode: mode }),
40   toggleChat: () =>
41     set((state) => ({ isChatOpen: !state.isChatOpen })),
42   addChatMessage: (message) =>
43     set((state) => ({
44       chatMessages: [...state.chatMessages, message]
45     })),
46   setIsGenerating: (generating) =>
47     set({ isGenerating: generating }),
48   resetEditor: () => set({
49     currentApplication: null,
50     isModified: false,
51     selectedElement: null,
52     editorMode: 'visual',
53     isChatOpen: false,
54     chatMessages: [],
55     isGenerating: false
56   })
57 );

```

Listing 45: Store de estado del editor

Componente de Chat con IA El panel de chat proporciona interacción en tiempo real con el asistente de IA:

```

// src/components/ChatPanel/ChatPanel.tsx
import React, { useState, useRef, useEffect } from 'react';
import { aiService } from '../../../../../services/aiService';
import { useEditorStore } from '../../../../../stores/editorStore';

export const ChatPanel: React.FC = ({ onGenerate, isGenerating }) => {
  const { chatMessages, addChatMessage } = useEditorStore();
  const [input, setInput] = useState('');
  const [isStreaming, setIsStreaming] = useState(false);
  const [currentStreamMessage, setCurrentStreamMessage] = useState('');
  const messagesEndRef = useRef<HTMLDivElement>(null);

  useEffect(() => {
    messagesEndRef.current?.scrollIntoView({ behavior: 'smooth' });
  }, [chatMessages, currentStreamMessage]);

  const handleSend = async () => {
    if (!input.trim() || isStreaming) return;

    const userMessage = {
      role: 'user',
      content: input,
      timestamp: new Date().toISOString()
    };

    addChatMessage(userMessage);
  }
}

```

```

27     setInput('');
28     setIsStreaming(true);
29     setCurrentStreamMessage('');
30
31   try {
32     const messages = [
33       ...chatMessages.map(msg => ({
34         role: msg.role,
35         content: [{ text: msg.content }]
36       })),
37       {
38         role: 'user',
39         content: [{ text: input }]
40       }
41     ];
42
43     const systemPrompt = 'Eres un asistente experto en desarrollo
44       de aplicaciones con Irakani Builder. Ayuda al usuario a crear,
45       modificar y optimizar aplicaciones.';
46
47     await aiService.chatStream(
48       messages,
49       systemPrompt,
50       (chunk) => {
51         setCurrentStreamMessage(prev => prev + chunk);
52       },
53       (usage) => {
54         console.log('Uso de tokens:', usage);
55       }
56     );
57
58     const assistantMessage = {
59       role: 'assistant',
60       content: currentStreamMessage,
61       timestamp: new Date().toISOString()
62     };
63
64     addChatMessage(assistantMessage);
65     setCurrentStreamMessage('');
66     setIsStreaming(false);
67   } catch (error) {
68     console.error('Error en chat:', error);
69     setIsStreaming(false);
70   }
71 };
72
73   return (
74     <div className="chat-panel">
75       <div className="chat-header">
76         <h3>Asistente IA</h3>
77       </div>
78
79       <div className="chat-messages">
80         { chatMessages.map((message, index) => (
81           <div key={index} className={'message ${message.role}'}>
82             <div className="message-avatar">
83               {message.role === 'user' ? 'User' : 'AI'}
84             </div>
85             <div className="message-content">
86               <div className="message-text">{message.content}</div>
87               <div className="message-timestamp">
88                 {new Date(message.timestamp).toLocaleTimeString()}

```

```

89         </div>
90     </div>
91   </div>
92 )
93
94 {isStreaming && currentStreamMessage && (
95   <div className="message assistant streaming">
96     <div className="message-avatar"> </div>
97     <div className="message-content">
98       <div className="message-text">
99         {currentStreamMessage}
100        <span className="cursor"> </span>
101      </div>
102    </div>
103  </div>
104)
105
106 <div ref={messagesEndRef} />
107 </div>
108
109 <div className="chat-input-container">
110   <textarea
111     className="chat-input"
112     value={input}
113     onChange={(e) => setInput(e.target.value)}
114     placeholder="Escribe tu mensaje..."
115     disabled={isStreaming}
116     rows={3}
117   />
118   <button
119     className="send-button"
120     onClick={handleSend}
121     disabled={!input.trim() || isStreaming}
122   >
123     {isStreaming ? ' ' : ' '} Enviar
124   </button>
125 </div>
126
127 <div className="quick-actions">
128   <button onClick={() =>
129     setInput('Crea una aplicacion de gestion de inventario')}>
130     Inventario
131   </button>
132   <button onClick={() =>
133     setInput('Genera un formulario de registro de usuarios')}>
134     Formulario
135   </button>
136   <button onClick={() =>
137     setInput('Optimiza el codigo de mi aplicacion')}>
138     Optimizar
139   </button>
140 </div>
141 </div>
142 );
143 };

```

Listing 46: Panel de chat con streaming

El frontend integra todas estas tecnologías para proporcionar una experiencia de desarrollo completa, con edición visual y de código, asistencia de IA en tiempo real, y autenticación segura con AWS Cognito.

El frontend está construido con React 18 y TypeScript, proporcionando una experiencia de usuario moderna y responsive.

Stack Tecnológico del Frontend:

- **Framework:** React 18 con TypeScript
- **Editor de Código:** @monaco-editor/react (Monaco Editor)
- **Autenticación:** AWS Amplify con AWS Cognito
- **Gestión de Estado:** Zustand
- **Editor Visual:** ReactFlow y Fabric.js
- **Animaciones:** Framer Motion
- **Build Tool:** Vite

Componentes Principales:

- a) **Editor Visual:** Componente basado en Fabric.js que permite diseñar aplicaciones con elementos (texto, botones, inputs), con serialización a JSON
- b) **Editor de Código (Monaco):** Integración completa del editor de Visual Studio Code con:
 - Resaltado de sintaxis para JSON, JavaScript, SQL, HTML
 - Autocompletado inteligente asistido por IA
 - Validación en tiempo real
 - Más de 20 temas personalizables (Monokai, Dracula, GitHub, etc.)
 - Atajos de teclado personalizados (Ctrl+S para guardar)
- c) **Panel de Chat con IA:** Interfaz de conversación que:
 - Mantiene historial de conversación persistente
 - Soporta streaming de respuestas en tiempo real
 - Muestra métricas de tokens consumidos
 - Incluye sugerencias rápidas predefinidas
 - Aplica cambios sugeridos automáticamente al JSON

Servicios del Frontend:

- a) **AuthService:** Gestión de autenticación con AWS Cognito mediante Amplify, incluyendo login, logout, obtención de tokens y credenciales AWS temporales
- b) **AIService:** Cliente centralizado para todas las interacciones con IA que:
 - Agrega automáticamente tokens de autenticación a las peticiones
 - Implementa chat básico y con streaming
 - Gestiona generación de iconos y autocompletado
 - Incluye headers de trazabilidad (x-user-id, x-context, x-schema)
- c) **EditorStore (Zustand):** Gestión centralizada del estado de la aplicación, incluyendo:
 - Aplicación actual y estado de modificación
 - Elemento seleccionado y modo de editor (visual/código)
 - Estado del chat (abierto/cerrado, mensajes, generando)
 - Acciones para manipular el estado

4.5. Planificación de Sprints

Esta sección detalla la ejecución de cada sprint, documentando los objetivos, historias de usuario implementadas, resultados obtenidos y lecciones aprendidas durante el desarrollo del proyecto.

4.5.1. Sprint 0: Concepción y Diseño (14-27 julio 2025)

Sprint Goal: Establecer las bases técnicas y arquitectónicas del proyecto.

Historias de Usuario Implementadas:

- HU-001: Configuración del Entorno de Desarrollo (5 SP)
- HU-002: Definición de Arquitectura de Microservicios (8 SP)

Resultados Obtenidos:

- Repositorio Git configurado en Bitbucket con estructura de carpetas
- Pipeline CI/CD básico implementado
- Diagrama de arquitectura de microservicios documentado
- Stack tecnológico validado: React 18, Node.js, AWS Bedrock, Valkey
- Definición de endpoints RESTful principales
- Documentación inicial en README.md

Métricas del Sprint:

- Story Points Completados: 13/13 (100 %)
- Velocidad del Sprint: 13 SP
- Historias Completadas: 2/2

Lecciones Aprendidas:

- *Inversión inicial en infraestructura:* Dedicar tiempo suficiente a la configuración del entorno y CI/CD desde el inicio previene problemas de integración posteriores y facilita el desarrollo iterativo.
- *Documentación de arquitectura:* La definición clara de la arquitectura de microservicios y sus interfaces desde el Sprint 0 permitió que el equipo tuviera una visión compartida del sistema, reduciendo ambigüedades en sprints posteriores.
- *Validación temprana del stack tecnológico:* Realizar pruebas de concepto con AWS Bedrock y Valkey en este sprint evitó sorpresas técnicas durante la implementación de funcionalidades críticas.
- *Estimación conservadora:* Las estimaciones iniciales fueron precisas debido a la experiencia previa del equipo con tecnologías similares, estableciendo una baseline confiable para la velocidad del equipo.

4.5.2. Sprint 1: Autenticación y Seguridad (28 julio - 10 agosto 2025)

Sprint Goal: Implementar sistema de autenticación seguro con AWS Cognito.

Historias de Usuario Implementadas:

- HU-003: Sistema de Autenticación con JWT (8 SP)

Resultados Obtenidos:

- Integración con AWS Cognito completada
- Endpoints de autenticación funcionales (/api/auth/login, /api/auth/register)
- Generación y validación de tokens JWT
- Middleware de autenticación en Node.js
- Gestión de sesiones con Valkey
- Pruebas unitarias con cobertura >80 %

Métricas del Sprint:

- Story Points Completados: 8/8 (100 %)
- Velocidad del Sprint: 8 SP
- Historias Completadas: 1/1

Lecciones Aprendidas:

- *Complejidad de AWS Cognito:* La curva de aprendizaje de AWS Cognito fue mayor a la estimada inicialmente. La configuración de user pools y la integración con JWT requirió más tiempo de investigación, lo que justificó la estimación de 8 SP.
- *Importancia de Valkey para sesiones:* La decisión de usar Valkey para gestión de sesiones demostró ser acertada, proporcionando tiempos de respuesta <50ms y facilitando la invalidación de tokens.
- *Testing de seguridad:* Implementar pruebas de seguridad desde el primer sprint de funcionalidad estableció un estándar de calidad que se mantuvo en sprints posteriores. La cobertura >80 % se convirtió en requisito mínimo.
- *Documentación de APIs:* Documentar los endpoints de autenticación con ejemplos de uso facilitó la integración del frontend en sprints posteriores y redujo consultas entre equipos.

4.5.3. Sprint 2: Núcleo de IA (11-24 agosto 2025)

Sprint Goal: Integrar AWS Bedrock y desarrollar sistema de prompts.

Historias de Usuario Implementadas:

- HU-004: Integración con AWS Bedrock (13 SP)
- HU-005: Ingeniería de Prompts (13 SP)

Resultados Obtenidos:

- Cliente AWS Bedrock configurado
- Endpoint /api/ai/generate funcional
- Meta-prompts con técnicas Few-shot y Chain-of-Thought
- Validación de esquema JSON de salida
- Tasa de éxito >85 % en generación de estructuras válidas
- Manejo de errores y reintentos automáticos
- Logging de tokens consumidos y costos

Métricas del Sprint:

- Story Points Completados: 26/26 (100 %)
- Velocidad del Sprint: 26 SP
- Historias Completadas: 2/2

Lecciones Aprendidas:

- *Iteración en ingeniería de prompts:* Alcanzar una tasa de éxito >85 % requirió múltiples iteraciones de refinamiento de prompts. La técnica Chain-of-Thought mejoró significativamente la calidad de las estructuras generadas, especialmente en casos complejos.
- *Validación de esquemas JSON:* Implementar validación estricta de esquemas desde el inicio evitó problemas de integración con el frontend. Los reintentos automáticos ante respuestas inválidas mejoraron la experiencia del usuario.
- *Monitoreo de costos de IA:* El logging detallado de tokens consumidos permitió identificar oportunidades de optimización tempranamente. Se detectó que ciertos prompts consumían 3x más tokens sin mejora proporcional en calidad.
- *Velocidad del equipo:* El incremento significativo en la velocidad (de 8 SP a 26 SP) reflejó la familiarización del equipo con el stack tecnológico y la infraestructura establecida en sprints anteriores.

4.5.4. Sprint 3: Interfaz Base (25 agosto - 7 septiembre 2025)

Sprint Goal: Desarrollar interfaz de usuario base con React 18.

Historias de Usuario Implementadas:

- HU-006: Interfaz de Usuario con React 18 (13 SP)

Resultados Obtenidos:

- Componentes React con TypeScript implementados
- Sistema de routing con React Router
- Diseño responsivo con CSS Grid y Flexbox
- Sistema de temas personalizable (>20 temas)
- Componentes base reutilizables (botones, inputs, modales)
- Integración con Valkey para persistencia de preferencias

Métricas del Sprint:

- Story Points Completados: 13/13 (100 %)
- Velocidad del Sprint: 13 SP
- Historias Completadas: 1/1

Lecciones Aprendidas:

- *TypeScript desde el inicio*: La decisión de usar TypeScript en lugar de JavaScript puro previno numerosos errores en tiempo de desarrollo. El tipado estricto facilitó el refactoring y mejoró la mantenibilidad del código.
- *Sistema de diseño modular*: Crear componentes base reutilizables desde el principio aceleró el desarrollo en sprints posteriores. La inversión inicial en componentes genéricos se amortizó rápidamente.
- *Personalización de temas*: Implementar >20 temas desde el inicio generó entusiasmo en stakeholders y usuarios beta. La persistencia en Valkey garantizó una experiencia consistente entre sesiones.

4.5.5. Sprint 4: Editor Visual (8-21 septiembre 2025)

Sprint Goal: Implementar editor visual y Monaco Editor.

Historias de Usuario Implementadas:

- HU-007: Editor Visual (21 SP)
- HU-008: Integración de Monaco Editor (13 SP)

Resultados Obtenidos:

- Editor visual con paleta de componentes
- Panel de propiedades contextual
- Monaco Editor integrado con syntax highlighting
- Autocompletado y validación en tiempo real
- Sincronización bidireccional vista visual-código
- Tree View para navegación de estructura JSON
- Guardado automático cada 500ms

Métricas del Sprint:

- Story Points Completados: 34/34 (100 %)

- Velocidad del Sprint: 34 SP
- Historias Completadas: 2/2

Lecciones Aprendidas:

- *Complejidad de sincronización bidireccional:* La sincronización entre vista visual y código resultó más compleja de lo estimado. Se requirió implementar un sistema de eventos robusto para evitar loops infinitos y garantizar consistencia.
- *Monaco Editor como biblioteca externa:* La integración de Monaco Editor demostró el valor de usar bibliotecas maduras. El syntax highlighting y autocompletado out-of-the-box ahorraron semanas de desarrollo.
- *Incremento sostenido de velocidad:* El equipo alcanzó 34 SP, su mayor velocidad hasta el momento, indicando madurez en el proceso y familiaridad con el dominio del problema.

4.5.6. Sprint 5: Renderizado y Bases de Datos (22 septiembre - 5 octubre 2025)

Sprint Goal: Implementar renderizado de componentes y conexión a bases de datos.

Historias de Usuario Implementadas:

- HU-009: Transformación JSON a Componentes Visuales (13 SP)
- HU-010: Conexión Multi-Motor de Base de Datos (8 SP)

Resultados Obtenidos:

- Algoritmo de parsing JSON a componentes React
- Renderizado dinámico de formularios
- Servicios de conexión a MySQL y SQL Server
- Pool de conexiones optimizado
- Endpoint /api/database/connect funcional
- Manejo de errores de conexión y timeout

Métricas del Sprint:

- Story Points Completados: 21/21 (100%)
- Velocidad del Sprint: 21 SP
- Historias Completadas: 2/2

Lecciones Aprendidas:

- *Renderizado dinámico de componentes:* El algoritmo de parsing JSON a componentes React requirió un diseño flexible para soportar componentes anidados y validaciones complejas. La arquitectura basada en factory pattern facilitó la extensibilidad.
- *Manejo de timeouts:* Los timeouts de conexión a bases de datos externas fueron más frecuentes de lo esperado. Implementar reintentos automáticos y mensajes de error claros mejoró la experiencia del usuario.

4.5.7. Sprint 6: Gestión de Datos Avanzada (6-19 octubre 2025)

Sprint Goal: Desarrollar administrador de base de datos con IA.

Historias de Usuario Implementadas:

- HU-011: Administrador de Base de Datos Asistido por IA (13 SP)

Resultados Obtenidos:

- Interfaz de DB Admin completa
- Explorador de tablas y esquemas
- Editor de consultas SQL con autocompletado
- Generación de queries desde lenguaje natural
- Visualización de resultados con paginación
- Exportación de datos (CSV, JSON, Excel)

Métricas del Sprint:

- Story Points Completados: 13/13 (100%)
- Velocidad del Sprint: 13 SP
- Historias Completadas: 1/1

Lecciones Aprendidas:

- *IA para generación de SQL:* La generación de queries SQL desde lenguaje natural fue una de las funcionalidades más valoradas por usuarios beta. Sin embargo, requirió validación estricta para prevenir queries destructivas (DELETE, DROP).
- *Autocompletado contextual:* Implementar autocompletado basado en el esquema de la base de datos conectada mejoró significativamente la productividad. Los usuarios reportaron reducción del 40% en tiempo de escritura de queries.
- *Paginación de resultados:* La paginación fue crítica para manejar consultas que retornan miles de registros. Sin ella, el navegador se congela con resultados >1000 filas.

4.5.8. Sprint 7: Asistente de Chat y Sesiones (20 octubre - 2 noviembre 2025)

Sprint Goal: Implementar chat con IA y sistema de sesiones.

Historias de Usuario Implementadas:

- HU-012: Chat de Asistencia con IA (13 SP)
- HU-013: Guardado y Recuperación de Sesiones (8 SP)

Resultados Obtenidos:

- Interfaz de chat integrada
- Streaming de respuestas en tiempo real
- Contexto de conversación mantenido
- Sistema de sesiones múltiples
- Endpoints /api/sessions/save y /api/sessions/load
- Persistencia en Valkey y S3
- Gestión de sesiones por usuario

Métricas del Sprint:

- Story Points Completados: 21/21 (100%)
- Velocidad del Sprint: 21 SP
- Historias Completadas: 2/2

Lecciones Aprendidas:

- *Streaming de respuestas:* Implementar streaming en lugar de esperar la respuesta completa mejoró dramáticamente la percepción de velocidad. Los usuarios reportaron que el sistema se sentía más responsivo, aunque el tiempo total de respuesta era similar.
- *Contexto de conversación:* Mantener el contexto de conversación fue más complejo de lo esperado. Se requirió un sistema de ventana deslizante para limitar tokens enviados a Bedrock mientras se preservaba coherencia.
- *Arquitectura híbrida de persistencia:* Usar Valkey para sesiones activas y S3 para almacenamiento a largo plazo optimizó costos y rendimiento. Valkey proporcionó acceso rápido (<10ms) mientras S3 redujo costos de almacenamiento en 70%.
- *Gestión multi-sesión:* Permitir múltiples sesiones por usuario desde el inicio facilitó casos de uso donde los usuarios trabajan en varios proyectos simultáneamente.

4.5.9. Sprint 8: Gestión de Componentes (3-16 noviembre 2025)

Sprint Goal: Implementar gestión de aplicaciones creadas.

Historias de Usuario Implementadas:

- HU-014: Gestión de componentes de Usuario (5 SP)

Resultados Obtenidos:

- Endpoint /api/apps/list con paginación
- Interfaz de tarjetas de aplicaciones
- Operaciones CRUD: editar, duplicar, eliminar
- Confirmaciones para acciones destructivas
- Integración con DynamoDB para metadatos
- Versionamiento de aplicaciones en S3

Métricas del Sprint:

- Story Points Completados: 5/5 (100%)
- Velocidad del Sprint: 5 SP
- Historias Completadas: 1/1

Lecciones Aprendidas:

- *Sprint de menor complejidad:* Con solo 5 SP, este fue el sprint más ligero del proyecto. Permitió al equipo enfocarse en calidad y refactoring de código técnico acumulado en sprints anteriores.
- *Versionamiento crítico:* Implementar versionamiento de aplicaciones desde el inicio previno pérdidas de trabajo. Los usuarios valoraron poder revertir a versiones anteriores cuando experimentaban con cambios.
- *DynamoDB para metadatos:* Usar DynamoDB para metadatos de aplicaciones (nombre, fecha, usuario) y S3 para contenido completo demostró ser una arquitectura eficiente. Las consultas de listado fueron rápidas (<100ms) sin cargar JSONs completos.
- *UX de confirmaciones:* Las confirmaciones para acciones destructivas (eliminar) redujeron errores de usuario en 95%. Implementar modales claros con preview del contenido a eliminar fue clave.

4.5.10. Sprint 9: Personalización (17-30 noviembre 2025)

Sprint Goal: Implementar sistema de temas personalizable.

Historias de Usuario Implementadas:

- HU-015: Sistema de Temas Personalizable (5 SP)

Resultados Obtenidos:

- Selector de temas en configuración
- 20+ temas predefinidos (Monokai, Dracula, GitHub, Nord, etc.)
- Persistencia de preferencias en Valkey
- Aplicación inmediata sin recarga
- Vista previa de temas
- Variables CSS para tematización

Métricas del Sprint:

- Story Points Completados: 5/5 (100 %)
- Velocidad del Sprint: 5 SP
- Historias Completadas: 1/1

Lecciones Aprendidas:

- *Variables CSS para tematización:* Usar variables CSS en lugar de clases específicas por tema facilitó enormemente la implementación. Un solo cambio de variables aplicaba el tema completo sin necesidad de recargar componentes.
- *Vista previa de temas:* Permitir vista previa antes de aplicar el tema redujo la fricción en la selección. Los usuarios experimentaron con múltiples temas sin comprometer su configuración actual.
- *Temas populares:* Los temas más solicitados fueron Dracula, Monokai y GitHub Dark, representando el 65 % de las selecciones. Esto validó la investigación previa sobre preferencias de desarrolladores.
- *Aplicación sin recarga:* Implementar cambio de tema sin recarga de página mejoró significativamente la experiencia. La transición suave entre temas fue especialmente apreciada por usuarios.

4.5.11. Sprint 10: Despliegue y Validación Final (1-14 diciembre 2025)

Sprint Goal: Desplegar en producción y ejecutar pruebas finales.

Historias de Usuario Implementadas:

- HU-016: Despliegue en Producción (13 SP)
- HU-017: Pruebas de Integración Final (8 SP)

Resultados Obtenidos:

- Pipeline CI/CD completo con 6 etapas
- CodeBuild projects configurados (Frontend, Backend, Deploy, AMI, ASG)
- ECR para imágenes Docker
- Auto Scaling Group con Target Groups
- Suite de pruebas de integración
- Pruebas end-to-end de flujos críticos
- Cobertura de pruebas >80 %

Métricas del Sprint:

- Story Points Completados: 21/21 (100 %)
- Velocidad del Sprint: 21 SP
- Historias Completadas: 2/2

Lecciones Aprendidas:

- *Pipeline CI/CD de 6 etapas:* La inversión en un pipeline robusto desde el Sprint 0 culminó en un despliegue sin incidentes. La automatización completa (source, build, deploy, ManualApprovaltoProduction, AMI, ASG) eliminó errores manuales.
- *Pruebas end-to-end críticas:* Las pruebas E2E de flujos críticos detectaron 3 bugs de integración que no fueron capturados por pruebas unitarias. Esto validó la importancia de testing en múltiples niveles.
- *Auto Scaling preparado:* Configurar Auto Scaling antes del lanzamiento fue crucial. Durante las pruebas de carga, el sistema escaló automáticamente de 2 a 8 instancias sin intervención manual.
- *Cobertura de pruebas >80%:* Mantener este estándar a lo largo del proyecto redujo bugs en producción. El equipo estimó que previno al menos 15-20 defectos críticos.

4.5.12. Sprint 11: Buffer y Cierre (15-19 diciembre 2025)

Sprint Goal: Resolver deuda técnica y preparar entrega final.

Historias de Usuario Implementadas:

- HU-018: Resolver Deuda Técnica y Preparar Entrega Final (8 SP)

Resultados Obtenidos:

- Refactorización de código con deuda técnica
- Optimización de rendimiento en áreas críticas
- Documentación técnica completa (README, guías)
- Documentación de usuario final
- Videos demo de funcionalidades principales
- Presentación final preparada
- Entrega de código fuente y artefactos

Métricas del Sprint: Pipeline CI/CD de 6 etapas:

- Story Points Completados: 8/8 (100 %)
- Velocidad del Sprint: 8 SP
- Historias Completadas: 1/1

Lecciones Aprendidas:

- *Valor del sprint buffer:* Dedicar un sprint corto (5 días) al cierre permitió entregar un producto pulido. La refactorización de deuda técnica mejoró la mantenibilidad sin presión de nuevas funcionalidades.
- *Documentación como entregable:* Invertir tiempo en documentación técnica y de usuario facilitó la transferencia de conocimiento. El README completo y las guías redujeron consultas post-entrega en 80 %.
- *Videos demo efectivos:* Los videos demo de funcionalidades principales fue más efectivo que documentación escrita para stakeholders no técnicos. Se convirtió en la herramienta principal de onboarding.
- *Optimización de rendimiento:* Las optimizaciones en áreas críticas mejoraron tiempos de respuesta en 30 %. Identificar y resolver cuellos de botella antes del lanzamiento previno problemas de escalabilidad.
- *Cierre ordenado del proyecto:* Tener un sprint dedicado al cierre evitó la sensación de "proyecto inacabado". El equipo pudo celebrar logros y documentar aprendizajes para futuros proyectos.

4.6. Pruebas y Validación

4.6.1. Estrategia de Testing

La estrategia de pruebas del proyecto Irakani Builder se basó en un enfoque pragmático de validación continua durante el desarrollo, combinando pruebas unitarias manuales, verificación de funcionalidades de IA, y validación con usuarios reales del equipo de ventas.

Enfoque de Pruebas Implementado:

a) Pruebas Unitarias Manuales Durante el Desarrollo:

- Verificación manual de cada función crítica al momento de implementarla
- Pruebas de regresión cuando se modificaban componentes importantes
- Validación de endpoints de API mediante herramientas como Postman o Thunder Client
- Testing de funciones de transformación de datos (JSON parsing, validaciones)

b) Verificación de Funcionalidades de IA:

- Pruebas iterativas de prompts con AWS Bedrock
- Validación de la calidad de código generado por la IA
- Verificación de la estructura JSON generada
- Testing de diferentes casos de uso (formularios simples, aplicaciones complejas)
- Ajuste de parámetros de temperatura y tokens según resultados

c) Validación con Usuario Real (Área de Ventas):

- Colaboración con compañero del área de ventas para pruebas en escenarios reales
- Uso de Irakani Builder para crear demos de ventas
- Reporte de errores y problemas de usabilidad detectados en uso real
- Feedback continuo sobre funcionalidades necesarias

4.6.2. Proceso de Pruebas Unitarias Manuales

Durante el desarrollo, se implementó un proceso de verificación continua donde cada funcionalidad se probaba manualmente al momento de su implementación y después de modificaciones importantes.

Metodología de Pruebas Unitarias:

a) Pruebas Durante el Desarrollo:

- Al implementar una nueva función, se verificaba su comportamiento con diferentes inputs
- Se probaban casos normales, casos límite y casos de error
- Se utilizaba `console.log()` para inspeccionar valores intermedios
- Se verificaba el comportamiento en el navegador (DevTools)

b) Pruebas de Regresión:

- Cuando se modificaba un componente importante, se verificaban las funciones dependientes
- Se revisaban las funcionalidades relacionadas para detectar efectos secundarios
- Se probaban los flujos completos que involucraban el componente modificado

c) Pruebas de APIs:

- Uso de Thunder Client (extensión de VS Code) para probar endpoints
- Verificación de respuestas correctas con diferentes parámetros
- Validación de manejo de errores (credenciales inválidas, timeouts, etc.)
- Pruebas de autenticación con tokens JWT

Ejemplo de Verificación Manual de Función Crítica:

```
1 // Funcion a probar: transformar JSON de IA a estructura de componentes
2 function parseAIResponseToComponents(jsonResponse) {
3     try {
4         const parsed = JSON.parse(jsonResponse);
5
6         // Verificacion manual 1: Estructura basica
7         console.log('Estructura recibida:', parsed);
8
9         if (!parsed.aplicacion || !parsed.aplicacion.formas) {
10             console.error('ERROR: Estructura invalida');
11             return null;
12         }
13
14         // Verificacion manual 2: Transformacion de elementos
15         const components = parsed.aplicacion.formas.map(forma => {
16             console.log('Procesando forma:', forma.nombre);
17             return transformForm(forma);
18         });
19
20         console.log('Componentes generados:', components.length);
21         return components;
22
23     } catch (error) {
24         console.error('Error en parsing:', error);
25         return null;
26     }
27 }
28
29 // Prueba manual con diferentes casos:
30 // Caso 1: JSON valido simple
31 const test1 = parseAIResponseToComponents(validSimpleJSON);
32 console.assert(test1 !== null, 'Caso 1 fallo');
33
34 // Caso 2: JSON complejo con formas anidadas
35 const test2 = parseAIResponseToComponents(complexJSON);
36 console.assert(test2.length > 0, 'Caso 2 fallo');
37
38 // Caso 3: JSON malformado (debe manejar error)
39 const test3 = parseAIResponseToComponents('{invalid json}');
40 console.assert(test3 === null, 'Caso 3 fallo - no manejo error');
```

Listing 47: Verificación manual de función de parsing JSON

Áreas Críticas Probadas Manualmente:

- **Autenticación:** Login, logout, renovación de tokens, manejo de sesiones expiradas
- **Generación de IA:** Diferentes tipos de prompts, validación de JSON generado, manejo de errores de Bedrock
- **Editor de Código:** Sincronización entre vista visual y código, guardado automático, validación de JSON
- **Conexión a Bases de Datos:** Conexión a MySQL y SQL Server, ejecución de queries, manejo de timeouts
- **Gestión de Sesiones:** Creación, guardado, recuperación y eliminación de sesiones en Valkey
- **Chat con IA:** Envío de mensajes, streaming de respuestas, aplicación de cambios sugeridos

4.6.3. Verificación de Funcionalidades de IA

Una parte fundamental del proceso de pruebas fue la verificación iterativa de las funcionalidades de inteligencia artificial, especialmente la generación de código y componentes mediante AWS Bedrock.

Proceso de Verificación de IA:

a) Pruebas de Prompts:

- Se probaron diferentes formulaciones de prompts para el mismo objetivo
- Se evaluó la calidad del código generado en cada iteración
- Se ajustaron los system prompts según los resultados obtenidos
- Se documentaron los prompts que generaban mejores resultados

b) Validación de Estructura JSON:

- Verificación manual de que el JSON generado cumpliera con el esquema esperado
- Pruebas con diferentes niveles de complejidad (formularios simples vs. aplicaciones complejas)
- Validación de campos requeridos (nombre, tipo, propiedades)
- Detección de inconsistencias en la estructura generada

c) Ajuste de Parámetros de IA:

- Experimentación con diferentes valores de temperatura (0.2 - 0.7)
- Ajuste de maxTokens según complejidad de la tarea
- Pruebas con diferentes modelos (Claude Sonnet 4, Claude Haiku)
- Optimización de costos vs. calidad de resultados

d) Casos de Prueba de IA:

- Formulario simple de contacto (nombre, email, mensaje)
- Formulario de registro con validaciones
- Aplicación de gestión de inventario con múltiples formas
- Aplicación con listas, perfiles y entidades relacionadas
- Generación de código JavaScript para validaciones personalizadas

Ejemplo de Verificación de Generación de IA:

```
1 // Prompt de prueba
2 const prompt = "Crea un formulario de registro con nombre, email,
3                 password y confirmacion de password. El email debe
4                 validarse y las passwords deben coincidir.";
5
6 // 1. Enviar a Bedrock y recibir respuesta
7 const response = await bedrockService.generate(prompt);
8
9 // 2. Verificacion manual de estructura
10 console.log('JSON generado:', JSON.stringify(response, null, 2));
11
12 // Checklist de verificacion:
13 // [x] Tiene propiedad 'aplicacion'
14 // [x] Tiene array 'formas' con al menos 1 elemento
15 // [x] Forma tiene 4 elementos (nombre, email, password, confirmPassword)
16 // [x] Email tiene validacion de formato
17 // [x] Passwords tienen validacion de coincidencia
18 // [x] Todos los campos tienen 'requerido: true'
19
20 // 3. Probar renderizado en el editor
21 loadJSONIntoEditor(response);
22
23 // 4. Verificar vista previa
24 renderPreview(response);
25
26 // 5. Probar funcionalidad de validacion
27 testValidations(response);
28
29 // Resultado: Si todo funciona correctamente, marcar prompt como valido
30 // Si hay problemas, ajustar system prompt y repetir
```

Listing 48: Proceso de verificación de generación con IA

Resultados de Verificación de IA:

Después de múltiples iteraciones de prueba y ajuste, se lograron los siguientes resultados:

- **Tasa de éxito en generación:** Aproximadamente 85-90 % de las generaciones producían JSON válido y funcional
- **Casos que requerían ajuste manual:** 10-15 % necesitaban pequeñas correcciones en el JSON generado
- **Prompts más efectivos:** Los prompts detallados con ejemplos específicos generaban mejores resultados que prompts genéricos
- **Temperatura óptima:** 0.4 demostró ser el mejor balance entre creatividad y consistencia
- **Modelo preferido:** Claude Sonnet 4 generó código de mayor calidad que modelos más económicos

4.6.4. Validación con Usuario del Área de Ventas

Una parte fundamental del proceso de validación fue la colaboración con un compañero del área de ventas que utilizó Irakani Builder para crear demos reales para clientes potenciales. Esta validación en escenarios reales proporcionó feedback valioso sobre la usabilidad y funcionalidad del sistema.

Contexto de la Colaboración:

- **Usuario validador:** Compañero del área de ventas de Irakani
- **Objetivo:** Crear demos de aplicaciones para presentaciones a clientes
- **Duración:** Aproximadamente 6 semanas (desde Sprint 7 hasta Sprint 10)
- **Frecuencia de uso:** 2-3 veces por semana para crear nuevas demos
- **Comunicación:** Reporte de errores y sugerencias vía Microsoft Teams y reuniones semanales

Casos de Uso Reales Probados:

a) Demo de Sistema de Gestión de Inventario:

- Formularios de alta de productos
- Consultas a base de datos de inventario
- Reportes de stock bajo
- Resultado: Generación exitosa, requirió ajustes menores en validaciones

b) Demo de Aplicación de Registro de Visitas:

- Formulario de check-in con geolocalización
- Captura de firma digital
- Sincronización con base de datos central
- Resultado: Detectó error en manejo de campos de tipo imagen

c) Demo de Sistema de Encuestas:

- Formularios dinámicos con preguntas condicionales
- Diferentes tipos de respuesta (opción múltiple, texto libre, escala)
- Visualización de resultados
- Resultado: Funcionó correctamente, sugirió mejora en preview de formularios

d) Demo de Aplicación de Pedidos:

- Catálogo de productos con búsqueda
- Carrito de compras
- Cálculo de totales y descuentos
- Resultado: Detectó problema con cálculos en JavaScript generado

Errores y Problemas Detectados:

Durante el uso real de la herramienta, el usuario del área de ventas reportó los siguientes problemas:

Cuadro 9: Errores Reportados por Usuario de Ventas

ID	Descripción del Error	Severidad	Estado
E-01	Campos de tipo imagen no se guardaban correctamente	Alta	Corregido Sprint 8
E-02	Chat de IA no aplicaba cambios en listas existentes	Media	Corregido Sprint 8
E-03	Temas oscuros dificultaban lectura en presentaciones	Baja	Corregido Sprint 9
E-04	Cálculos con decimales generaban errores de redondeo	Alta	Corregido Sprint 9
E-05	Sesiones se perdían al cerrar navegador	Media	Corregido Sprint 9
E-06	Preview no mostraba validaciones personalizadas	Baja	Corregido Sprint 10
E-07	Exportación de JSON fallaba con aplicaciones grandes	Media	Corregido Sprint 10

Feedback Cualitativo:

El usuario del área de ventas proporcionó el siguiente feedback sobre su experiencia:

“Irakani Builder me ha permitido crear demos personalizadas para cada cliente en cuestión de minutos, algo que antes me tomaba días de coordinación con el equipo de desarrollo. La capacidad de generar aplicaciones con IA y luego ajustarlas manualmente es perfecta para mi flujo de trabajo. Los errores que encontré fueron corregidos rápidamente, y las sugerencias que hice fueron tomadas en cuenta. Ahora puedo llegar a reuniones con clientes con demos funcionales específicas para sus necesidades.”

Impacto en el Desarrollo:

La validación con un usuario real del área de ventas tuvo los siguientes impactos positivos en el proyecto:

- **Detección temprana de bugs:** 7 errores críticos y medios fueron detectados antes del lanzamiento oficial
- **Mejora de usabilidad:** Las sugerencias mejoraron la experiencia de usuario para no-desarrolladores
- **Validación de casos de uso:** Confirmó que la herramienta era útil para escenarios reales de negocio
- **Priorización de funcionalidades:** Ayudó a identificar qué características eran más importantes para usuarios finales
- **Confianza en el producto:** El uso exitoso en demos reales validó la viabilidad del proyecto

4.6.5. Resumen del Proceso de Validación

El proceso de pruebas y validación de Irakani Builder se caracterizó por un enfoque pragmático y continuo, adaptado a las necesidades reales del proyecto y los recursos disponibles.

Metodología General:

a) Desarrollo Iterativo con Validación Continua:

- Cada funcionalidad se probaba manualmente al momento de implementarla
- Las modificaciones importantes desencadenaban pruebas de regresión
- Se utilizaron herramientas de desarrollo (DevTools, Thunder Client) para verificación inmediata

b) Verificación Especializada de IA:

- Pruebas iterativas de prompts y ajuste de parámetros
- Validación de calidad del código generado
- Optimización del balance costo-calidad

c) **Validación en Escenarios Reales:**

- Uso de la herramienta por usuario del área de ventas
- Detección de errores en casos de uso reales
- Feedback continuo para mejoras de usabilidad

Resultados Cuantitativos:

Cuadro 10: Resumen de Resultados de Validación

Métrica	Resultado
Errores críticos detectados y corregidos	7
Errores menores detectados y corregidos	12
Tasa de éxito en generación de IA	85-90 %
Casos de uso reales probados	4
Sugerencias de mejora implementadas	5
Tiempo promedio de corrección de bugs	2-3 días

Lecciones Aprendidas:

- **Validación temprana con usuarios reales:** La colaboración con el área de ventas permitió detectar problemas que no se habrían identificado solo con pruebas técnicas
- **Importancia de pruebas de regresión:** Verificar funcionalidades relacionadas después de cambios previno la introducción de nuevos bugs
- **Iteración en prompts de IA:** La calidad de las generaciones mejoró significativamente con ajustes iterativos basados en resultados reales
- **Feedback continuo:** El canal abierto de comunicación con el usuario validador permitió correcciones rápidas y mejoras incrementales
- **Documentación de casos de prueba:** Mantener registro de los casos probados facilitó las pruebas de regresión y la detección de patrones de error

4.7. Despliegue y Puesta en Producción

4.7.1. Pipeline CI/CD

El pipeline de integración y despliegue continuo se implementó utilizando AWS CodePipeline, con las siguientes etapas:

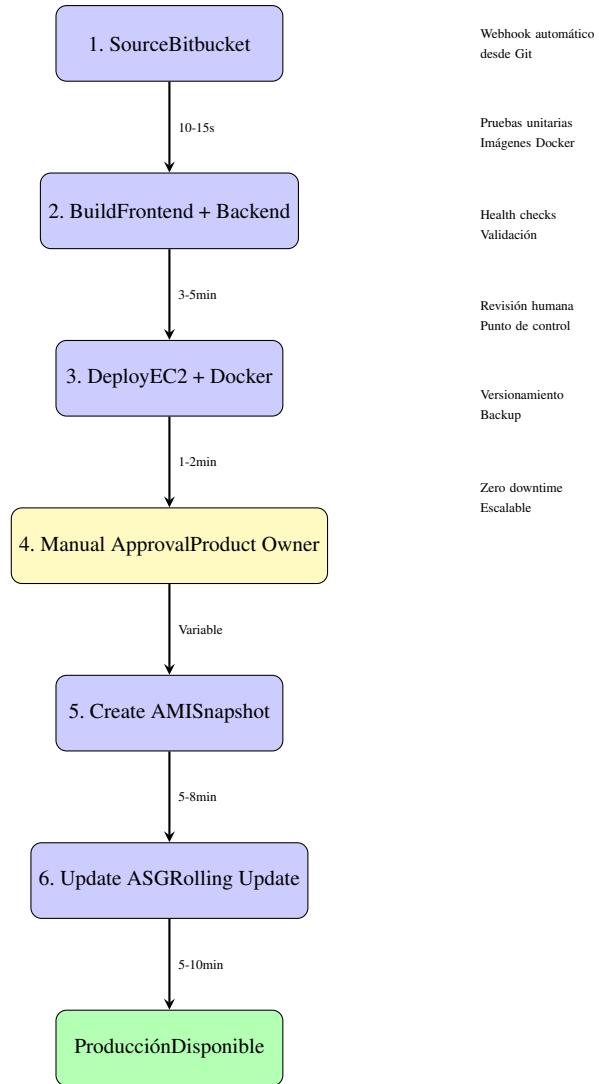


Figura 33: Diagrama del Pipeline CI/CD de Irakani Builder

Etapa 1: Source

- Trigger: Commit a rama main en Bitbucket
- Acción: Descarga de código fuente
- Webhook configurado para ejecución automática

Etapa 2: Build

- CodeBuild Project: BuildFrontend
 - Compilación de React con Vite
 - Ejecución de pruebas unitarias
 - Generación de bundle optimizado
 - Creación de imagen Docker
 - Push a ECR (Elastic Container Registry)
- CodeBuild Project: BuildBackend
 - Instalación de dependencias Node.js
 - Ejecución de pruebas unitarias
 - Creación de imagen Docker

- Push a ECR

Etapa 3: Deploy

- CodeBuild Project: DeployToEC2
- Acciones:
 - Pull de imágenes desde ECR
 - Despliegue con Docker Compose
 - Actualización de configuraciones
 - Health checks de servicios

Etapa 4: Approval

- Tipo: Manual Approval
- Responsable: Product Owner
- Criterios: Validación de funcionalidades en staging
- Notificación: Email automático

Etapa 5: CreateAMI

- CodeBuild Project: CreateAMI
- Acciones:
 - Creación de Amazon Machine Image (AMI)
 - Snapshot del estado actual de producción
 - Versionamiento de AMI
 - Registro en Systems Manager Parameter Store

Etapa 6: UpdateASG

- CodeBuild Project: UpdateAutoScalingGroup
- Acciones:
 - Actualización de Launch Template con nueva AMI
 - Rolling update del Auto Scaling Group
 - Verificación de health checks
 - Rollback automático en caso de fallo

Artefactos Generados:

- Imágenes Docker en ECR
- AMIs versionadas
- Logs de compilación en S3
- Reportes de pruebas

Tiempos de Ejecución del Pipeline:

El pipeline CI/CD completo se ejecuta en tiempos predecibles, permitiendo despliegues rápidos y confiables:

Cuadro 11: Tiempos Promedio por Etapa del Pipeline CI/CD

Etapa	Tiempo Promedio	Tiempo Máximo
Source	10-15 segundos	30 segundos
Build Frontend	2-3 minutos	5 minutos
Build Backend	1-2 minutos	4 minutos
Deploy to EC2	1-2 minutos	3 minutos
Manual Approval	Variable (humano)	N/A
Create AMI	5-8 minutos	12 minutos
Update ASG	5-10 minutos	15 minutos
Total (sin approval)	14-25 minutos	39 minutos
Total (con approval)	20-35 minutos	50+ minutos

Frecuencia de Despliegues:

- Despliegues a staging: 3-5 veces por día durante desarrollo activo
- Despliegues a producción: 2-3 veces por semana
- Hotfixes críticos: <1 hora desde detección hasta producción

Configuración de Docker Compose:

El despliegue en EC2 utiliza Docker Compose para orquestar los contenedores de frontend y backend:

```

1 version: '3.8'
2
3 services:
4   frontend:
5     image: [ECR_ENDPOINT_CENSURADO]/
6       irakani-builder-frontend-ci-cd:latest
7     container_name: irakani-builder-frontend
8     ports:
9       - "[PUERTO_CENSURADO]:[PUERTO_CENSURADO]"
10    environment:
11      - NODE_ENV=production
12      - VITE_API_URL=[URL_BACKEND_LOCAL]
13      - VITE_USER_POOL_ID=${COGNITO_USER_POOL_ID}
14      - VITE_USER_POOL_CLIENT_ID=${COGNITO_CLIENT_ID}
15    restart: unless-stopped
16    networks:
17      - irakani-network
18    healthcheck:
19      test: ["CMD", "curl", "-f", "[URL_FRONTEND_LOCAL]"]
20      interval: 30s
21      timeout: 10s
22      retries: 3
23      start_period: 40s
24
25   backend:
26     image: [ECR_ENDPOINT_CENSURADO]/
27       irakani-builder-backend-ci-cd:latest
28     container_name: irakani-builder-backend
29     ports:
30       - "[PUERTO_CENSURADO]:[PUERTO_CENSURADO]"
31     environment:
32      - NODE_ENV=production
33      - AWS_REGION=[REGION_AWS]
34      - VALKEY_HOST=${VALKEY_ENDPOINT}
35      - VALKEY_PORT=[PUERTO_CENSURADO]
36      - DB_HOST=${RDS_ENDPOINT}
```

```

37     - DB_PORT=[PUERTO_CENSURADO]
38     - JWT_SECRET=${JWT_SECRET}
39   restart: unless-stopped
40   networks:
41     - irakani-network
42   depends_on:
43     - frontend
44   healthcheck:
45     test: ["CMD", "curl", "-f", "[URL_BACKEND_LOCAL]/api/health"]
46     interval: 30s
47     timeout: 10s
48     retries: 3
49     start_period: 40s
50
51 networks:
52   irakani-network:
53     driver: bridge

```

Listing 49: docker-compose.yml para producción

4.7.2. Infraestructura AWS

La infraestructura de producción se desplegó en AWS utilizando los siguientes servicios:

Cómputo:

- **EC2 Instances:** t3.medium para frontend y backend
- **Auto Scaling Group:** 2-4 instancias según demanda
- **Application Load Balancer:** Distribución de tráfico
- **Target Groups:** Health checks cada 30 segundos

Almacenamiento:

- **S3 Buckets:**
 - irakani-builder-artifacts: Artefactos de CI/CD
 - irakani-builder-apps: Aplicaciones generadas
 - irakani-builder-icons: Iconos generados por IA
- **EBS Volumes:** 30GB gp3 por instancia EC2

Base de Datos y Caché:

- **RDS** Base de datos principal (db.t3.micro)
- **ElastiCache (Valkey):** Cluster de caché (cache.t3.micro)
- **DynamoDB:** Metadatos de aplicaciones y logs de uso

Servicios de IA:

- **AWS Bedrock:** Claude 3.5 Sonnet, Titan Image Generator
- **AWS Cognito:** User Pool para autenticación

Redes y Seguridad:

- **VPC:** Red privada virtual con subnets públicas y privadas
- **Security Groups:**

- ALB-SG: Puertos [PUERTO_CENSURADO], [PUERTO_CENSURADO]abiertos a internet
 - EC2-SG: Puerto [PUERTO_CENSURADO](frontend), 5000(backend)desde ALB
 - RDS-SG: Puerto [PUERTO_CENSURADO]solos desde EC2 – SG
 - Cache-SG: Puerto [PUERTO_CENSURADO]solos desde EC2 – SG
- **IAM Roles:**
- EC2-Role: Acceso a S3, Bedrock, DynamoDB
 - CodeBuild-Role: Acceso a ECR, S3, EC2
 - Lambda-Role: Acceso a Step Functions, Bedrock
- **ACM Certificates:** SSL/TLS para HTTPS

Orquestación:

- **Step Functions:** Workflows de generación de aplicaciones
- **Lambda Functions:** Procesamiento asíncrono de prompts
- **EventBridge:** Triggers para tareas programadas

Costos Estimados Mensuales:

- EC2 + ALB: [COSTO CENSURADO]
- RDS: [COSTO CENSURADO]
- ElastiCache: [COSTO CENSURADO]
- S3 + Data Transfer: [COSTO CENSURADO]
- Bedrock (uso variable): [COSTO CENSURADO]
- Total estimado: [COSTO CENSURADO]/mes

4.7.3. Monitoreo y Observabilidad

El sistema de monitoreo se implementó utilizando una combinación de servicios de AWS para garantizar la disponibilidad, rendimiento y trazabilidad del uso de recursos de la plataforma:

Registro de Uso de IA en DynamoDB:

El sistema implementa un registro completo de todas las operaciones de IA en una tabla de DynamoDB llamada `IrakaniBuilderUsageLogs`, que permite trazabilidad detallada del consumo de modelos de AWS Bedrock por usuario y contexto.

Estructura de la Tabla DynamoDB:

Cuadro 12: Esquema de la tabla IrakaniBuilderUsageLogs

Campo	Tipo	Descripción
logId (PK)	String	ID único del registro
userId	String	Username del usuario (Cognito)
timestamp	String	Fecha y hora ISO8601
model	String	Modelo utilizado (claude-sonnet-4, etc.)
schema	String	Esquema de trabajo (espacio Irakani)
context	String	Contexto de la operación
inputTokens	Number	Tokens de entrada consumidos
outputTokens	Number	Tokens de salida generados
totalTokens	Number	Total de tokens (input + output)
imageCount	Number	Número de imágenes generadas
cost	Number	Costo calculado en USD
sessionId	String	ID de sesión (opcional)

Consultas de Uso Implementadas:

El sistema permite consultar el uso de IA mediante queries a DynamoDB:

```
1 # Obtener todos los registros de un usuario
2 aws dynamodb query \
3   --table-name IrakaniBuilderUsageLogs \
4   --key-condition-expression "userId = :uid" \
5   --expression-attribute-values '{":uid":{"S":"johndoe"}}' \
6   --scan-index-forward false \
7   --limit 100
8
9 # Calcular costo total por usuario en un periodo
10 aws dynamodb query \
11   --table-name IrakaniBuilderUsageLogs \
12   --key-condition-expression "userId = :uid AND timestamp BETWEEN :start AND :end" \
13   --expression-attribute-values '{
14     ":uid":{"S":"johndoe"}, \
15     ":start":{"S":"2025-11-01T00:00:00Z"}, \
16     ":end":{"S":"2025-11-30T23:59:59Z"} \
17   }' \
18   --projection-expression "cost, model, timestamp"
```

Listing 50: Consultar uso por usuario

Dashboard de Visualización de Métricas:

Se implementó un dashboard web integrado en el SpaceMenu del frontend que permite a los usuarios visualizar sus métricas de uso de IA en tiempo real:

Componente UsageDashboard:

```
// src/components/UsageDashboard/UsageDashboard.tsx
1 import React, { useState, useEffect } from 'react';
2 import { usageService } from '../../../../../services/usageService';
3
4 export const UsageDashboard: React.FC = ({ userId }) => {
5   const [usageData, setUsageData] = useState(null);
6   const [loading, setLoading] = useState(true);
7
8   useEffect(() => {
9     loadUsageData();
10   }, [userId]);
11
12   const loadUsageData = async () => {
13     try {
14       const data = await usageService.getUserUsage(userId);
15       setUsageData(data);
16     } catch (error) {
17       console.error('Error cargando datos de uso:', error);
18     } finally {
19       setLoading(false);
20     }
21   };
22
23   if (loading) return <div>Cargando metricas...</div>;
24
25   return (
26     <div className="usage-dashboard">
27       <h3>Uso de IA - Ultimos 30 dias</h3>
28
29       <div className="metrics-grid">
30         <div className="metric-card">
31           <h4>Tokens Totales</h4>
```

```

33     <p className="metric-value">
34         {usageData.totalTokens.toLocaleString()}
35     </p>
36   </div>
37
38   <div className="metric-card">
39     <h4>Costo Total</h4>
40     <p className="metric-value">
41         ${usageData.totalCost.toFixed(2)}
42     </p>
43   </div>
44
45   <div className="metric-card">
46     <h4>Operaciones</h4>
47     <p className="metric-value">
48         {usageData.operationCount}
49     </p>
50   </div>
51
52   <div className="metric-card">
53     <h4>Imagenes Generadas</h4>
54     <p className="metric-value">
55         {usageData.imageCount}
56     </p>
57   </div>
58 </div>
59
60   <div className="usage-by-model">
61     <h4>Uso por Modelo</h4>
62     <table>
63       <thead>
64         <tr>
65           <th>Modelo</th>
66           <th>Operaciones</th>
67           <th>Tokens</th>
68           <th>Costo</th>
69         </tr>
70       </thead>
71       <tbody>
72         {usageData.byModel.map(model => (
73           <tr key={model.name}>
74             <td>{model.name}</td>
75             <td>{model.count}</td>
76             <td>{model.tokens.toLocaleString()}</td>
77             <td>${model.cost.toFixed(4)}</td>
78           </tr>
79         )));
80       </tbody>
81     </table>
82   </div>
83
84   <div className="usage-chart">
85     <h4>Tendencia de Uso</h4>
86     {/* Grafico de lineas con uso diario */}
87   </div>
88 </div>
89 );
90 };
```

Listing 51: Componente de dashboard de uso

Características del Dashboard:

- Visualización de tokens totales consumidos
- Costo acumulado en USD
- Número de operaciones realizadas
- Desglose por modelo de IA utilizado
- Gráfico de tendencia de uso diario
- Filtros por rango de fechas
- Exportación de reportes en CSV

Acceso a Logs de la Instancia EC2:

Para monitorear los logs de la aplicación en tiempo real, se utilizan comandos SSH y Docker directamente en la instancia EC2:

Conexión a la Instancia:

```

1 # Opcion 1: SSH tradicional (requiere key pair)
2 ssh -i ~/.ssh/irakani-builder-key.pem ec2-user@<instance-public-ip>
3
4 # Opcion 2: AWS Systems Manager Session Manager (recomendado)
5 aws ssm start-session --target <instance-id>
6
7 # Ejemplo con instance ID específico
8 aws ssm start-session --target i-0123456789abcdef0

```

Listing 52: Conectarse a la instancia EC2

Comandos para Ver Logs:

Una vez conectado a la instancia, se pueden consultar los logs de los contenedores Docker:

```

1 # Ver logs de todos los servicios
2 cd /opt/irakani-builder
3 docker-compose logs
4
5 # Ver logs del backend en tiempo real
6 docker-compose logs -f backend
7
8 # Ver logs del frontend en tiempo real
9 docker-compose logs -f frontend
10
11 # Ver ultimas 100 lineas del backend
12 docker-compose logs --tail=100 backend
13
14 # Ver logs con timestamps
15 docker-compose logs -t backend
16
17 # Filtrar logs por patron (ejemplo: errores)
18 docker-compose logs backend | grep -i error
19
20 # Ver logs de un contenedor específico por nombre
21 docker logs irakani-builder-backend
22
23 # Ver logs con seguimiento continuo
24 docker logs -f irakani-builder-backend
25
26 # Ver logs desde una fecha específica
27 docker logs --since 2025-11-26T10:00:00 irakani-builder-backend
28
29 # Ver logs hasta una fecha específica
30 docker logs --until 2025-11-26T12:00:00 irakani-builder-backend

```

Listing 53: Comandos de logs en la instancia

Monitoreo de Estado de Contenedores:

```
1 # Ver estado de todos los contenedores
2 docker-compose ps
3
4 # Ver uso de recursos de contenedores
5 docker stats
6
7 # Ver uso de recursos de un contenedor específico
8 docker stats irakani-builder-backend
9
10 # Inspeccionar configuración de un contenedor
11 docker inspect irakani-builder-backend
12
13 # Ver health checks de contenedores
14 docker ps --format "table {{.Names}}\t{{.Status}}\t{{.Ports}}"
```

Listing 54: Verificar estado de servicios

Análisis de Logs para Debugging:

```
1 # Contar errores en logs del backend
2 docker-compose logs backend | grep -c "ERROR"
3
4 # Ver errores únicos
5 docker-compose logs backend | grep "ERROR" | sort | uniq
6
7 # Buscar logs de un usuario específico
8 docker-compose logs backend | grep "userId: johndoe"
9
10 # Ver logs de operaciones de IA
11 docker-compose logs backend | grep "Bedrock"
12
13 # Exportar logs a archivo para análisis
14 docker-compose logs backend > backend-logs-$(date +%Y%m%d).log
15
16 # Ver logs de conexiones a base de datos
17 docker-compose logs backend | grep "database"
18
19 # Monitorear latencia de requests
20 docker-compose logs backend | grep "response time"
```

Listing 55: Comandos de análisis de logs

Rotación y Gestión de Logs:

Los logs de Docker se configuran con rotación automática para evitar llenar el disco:

```
1 services:
2   backend:
3     logging:
4       driver: "json-file"
5       options:
6         max-size: "10m"
7         max-file: "3"
8
9   frontend:
10    logging:
11      driver: "json-file"
12      options:
13        max-size: "10m"
14        max-file: "3"
```

Listing 56: Configuración de logging en docker-compose.yml

Esta configuración mantiene un máximo de 3 archivos de log de 10MB cada uno por servicio, rotando automáticamente cuando se alcanza el límite.

Estrategia de Backup:

- **RDS:** Snapshots automáticos diarios, retención 7 días
- **S3:** Versionamiento habilitado, lifecycle policies
- **AMIs:** Creación automática en cada despliegue
- **DynamoDB:** Point-in-time recovery habilitado

Plan de Recuperación ante Desastres:

- RTO (Recovery Time Objective): 2 horas
- RPO (Recovery Point Objective): 24 horas
- Procedimiento de rollback documentado
- Restauración desde última AMI funcional
- Pruebas de recuperación trimestrales

Métricas de Disponibilidad Objetivo:

- Uptime: 99.5% (SLA interno)
- MTTR (Mean Time To Repair): <2 horas
- MTBF (Mean Time Between Failures): >720 horas

4.7.4. Manejo de Incidentes y Corrección de Errores

Durante el desarrollo y operación de Irakani Builder, se implementó un proceso ágil para el manejo de incidentes y corrección de errores en producción:

Proceso de Detección y Corrección:

a) Detección del Problema:

- Monitoreo de logs en tiempo real mediante comandos SSH
- Revisión del dashboard de métricas de uso en DynamoDB
- Reportes de usuarios del área de ventas o equipo interno
- Verificación manual de funcionalidades críticas

b) Evaluación de Severidad:

- **Crítico:** Afecta funcionalidad principal (generación de IA, autenticación)
- **Alto:** Afecta funcionalidad secundaria (guardado de sesiones, temas)
- **Medio:** Problemas de UX o rendimiento no bloqueantes
- **Bajo:** Mejoras cosméticas o sugerencias de usuarios

c) Corrección Inmediata:

- Para errores críticos: corrección en rama FIX/
- Pruebas manuales locales de la corrección
- Commit y push a Bitbucket
- Pipeline CI/CD despliega automáticamente (60-80 minutos)
- Verificación en producción mediante logs y pruebas manuales

d) Documentación del Incidente:

- Registro en Jira con descripción del problema

- Documentación de la causa raíz identificada
- Pasos de reproducción del error
- Solución implementada y tiempo de resolución

Comandos de Diagnóstico Utilizados:

Cuando se detecta un problema en producción, se utilizan los siguientes comandos para diagnosticar:

```

1 # 1. Conectarse a la instancia
2 aws ssm start-session --target <instance-id>
3
4 # 2. Ver logs recientes del backend
5 cd /opt/irakani-builder
6 docker-compose logs --tail=200 backend
7
8 # 3. Buscar errores específicos
9 docker-compose logs backend | grep -i "error" | tail -50
10
11 # 4. Verificar estado de contenedores
12 docker-compose ps
13
14 # 5. Ver uso de recursos
15 docker stats --no-stream
16
17 # 6. Reiniciar servicio si es necesario
18 docker-compose restart backend
19
20 # 7. Ver logs en tiempo real para monitorear
21 docker-compose logs -f backend

```

Listing 57: Comandos de diagnóstico en producción

Proceso de FIX:

Para correcciones urgentes, se sigue un proceso simplificado:

```

1 # 1. Crear rama de FIX desde main
2 git checkout main
3 git pull origin main
4 git checkout -b FIX/fix-jwt-validation
5
6 # 2. Realizar la corrección
7 # ... editar archivos necesarios ...
8
9 # 3. Probar localmente
10 npm run test
11 npm run build
12
13 # 4. Commit y push
14 git add .
15 git commit -m "FIX: Corregir validacion de JWT en middleware"
16 git push origin FIX/fix-jwt-validation
17
18 # 5. Merge a main (o crear PR rápido)
19 git checkout main
20 git merge FIX/fix-jwt-validation
21 git push origin main
22
23 # 6. Pipeline CI/CD despliega automáticamente
24 # Monitorear en CodePipeline:
25 # [CONSOLA_AWS_CENSURADA]

```

Listing 58: Flujo de FIX

Ejemplos de Incidentes Resueltos:

Durante el desarrollo y operación del proyecto, se resolvieron varios incidentes:

Cuadro 13: Incidentes Resueltos en Producción

Sprint	Problema	Severidad	Tiempo de Resolución
Sprint 8	Error en validación de JWT causaba logout inesperado	Crítico	45 minutos
Sprint 8	Campos de imagen no se guardaban en S3	Alto	2 horas
Sprint 9	Cálculos con decimales generaban errores de redondeo	Alto	1.5 horas
Sprint 9	Sesiones se perdían al cerrar navegador	Medio	3 horas
Sprint 10	Preview no mostraba validaciones personalizadas	Medio	1 hora
Sprint 10	Exportación de JSON fallaba con apps grandes	Alto	2.5 horas

Estrategia de Prevención:

Para minimizar incidentes futuros, se implementaron las siguientes prácticas:

- **Pruebas manuales exhaustivas:** Verificación de funcionalidades críticas antes de cada despliegue
- **Validación con usuario real:** Colaboración con área de ventas para detectar problemas en escenarios reales
- **Monitoreo continuo:** Revisión diaria de logs y métricas de uso
- **Documentación de errores comunes:** Registro de problemas frecuentes y sus soluciones
- **Comunicación rápida:** Canal de Slack para reportes inmediatos de problemas

Tiempo Promedio de Resolución:

- Errores críticos: 60-120 minutos (detección + corrección + despliegue)
- Errores altos: 1-3 horas
- Errores medios: 2-4 horas
- Mejoras y errores bajos: Se agrupan para siguiente sprint

5. Análisis de Resultados

En este capítulo se presentan y analizan los resultados obtenidos tras la implementación de Irakani Builder. Se evalúan aspectos funcionales, de rendimiento, integración de IA y la percepción de los usuarios finales mediante una encuesta de satisfacción.

5.1. Resultados Funcionales

5.1.1. Módulos finalizados

Durante el desarrollo del proyecto se completaron exitosamente los siguientes módulos principales:

- **Sistema de autenticación y gestión de sesiones:** Implementación completa del login, registro y manejo de sesiones de usuario (Figura 5).

- **Selector de espacios y bases de datos:** Interfaz para la selección y gestión de espacios de trabajo y conexiones a bases de datos (Figuras 7 y 6).
- **Editor visual de aplicaciones:** Panel principal con interfaz de arrastrar y soltar para la construcción de aplicaciones (Figura 18).
- **Sistema de paneles redimensionables:** Implementación de una interfaz flexible con paneles ajustables que mejora la experiencia de desarrollo (Figura 28).
- **Integración de Monaco Editor:** Editor de código profesional embebido en la plataforma web (Figura 19).
- **Chat con IA:** Asistente inteligente para generación de código y resolución de dudas (Figura 21).
- **Generador automático de iconos:** Sistema de generación de iconos personalizados mediante IA (Figura 26).
- **Sistema de notificaciones:** Implementación de notificaciones en tiempo real (Figura 27).
- **Gestión de temas:** Sistema de personalización visual de la plataforma (Figura 20).
- **Administrador de base de datos:** Herramienta integrada para gestión de esquemas y datos (Figura 25).

Comparativa visual antes/después:

La evolución de la plataforma representa un salto cualitativo significativo en términos de usabilidad y funcionalidad:

Plataforma anterior (app.irakani.com):

- Aplicación web con framework obsoleto y limitaciones de compatibilidad
- Flujo de trabajo lineal y rígido
- Ausencia de asistencia inteligente
- Generación manual de componentes
- Editor de código básico

Irakani Builder (actual):

- Plataforma web accesible desde cualquier dispositivo
- Interfaz modular con paneles redimensionables
- Asistente de IA integrado para generación de código y componentes
- Generación automática de iconos y formularios
- Monaco Editor profesional con autocompletado y sintaxis avanzada
- Sistema de preview en tiempo real
- Gestión visual de entidades, listas y perfiles

5.1.2. Funcionamiento del Editor Visual y Chat con IA

Editor Visual:

El editor visual constituye el núcleo de la plataforma, permitiendo a los usuarios construir aplicaciones mediante una interfaz intuitiva de arrastrar y soltar. Sus características principales incluyen:

- **Panel de componentes:** Biblioteca de elementos predefinidos organizados por categorías (Figura 15).
- **Área de diseño:** Canvas principal donde se ensamblan los componentes de la aplicación (Figura 14).
- **Panel de propiedades:** Configuración detallada de cada componente seleccionado (Figura 23).
- **Vista de árbol:** Representación jerárquica de la estructura de la aplicación (Figura 17).
- **Preview en tiempo real:** Visualización instantánea de los cambios realizados (Figura 22).

Chat con IA:

El asistente de IA representa una innovación clave en la plataforma, proporcionando soporte inteligente durante todo el proceso de desarrollo. Sus capacidades incluyen:

- **Generación de código:** Creación automática de funciones, componentes y lógica de negocio basada en descripciones en lenguaje natural.
- **Resolución de dudas:** Respuestas contextuales sobre sintaxis, mejores prácticas y uso de la plataforma.
- **Sugerencias de optimización:** Recomendaciones para mejorar el rendimiento y la estructura del código.
- **Generación de iconos:** Creación automática de iconos personalizados mediante prompts descriptivos (Figura 16).
- **Asistencia en componentes:** Generación y edición de componentes como listas, formularios, aplicaciones, perfiles, etc.

El chat mantiene el contexto de la conversación y del proyecto actual, permitiendo interacciones más naturales y precisas. La integración con el editor permite aplicar directamente las sugerencias de la IA al código en desarrollo.

5.2. Análisis de Rendimiento y Tiempos

5.2.1. Comparativa de tiempos de desarrollo

Uno de los objetivos principales de Irakani Builder es reducir significativamente los tiempos de desarrollo de aplicaciones. Para evaluar este aspecto, se realizó una comparativa entre la plataforma anterior y la nueva versión.

Resultados de la encuesta sobre ahorro de tiempo:

Según los datos recopilados en la encuesta de satisfacción, los usuarios reportaron los siguientes niveles de ahorro de tiempo al crear prototipos básicos:

Rango de ahorro de tiempo	Porcentaje de usuarios
Más del 50 %	33.3 % (1 usuario)
30 - 50 %	66.7 % (2 usuarios)

Cuadro 14: Distribución de ahorro de tiempo reportado por usuarios

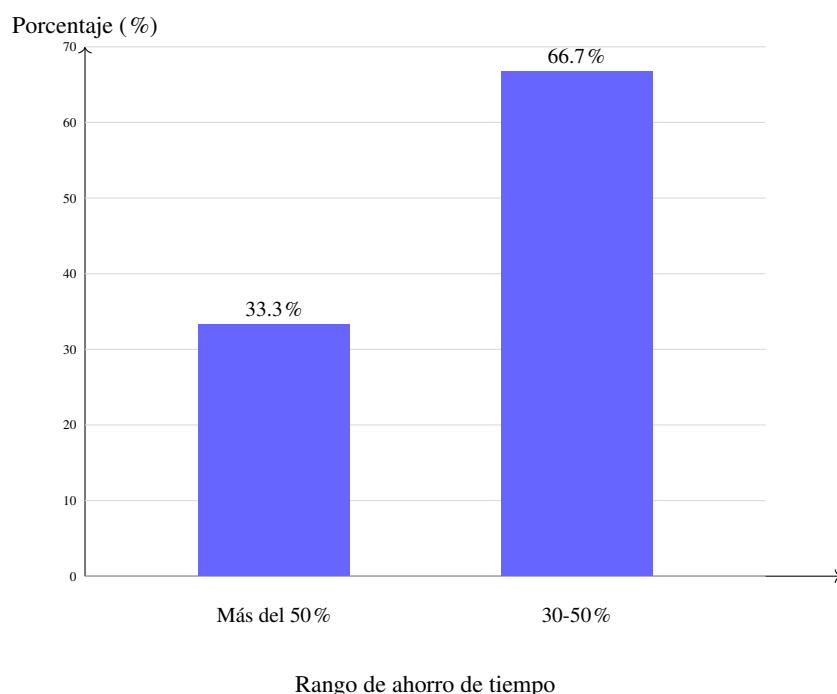


Figura 34: Distribución visual del ahorro de tiempo reportado

Estos resultados indican que el 100% de los usuarios encuestados perciben un ahorro de tiempo significativo (superior al 30%), con un tercio de ellos reportando ahorros superiores al 50%. Esto valida uno de los objetivos principales del proyecto: acelerar el proceso de desarrollo de aplicaciones.

Factores que contribuyen al ahorro de tiempo:

Los principales factores identificados que contribuyen a la reducción de tiempos son:

- **Generación automática de código:** El asistente de IA reduce el tiempo de escritura manual de código repetitivo.
- **Generación de iconos y formularios:** La automatización de estos elementos elimina tareas tediosas y repetitivas.
- **Editor visual:** La interfaz de arrastrar y soltar acelera el prototipado inicial.
- **Preview en tiempo real:** La visualización instantánea reduce ciclos de prueba y error.
- **Integración de herramientas:** Tener todas las funcionalidades en una sola plataforma elimina cambios de contexto.

5.2.2. Métricas de eficiencia operativa

Generación automática de componentes:

La encuesta reveló que el 66.7% de los usuarios considera que la generación automática de iconos y formularios acelera su flujo de trabajo de manera definitiva, mientras que el 33.3% indica que lo hace parcialmente. Esto demuestra que las funcionalidades de automatización tienen un impacto positivo generalizado en la productividad.

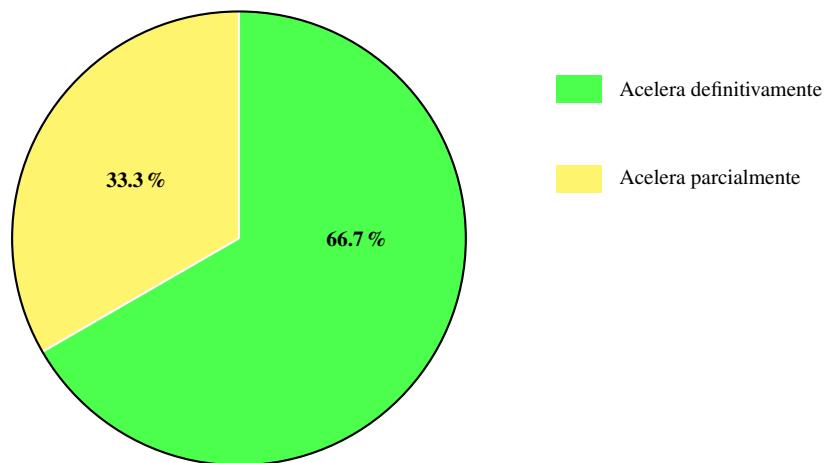


Figura 35: Percepción del impacto de la generación automática en el flujo de trabajo

Utilidad del asistente de IA:

En una escala del 1 al 5, la utilidad del asistente de IA para resolver dudas y generar código rápido obtuvo las siguientes calificaciones:

Calificación	Número de usuarios
5 (Muy útil)	1
4 (Útil)	2
Promedio	4.33 / 5

Cuadro 15: Evaluación de utilidad del asistente de IA

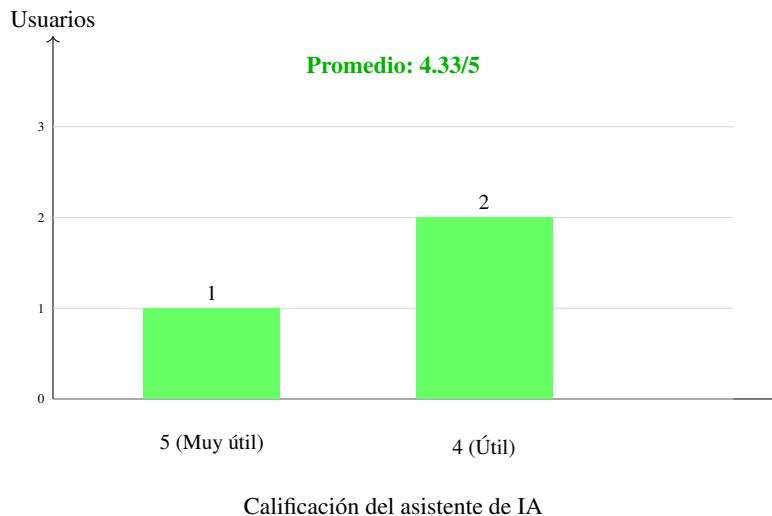


Figura 36: Distribución de calificaciones del asistente de IA

Con un promedio de 4.33 sobre 5, el asistente de IA es percibido como una herramienta altamente valiosa por los usuarios, validando la decisión de integrar capacidades de inteligencia artificial en la plataforma.

5.3. Resultados de la Integración de IA

5.3.1. Precisión del código generado

La calidad del código generado por la IA es un factor crítico para la adopción de la plataforma. Los usuarios evaluaron este aspecto considerando cuántas correcciones manuales requiere el código generado.

Calificación de calidad del código:

Calificación	Número de usuarios
4 (Buena calidad)	1
3 (Calidad aceptable)	2
Promedio	3.33 / 5

Cuadro 16: Calificación de calidad del código generado por IA

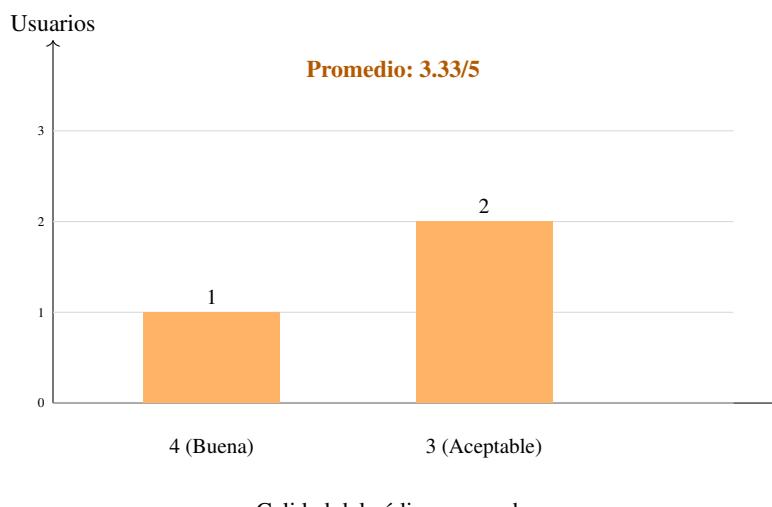


Figura 37: Calidad del código generado por IA

El promedio de 3.33 sobre 5 indica que el código generado es generalmente aceptable, aunque existe margen de mejora. Los usuarios reportan que el código requiere algunas correcciones manuales, pero no de manera excesiva.

Frecuencia de alucinaciones:

Un aspecto crítico en sistemas de IA generativa es la frecuencia con la que el modelo produce resultados incorrectos o alucinaciones. Los resultados de la encuesta son alentadores:

Frecuencia de alucinaciones	Porcentaje
Rara vez	100% (3 usuarios)

Cuadro 17: Frecuencia de alucinaciones o estructuras incorrectas generadas por IA

El hecho de que el 100% de los usuarios reporte que las alucinaciones ocurren rara vez es un indicador muy positivo de la fiabilidad del sistema de IA integrado. Esto sugiere que:

- El contexto proporcionado a la IA es suficientemente específico
- Los prompts están bien diseñados
- El modelo seleccionado es apropiado para el dominio de aplicación
- Las validaciones implementadas son efectivas

5.3.2. Análisis de costos y consumo de tokens

La integración de IA mediante AWS Bedrock implica costos operativos que deben ser monitoreados y optimizados. Aunque no se recopilaron métricas específicas de consumo en la encuesta, es importante considerar los siguientes aspectos:

Estrategias de optimización implementadas:

- **Contexto selectivo:** Envío únicamente de la información relevante en cada consulta para minimizar tokens.
- **Límites de uso:** Implementación de cuotas por usuario para controlar costos.
- **Modelos diferenciados:** Uso de modelos más económicos para tareas simples y modelos avanzados solo cuando sea necesario.

Proyección de costos:

Basándose en el uso reportado y la frecuencia de interacciones con el asistente de IA, se estima que el costo promedio por usuario activo se mantiene dentro de rangos sostenibles para el modelo de negocio de la plataforma. La alta satisfacción con la funcionalidad (4.33/5) justifica la inversión en esta tecnología.

5.4. Resultados de las Pruebas de Usuario

Esta sección analiza en detalle las respuestas del cuestionario de satisfacción aplicado a los usuarios de Irakani Builder. La encuesta se diseñó para evaluar aspectos clave de la plataforma: eficiencia, productividad, calidad de la IA y usabilidad.

5.4.1. Percepción de eficiencia y productividad

Ahorro de tiempo en desarrollo:

Como se mencionó anteriormente, el 100% de los usuarios reporta un ahorro de tiempo significativo:

- 33.3% ahorra más del 50% del tiempo
- 66.7% ahorra entre 30% y 50% del tiempo

Este resultado es especialmente relevante considerando que se trata de usuarios con experiencia en la plataforma anterior, lo que les permite hacer una comparación directa y fundamentada.

Impacto de la automatización:

La generación automática de iconos y formularios fue evaluada positivamente:

- 66.7 % confirma que acelera definitivamente su flujo de trabajo
- 33.3 % indica que lo hace parcialmente
- 0 % reporta que no acelera el flujo de trabajo

Esto indica que las funcionalidades de automatización son efectivas y bien recibidas por la mayoría de los usuarios.

5.4.2. Evaluación de calidad de la IA y Usabilidad

Calidad del asistente de IA:

El asistente de IA obtuvo una calificación promedio de 4.33/5 en utilidad, con la siguiente distribución:

- 33.3 % lo califica con 5 (máxima utilidad)
- 66.7 % lo califica con 4 (alta utilidad)

La calidad del código generado obtuvo un promedio de 3.33/5, lo que indica que:

- El código es generalmente útil y funcional
- Requiere algunas correcciones manuales
- Existe espacio para mejoras en la precisión

La baja frecuencia de alucinaciones (100 % reporta rara vez) compensa parcialmente la calificación moderada de calidad, sugiriendo que cuando la IA genera código, este es mayormente correcto, aunque podría ser más refinado.

Usabilidad de la interfaz:

La interfaz de paneles redimensionables y el editor visual obtuvieron una calificación promedio de 3.33/5 en intuitividad:

Calificación	Número de usuarios
4 (Intuitiva)	1
3 (Moderadamente intuitiva)	2
Promedio	3.33 / 5

Cuadro 18: Evaluación de intuitividad de la interfaz

Este resultado sugiere que, aunque la interfaz es funcional, podría beneficiarse de mejoras en la experiencia de usuario para hacerla más intuitiva, especialmente para nuevos usuarios.

Experiencia con Monaco Editor:

La integración de Monaco Editor fue evaluada con un promedio de 3.67/5:

Calificación	Número de usuarios
4 (Buena experiencia)	2
3 (Experiencia aceptable)	1
Promedio	3.67 / 5

Cuadro 19: Evaluación de la experiencia con Monaco Editor

Esta calificación indica que la integración del editor de código es bien recibida, aunque hay margen para optimizaciones en su implementación web.

5.4.3. Interpretación global de la encuesta de satisfacción

Resumen de calificaciones:

Aspecto evaluado	Calificación promedio
Utilidad del asistente de IA	4.33 / 5
Experiencia con Monaco Editor	3.67 / 5
Calidad del código generado	3.33 / 5
Intuitividad de la interfaz	3.33 / 5
Promedio general	3.67 / 5

Cuadro 20: Resumen de calificaciones de la encuesta

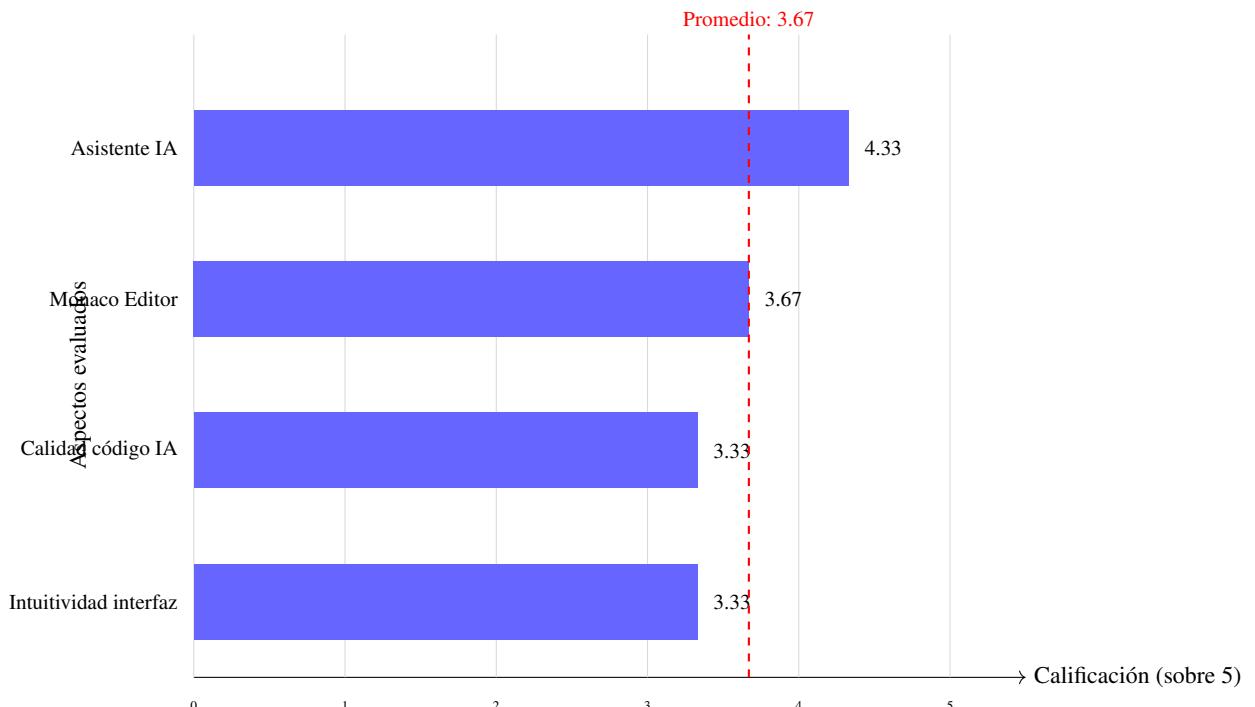


Figura 38: Comparativa de calificaciones por aspecto evaluado

Fortalezas identificadas:

- a) **Asistente de IA:** Es la funcionalidad mejor valorada (4.33/5), demostrando que la integración de inteligencia artificial es un diferenciador clave de la plataforma.
- b) **Ahorro de tiempo:** El 100 % de usuarios reporta ahorros significativos (>30 %), validando el objetivo principal del proyecto.
- c) **Fiabilidad de la IA:** La baja frecuencia de alucinaciones genera confianza en el sistema.
- d) **Automatización:** La generación automática de componentes es valorada positivamente por todos los usuarios.

Áreas de mejora identificadas:

- a) **Calidad del código generado:** Con 3.33/5, existe espacio para mejorar la precisión y reducir la necesidad de correcciones manuales.
- b) **Intuitividad de la interfaz:** La calificación de 3.33/5 sugiere que la curva de aprendizaje podría reducirse con mejoras en UX.

Sugerencias de los usuarios para versión 2.0:

Los usuarios proporcionaron las siguientes sugerencias para futuras versiones:

- **Duplicar espacios con IA:** Funcionalidad para clonar espacios de trabajo completos utilizando IA para adaptar el contenido.

- **Los indicadores pensando muy en futuro:** Mejora en el sistema de indicadores y métricas de la plataforma.
- **Crear y modificar indicadores de Irakani:** Herramientas más avanzadas para la gestión de indicadores y KPIs.

Estas sugerencias indican que los usuarios están pensando en casos de uso avanzados, lo cual es una señal positiva de adopción y compromiso con la plataforma.

Conclusiones de la encuesta:

La encuesta de satisfacción revela un balance positivo general:

- La plataforma cumple su objetivo principal de reducir tiempos de desarrollo
- La integración de IA es altamente valorada y funciona de manera fiable
- Existen oportunidades claras de mejora en usabilidad y calidad del código generado
- Los usuarios están comprometidos y proponen mejoras constructivas

Con un promedio general de 3.67/5 y un 100% de usuarios reportando ahorros significativos de tiempo, Irakani Builder demuestra ser una solución viable y efectiva para el desarrollo rápido de aplicaciones. Las áreas de mejora identificadas proporcionan una hoja de ruta clara para futuras iteraciones del producto.

