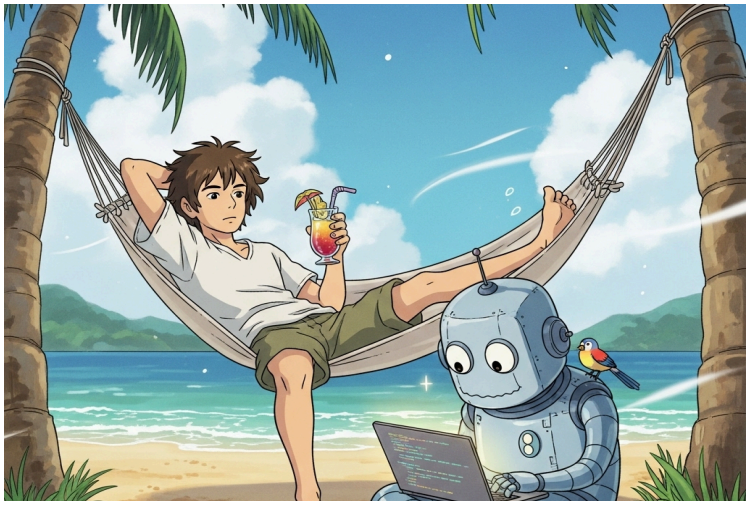
UNRN Universidad Nacional de Río Negro

Licenciatura en Sistemas

PRÁCTICAS RECOMENDADAS PARA UN DESARROLLO EFICIENTE DE "VIBE CODING".



Autor: Antual, Adrian Tomas.

Director: Molinari, Enrique Pablo.

Viedma, Río Negro.

2025.

Introducción y planteo del problema..... 3
IA a primera vista..... 3
Promesas brillantes, fantasmas molestos..... 4
Objetivos..... 5
Objetivo General..... 5
Objetivos Específicos..... 5
Herramientas que programan: cartografía del Vibe Coding..... 6
Cuatro familias..... 6
Cinco miradas..... 8
Tablero de contrastes..... 11
Un nuevo lenguaje..... 12
Vibe Coding..... 13
LLM (Large Language Model)..... 13
Coding Agents (Agentes de programación)..... 13
Prompts..... 14
MCP (Model Context Protocol)..... 14
PRD (Product Requirements Document)..... 14
Más allá de las palabras..... 14
Persistencia de contexto..... 14
Reglas y archivos de configuración en Cursor..... 14
Modelos de agentes en Cursor..... 15
Contexto en Cursor..... 16
Forks..... 16
Diseño en un archivo Markdown..... 16
Caja de herramientas..... 18
Cursor..... 18
El peso de la decisión..... 18
Desarrollo de la aplicación..... 20
Caso de estudio:..... 20
Arquitectura del sistema..... 21
Tecnologías utilizadas..... 22
Paso a paso..... 24
Manejo de frameworks y entornos de desarrollo..... 25
Definición de reglas y uso de MCP..... 27
REGLAS..... 27
MCP (Model Context Protocol)..... 39
Práctica: Generación del PRD..... 41
Práctica: Generación del Documento de Especificaciones Técnicas..... 44
Práctica: Plan de Ejecución en Etapas y Fases..... 47
Ejecución del Plan con Rol de Programador en Cursor..... 51
Práctica: Resolución de Problemas y Depuración..... 57

Buenas prácticas para el desarrollo de software con Inteligencia Artificial..... 59
¿El éxito o la ilusión del éxito?..... 62
Entre la exploración y la disciplina..... 64
Problemas..... 66
Alucinaciones..... 67
Evidencia de problemas:..... 67
Caso 1:..... 67
Caso 2:..... 69
Trabajos Relacionados: Discusión..... 71
GitHub Spec-Kit..... 71
Un acercamiento práctico al nuevo paradigma..... 73
Conclusiones..... 75
El final..... 75
Mas alla del código..... 76
Referencias..... 79
Resultados..... 80

Introducción y planteo del problema

IA a primera vista

En los últimos años, la inteligencia artificial (IA) ha transformado de manera significativa la industria del software, integrándose en múltiples etapas del ciclo de desarrollo y ofreciendo nuevas posibilidades para agilizar tareas, optimizar procesos y potenciar la creatividad de los programadores. Herramientas basadas en modelos de lenguaje de gran escala (LLM) como GitHub Copilot, Cursor o Claude Code permiten hoy generar código, detectar errores y proponer mejoras a partir de simples instrucciones en lenguaje natural.

Dentro de este contexto, surgió recientemente un concepto denominado Vibe Coding, popularizado en redes sociales por Andrej Karpathy. En su publicación original, Karpathy describió con tono irónico la idea de "programar dejándose llevar por la vibra", aceptando automáticamente cambios generados por la IA, copiando y pegando mensajes de error hasta que desaparecen y sin leer en detalle el código. Aunque la propuesta nació como una broma sobre proyectos descartables de fin de semana, rápidamente fue resignificada en la práctica. Hoy se entiende el Vibe Coding como un enfoque de desarrollo que combina programación tradicional con la asistencia intensiva de IA, utilizando interacciones en lenguaje natural para construir aplicaciones de manera ágil.

El tweet original es el siguiente:

> "*Hay un nuevo tipo de programación que llamo "vibe coding", donde te entregás completamente a la vibra, abrazás lo exponencial y te olvidás de que el código siquiera existe. Esto es posible porque los LLMs (por ejemplo, Cursor Composer con Sonnet) ya son demasiado buenos. Además, hablo con Composer usando SuperWhisper, así que casi ni toco el teclado.*
>
> *Pido las cosas más tontas, como "reducí el padding de la barra lateral a la mitad", porque me da pereza buscarlo. Siempre le doy a "Accept All", ya ni leo los diffs. Cuando me aparecen mensajes de error simplemente los copio y pego sin ningún comentario, y la mayoría de las veces eso lo arregla.*

El código crece más allá de mi comprensión habitual, tendría que sentarme a leerlo bastante para entenderlo. A veces los LLMs no logran corregir un bug, entonces lo esquivo o pido cambios aleatorios hasta que desaparece.

No está tan mal para proyectos descartables de fin de semana, pero sigue siendo bastante divertido. Estoy armando un proyecto o una webapp, pero en realidad no es programar: solo miro cosas, digo cosas, corro cosas y copio/pego cosas, y la mayoría de las veces funciona."

(Traducción propia)

Promesas brillantes, fantasmas molestos

La aparición de este enfoque abre oportunidades, pero también plantea desafíos. El ecosistema de herramientas crece a gran velocidad, sin que existan todavía metodologías claras ni consensos sobre cómo usarlas de forma efectiva. Esto genera incertidumbre en los desarrolladores, quienes deben distinguir qué aporta valor real al proceso y qué introduce riesgos, como la dependencia excesiva del asistente, la pérdida de control sobre el código o la dificultad de mantener proyectos a largo plazo.

En este marco se inscribe el presente Trabajo Final de Carrera, cuyo propósito es analizar y documentar el uso de herramientas de IA en el desarrollo de una aplicación web sencilla, bajo el enfoque de Vibe Coding. El objetivo central es identificar y proponer un conjunto de buenas prácticas que orienten a futuros desarrolladores en la adopción responsable y eficiente de estas tecnologías.

Objetivos

Este Trabajo Final de Carrera busca explorar el enfoque de Vibe Coding, entendido como la integración de herramientas de inteligencia artificial en el proceso de desarrollo de software. Para ello, se propone un conjunto de objetivos que orientan la investigación y el trabajo.

Objetivo General

Establecer un conjunto de prácticas recomendadas que permitan aprovechar herramientas de desarrollo asistido por IA de forma eficiente y replicable en proyectos web.

Objetivos Específicos

- I*nvestigar herramientas de desarrollo de software asistidas por IA.*
- *Indagar terminologías nuevas que incorpora esta forma moderna de programar.*
- *Seleccionar un conjunto de herramientas para relevar.*
- *Desarrollar una aplicación web utilizando las herramientas de IA seleccionadas.*
- *Extraer y documentar prácticas recomendadas basadas en la experiencia de la práctica.*
- *Documentar la experiencia de desarrollo como el entregable principal del TFC.*

Herramientas que programan: cartografía del Vibe Coding

Cuatro familias

Este capítulo aborda el primer objetivo específico del trabajo: *"investigar herramientas de desarrollo de software asistidas por inteligencia artificial"*. Para ello, se recopilaron y analizaron distintos agentes y entornos de programación que actualmente conforman el ecosistema emergente del llamado Vibe Coding.

Durante el último año ha surgido un abanico creciente de soluciones que permiten a los desarrolladores apoyarse en modelos de lenguaje (LLM) para escribir, modificar y probar código. Estas herramientas no son homogéneas, sino que se presentan en formas diversas, según el grado de integración de la IA y el tipo de experiencia que priorizan.

En términos generales, es posible agruparlas en cuatro grandes categorías. El primer grupo está conformado por las plataformas visuales de desarrollo full stack, diseñadas para construir aplicaciones completas en poco tiempo mediante prompts y editores gráficos. Ejemplos representativos son *Tempo Labs, Bolt.new y Lovable.dev*, que ofrecen autenticación, base de datos y despliegue en la nube, con la posibilidad de importar proyectos desde repositorios o maquetas de diseño. A estas se suman *Replit* y *Base44*, que apuntan a públicos con diferentes niveles de experiencia, desde principiantes hasta desarrolladores avanzados.

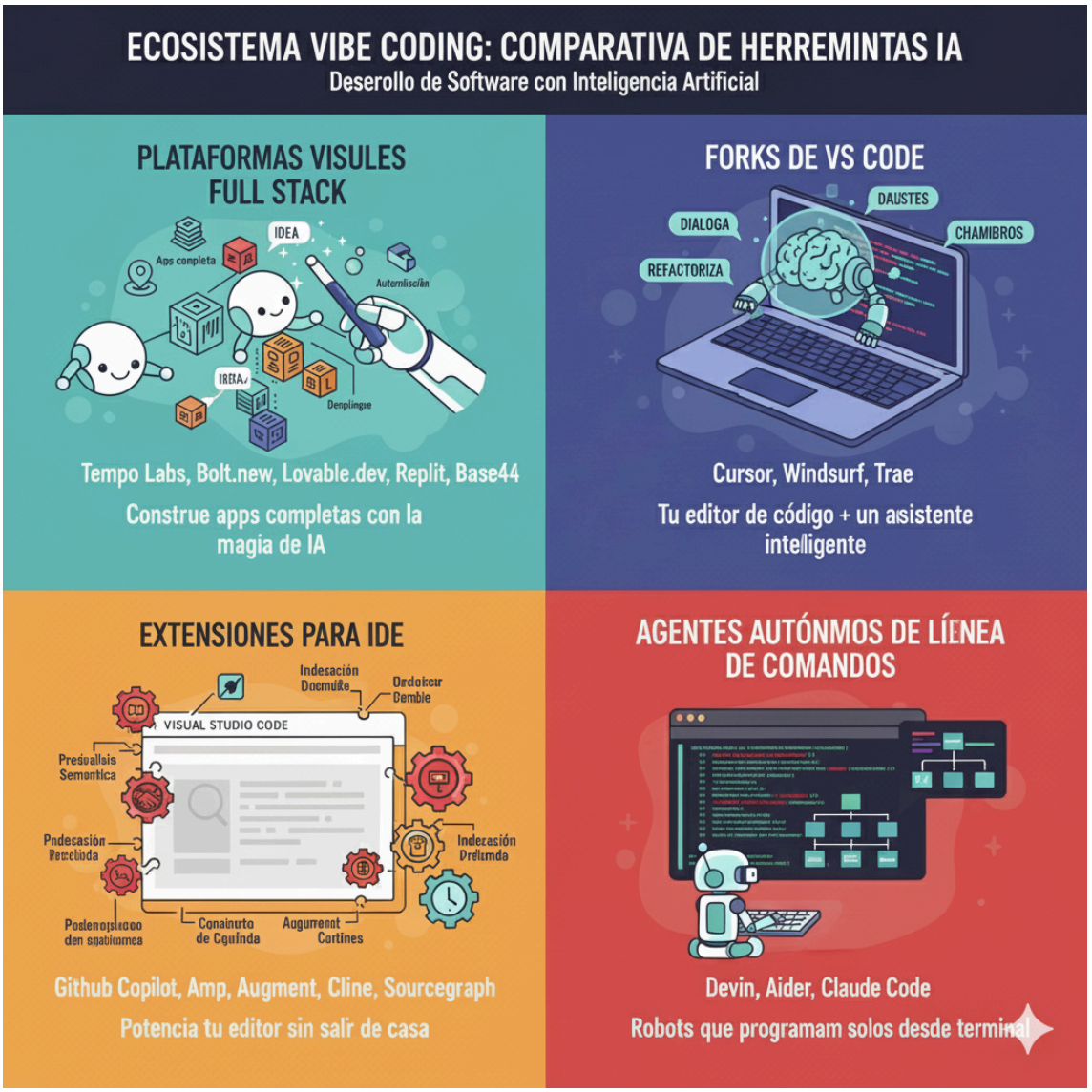
El segundo grupo lo integran los forks de VS Code, entre los que destacan *Cursor, Windsurf y Trae*. Estas propuestas replican la experiencia de un IDE tradicional, pero con funciones avanzadas de IA integradas. Permiten dialogar con un agente para solicitar cambios, realizar ajustes en el código de manera conversacional y, en algunos casos, conectar servicios externos mediante el protocolo MCP (Model Context Protocol). Aunque son muy útiles en fases iniciales de construcción, todavía presentan limitaciones cuando el proyecto crece en complejidad y requiere mayor control del contexto.

Un tercer conjunto lo constituyen las extensiones para IDE, especialmente en *Visual Studio Code*. Entre ellas se destacan GitHub Copilot, windsurf, *Amp, Augment, Continue, Cline y Sourcegraph (Cody)*. Su atractivo radica en que no

obligan a abandonar el entorno de trabajo habitual, sino que se integran como complementos. Cada una cumple un rol particular: *Amp* fomenta la colaboración en equipos, *Augment* y *Sourcegraph* ofrecen indexación profunda de repositorios y búsqueda semántica, mientras que *Cline* y *Continue* se orientan a la predicción de cambios y la automatización de tareas. También aparecen propuestas más experimentales, como *Fynix* y *Pythagora*, enfocadas en analizar la evolución del código o generar aplicaciones Node.js desde cero.

Finalmente, se ubican los agentes autónomos de línea de comandos, como *Devin, Aider y Claude Code*. Estos funcionan directamente desde la terminal e integran repositorios locales, permitiendo planificar, implementar y probar funcionalidades con mínima intervención del programador. Su principal fortaleza es la automatización de tareas complejas y la interacción natural mediante prompts, aunque presentan desafíos en cuanto a consumo de recursos, costos y la necesidad de validación humana para evitar errores.

En conjunto, este panorama muestra que el ecosistema de Vibe Coding está en plena expansión y que, si bien ninguna herramienta por sí sola garantiza un flujo de trabajo completo y confiable, cada una aporta componentes valiosos que pueden ser aprovechados de manera combinada.