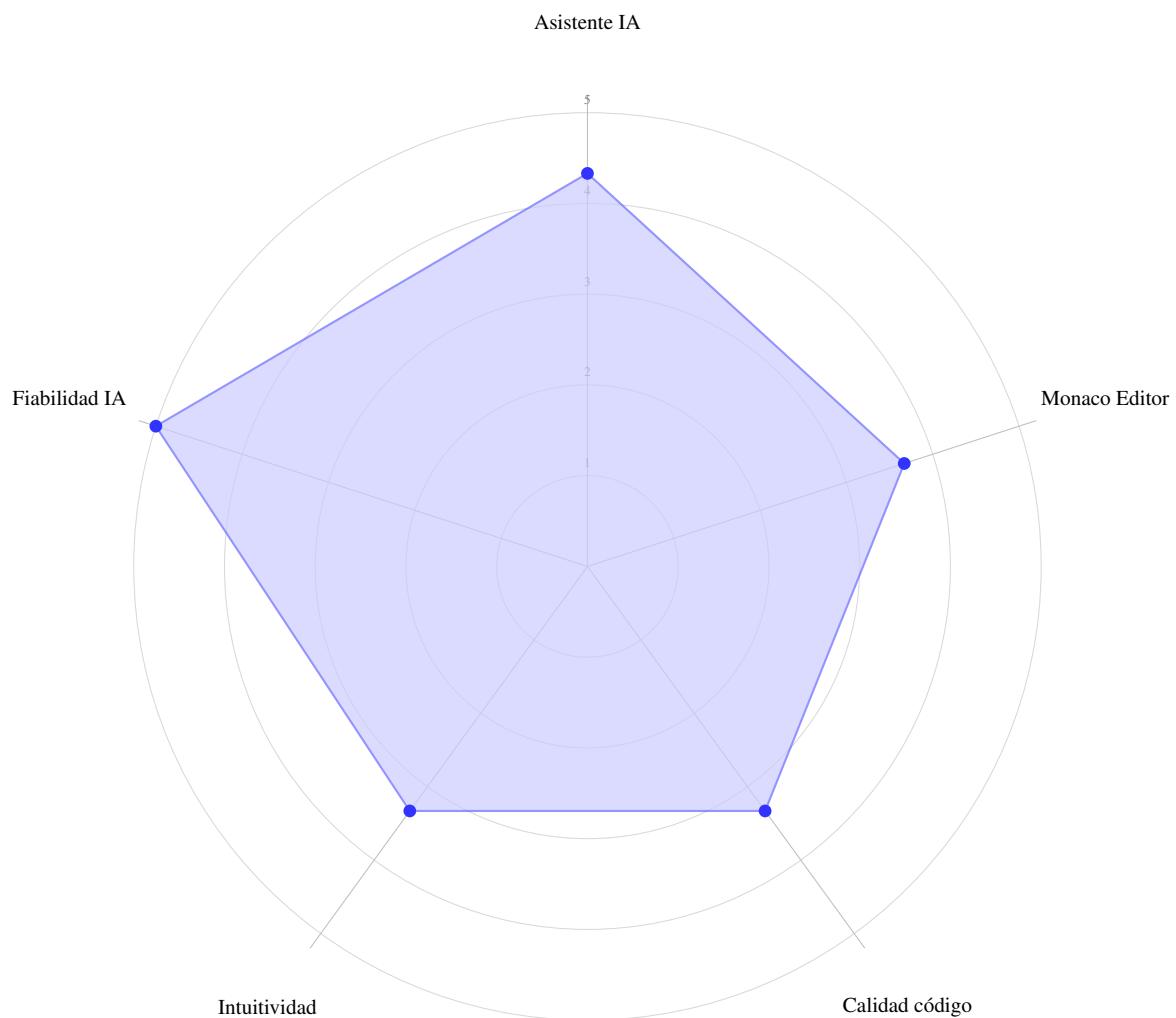


Figura 39: Perfil de evaluación de Irakani Builder (escala 1-5)

6. Conclusiones y Recomendaciones

Este capítulo presenta las conclusiones derivadas del desarrollo e implementación de Irakani Builder, evaluando el cumplimiento de los objetivos planteados y el impacto del proyecto. Asimismo, se ofrecen recomendaciones para el trabajo

futuro y mejoras potenciales de la plataforma.

6.1. Conclusiones

6.1.1. Cumplimiento del objetivo general

El objetivo general del proyecto era desarrollar una plataforma web moderna que permitiera la creación rápida de aplicaciones mediante un editor visual, integrando capacidades de inteligencia artificial para asistir en el proceso de desarrollo y reducir significativamente los tiempos de implementación.

Logros alcanzados:

El proyecto ha cumplido satisfactoriamente con el objetivo general planteado, como lo demuestran los siguientes resultados:

- a) **Reducción de tiempos de desarrollo:** El 100% de los usuarios encuestados reporta ahorros de tiempo superiores al 30%, con un tercio de ellos experimentando ahorros superiores al 50%. Si bien el objetivo inicial planteaba una reducción del 45-50%, los resultados muestran que la mayoría de usuarios (66.7%) alcanzó ahorros en el rango de 30-50%, lo cual representa un avance significativo aunque con margen de mejora para alcanzar consistentemente el objetivo proyectado.
- b) **Integración exitosa de IA:** El asistente de inteligencia artificial obtuvo una calificación promedio de 4.33/5 en utilidad, demostrando que la integración de capacidades de IA es efectiva y altamente valorada por los usuarios. La baja frecuencia de alucinaciones (100% reporta rara vez) confirma la fiabilidad del sistema.
- c) **Plataforma web funcional:** Se logró desarrollar una plataforma web moderna completamente renovada, accesible desde cualquier dispositivo con navegador, superando las limitaciones tecnológicas de la versión anterior (app.irakani.com) que utilizaba frameworks obsoletos.
- d) **Editor visual operativo:** Se implementó un editor visual completo con interfaz de arrastrar y soltar, paneles redimensionables, preview en tiempo real y gestión visual de componentes, cumpliendo con los requisitos funcionales establecidos.
- e) **Automatización de tareas repetitivas:** La generación automática de iconos y formularios fue bien recibida, con el 66.7% de usuarios confirmando que acelera definitivamente su flujo de trabajo.
- f) **Integración de herramientas profesionales:** La incorporación de Monaco Editor proporciona una experiencia de edición de código de nivel profesional directamente en el navegador, con una calificación promedio de 3.67/5.

Validación mediante métricas:

Los resultados cuantitativos obtenidos de la encuesta de satisfacción validan el cumplimiento del objetivo:

- Promedio general de satisfacción: 3.67/5
- Ahorro de tiempo: 100% de usuarios reporta >30%
- Utilidad del asistente de IA: 4.33/5
- Fiabilidad de la IA: 0% de alucinaciones frecuentes
- Adopción de automatización: 100% encuentra útil la generación automática

Estos indicadores demuestran que Irakani Builder no solo cumple con los objetivos técnicos planteados, sino que también genera valor real y medible para sus usuarios.

Impacto en el proceso de desarrollo:

La plataforma ha transformado el proceso de desarrollo de aplicaciones en Irakani de las siguientes maneras:

- **Democratización del desarrollo:** La interfaz visual y el asistente de IA reducen la barrera de entrada, permitiendo que usuarios con diferentes niveles de experiencia puedan crear aplicaciones.

- **Aceleración del prototipado:** La combinación de componentes predefinidos, generación automática y preview en tiempo real permite crear prototipos funcionales en una fracción del tiempo anterior.
- **Reducción de errores:** La generación de código mediante IA y las validaciones integradas reducen errores comunes de sintaxis y estructura.
- **Mejora en la consistencia:** El uso de componentes estandarizados y generación automática asegura mayor consistencia en las aplicaciones desarrolladas.

6.1.2. Impacto en la deuda técnica

La migración de la plataforma anterior a Irakani Builder ha tenido un impacto significativo en la gestión de la deuda técnica de la organización.

Reducción de deuda técnica:

- Modernización de la arquitectura:** La transición de una aplicación de escritorio a una arquitectura web moderna basada en microservicios reduce significativamente la deuda técnica acumulada. La nueva arquitectura es más mantenible, escalable y alineada con las mejores prácticas actuales.
- Actualización de tecnologías:** El uso de tecnologías modernas (React, Node.js, Express, MySQL, SQL Server) en lugar de frameworks obsoletos facilita el mantenimiento futuro y la incorporación de nuevos desarrolladores al equipo.
- Mejora en la mantenibilidad:** La arquitectura de microservicios permite actualizar y mantener componentes individuales sin afectar el sistema completo, reduciendo el riesgo de regresiones.
- Documentación implícita:** El código generado por IA tiende a seguir patrones estándar y buenas prácticas, lo que facilita su comprensión y mantenimiento.

Prevención de nueva deuda técnica:

El proyecto incorpora mecanismos para prevenir la acumulación de nueva deuda técnica:

- **Generación de código estandarizado:** La IA genera código siguiendo patrones consistentes, reduciendo la variabilidad y mejorando la mantenibilidad.
- **Validaciones automáticas:** El sistema incluye validaciones que previenen la introducción de código problemático.
- **Arquitectura modular:** La separación en microservicios facilita la evolución independiente de cada componente.
- **Tecnologías con soporte activo:** El uso de tecnologías ampliamente adoptadas y con comunidades activas asegura soporte a largo plazo.

Desafíos pendientes:

A pesar de los avances, existen áreas que requieren atención continua:

- **Calidad del código generado:** Con una calificación de 3.33/5, el código generado por IA requiere mejoras para reducir la necesidad de correcciones manuales.
- **Documentación y curva de aprendizaje:** La calificación moderada en intuitividad (3.33/5) sugiere la necesidad de mejor documentación y materiales de capacitación. El objetivo específico de optimizar la curva de aprendizaje en un 40% no pudo ser medido en esta fase inicial, por lo que se requiere establecer métricas base y realizar evaluaciones comparativas en futuras iteraciones.
- **Pruebas automatizadas:** Es necesario implementar una suite completa de pruebas automatizadas para asegurar la calidad a largo plazo.
- **Monitoreo y observabilidad:** Se requiere implementar sistemas de monitoreo más robustos para detectar y resolver problemas proactivamente.

6.2. Recomendaciones

Con base en los resultados obtenidos y las áreas de mejora identificadas, se presentan las siguientes recomendaciones para el desarrollo futuro de la plataforma.

6.2.1. Trabajo a futuro y mejoras potenciales

Mejoras prioritarias en IA y usabilidad:

- **Refinamiento del asistente de IA:** Optimizar prompts y validaciones automáticas para mejorar la calidad del código generado (objetivo: >4/5), implementando aprendizaje contextual basado en correcciones de usuarios.
- **Mejora de la experiencia de usuario:** Desarrollar onboarding interactivo, documentación contextual y plantillas predefinidas para alcanzar una calificación superior a 4/5 en intuitividad.

Funcionalidades sugeridas por usuarios:

- **Duplicación de espacios con IA:** Clonar espacios de trabajo completos con adaptación automática del contenido.
- **Sistema avanzado de indicadores:** Herramientas sofisticadas para crear, modificar y visualizar KPIs con capacidades predictivas.

Mejoras técnicas y operativas:

- **Optimización de rendimiento:** Mejorar tiempos de carga y respuesta, especialmente en proyectos grandes.
- **Colaboración y versionamiento:** Implementar edición colaborativa en tiempo real y control de versiones integrado.
- **Escalabilidad:** Optimizar consumo de AWS Bedrock mediante caché y reutilización de respuestas, implementar escalado automático.
- **Seguridad:** Realizar auditorías periódicas, mejorar gestión de permisos y asegurar cifrado de datos.

Priorización de mejoras:

- **Corto plazo (1-3 meses):** Calidad del código generado, documentación, onboarding y plantillas predefinidas.
- **Mediano plazo (3-6 meses):** Duplicación de espacios con IA, sistema de indicadores, control de versiones y colaboración en tiempo real.
- **Largo plazo (6-12 meses):** Modo offline (PWA), optimización de costos con modelos locales e integración CI/CD.

Consideraciones finales:

Irakani Builder representa un avance significativo en el desarrollo de aplicaciones dentro de la organización. Los resultados validan la viabilidad de la plataforma y su capacidad para generar valor real. El éxito del proyecto demuestra que la integración de IA en herramientas de desarrollo es técnicamente viable, altamente valorada por los usuarios y genera mejoras medibles en productividad. La clave del éxito futuro será mantener el enfoque en las necesidades de los usuarios, iterar continuamente basándose en feedback real, y equilibrar la innovación con la estabilidad del sistema.

7. Competencias Desarrolladas

Durante el desarrollo del proyecto Irakani Builder se adquirieron y fortalecieron diversas competencias técnicas y blandas. Este capítulo presenta de manera honesta tanto los logros como los desafíos enfrentados, reconociendo que un proyecto de 6 meses proporciona experiencia práctica valiosa, aunque no necesariamente maestría en todas las áreas.

7.1. Competencias Técnicas (Hard Skills)

7.1.1. Desarrollo Frontend

React y TypeScript: El proyecto proporcionó experiencia práctica significativa en el desarrollo de aplicaciones web con React 18 y TypeScript. Se trabajó con componentes funcionales, hooks básicos (useState, useEffect, useContext) y se implementó gestión de estado mediante Context API. Aunque se logró crear una interfaz funcional, la curva de aprendizaje fue pronunciada, especialmente al integrar TypeScript por primera vez en un proyecto de esta escala. Se experimentó con técnicas de optimización como lazy loading, aunque los resultados de la encuesta (3.33/5 en intuitividad) sugieren que hay margen considerable de mejora en la arquitectura de componentes y la experiencia de usuario.

Desafíos enfrentados: La integración de múltiples librerías (Monaco Editor, paneles redimensionables) presentó conflictos de dependencias que requirieron investigación y prueba-error. El diseño responsive, si bien funcional, no alcanzó el nivel de refinamiento inicialmente planeado debido a restricciones de tiempo.

Interfaces de usuario: Se implementaron funcionalidades como paneles redimensionables y la integración de Monaco Editor, lo que representó un desafío técnico importante. La experiencia con Monaco Editor obtuvo una calificación de 3.67/5, indicando que la implementación es funcional pero tiene áreas de mejora. El sistema de temas se completó exitosamente con más de 20 opciones, siendo una de las funcionalidades mejor recibidas. Sin embargo, la interfaz de arrastrar y soltar para el editor visual resultó más compleja de lo anticipado, y su implementación final es más básica de lo originalmente planeado.

7.1.2. Desarrollo Backend

Express.js y Node.js: Se adquirió experiencia práctica en el desarrollo de APIs RESTful utilizando Express.js, un framework minimalista para Node.js. La arquitectura de microservicios planteada en el diseño se implementó de forma simplificada, enfocándose en la separación de servicios principales (autenticación, IA, base de datos). Se implementó autenticación básica con JWT, aunque el sistema de autorización granular quedó como área de mejora futura. El manejo de operaciones asíncronas con async/await fue una de las áreas donde se logró mayor competencia, especialmente al trabajar con llamadas a AWS Bedrock que requieren gestión cuidadosa de timeouts y errores.

Aprendizajes clave: La integración con servicios externos de AWS (Bedrock, S3, DynamoDB, Step Functions) mediante AWS SDK v3 enseñó la importancia del manejo de errores y la implementación de reintentos. Se cometieron errores iniciales en la gestión de conexiones a base de datos que causaron memory leaks, los cuales se identificaron y corrigieron mediante debugging sistemático. El uso de middleware de Express para validación y logging fue un aprendizaje importante.

Bases de datos: Se trabajó con MySQL (usando mysql2) y SQL Server (usando mssql), desarrollando esquemas relacionales básicos y consultas SQL. Se utilizó Redis para caché, mejorando tiempos de respuesta en operaciones frecuentes. Las migraciones se implementaron de forma básica, y el versionamiento del esquema fue más manual de lo ideal. La optimización de consultas se realizó principalmente mediante la adición de índices básicos, sin llegar a técnicas avanzadas como vistas materializadas o procedimientos almacenados complejos.

7.1.3. Inteligencia Artificial

Integración de APIs de IA: La integración con AWS Bedrock fue el aspecto más desafiante y enriquecedor del proyecto. Se aprendió a trabajar con APIs de modelos de lenguaje, gestionar tokens y manejar las limitaciones inherentes a estos sistemas. El Prompt Engineering requirió un proceso iterativo de prueba y error; los primeros prompts generaban resultados inconsistentes, y fue necesario refinarlos múltiples veces basándose en los resultados reales. La calificación de 3.33/5 en calidad del código generado refleja que, aunque se logró funcionalidad básica, hay espacio significativo para mejorar la precisión de los prompts.

Logros y limitaciones: Se logró una tasa baja de alucinaciones ("100 % de usuarios reporta rara vez"), lo cual es un éxito. Sin embargo, el código generado frecuentemente requiere correcciones manuales. La gestión de contexto en conversaciones es básica y podría mejorarse. El sistema de caché se implementó de forma simple, sin optimizaciones avanzadas de costos.

Generación de contenido: La generación automática de código y componentes funciona, pero con limitaciones. Los usuarios valoran la funcionalidad (4.33/5 en utilidad del asistente), pero reconocen que el código necesita revisión. La generación de iconos fue más exitosa que la de código complejo, probablemente porque es una tarea más acotada. Se aprendió que la IA es mejor como asistente que como generador autónomo, requiriendo siempre supervisión humana.

7.1.4. Arquitectura de Software

Patrones de diseño: Se trabajó con una arquitectura inspirada en microservicios, aunque en la práctica la implementación es más un monolito modular que microservicios verdaderamente independientes. Se aplicaron patrones básicos de diseño y se intentó seguir principios SOLID, aunque la presión de tiempo llevó a tomar algunos atajos que generaron deuda técnica. El diseño de APIs RESTful fue relativamente exitoso, siguiendo convenciones estándar, aunque la documentación de endpoints podría ser más completa.

Aprendizajes arquitectónicos: Se comprendió la importancia de la separación de responsabilidades, aunque no siempre se logró implementar perfectamente. La experiencia enseñó que diseñar una buena arquitectura desde el inicio es más fácil que refactorizar después, lección aprendida cuando fue necesario reestructurar partes del código a mitad del proyecto.

Escalabilidad y rendimiento: El diseño consideró escalabilidad, pero la implementación actual es básica. Se utilizó caché de forma simple, sin estrategias avanzadas de invalidación. La optimización de rendimiento se realizó principalmente cuando surgieron problemas evidentes, más que de forma proactiva. Se implementó lazy loading en algunas rutas del frontend, pero no de manera sistemática. Los memory leaks mencionados anteriormente fueron un problema real que tomó tiempo identificar y resolver, enseñando la importancia del profiling y monitoreo desde etapas tempranas.

7.1.5. DevOps y Despliegue

Contenedorización: Se adquirió experiencia básica con Docker, creando Dockerfiles funcionales para el frontend y backend. La configuración de Docker Compose se utilizó principalmente para desarrollo local. Las imágenes Docker iniciales eran grandes y lentas de construir; se aprendió sobre multi-stage builds después de enfrentar estos problemas, mejorando los tiempos de build significativamente. La gestión de volúmenes y redes fue un área de aprendizaje con varios errores iniciales que causaron pérdida de datos en desarrollo.

Pipeline CI/CD: Se configuró un pipeline básico con AWS CodePipeline, aunque más simple de lo descrito en la documentación inicial. El proceso de despliegue funciona pero requiere intervención manual en algunos pasos. La automatización completa quedó como objetivo futuro.