Elaboración propia realizada con Gemini

Cinco miradas.

El ecosistema de herramientas de Vibe Coding es amplio y diverso. Para este Trabajo Final de Carrera se propone una clasificación propia que, más que describir productos específicos, busca establecer **dimensiones analíticas** que permitan evaluar el rol de cada herramienta dentro del ciclo de vida del software. Estas dimensiones surgen del estudio y análisis de artículos recientes, cuadros comparativos disponibles en internet e investigaciones de autores acerca de este tema. [1], [2]

Nivel de integración con el flujo de trabajo

La primera dimensión se centra en analizar cuánto se modifica el flujo tradicional de desarrollo al incorporar herramientas de inteligencia artificial. Según el grado de integración que proponen, pueden identificarse tres tipos principales:

- **Independientes:** crean su propio entorno cerrado de trabajo, donde todas las tareas se desarrollan dentro de la misma plataforma, sin depender de herramientas externas.
- **Integradas:** se acoplan a entornos consolidados (como los IDEs) y amplían sus funciones, sin alterar la dinámica de trabajo existente.
- **Complementarias:** se limitan a asistir etapas específicas del proceso (por ejemplo, documentación o testing) sin intervenir en el ciclo completo del desarrollo.

Tipo de interacción con la IA

Una segunda dimensión se refiere a la forma en que el desarrollador se comunica con la inteligencia artificial. En este sentido, las herramientas pueden clasificarse en tres enfoques principales:

- **Conversacionales:** permiten una interacción directa mediante prompts en lenguaje natural, de estilo chat, donde el diálogo guía la generación de resultados.
- **Predictivas:** operan en segundo plano, sugiriendo código, completando funciones o detectando posibles errores mientras se escribe.
- **Híbridas:** combinan ambos enfoques, ofreciendo tanto sugerencias automáticas como la posibilidad de mantener conversaciones contextuales con el sistema.

Gestión del contexto

La capacidad de una herramienta para conservar información del proyecto a lo largo del tiempo es un factor determinante en la eficiencia del Vibe Coding. Según el modo en que gestionan dicho contexto, pueden distinguirse tres tipos:

- **Estáticas:** responden únicamente a la instrucción puntual del usuario, sin conservar memoria de interacciones previas.
- **Persistentes:** almacenan un historial o conjunto de reglas que permiten retomar conversaciones o tareas anteriores con coherencia.
- **Mixtas:** combinan ambas estrategias, conservando sólo fragmentos del contexto relevantes para la tarea actual.

Alcance funcional

Otra dimensión clave radica en cuán amplio es el espectro de tareas que una herramienta puede cubrir dentro del proceso de desarrollo. En este plano se pueden identificar tres niveles:

- **De prototipado rápido:** permiten levantar en minutos una aplicación mínima o un modelo inicial, útil para validar ideas o probar arquitecturas.
- **De desarrollo completo:** gestionan de forma integral tanto el frontend como el backend y la base de datos, posibilitando ciclos de desarrollo más autónomos.
- **Especializadas:** se orientan a funciones concretas, como pruebas automatizadas, documentación, refactorización o auditoría de código.

Curva de aprendizaje

Finalmente, la facilidad de adopción es una variable determinante en la expansión del Vibe Coding. No todas las herramientas requieren el mismo nivel de conocimiento técnico, por lo que pueden clasificarse de la siguiente manera:

- **Accesibles:** diseñadas para principiantes o usuarios sin experiencia en programación, priorizando la simplicidad de uso.
- **Intermedias:** requieren cierto conocimiento previo sobre flujos de trabajo, comandos o conceptos básicos de desarrollo.
- **Avanzadas:** están pensadas para equipos técnicos con experiencia en integración de repositorios, control de versiones y automatización de procesos.

Tablero de contrastes.

Herramienta.	**Rol en el flujo de trabajo.**	**Grado de autonomía del agente.**	**Nivel de integración.**	**Complejidad / curva de aprendizaje.**	**Ejemplo de aporte concreto.**
Cursor.	IDE principal.	Medio-Alto (Composer, MCP, Elección de modelos).	Total (entorno completo).	Alta (requiere experiencia en IDE y Git).	Modificación conversacional de código en proyectos grandes.
GitHub Copilot.	Asistente dentro del IDE.	Medio-Alto (MCP, Elección de modelos).	Integrado (extensión).	Media (entorno familiar).	Sugerencias rápidas y generación de pruebas unitarias.
Claude Code.	Terminal autónoma.	Alto (planifica e implementa).	Independiente.	Media-Alta.	Desarrollo de nuevos módulos sin usar IDE.

Lovable.dev.	Constructor full stack visual.	Medio (prompt + edición visual).	Independiente.	Baja (apto para no-programadores).	Prototipos completos con base de datos y autenticación.
Sourcegraph (Cody).	Indexación y búsqueda semántica.	Medio (contexto + completado).	Complementaria.	Alta (uso en equipos grandes).	Refactorización y búsqueda en repositorios masivos.
Replit.	Plataforma online unificada.	Medio (despliegue + IA).	Independiente.	Baja-Media.	Construcción y despliegue rápido de apps.
Continue.	Extensión de VS Code.	Medio (chat + agente).	Integrada.	Media.	Automatización de tareas repetitivas dentro del IDE.

Cuadro realizado como elaboración propia

Un nuevo lenguaje

El enfoque de Vibe Coding no solo trae consigo nuevas formas de programar, sino también un vocabulario propio que se fue consolidando con la aparición de herramientas basadas en inteligencia artificial. Estas terminologías son necesarias para comprender cómo se articula esta metodología y para interpretar con claridad los resultados del presente trabajo.

Inteligencia Artificial (IA).

Disciplina de la informática que busca desarrollar sistemas capaces de realizar tareas que normalmente requieren inteligencia humana, como el razonamiento, el aprendizaje o la comprensión del lenguaje natural. En el contexto del desarrollo de software, la IA se materializa principalmente a través de modelos de lenguaje (LLM) y agentes de programación que asisten a los desarrolladores en distintas etapas del ciclo de vida del software. Su incorporación al proceso de codificación representa un cambio de paradigma: del pensamiento algorítmico lineal hacia una colaboración interactiva entre humano y máquina.

Vibe Coding.

Forma de programar que combina el desarrollo tradicional con la asistencia de modelos de inteligencia artificial, utilizando el lenguaje natural como medio principal de interacción. El término fue acuñado de manera irónica por Andrej Karpathy, pero posteriormente adoptado y reinterpretado por la comunidad tecnológica como una metodología emergente orientada al desarrollo asistido por IA.

LLM (Large Language Model).

Un modelo de lenguaje grande es un sistema de inteligencia artificial entrenado con enormes volúmenes de texto y código para comprender y generar lenguaje natural de forma contextual.

En el marco del Vibe Coding, los LLM son el núcleo operativo detrás de herramientas como Cursor o GitHub Copilot, que pueden sugerir, corregir o incluso escribir código completo. Su comportamiento es probabilístico, no determinista, lo que implica que sus respuestas pueden variar ante la misma instrucción.

Coding Agents (Agentes de programación).

Herramientas de IA que no se limitan a sugerir fragmentos de código, sino que pueden planificar, modificar, probar y depurar aplicaciones de manera semiautónoma. Actúan como asistentes técnicos capaces de ejecutar instrucciones complejas dentro de entornos de desarrollo integrados.

Prompts.

Instrucciones redactadas en lenguaje natural que guían el comportamiento del modelo. En el contexto del desarrollo, un prompt puede ir desde una orden simple (“agregá un botón a la vista principal”) hasta una directiva compleja (“implementá arquitectura hexagonal con repositorios y servicios desacoplados”). La calidad del resultado depende en gran medida de la claridad, precisión y contexto del prompt.

MCP (Model Context Protocol).

Protocolo abierto que conecta a los LLM con datos, servicios y herramientas externas. Permite que agentes como Cursor, Continue o Claude Workbench accedan a APIs, bases de datos o archivos del proyecto, ampliando así su conocimiento contextual. Es una de las innovaciones clave que permiten que la IA no trabaje de manera aislada, sino integrada al ecosistema de desarrollo.

PRD (Product Requirements Document).

Documento de requisitos del producto que, en entornos de Vibe Coding, puede generarse automáticamente a partir de interacciones con IA. Contiene flujos de usuario, diagramas y especificaciones funcionales, y actúa como una hoja de ruta del desarrollo que mantiene la coherencia entre los requerimientos, el diseño y el código.

Más allá de las palabras.

Persistencia de contexto.

Mecanismo que permite conservar información a lo largo del ciclo de desarrollo, evitando que el modelo “olvide” los detalles del proyecto entre sesiones. En la práctica, esto se logra mediante documentos de contexto, archivos de reglas o el uso de protocolos como MCP, que permiten a la IA retomar conversaciones o implementaciones anteriores sin pérdida de coherencia.

Reglas y archivos de configuración en Cursor.

Son conjuntos de lineamientos definidos por el usuario para orientar el accionar de la IA en proyectos complejos. Funcionan como un manual de estilo

automatizado, ayudando a mantener consistencia entre capas, evitar errores repetitivos y respetar convenciones arquitectónicas.

Hoy existen repositorios abiertos que contienen configuraciones prediseñadas por ejemplo, Cursor Directory o jabrena/cursor-rules-spring-boot, e incluso frameworks como Laravel Boost, que incorporan buenas prácticas y MCPs recomendados desde su instalación inicial.

Modelos de agentes en Cursor.

Cursor ofrece diferentes modos de agente que definen el nivel de autonomía, las capacidades técnicas y el tipo de interacción que la inteligencia artificial mantiene con el entorno de desarrollo. Cada modo representa una forma distinta de colaboración entre el programador y el asistente, ajustando el equilibrio entre exploración, ejecución y control humano.

Los modos principales disponibles son los siguientes:

- **Agent:** Es el modo más completo y autónomo. Permite a la IA realizar tareas complejas como refactorización, depuración o modificaciones simultáneas en múltiples archivos. En este modo, el agente tiene acceso a todas las herramientas internas de Cursor y puede explorar el proyecto de forma activa, lo que lo convierte en la opción ideal para fases de implementación o reestructuración del código.

- **Ask:** Diseñado para la exploración y el aprendizaje del código sin realizar modificaciones. En este modo, la IA actúa en un entorno de solo lectura, lo que permite al desarrollador formular preguntas, pedir explicaciones o solicitar sugerencias de planificación. Solo las herramientas de análisis y búsqueda están habilitadas, garantizando que no se altere el código existente.

- **Plan:** Utilizado cuando se requiere abordar funciones o tareas que demandan planificación previa. En este modo, el agente elabora planes de acción detallados antes de ejecutar cualquier cambio, realizando preguntas de clarificación si es necesario. Se encuentra habilitado para usar todas las herramientas, pero prioriza la comprensión del contexto y

la precisión antes de modificar el código.

- **Custom:** Pensado para flujos de trabajo personalizados. El usuario puede definir configuraciones propias del agente, ajustando las herramientas disponibles y el comportamiento del modelo según sus necesidades. Este modo resulta especialmente útil en entornos profesionales donde se busca estandarizar procesos o replicar configuraciones de trabajo entre distintos proyectos.

Contexto en Cursor.

Hace referencia a la capacidad del entorno para comprender la estructura completa del proyecto. Incluye la indexación del repositorio, la persistencia de estado (recordando cambios y decisiones) y el conocimiento acumulado a partir de interacciones previas.

Gracias a esto, la IA puede realizar modificaciones coherentes con el diseño existente, en lugar de actuar sobre fragmentos aislados de código.

Forks.

En el contexto del Vibe Coding, los forks se refieren a versiones derivadas de entornos de desarrollo tradicionales, como Visual Studio Code, que son extendidos con capacidades de inteligencia artificial. Ejemplos relevantes incluyen:

- ➢ Cursor, que integra un agente de IA con comprensión contextual.
- ➢ Windsurf, enfocado en la colaboración y el trabajo en equipo.
- ➢ Trae, orientado a la generación automática de código a partir de lenguaje natural.

Diseño en un archivo Markdown.

Algunas herramientas de Vibe Coding permiten generar documentos de diseño en formato Markdown, que funcionan como una guía de desarrollo viva y trazable. Estos documentos suelen incluir:

- Flujos de usuario, describiendo la interacción esperada.

- Diagramas y mockups, que visualizan la interfaz y arquitectura.
- Especificaciones funcionales y vínculos con el código fuente. El uso de Markdown unifica comunicación, diseño y documentación técnica en un formato liviano, editable y accesible desde cualquier entorno de desarrollo.

Caja de herramientas.

En el marco de este Trabajo Final de Carrera, probar el enfoque metodológico de Vibe Coding requiere elegir un conjunto reducido de herramientas que resulten representativas, funcionales y viables para el desarrollo del caso de estudio: una aplicación web de clasificados.

De la amplia variedad de herramientas relevadas en capítulos anteriores, se optó por Cursor y GitHub Copilot.

Cursor.

Cursor es un *fork* de Visual Studio Code que incorpora de manera nativa un agente de inteligencia artificial orientado al desarrollo de software. Se ha convertido en una de las herramientas más representativas del Vibe Coding debido a su capacidad de asistencia en tiempo real, que permite solicitar cambios directamente en lenguaje natural sobre el código. Además, incorpora la función *Composer*, diseñada para ejecutar instrucciones complejas y delegar tareas completas al agente, lo que lo convierte en un aliado poderoso en procesos iterativos. Otra de sus virtudes es la integración con el protocolo MCP (*Model Context Protocol*), que habilita la conexión con servicios y APIs externas para enriquecer el contexto del trabajo. También destaca por su persistencia de contexto, ya que mantiene información sobre el estado del proyecto mediante reglas y archivos de configuración, aspecto crucial cuando el código comienza a escalar en tamaño y complejidad.

En este Trabajo Final de Carrera, Cursor será la herramienta principal para la generación y modificación del backend y frontend de la aplicación, poniendo a prueba su capacidad de interpretar *prompts* complejos y de gestionar el contexto en las distintas fases de desarrollo. En este sentido, Cursor puede considerarse la encarnación más clara del concepto de Vibe Coding, dado que ofrece una interacción fluida entre lenguaje natural y código, reduciendo el esfuerzo manual y acelerando las iteraciones.

El peso de la decisión.

La elección de Cursor no se fundamenta únicamente en sus características técnicas, sino también en su relevancia y el impacto que han tenido en la

comunidad de desarrolladores. Cursor fue seleccionada por representar de manera fiel la esencia del Vibe Coding y por haber ganado notoriedad rápidamente como una de las primeras herramientas en poner a prueba esta forma de programar.

Desarrollo de la aplicación.

Para llevar adelante este Trabajo Final de Carrera se plantea el desarrollo de una aplicación web de clasificados, que funcionará como caso de estudio para analizar y poner en práctica el enfoque de Vibe Coding. La elección de este tipo de sistema responde a la necesidad de contar con un proyecto de alcance acotado, manejable en tiempo y complejidad, pero que al mismo tiempo permita incorporar componentes centrales de cualquier proceso de desarrollo de software, tales como gestión de usuarios, seguridad, persistencia de datos y validación de información.