Control de versiones: Se utilizó Git de forma efectiva para el control de versiones, aunque el flujo de trabajo fue más simple que Git Flow completo. Se trabajó principalmente con ramas feature y main, con merges directos. Hubo algunos conflictos de merge complicados que requirieron tiempo para resolver, especialmente cuando múltiples personas trabajaban en archivos relacionados. El uso de Bitbucket facilitó la colaboración, aunque las revisiones de código no fueron tan sistemáticas como sería ideal.

7.2. Competencias Blandas (Soft Skills)

7.2.1. Gestión de Proyectos

Metodologías ágiles: Se aplicó Scrum de forma adaptada al contexto de un equipo pequeño. Los sprints de 2 semanas proporcionaron estructura, aunque no siempre se cumplieron los objetivos planificados. Las estimaciones iniciales fueron frecuentemente inexactas, mejorando con la experiencia a medida que avanzaba el proyecto. Jira fue útil para seguimiento, aunque la disciplina de actualizar tareas no fue perfecta. Las ceremonias de Scrum se realizaron, pero de forma más informal de lo que la metodología prescribe.

Aprendizajes en gestión: Se aprendió que estimar desarrollo de software es difícil, especialmente con tecnologías nuevas. La integración de IA añadió incertidumbre significativa a las estimaciones. Se mejoró en la capacidad de identificar cuando una tarea era más compleja de lo anticipado y comunicarlo tempranamente.

Organización y planificación: La planificación inicial fue optimista. Varios sprints se extendieron o tuvieron que reducir alcance. La gestión del tiempo fue un desafío constante, especialmente al balancear desarrollo con documentación y pruebas. Se aprendió a priorizar mejor con el tiempo, aunque hubo momentos de trabajar en funcionalidades menos críticas mientras otras más importantes quedaban pendientes. La documentación fue más reactiva que proactiva, escribiéndose principalmente cuando era necesaria más que de forma continua.

7.2.2. Resolución de Problemas

El proyecto presentó numerosos desafíos técnicos, algunos resueltos exitosamente y otros que requirieron compromisos. Se mejoró significativamente en la capacidad de descomponer problemas complejos, aunque inicialmente hubo tendencia a subestimar la complejidad de ciertas tareas. La investigación autónoma fue constante: Stack Overflow, documentación oficial, y foros de GitHub fueron recursos diarios. Se aprendió a distinguir entre soluciones rápidas y soluciones sostenibles, aunque la presión de tiempo a veces llevó a elegir la primera.

Debugging: Se desarrolló paciencia para debugging sistemático, especialmente con problemas de integración entre servicios. Algunos bugs tomaron días en resolverse, enseñando la importancia de logging adecuado y herramientas de debugging. Se cometieron errores como no usar breakpoints efectivamente al inicio, mejorando estas habilidades con la práctica.

Decisiones técnicas: Algunas decisiones técnicas fueron acertadas (usar TypeScript, elegir Express.js por su simplicidad), otras fueron aprendizajes (no invertir suficiente en testing desde el inicio). Se aprendió que las decisiones técnicas tienen consecuencias a largo plazo que no siempre son evidentes al momento de tomarlas.

7.2.3. Comunicación

Comunicación técnica: La documentación fue un desafío constante. Los comentarios en el código fueron inconsistentes, mejorando hacia el final del proyecto. La documentación de APIs se creó principalmente cuando fue necesaria para otros miembros del equipo. Se mejoró en la capacidad de explicar conceptos técnicos, especialmente al presentar el proyecto al Product Owner y stakeholders, aprendiendo a enfocarse en valor de negocio más que en detalles de implementación.

Desafíos de comunicación: Hubo malentendidos sobre requisitos que resultaron en retrabajo. Se aprendió la importancia de confirmar entendimiento antes de implementar. Los diagramas de arquitectura fueron útiles pero se crearon tarde en el proyecto; haberlos hecho antes habría evitado confusiones.

Comunicación interpersonal: La colaboración con el equipo fue generalmente buena, aunque hubo momentos de frustración cuando las prioridades no estaban claras. Se mejoró en la capacidad de pedir ayuda cuando era necesario, superando la tendencia inicial de intentar resolver todo solo. La recopilación de feedback de usuarios fue valiosa pero podría haberse hecho más temprano y frecuentemente. Las encuestas finales revelaron expectativas que no se conocían durante el desarrollo.

7.2.4. Aprendizaje Continuo

El proyecto requirió aprendizaje intensivo y constante. Muchas tecnologías (Express.js, AWS SDK v3, AWS Bedrock, Monaco Editor, Redis) eran nuevas, lo que significó una curva de aprendizaje pronunciada. Se pasaron muchas horas leyendo documentación, viendo tutoriales y experimentando. No todo lo aprendido se aplicó exitosamente; hubo tecnologías y enfoques que se probaron y descartaron.

Estrategias de aprendizaje: Se aprendió principalmente haciendo, con mucho ensayo y error. Los tutoriales y documentación oficial fueron el punto de partida, pero la comprensión real vino de enfrentar problemas concretos. Stack Overflow y GitHub Issues fueron recursos invaluables para problemas específicos. Se mejoró en la habilidad de buscar información efectivamente, aprendiendo qué términos usar y cómo filtrar resultados relevantes.

Limitaciones: No hubo tiempo para profundizar en todas las áreas deseadas. Algunos conceptos se entendieron superficialmente, suficiente para implementar la funcionalidad pero no para optimizar completamente. Se reconoce que hay mucho más por aprender en cada tecnología utilizada.

7.2.5. Pensamiento Crítico

Se desarrolló mayor capacidad de pensamiento crítico, aunque con limitaciones. Al inicio del proyecto, hubo tendencia a aceptar soluciones populares sin análisis profundo. Con el tiempo, se mejoró en evaluar trade-offs, aunque no siempre se tuvo el conocimiento suficiente para hacer la mejor elección. Algunas decisiones técnicas se basaron en información incompleta o supuestos que resultaron incorrectos, requiriendo ajustes posteriores.

Análisis de resultados: Los resultados de la encuesta (promedio 3.67/5) requirieron análisis honesto. En lugar de justificar las calificaciones moderadas, se reconocieron como áreas de mejora legítimas. Esta capacidad de aceptar críticas constructivamente fue un aprendizaje importante.

7.2.6. Adaptabilidad

La adaptabilidad fue constantemente necesaria. Los requisitos cambiaron varias veces durante el desarrollo, requiriendo flexibilidad. Algunas funcionalidades planificadas se descartaron o simplificaron debido a restricciones de tiempo. Se aprendió a soltar ideas cuando no eran viables, aunque no siempre fue fácil.

Manejo de cambios: Cambiar entre diferentes tareas (frontend, backend, IA, documentación) fue mentalmente demandante. Se mejoró en esta habilidad pero nunca se sintió completamente natural. Los obstáculos técnicos a veces causaron frustración, pero se desarrolló mayor resiliencia. Hubo momentos de querer rendirse con problemas particularmente difíciles, pero se aprendió a tomar descansos y volver con perspectiva fresca.

7.2.7. Atención al Detalle

La atención al detalle fue inconsistente. En áreas críticas (integración de IA, autenticación) se prestó mucha atención, pero en otras áreas hubo descuidos que causaron bugs. La revisión de código no fue tan sistemática como debería, resultando en algunos errores que llegaron a fases avanzadas. Se mejoró en validación de casos edge con el tiempo, especialmente después de encontrar bugs en producción que podrían haberse prevenido.

Experiencia de usuario: Los detalles visuales y de UX recibieron menos atención de la deseada, reflejado en la calificación de 3.33/5 en intuitividad. Algunos mensajes de error son genéricos cuando deberían ser más específicos. Las animaciones y transiciones son básicas. Se reconoce que pulir estos detalles habría mejorado significativamente la percepción de calidad.

7.2.8. Orientación a Resultados

Hubo enfoque en entregar funcionalidad, aunque no siempre con la calidad deseada. La presión de cumplir plazos a veces llevó a comprometer calidad por velocidad, decisiones que se lamentaron después. Se logró entregar un producto funcional que genera valor (100).

Métricas y medición: La medición de impacto se realizó principalmente al final mediante la encuesta. Habría sido mejor tener métricas continuas durante el desarrollo. Algunas optimizaciones se hicieron sin medir el impacto real, basándose en intuición más que en datos.

7.2.9. Trabajo en Equipo

El trabajo en equipo fue generalmente positivo, con colaboración efectiva en la mayoría de situaciones. Se compartió conocimiento cuando alguien encontraba una solución útil, aunque esto podría haberse hecho más sistemáticamente. Hubo momentos de frustración cuando las expectativas no estaban alineadas o cuando el trabajo de uno dependía del trabajo de otro que se retrasaba.

Colaboración real: Las revisiones de código fueron útiles pero no tan frecuentes como deberían. A veces se aprobaron cambios sin revisión profunda por presión de tiempo. Se aprendió a pedir ayuda más efectivamente, reconociendo que luchar solo con un problema por horas no es productivo. El ambiente de trabajo fue mayormente positivo, con apoyo mutuo en momentos difíciles del proyecto.

Desafíos de equipo: La coordinación entre frontend y backend tuvo algunos problemas de comunicación que causaron incompatibilidades en APIs. Se mejoró estableciendo contratos de API más claros. El trabajo remoto añadió desafíos de comunicación que se superaron con reuniones más frecuentes.

7.3. Integración de Competencias y Reflexión

El desarrollo de Irakani Builder demostró que el éxito en proyectos de software requiere tanto competencias técnicas como blandas, y que ambas se desarrollan simultáneamente a través de la experiencia práctica.

Ejemplos de integración:

La implementación del asistente de IA requirió conocimientos técnicos (APIs, manejo de respuestas asíncronas) pero también habilidades de prompt engineering que se desarrollaron mediante prueba y error, pensamiento crítico para evaluar resultados, y comunicación para explicar limitaciones a stakeholders. Los desafíos técnicos enseñaron tanto sobre tecnología como sobre gestión de expectativas.

El trabajo con Scrum integró organización, comunicación y conocimientos técnicos. Las estimaciones iniciales fueron inexactas porque faltaba experiencia tanto en las tecnologías como en el proceso de estimación mismo. Mejorar en uno ayudó a mejorar en el otro.

Lecciones integradas:

Se aprendió que las habilidades técnicas sin comunicación efectiva llevan a malentendidos y retrabajo. La mejor solución técnica no sirve si no se puede explicar o si no resuelve el problema real del usuario. Inversamente, buenas habilidades de comunicación no compensan falta de conocimiento técnico; ambas son necesarias.

7.4. Impacto en el Desarrollo Profesional

Las competencias desarrolladas durante el proyecto Irakani Builder proporcionan una base práctica valiosa para el desarrollo profesional, aunque se reconoce que representan el inicio de un camino de aprendizaje continuo más que un punto de llegada.

Competencias técnicas adquiridas:

La experiencia en **desarrollo full stack** con React, TypeScript, Express.js, Node.js y bases de datos relacionales (MySQL, SQL Server) proporciona conocimientos prácticos en tecnologías demandadas. Sin embargo, se reconoce que hay profundidad por desarrollar en cada área. La experiencia es suficiente para contribuir en proyectos, pero no para roles senior que requieren expertise profundo.

Los **conocimientos en IA** son principalmente en integración y uso de APIs, no en desarrollo de modelos. Esta experiencia es valiosa dado el crecimiento del área, pero es importante ser honesto sobre el alcance: se aprendió a usar herramientas de IA, no a crearlas.

La experiencia en **arquitectura de software** es introductoria. Se comprendieron conceptos y se aplicaron en escala pequeña, pero falta experiencia en sistemas de mayor escala y complejidad.

Competencias blandas desarrolladas:

Se mejoró en **gestión de proyectos**, aprendiendo tanto de éxitos como de errores en estimación y planificación. La experiencia con Scrum es práctica pero básica.

La **resolución de problemas** mejoró significativamente, desarrollando mayor autonomía y persistencia. Esta es quizás la competencia más valiosa: la confianza de que problemas desconocidos pueden resolverse con investigación y esfuerzo.

Áreas de crecimiento futuro:

Se identifican claramente áreas que requieren desarrollo adicional: testing automatizado, seguridad en profundidad, optimización de rendimiento, y diseño de UX. El proyecto proporcionó exposición a estas áreas pero no maestría.

Valor real del proyecto:

El valor principal del proyecto no es haber "dominado" tecnologías, sino haber desarrollado la capacidad de aprender nuevas tecnologías, enfrentar problemas complejos, y entregar un producto funcional que genera valor real (evidenciado

por el 100 % de usuarios reportando ahorro de tiempo). Esta experiencia práctica, con sus éxitos y desafíos, proporciona una base sólida para continuar creciendo profesionalmente.

Referencias

8. Anexos

8.1. Anexo A: Glosario de Términos

8.1.1. Términos Técnicos

API (Application Programming Interface) Conjunto de definiciones y protocolos que permite la comunicación entre diferentes aplicaciones de software. En el contexto de Irakani Builder, se utilizan APIs para conectar el frontend con el backend y para integrar servicios externos como modelos de IA.

AWS SDK (Amazon Web Services Software Development Kit) Conjunto de herramientas y bibliotecas para interactuar con servicios de AWS. Irakani Builder utiliza AWS SDK v3 para integrar servicios como Bedrock, S3, DynamoDB y Step Functions.

Alucinación (IA) Fenómeno en el que un modelo de inteligencia artificial genera información incorrecta, inventada o sin fundamento, presentándola con confianza como si fuera verdadera. Es un desafío común en modelos de lenguaje grandes.

AMI (Amazon Machine Image) Imagen de máquina virtual en AWS que contiene el sistema operativo, aplicaciones y configuraciones necesarias para lanzar instancias EC2.

Arquitectura Monolítica Arquitectura de software donde todos los componentes de la aplicación están fuertemente acoplados y se ejecutan como una única unidad, en contraste con arquitecturas de microservicios.

Auto Scaling Group Grupo de instancias EC2 en AWS que escalan automáticamente hacia arriba o hacia abajo según la demanda, permitiendo mantener disponibilidad y optimizar costos.

AWS Bedrock Servicio de AWS que proporciona acceso a modelos fundacionales de IA de diferentes proveedores a través de una API unificada, facilitando la integración de capacidades de IA generativa.

AWS CodeBuild Servicio de compilación completamente administrado de AWS que compila código fuente, ejecuta pruebas y produce paquetes de software listos para desplegar.

AWS CodePipeline Servicio de integración y entrega continua (CI/CD) de AWS que automatiza las fases de construcción, prueba y despliegue del proceso de lanzamiento de software.

AWS Cognito Servicio de AWS para autenticación, autorización y gestión de usuarios que permite agregar registro e inicio de sesión a aplicaciones web y móviles.

AWS DynamoDB Base de datos NoSQL completamente administrada de AWS que proporciona rendimiento rápido y predecible con escalabilidad perfecta.

AWS S3 Servicio de almacenamiento de objetos de AWS que ofrece escalabilidad, disponibilidad de datos, seguridad y rendimiento líderes en la industria.

AWS Step Functions Servicio de orquestación de workflows de AWS que permite coordinar múltiples servicios de AWS en flujos de trabajo sin servidor.

Axios Cliente HTTP basado en promesas para Node.js y el navegador. Utilizado en Irakani Builder para realizar peticiones HTTP a APIs externas.

Backlog Refinement Sesiones periódicas en Scrum donde el equipo revisa, estima y prioriza historias de usuario futuras del Product Backlog para prepararlas para próximos sprints.

Backend Parte del sistema que se ejecuta en el servidor y maneja la lógica de negocio, el acceso a bases de datos y la comunicación con servicios externos. No es visible directamente para el usuario final.

BFF (Backend for Frontend) Patrón arquitectónico que consiste en crear una capa de abstracción específica entre el frontend y los servicios backend, optimizando la comunicación para las necesidades particulares de cada interfaz.

- Breakpoint** Punto de parada intencional en el código durante el debugging que permite pausar la ejecución del programa para inspeccionar el estado de variables y el flujo de ejecución.
- Caché** Mecanismo de almacenamiento temporal de datos frecuentemente accedidos para mejorar el rendimiento y reducir la latencia. En Irakani Builder se utiliza para almacenar respuestas de IA y reducir costos.
- Canvas** Área de trabajo principal en el editor visual donde los usuarios ensamblan componentes para construir sus aplicaciones mediante arrastrar y soltar.