El objetivo de trabajar sobre un sistema de este tipo no es construir un producto final listo para su explotación comercial, sino disponer de un laboratorio controlado en el que se puedan observar con claridad las intervenciones de la inteligencia artificial y evaluar cómo influyen en cada etapa del ciclo de vida del software. En este sentido, se prioriza la simplicidad en la definición de requerimientos iniciales, pero al mismo tiempo se incluyen ciertos elementos “no negociables”, como la seguridad en el inicio de sesión, la creación de pruebas automáticas y la moderación de contenido, de modo de garantizar que el caso de estudio representa desafíos reales de la práctica profesional.

Caso de estudio:

El sistema que se desarrollará como caso de estudio consiste en una aplicación web de clasificados en línea, concebida como un entorno simple pero con los elementos necesarios para poner a prueba las prácticas de desarrollo asistido por inteligencia artificial.

En términos generales, la aplicación permitirá que cualquier visitante, sin necesidad de autenticación, pueda visualizar los clasificados activos disponibles. Cada clasificado contendrá información básica: un título, una descripción, un precio, un estado (activo, vendido, pendiente o desactivado), el vendedor que lo publica y la categoría a la que pertenece.

Las categorías estarán definidas por un nombre y la lista de clasificados asociados, permitiendo organizar los avisos de manera jerárquica.

El sistema contará con usuarios registrados, diferenciados por dos roles principales: vendedor y administrador.

- ➢ Todo nuevo usuario que se registre recibirá por defecto el rol de vendedor. Los vendedores tendrán la posibilidad de gestionar su perfil (editar datos personales como nombre, apellido, correo, nombre de usuario, teléfono y contraseña), así como administrar sus propios clasificados: crearlos, editarlos, eliminarlos y consultarlos.
- ➢ Los administradores tendrán responsabilidades de gestión global. Serán los encargados de aprobar los clasificados creados por los vendedores antes de su publicación, garantizando un control de calidad sobre los avisos que llegan a la plataforma. Además, podrán asignar el rol de administrador a otros usuarios, así como crear, editar y eliminar categorías.

De esta manera, la aplicación integra funcionalidades mínimas pero representativas de un sistema web moderno: autenticación de usuarios, manejo de roles y permisos, CRUD de entidades principales y control administrativo.

Es importante remarcar que no se incluirá lógica de negocio compleja (como pasarelas de pago, sistemas de mensajería, algoritmos de recomendación o métricas de comportamiento). El objetivo es mantener un alcance limitado y controlado, lo suficientemente sencillo para centrarse en la exploración de prácticas de Vibe Coding, pero con desafíos relevantes como seguridad, validación de datos, flujo de aprobación y gestión de roles.

Arquitectura del sistema

Para el desarrollo de la aplicación se adoptó una arquitectura en capas, un enfoque clásico en la ingeniería de software que facilita la separación de responsabilidades, la mantenibilidad y la escalabilidad del sistema. Este modelo organiza el código en diferentes módulos, donde cada uno cumple un rol específico dentro del flujo de la aplicación.

En el caso particular de este proyecto, la estructura del backend se organizó en las siguientes carpetas:

- models: contiene las entidades del sistema, que representan la estructura de los datos principales (usuarios, clasificados, categorías, roles y estados).
- repository: implementa el acceso a la base de datos y las operaciones CRUD sobre las entidades.
- DTO(Data Transfer Objects): define objetos intermedios que permiten trasladar información entre capas de manera más controlada y eficiente.
- services: encapsula la lógica de negocio y coordina la interacción entre repositorios y controladores.
- controllers: expone las funcionalidades a través de endpoints, gestionando las solicitudes y respuestas de la aplicación.
- exceptions: centraliza el manejo de errores y excepciones, asegurando respuestas consistentes y mejorando la robustez del sistema.

Esta organización busca que cada parte del código tenga un propósito bien definido, reduciendo la complejidad y favoreciendo la reutilización.

Tecnologías utilizadas

El sistema se construyó utilizando un conjunto de tecnologías modernas, seleccionadas por su madurez, soporte de comunidad y capacidad para integrarse de manera eficiente:

Backend: se empleó Spring Boot sobre Java, un framework ampliamente adoptado para el desarrollo de aplicaciones web y servicios REST. Permite estructurar el proyecto bajo una arquitectura en capas de forma natural, con un ecosistema rico en librerías y extensiones.

Frontend: se utilizó React con Vite como entorno de construcción y desarrollo. React ofrece una estructura modular basada en componentes, lo que facilita la construcción de interfaces dinámicas y escalables, mientras que Vite mejora el rendimiento en el proceso de compilación y despliegue.

Base de datos: se eligió PostgreSQL, un sistema de gestión de bases de datos relacional reconocido por su robustez, seguridad y soporte para funcionalidades avanzadas, lo que lo convierte en una opción confiable para proyectos web de mediana y gran escala.
La elección de este stack tecnológico responde no solo a sus características individuales, sino también a su integración armónica, que permite un flujo de trabajo coherente desde la interfaz de usuario hasta la persistencia de datos. Además, todas son tecnologías consolidadas en la industria, lo que garantiza documentación abundante, buenas prácticas probadas y la posibilidad de extender el proyecto en caso de ser necesario.

De esta manera, la aplicación de clasificados se convierte en un instrumento metodológico: más que un fin en sí mismo: es el medio a través del cual se pondrá a prueba la hipótesis central de este trabajo, que sostiene que el Vibe Coding puede ser utilizado como un enfoque válido y productivo en proyectos de software, siempre que se acompañe de un conjunto de prácticas claras y replicables.

Paso a paso

El desarrollo de software asistido por inteligencia artificial, en particular bajo el enfoque del Vibe Coding, supone un cambio profundo respecto a los métodos tradicionales. No se trata únicamente de “aceptar sugerencias” de un asistente, sino de construir un flujo de trabajo donde las herramientas de IA puedan asumir distintos roles (gestor de proyecto, ingeniero de sistemas, programador), siempre bajo la supervisión crítica del desarrollador humano.

A lo largo de este Trabajo Final de Carrera se identificaron una serie de prácticas que permiten aprovechar de manera más eficiente herramientas como Cursor y GitHub Copilot. Estas prácticas se presentan organizadas siguiendo el ciclo de vida del software: primero la preparación del entorno, luego la documentación inicial, más tarde la planificación de la implementación, la ejecución del código, la resolución de problemas y, finalmente, la integración entre componentes.

El propósito de este capítulo no es establecer un manual rígido, sino sistematizar lo aprendido en la experiencia práctica. Cada práctica documentada surge de pruebas reales, donde se registraron aciertos, dificultades y ajustes necesarios. En cada apartado se incluyen los *prompts* empleados, con el objetivo de ilustrar cómo se llevó a cabo la interacción con la IA y dar mayor transparencia al proceso.

Si bien para este caso de estudio se eligió un conjunto específico de tecnologías (Spring Boot en el backend, React con Vite en el frontend y PostgreSQL como base de datos) la guía busca mantener un carácter universal. Es decir, las prácticas documentadas están pensadas para ser adaptables a otros lenguajes, frameworks o herramientas de IA, de manera que puedan servir como referencia para proyectos de distinta naturaleza y escala.

De este modo, el capítulo funciona como una guía aplicable para futuros desarrolladores que busquen aplicar el Vibe Coding en proyectos propios, mostrando tanto los beneficios alcanzados como los cuidados que deben tenerse para no perder control sobre el resultado final.

Manejo de frameworks y entornos de desarrollo

Al trabajar con Cursor en proyectos que utilizan frameworks basados en Java, como Spring Boot, se observaron ciertas limitaciones que condicionan la preparación del entorno. En particular, la herramienta no es capaz de descargar directamente el archivo comprimido generado por Spring Initializr, que constituye el punto de partida habitual de un proyecto. Este paso requiere una intervención manual por parte del programador, quien debe crear el proyecto inicial desde el portal oficial y luego cargarlo en el repositorio que será usado en Cursor.

Una vez resuelto este paso inicial, Cursor sí puede manipular con normalidad la configuración del entorno: agregar dependencias en pom.xml o build.gradle, definir el archivo application.properties (o application.yml), organizar paquetes y, en general, adaptar la estructura de acuerdo con los requerimientos del proyecto.

A partir de esta experiencia, la práctica recomendada es la siguiente:

- **Generar manualmente el proyecto base** en Spring Initializr, incluyendo únicamente las dependencias esenciales (web, seguridad, JPA y PostgreSQL).
- **Subir este esqueleto inicial al repositorio**, asegurándose de que compile y se ejecute correctamente antes de comenzar a interactuar con la IA.
- **Cargar el proyecto al contexto de cursor** y delegarle la tarea de ajustar el entorno y añadir dependencias adicionales en función de las necesidades específicas del proyecto.

A diferencia de lo que ocurre con Spring Boot, donde la IA no puede crear por sí sola el proyecto inicial, en entornos de frontend como React esta restricción no existe. Cursor puede inicializar un proyecto de frontend directamente mediante comandos como *npm create vite@latest o npx create-react-app*. Esto evidencia que el grado de intervención manual depende del tipo de framework:

en algunos casos, como React, la IA puede encargarse de todo el arranque; en otros, como Spring Boot, necesita que el desarrollador le prepare la base inicial.

En definitiva, la clave está en conocer la herramienta seleccionada de vibe coding para entender estas limitaciones y anticiparse a ellas. Si el entorno está correctamente configurado desde el inicio, la IA puede dedicarse a aportar valor en la escritura de lógica de negocio, pruebas y refactorización, en lugar de quedar bloqueada en tareas administrativas.

Definición de reglas y uso de MCP

El segundo hallazgo importante está relacionado con la definición temprana de reglas de trabajo y la utilización de MCP (Model Context Protocol).

REGLAS

Cursor permite definir archivos de reglas y protocolos de contexto que orientan el accionar de la inteligencia artificial durante el desarrollo. Estas reglas funcionan de manera similar a un manual de estilo o a un documento de lineamientos arquitectónicos, ya que establecen qué se espera del sistema, cómo debe estructurarse el código y bajo qué convenciones debe operar la IA.

Las reglas se redactan en formato Markdown y pueden ser creadas por el propio programador según las necesidades del proyecto. Sin embargo, con la expansión del Vibe Coding y el trabajo colaborativo en comunidades técnicas, comenzaron a surgir repositorios y bibliotecas públicas que ofrecen conjuntos de reglas ya estructuradas para distintos lenguajes, frameworks y metodologías. Ejemplos de estos recursos incluyen el directorio Cursor Directory [7], que reúne configuraciones predefinidas para diversos entornos, y el repositorio jabrena/cursor-rules-spring-boot [6], donde se publican reglas optimizadas para proyectos en Java con Spring Boot.

De forma similar, frameworks como Laravel han comenzado a integrar estas prácticas de manera nativa a través de extensiones como Laravel Boost, una dependencia que incorpora reglas, estructuras de prompt y MCPs recomendados para proyectos desarrollados en este ecosistema. Este tipo de herramientas marca una tendencia hacia entornos de desarrollo cada vez más estandarizados y listos para trabajar con IA desde su instalación inicial.
Estos espacios de intercambio facilitan la adopción de buenas prácticas, permitiendo que los equipos de desarrollo partan de una base sólida y la adapten a los requerimientos de cada caso en lugar de redactar todo desde cero.

En el marco de este proyecto, se elaboró un conjunto propio de reglas orientadas a mantener la coherencia, la organización por capas y la calidad del código dentro del entorno Spring Boot. Para este caso de estudio, las reglas

fueron redactadas por el programador y se dividieron en dos grupos: reglas generales, aplicables a toda la arquitectura del sistema, y reglas específicas por capa, destinadas a guiar el comportamiento de la IA en controladores, servicios, repositorios y modelos. A continuación, se presentan las reglas implementadas.

General (Aplica a toda la aplicación)

Estilo y estructura del código

- *Escribe código limpio, eficiente y bien documentado.Comenta cada bloque lógico relevante para explicar su propósito.*
- *Usa nombres descriptivos de métodos, variables y clases.*
- *Convenciones de nomenclatura:*
    - *Clases: PascalCase → UsuarioController, OrderService*
    - *Métodos y variables: camelCase → findUserById, isOrderValid*
    - *Constantes: ALL_CAPS → MAX_RETRY_ATTEMPTS, DEFAULT_PAGE_SIZE*
- *Mantén un diseño cohesivo y de bajo acoplamiento, siguiendo los principios SOLID.*

Modelo (model/)

Construcción y validez del objeto

- *Siempre construir por constructor.*
- *No uses setters para inicializar objetos.*
- *Valida todos los atributos en el constructor.*
- *Cada validación se implementa como un método privado assert{Condición}.*

- *Lanza excepciones de tipo RuntimeException o subclases específicas (ej. IllegalArgumentException, InvalidOperationException) cuando una validación falla.*
- *Define mensajes de error como constantes static final para mantener consistencia.*

  ejemplo: static final String ERROR_NOMBRE_INVALIDO =

  "El nombre no puede estar vacío.";

```
public Usuario(String nombre, String email) {
    assertNombreValido(nombre);
    this.nombre = nombre;
    this.email = email;
}
private void assertNombreValido(String nombre) {
    if (nombre == null || nombre.isBlank()) {
                                        throw new
IllegalArgumentException(ERROR_NOMBRE_INVALIDO);
    }
}
```

Inmutabilidad y acceso al estado

- *Evita getters y setters anémicos. Solo define métodos de acceso si aportan semántica o lógica de negocio (ej. esActivo()).*
- *Para colecciones internas:*
  - *Devuelve siempre una vista inmodificable (Collections.unmodifiableList()).*

- *Agrega o elimina elementos solo mediante métodos de negocio explícitos (usuario.agregarRol()).*

Comportamiento y lógica de negocio

- *Tell, don't ask: no expongas el estado interno del objeto; delega la acción al propio objeto.*
- *Toda modificación de estado debe realizarse mediante un método de negocio expresivo, que encapsule sus reglas y validaciones.*

```
public void publicar() {
    if (this.estado != EstadoClasificado.BORRADOR) {
        throw new InvalidOperationException("Solo se puede publicar un clasificado en estado BORRADOR.");
    }
    this.estado = EstadoClasificado.PUBLICADO;
    this.fechaPublicacion = LocalDateTime.now();
}
```

Servicios (service/)

Principios generales

- *Cada servicio debe encargarse de un único agregado o contexto de dominio.*
- *No mezcles responsabilidades.*

Ejemplo: UsuarioService gestiona usuarios, AuthService gestiona autenticación.