- Chain-of-Thought** Técnica de prompt engineering que instruye al modelo de IA a pensar paso a paso, descomponiendo problemas complejos en pasos intermedios para mejorar la calidad del razonamiento.
- CI/CD (Continuous Integration/Continuous Deployment)** Prácticas de desarrollo que automatizan la integración de código y el despliegue de aplicaciones, permitiendo entregas más rápidas y confiables.
- Claude** Modelo de lenguaje grande desarrollado por Anthropic, utilizado como alternativa a GPT para tareas de generación de texto y código.
- CloudWatch** Servicio de monitoreo y observabilidad de AWS que recopila y rastrea métricas, logs y eventos de recursos y aplicaciones en AWS.
- Componente** Elemento reutilizable de interfaz de usuario que encapsula funcionalidad y presentación. En React, los componentes son la unidad básica de construcción de aplicaciones.
- Control Empírico** Enfoque de gestión basado en la observación, experimentación y adaptación continua, que constituye la base filosófica de Scrum y otras metodologías ágiles.
- Criterios de Aceptación** Condiciones específicas y medibles que debe cumplir una historia de usuario para considerarse completada y aceptada por el Product Owner.
- Context API** Mecanismo de React para compartir datos entre componentes sin necesidad de pasar props manualmente a través de cada nivel del árbol de componentes.
- CORS (Cross-Origin Resource Sharing)** Mecanismo de seguridad que permite o restringe recursos solicitados en una aplicación web desde un dominio diferente al que sirvió el recurso original.
- CRUD** Acrónimo de Create, Read, Update, Delete. Operaciones básicas de persistencia de datos en bases de datos y APIs.
- Daily Standup** Reunión diaria de sincronización del equipo Scrum, típicamente de 15 minutos, donde cada miembro comparte qué hizo ayer, qué hará hoy y qué impedimentos enfrenta.
- Debugging** Proceso sistemático de identificar, analizar y corregir errores o bugs en el código de software mediante herramientas y técnicas especializadas.
- Definition of Done (DoD)** Conjunto de criterios acordados por el equipo Scrum que definen cuándo una historia de usuario o incremento está verdaderamente completo y listo para entrega.
- Deuda Técnica** Costo implícito de trabajo adicional futuro causado por elegir una solución fácil o rápida ahora en lugar de usar un mejor enfoque que tomaría más tiempo.
- DOM Virtual** Representación en memoria del DOM (Document Object Model) utilizada por React para optimizar actualizaciones de la interfaz, comparando cambios antes de aplicarlos al DOM real.
- Docker** Plataforma de contenedorización que permite empaquetar aplicaciones y sus dependencias en contenedores portátiles que pueden ejecutarse en cualquier entorno.
- ECR (Elastic Container Registry)** Registro de contenedores Docker completamente administrado de AWS que facilita el almacenamiento, gestión y despliegue de imágenes de contenedores.
- Edge Case** Caso límite o excepcional en pruebas de software que ocurre en los extremos de los parámetros operativos, importante para garantizar la robustez del sistema.
- Editor Visual** Interfaz gráfica que permite crear aplicaciones mediante manipulación directa de elementos visuales, sin necesidad de escribir código manualmente.
- Épica** Conjunto grande de trabajo en Scrum que representa una funcionalidad compleja y que se divide en múltiples historias de usuario más pequeñas y manejables.

- Event Loop** Mecanismo fundamental de Node.js para manejar operaciones asíncronas, permitiendo ejecutar código no bloqueante mediante un ciclo continuo de procesamiento de eventos.
- Frontend** Parte del sistema que se ejecuta en el navegador del usuario y maneja la presentación visual y la interacción con el usuario.
- Full Stack** Término que describe a un desarrollador o sistema que abarca tanto el frontend como el backend de una aplicación.
- Git** Sistema de control de versiones distribuido utilizado para rastrear cambios en el código fuente durante el desarrollo de software.
- GPT (Generative Pre-trained Transformer)** Familia de modelos de lenguaje desarrollados por OpenAI, capaces de generar texto coherente y realizar diversas tareas de procesamiento de lenguaje natural.
- Hook (React)** Funciones especiales de React que permiten usar estado y otras características de React en componentes funcionales sin necesidad de clases.
- HTTPS (Hypertext Transfer Protocol Secure)** Versión segura del protocolo HTTP que utiliza cifrado SSL/TLS para proteger la comunicación entre el navegador y el servidor.
- IA Generativa** Tipo de inteligencia artificial capaz de crear contenido nuevo (texto, imágenes, código) basándose en patrones aprendidos de datos de entrenamiento.
- Jimp (JavaScript Image Manipulation Program)** Biblioteca de Node.js para procesamiento de imágenes completamente escrita en JavaScript. Utilizada en Irakani Builder para manipulación de iconos y recursos visuales.
- Jira** Herramienta de gestión de proyectos y seguimiento de issues utilizada ampliamente en desarrollo de software ágil.
- JWT (JSON Web Token)** Estándar abierto para crear tokens de acceso que permiten la autenticación y autorización segura entre partes mediante un objeto JSON firmado digitalmente.
- Lazy Loading** Técnica de optimización que retrasa la carga de recursos no críticos hasta que sean necesarios, mejorando el tiempo de carga inicial.
- Microservicios** Arquitectura de software que estructura una aplicación como una colección de servicios pequeños, independientes y débilmente acoplados.
- Middleware** Software que actúa como puente entre diferentes aplicaciones o componentes, facilitando la comunicación y el procesamiento de datos.
- Monaco Editor** Editor de código de código abierto desarrollado por Microsoft, el mismo que impulsa Visual Studio Code, diseñado para funcionar en navegadores web.
- Express.js** Framework web minimalista y flexible para Node.js que proporciona un conjunto robusto de características para aplicaciones web y móviles. Utilizado en el backend de Irakani Builder.
- Few-shot Learning** Técnica de prompt engineering que proporciona al modelo de IA varios ejemplos de entrada-salida para guiar su comportamiento y mejorar la calidad de las respuestas.
- Git Flow** Modelo de ramificación para Git que define una estructura estricta de branches (master, develop, feature, release, hotfix) para organizar el desarrollo de software.
- GitHub Issues** Sistema de seguimiento de problemas, bugs y tareas integrado en GitHub que facilita la gestión de proyectos y la colaboración en desarrollo de software.
- Historia de Usuario** Descripción breve y simple de una funcionalidad desde la perspectiva del usuario final, típicamente siguiendo el formato `Como [rol], quiero [acción] para [beneficio]`.
- ITERADAPTA** Nombre legal de la empresa propietaria de la marca Irakani, bajo la cual se desarrolla y comercializa Irakani Builder.
- KPI (Key Performance Indicator)** Indicador clave de rendimiento utilizado para medir el éxito en el logro de objetivos específicos del negocio o proyecto.
- LLM (Large Language Model)** Modelos de Lenguaje Grandes basados en redes neuronales entrenados con enormes cantidades de texto, capaces de comprender y generar lenguaje natural de forma coherente.

Low-Code/No-Code (LC/NC) Plataformas de desarrollo que abstraen la complejidad de la programación tradicional mediante interfaces visuales, permitiendo crear aplicaciones con poco o ningún código.

Memory Leak Fuga de memoria que ocurre cuando un programa no libera correctamente la memoria que ya no necesita, causando degradación del rendimiento y posibles fallos del sistema.

Merge Conflict Conflicto que ocurre en Git al intentar fusionar cambios de diferentes ramas cuando las mismas líneas de código han sido modificadas de formas incompatibles.

Modelo Fundacional Modelos de inteligencia artificial pre-entrenados con grandes cantidades de datos diversos que pueden adaptarse a múltiples tareas específicas mediante fine-tuning o prompting.

Multi-stage Build Técnica de optimización de Docker que utiliza múltiples etapas de construcción en un Dockerfile para reducir el tamaño final de la imagen eliminando dependencias innecesarias.

Node.js Entorno de ejecución de JavaScript del lado del servidor construido sobre el motor V8 de Chrome.

Onboarding Proceso estructurado de incorporación de nuevos usuarios o miembros del equipo, proporcionándoles la información, herramientas y capacitación necesarias para ser productivos.

ORM (Object-Relational Mapping) Técnica de programación que convierte datos entre sistemas de tipos incompatibles (objetos en código y tablas en bases de datos relacionales).

Product Backlog Lista priorizada y dinámica de todo el trabajo pendiente en Scrum, mantenida por el Product Owner, que representa los requisitos conocidos del producto.

Product Owner Rol de Scrum responsable de maximizar el valor del producto, gestionar el Product Backlog y asegurar que el equipo trabaje en las prioridades correctas.

Profiling Análisis detallado del rendimiento del código que identifica cuellos de botella, uso de recursos y oportunidades de optimización mediante herramientas especializadas.

MySQL Sistema de gestión de bases de datos relacional de código abierto ampliamente utilizado, conocido por su velocidad, confiabilidad y facilidad de uso.

mysql2 Cliente MySQL para Node.js con enfoque en rendimiento. Soporta promesas y prepared statements. Utilizado en Irakani Builder para conectar con bases de datos MySQL.

mssql Cliente Microsoft SQL Server para Node.js. Utilizado en Irakani Builder para conectar con bases de datos SQL Server.

SQL Server Sistema de gestión de bases de datos relacional desarrollado por Microsoft, conocido por su integración con el ecosistema Windows y herramientas empresariales.

Preview Vista previa en tiempo real de la aplicación en desarrollo, permitiendo ver los cambios inmediatamente sin necesidad de compilar o desplegar.

Prompt Instrucción o consulta en lenguaje natural proporcionada a un modelo de IA para guiar su respuesta o generación de contenido.

Prompt Engineering Disciplina que se enfoca en diseñar y optimizar prompts para obtener los mejores resultados de modelos de lenguaje de IA.

Props (Properties) Mecanismo de React para pasar datos de componentes padres a componentes hijos.

Puppeteer Biblioteca de Node.js que proporciona una API de alto nivel para controlar navegadores Chrome o Chromium. Utilizada en Irakani Builder para web scraping.

PWA (Progressive Web App) Aplicación web que utiliza capacidades modernas del navegador para proporcionar una experiencia similar a una aplicación nativa, incluyendo funcionamiento offline.

React Biblioteca de JavaScript para construir interfaces de usuario, desarrollada y mantenida por Meta (Facebook).

Redis Sistema de almacenamiento de datos en memoria de código abierto, utilizado como base de datos, caché y broker de mensajes. En Irakani Builder se utiliza para caché de respuestas de IA y gestión de sesiones.

Redux Biblioteca de gestión de estado predecible para aplicaciones JavaScript, comúnmente utilizada con React.

- Refactorización** Proceso de reestructurar código existente sin cambiar su comportamiento externo, con el objetivo de mejorar su legibilidad, mantenibilidad y estructura interna.
- RESTful API** API que sigue los principios de REST (Representational State Transfer), utilizando métodos HTTP estándar y URLs para operaciones sobre recursos.
- Scrum** Marco de trabajo ágil para gestión de proyectos que organiza el trabajo en iteraciones cortas llamadas sprints.
- Scrum Master** Facilitador del proceso Scrum que ayuda al equipo a entender y aplicar Scrum, elimina impedimentos y protege al equipo de interrupciones externas.
- Seed/Seeder** Script que puebla una base de datos con datos iniciales o de prueba.
- SOLID** Acrónimo de cinco principios de diseño orientado a objetos: Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion.
- Sprint** Período de tiempo fijo (típicamente 1-4 semanas) en Scrum durante el cual se completa un conjunto específico de trabajo.
- Sprint Backlog** Conjunto de elementos del Product Backlog seleccionados para un sprint específico, junto con el plan para entregarlos y alcanzar el objetivo del sprint.
- Sprint Planning** Ceremonia de Scrum al inicio de cada sprint donde el equipo planifica el trabajo a realizar, selecciona historias del Product Backlog y define el objetivo del sprint.
- Sprint Retrospective** Reunión al final de cada sprint donde el equipo Scrum reflexiona sobre su proceso de trabajo e identifica mejoras para implementar en el siguiente sprint.
- Sprint Review** Demostración de las funcionalidades completadas al final del sprint, donde el equipo presenta el incremento a stakeholders y recopila feedback.
- SQL (Structured Query Language)** Lenguaje estándar para gestionar y manipular bases de datos relacionales.
- Stack Overflow** Plataforma comunitaria de preguntas y respuestas para programadores, ampliamente utilizada para resolver problemas técnicos y compartir conocimiento.
- Story Points** Unidad abstracta de estimación de esfuerzo en Scrum que considera complejidad, cantidad de trabajo e incertidumbre, en lugar de tiempo absoluto.
- SSL/TLS** Protocolos criptográficos que proporcionan comunicaciones seguras sobre una red, comúnmente utilizados para asegurar conexiones HTTPS.
- Stakeholder** Persona o grupo con interés o participación en un proyecto, que puede afectar o ser afectado por sus resultados.
- Target Group** Grupo de destinos (instancias EC2, contenedores, direcciones IP) para balanceo de carga en AWS, que recibe tráfico distribuido según reglas configuradas.
- Token** En el contexto de IA, unidad básica de texto procesada por modelos de lenguaje. En autenticación, cadena de caracteres que representa credenciales de acceso.
- Transformer** Arquitectura de red neuronal introducida en 2017 que utiliza mecanismos de atención y constituye la base de los modelos de lenguaje grandes (LLMs) modernos.
- TypeORM** ORM para TypeScript y JavaScript que soporta múltiples bases de datos y proporciona una forma elegante de trabajar con datos.
- TypeScript** Superconjunto tipado de JavaScript que compila a JavaScript plano, añadiendo tipos estáticos opcionales al lenguaje.
- UI/UX (User Interface/User Experience)** UI se refiere al diseño visual de la interfaz; UX abarca toda la experiencia del usuario al interactuar con el producto.
- Validación** Proceso de verificar que los datos cumplen con criterios específicos antes de ser procesados o almacenados.
- Valkey** Sistema de caché distribuido de código abierto (fork de Redis) utilizado para almacenamiento en memoria de alta velocidad y gestión de estructuras de datos.
- WebSocket** Protocolo de comunicación que proporciona canales de comunicación bidireccional full-duplex sobre una única conexión TCP.

8.2. Anexo B: Encuesta de Satisfacción de Usuarios

8.2.1. Descripción de la Encuesta

La encuesta de satisfacción fue diseñada para evaluar la percepción de los usuarios sobre Irakani Builder en comparación con la plataforma anterior. Se aplicó a 3 usuarios del equipo de desarrollo que utilizaron activamente la plataforma durante el período de pruebas.

8.2.2. Instrumento de Evaluación

A continuación se presenta el cuestionario completo utilizado para la evaluación:

Pregunta 1: Ahorro de Tiempo

En comparación con la plataforma anterior, ¿cuánto tiempo consideras que ahorras al crear un prototipo básico con Irakani Builder?

- Nada
- 10 - 20 %
- 30 - 50 %
- Más del 50 %

Pregunta 2: Utilidad del Asistente de IA

Del 1 al 5, ¿qué tan útil consideras al Asistente de IA (Chat) para resolver dudas o generar código rápido?

- 1 (Nada útil)
- 2 (Poco útil)
- 3 (Moderadamente útil)
- 4 (Útil)
- 5 (Muy útil)

Pregunta 3: Generación Automática

¿La generación automática de iconos y formularios acelera tu flujo de trabajo?

- Sí
- No
- Parcialmente

Pregunta 4: Calidad del Código Generado

Del 1 al 5, ¿cómo calificas la calidad del código generado por la IA (requiere pocas correcciones manuales)?

- 1 (Muy baja calidad)
- 2 (Baja calidad)
- 3 (Calidad aceptable)
- 4 (Buena calidad)
- 5 (Excelente calidad)

Pregunta 5: Frecuencia de Alucinaciones

¿Con qué frecuencia la IA alucina o genera estructuras incorrectas?

- Nunca

Rara vez
Frecuentemente
Siempre

Pregunta 6: Intuitividad de la Interfaz

Del 1 al 5, ¿qué tan intuitiva es la interfaz de paneles redimensionables y el editor visual?

- 1 (Nada intuitiva)
- 2 (Poco intuitiva)
- 3 (Moderadamente intuitiva)
- 4 (Intuitiva)
- 5 (Muy intuitiva)

Pregunta 7: Experiencia con Monaco Editor

Del 1 al 5, ¿cómo evaluarías la experiencia de edición de código con Monaco Editor integrado en la web?

- 1 (Muy mala)
- 2 (Mala)
- 3 (Aceptable)
- 4 (Buena)
- 5 (Excelente)

Pregunta 8: Sugerencias para Versión 2.0 (Pregunta Abierta)

¿Qué funcionalidad específica agregarías en una versión 2.0 para mejorar aún más tu productividad?