- *Los servicios interactúan solo con repositorios, nunca con EntityManager.*

- *No expongas repositorios a controladores.*
- *No dupliques las validaciones que ya están en el modelo.*

Rol del servicio: Orquestador

- *Los servicios deben ser delgados: orquestan operaciones, pero no implementan lógica de negocio.*

  Flujo típico:

  - *Recuperar entidades del repositorio.*
  - *Invocar métodos del dominio.*
  - *Guardar las entidades modificadas.*

Ejemplo correcto:

```
@Transactional
public void publicarClasificado(Long id) {
    Clasificado clasificado = clasificadoRepository.findById(id)
                                        .orElseThrow(() -> new
ClasificadoNotFoundException("Clasificado no encontrado"));
    clasificado.publicar();
    clasificadoRepository.save(clasificado);
}
```

Transacciones

- *Usa @Transactional en métodos que modifican estado.*
- *Usa @Transactional(readOnly = true) en métodos de solo lectura.*

Mapeo de DTOs

- *El servicio es responsable de:*
  - *Convertir RequestDTO → Entidad.*

- *Convertir Entidad → ResponseDTO.*

- *Los DTOs no deben ser pasados al modelo.*
- *Si el mapeo es complejo, crea una clase Mapper.*

private UsuarioResponseDTO toResponse(Usuario usuario) {

return new UsuarioResponseDTO(usuario.getId(), usuario.getNombre(), usuario.getEmail());

}

Inyección de dependencias

- *Usa inyección por constructor.*
- *Declara las dependencias como private final.*
- *No uses @Autowired sobre campos.*

ejemplo bien:

@Service

public class ClasificadoService {

private final ClasificadoRepository clasificadoRepository;

public ClasificadoService(ClasificadoRepository clasificadoRepository) {

this.clasificadoRepository = clasificadoRepository;

}

}

Manejo de excepciones

- *No captures excepciones de negocio; deja que se propaguen al @ControllerAdvice.*

- *Solo captura y adapta excepciones de infraestructura.*

Controladores (controller/)

Rol y responsabilidad

- *Los controladores representan la interfaz pública (API REST).*
- *No contienen lógica de negocio ni de persistencia.*
- *Solo traducen entre HTTP ↔ dominio.*

Diseño RESTful

- *Usa verbos HTTP apropiados y códigos de estado coherentes:*
    - *GET → lectura (200 OK)*
    - *POST → creación (201 Created)*
    - *PUT/PATCH → actualización (200/204)*
    - *DELETE → eliminación (204)*
- *No retornes entidades del dominio.*
- *Siempre retorna ResponseDTOs.*

Interacción con servicios

```
@PostMapping
public ResponseEntity<UsuarioResponseDTO> registrar(@Valid @RequestBody UsuarioRequestDTO dto) {
        UsuarioResponseDTO response = usuarioService.registrarUsuario(dto);
        return ResponseEntity.status(HttpStatus.CREATED).body(response);
}
```

Validaciones

- *Valida con @Valid y anotaciones de Bean Validation.*
- *No repitas validaciones del modelo o servicio.*

Manejo centralizado de errores

- *Usa @ControllerAdvice para capturar excepciones y traducirlas a respuestas HTTP.*

```
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(UsuarioNotFoundException.class)
                    public ResponseEntity<ErrorResponseDTO>
handleUsuarioNotFound(UsuarioNotFoundException ex) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
            .body(new ErrorResponseDTO("Usuario no encontrado"));
    }
}
```

Seguridad

- *Configura CORS y seguridad globalmente.*
- *Usa @PreAuthorize o @Secured para autorización.*
- *No verifiques roles ni tokens manualmente en los controladores.*

Repositorios (repository/)

Propósito

- *Gestionan el acceso a datos persistentes.*
- *Aíslan la lógica de persistencia del resto del sistema.*

Principios fundamentales

- *No incluir lógica de negocio.*
- *Usar interfaces que extienden de JpaRepository o similares.*
- *Usar nombres descriptivos en los métodos (findByEmail, buscarPorRangoDePrecio).*
- *Manejar las transacciones desde la capa de servicio, no en el repositorio.*

Ejemplo correcto

```
public interface ProductoRepository extends JpaRepository<Producto, Long> {

    Optional<Producto> findByCodigo(String codigo);

    List<Producto> findByCategoria(String categoria);

    @Query("SELECT p FROM Producto p WHERE p.precio BETWEEN :min AND :max")
    List<Producto> buscarPorRangoDePrecio(@Param("min") double min, @Param("max") double max);
}
```

 # Testing Automatizado con JUnit 5.13

1. Nombre claro y descriptivo

- Nombra los métodos de test siguiendo este patrón:

`cuestionAEvaluar_resultadoEsperado`

- Agrega la anotacion `@DisplayName` explicando en lenguaje natural el

objetivo del test con este formato:

```java

@DisplayName("CuestonAEvaluar resultadoEsperado")

```

2. No uses Mock, Stubs o Fakes

- Los tests unitarios corren todos en memoria ejutando el código real.

- No uses mocks, stubs o fakes, ya que no son necesarios y complican

la lectura del test.

- Solo usaremos mocks en consumo de servicios externos (pagos online,

emails, etc).

3. Estructura del test

- Cada test debe tener la siguiente estructura:

```java

@Test

@DisplayName("Nombre del test")

void testNombreDelMetodo() {

    // Setup: Preparar el escenario

    // Ejercitación: Ejecutar la acción a probar
```

// Verificación: Verificar el resultado esperado

}

```

- Incorpora comentarios de la estructura del test para facilitar la comprensión del código.

## 4. Un solo caso de prueba por test

- Cada test debe evaluar un único caso de prueba.

- Si necesitas evaluar múltiples casos, crea un test separado para cada uno.

## 5. Usa Asserts claros y descriptivos

- Utiliza aserciones claras y descriptivas agregando mensajes que expliquen el propósito de la verificación.

- Por ejemplo:

```java

assertEquals(expectedValue, actualValue, "El valor esperado no coincide con el valor actual");

```

6. Probá Casos límites

- Valores nulos (null)

- Listas vacías o inputs vacíos

- Números negativos o fuera de rango

- Estados inválidos o excepciones esperadas
```

7. Verificá excepciones correctamente

- Utiliza `assertThrows` para verificar que se lanza la excepción esperada.

- En general mi código lanzará RuntimeException, pero verificaló primero.

- Ejemplo:

```java
var ex = assertThrows(RuntimeException.class, () -> {
        // Código que debería lanzar la excepción
    });
    assertEquals("Mensaje de error esperado", ex.getMessage());
```

- Sobre el "Mensaje de error esperado", antes de poner a constante dura en el test verifica que la constante definida en el codigo real y si es asi úsala.

8. Testing de Integración

- Usamos test-data.sql como set up inicial de la BD.

- Siempre usar como beforeEach el truncate ya que resetea la base de datos despues de cada test que corre:

```java
void beforeEach() {
    emf.getSchemaManager().truncate();
}
```

- No incluyas casos de tests que pueden ser testeados con tests unitarios

MCP (Model Context Protocol)

Uno de los aportes más significativos de Cursor es la posibilidad de integrar MCP (Model Context Protocol), un estándar abierto diseñado para que los agentes de IA puedan conectarse con fuentes de información externas. En términos simples, el MCP funciona como un “puente” que enriquece el contexto de la herramienta de IA; en lugar de limitarse al archivo actual o a la memoria temporal de la sesión, el agente puede consultar datos adicionales y utilizarlos en su razonamiento.

El impacto actual del MCP en herramientas como Cursor es notable. Al permitir que la IA tenga acceso a un “universo ampliado de información”, se reduce el riesgo de que el agente produzca código incoherente o fuera de contexto. Además, abre la puerta a escenarios más complejos: integración con repositorios Git para leer el historial de cambios, o conexión con plataformas de despliegue para ajustar pipelines de CI/CD.

Podría decirse que el MCP es uno de los pilares del Vibe Coding: ya no se trata solo de escribir en lenguaje natural y obtener código, sino de dotar a la IA de un contexto vivo y dinámico que le permita tomar mejores decisiones. En este sentido, su uso representa una forma de ingeniería de contexto, donde el desarrollador define cuidadosamente qué información extra debe estar disponible para la IA, evitando improvisaciones y asegurando un flujo de trabajo más confiable.

Para el caso de estudio propuesto, se utilizaron los siguientes MCP:

Task-Master-AI: pensado como un asistente externo que organiza y gestiona tareas. Su función es coordinar pedidos complejos, descomponerlos en subtareas y mejorar el flujo de trabajo dentro del agente.

"task-master-ai":

{ "command": "npx",

"args": ["-y", "--package=task-master-ai", "task-master-ai"],

"env":

}

Context7: servicio en línea que proporciona contexto adicional desde una API externa. Permite enriquecer al agente con información de referencia alojada fuera del proyecto local, mejorando la calidad de las respuestas y la coherencia en las decisiones.

"context7":

{

"url": "https://mcp.context7.com/mcp"

}

Es importante conocer y analizar en detalle las herramientas seleccionadas. Estas prácticas pretenden ser lo más completas posibles, integrando conceptos como reglas de trabajo y MCP para lograr un producto robusto. Sin embargo, no todas las herramientas asociadas al Vibe Coding son capaces de soportar estas integraciones. Cada una presenta limitaciones distintas en cuanto a manejo de contexto, compatibilidad de reglas o acceso a protocolos externos.

Por ello, resulta clave evaluar las capacidades de cada herramienta antes de adoptarla en un proyecto real, eligiendo la más adecuada según las necesidades específicas.

Práctica: Generación del PRD

Descripción breve: El Product Requirements Document (PRD) es un documento clave que traduce los requisitos de usuario en especificaciones claras y organizadas. Define objetivos, historias de usuario y criterios de aceptación, sirviendo como guía inicial y referencia continua a lo largo del desarrollo. En este caso, la IA asume el rol de project manager para elaborar este documento de manera dinámica

[Contexto del sistema]: Este bloque introduce el sistema sobre el cual se va a trabajar. Su función es darle a la IA la visión inicial del proyecto, no debe ser una línea vaga ni algo “sencillo”. Debe explicar con claridad qué sistema queremos obtener: su propósito, alcance y restricciones. Cuanto más detallado y concreto sea, mejor resultado se obtendrá.

[Rol asignado a la IA]: Aquí se define desde qué perspectiva trabajará la IA. Para un PRD, el rol natural es el **Project Manager**, ya que es quien se encarga de traducir necesidades en requisitos.

[Tarea principal]: Es la instrucción central, lo que se espera que haga la IA en este contexto. Aca se establece claramente el objetivo, para evitar ambigüedades en la salida. Se pedirá que analice la descripción del sistema y lo transforme en un PRD. Se puede ampliar indicando que el PRD será la base del backlog, que debe detectar inconsistencias o que debe formular preguntas si falta información.

[Contenido requerido] Se especifica qué secciones debe incluir el PRD. Esto asegura que el documento generado tenga los elementos mínimos necesarios:

- Historias de usuario.
- Criterios de aceptación.
- Preguntas de aclaración.

De forma opcional se pueden sumar otros elementos: supuestos, requisitos no funcionales, riesgos, dependencias, métricas de éxito o priorización de funcionalidades.

[Reglas de interacción]

Define cómo debe comportarse la IA si encuentra vacíos de información o contradicciones. La clave está en que pregunte antes de inventar datos, asegurando precisión y relevancia en el documento.

[Formato de entrega] Indica cómo debe presentarse el PRD. Lo más común es un archivo en Markdown, guardado dentro del directorio docs/ del repositorio, con un nombre estandarizado (ejemplo: docs/prd.md). También se puede definir la estructura interna del documento (ejemplo: títulos como "Introducción", "Historias de Usuario", etc.).

[Restricciones] Por último, este bloque fija los límites. El más importante: la IA **NO** debe generar código, solo el documento. También se puede aclarar que **NO** debe agregar explicaciones fuera del PRD.

prompt:

> "**[Descripción del sistema que se desea realizar.]**
>
> **[Rol asignado a la IA]** Sos el project manager de esta aplicación. **[Tarea principal]** Tu tarea es analizar el sistema descrito arriba y convertir los requisitos de usuario en un documento de requisitos de producto (PRD) **[Contenido requerido]** que incluyan historias de usuario para nuevas funcionalidades. Añade criterios de aceptación. **[Reglas de interacción]** Si no tienes suficiente información, pregúntame sobre la funcionalidad. **[Formato de entrega]** Inserta el diseño en un archivo Markdown en el directorio docs del repositorio. El nombre del archivo debe ser [*nombre del sistema*]-prd.md . El archivo debe estar formateado en Markdown e incluir encabezados y viñetas. **[Restricciones] No** codifiques nada, solo crea el documento y guardalo para futuras consultas."

Sugerencias para esta etapa

- Mantener el PRD bajo control de versiones dentro del repositorio (/docs) para garantizar trazabilidad y facilitar actualizaciones.

- pedir explícitamente que no se codifique ya que es una etapa de análisis.
- Leer y modificar el PRD las veces que sean necesarias hasta conseguir el resultado que queremos.
- No especificar elementos técnicos de la programación.

Práctica: Generación del Documento de Especificaciones Técnicas

Descripción breve: El documento de especificaciones técnicas traduce los requerimientos funcionales definidos en el PRD a un diseño detallado de implementación. Incluye la arquitectura, entidades, endpoints y estrategias técnicas necesarias para guiar la construcción del sistema. Con la asistencia de IA, este documento puede generarse automáticamente a partir del PRD y del estado del código existente, actuando como un puente entre el análisis y la implementación.

Estructura del prompt:

[Contexto del sistema]

Este bloque explica el punto de partida: el PRD ya fue generado y contiene los requisitos funcionales. Ahora se requiere traducir ese documento en un diseño técnico detallado. El foco está en cómo se implementará el sistema, asegurando que los criterios de aceptación definidos en el PRD puedan cumplirse.

[Rol asignado a la IA]

Aquí se define que la IA debe actuar como Arquitecto de Software, rol encargado de diseñar soluciones técnicas a partir de requisitos funcionales. Este rol debe pensar en términos de arquitectura, integración y pasos de implementación, **no de programación directa.**

[Tarea principal]

La IA debe convertir el PRD en un documento técnico que describa cómo se implementará el sistema. Su función es definir la arquitectura, las entidades, los endpoints y las relaciones entre componentes, estableciendo los pasos y criterios necesarios para guiar la fase de programación. No debe generar código, sino un diseño técnico claro, completo y coherente con los requisitos definidos.

[Contenido requerido]

El documento de Especificaciones Técnicas debe incluir como mínimo:

- ➢ Descripción de la arquitectura o diseño propuesto.
- ➢ Pasos detallados de implementación.
- ➢ Puntos de integración con código existente.
- ➢ Supuestos explícitos si algo no está claro.
- ➢ Referencias a los criterios de aceptación del PRD.

Puede también incluir: riesgos técnicos, dependencias externas, tecnologías utilizadas, convenciones o limitaciones conocidas. No está de más recordar, que cuanto más detallamos los requerimientos, mejor resultado obtendremos.

[Reglas de interacción]

Si algo en el PRD no está claro, la IA debe preguntar al usuario antes de inventar datos.Si necesita asumir algo, debe dejarlo expresado en el documento como suposición explícita.

[Formato de entrega]

En este apartado, se define la forma en que se entrega el resultado, una de las más comunes es: diseño de un archivo Markdown, ubicado en el directorio docs/ del repositorio.El nombre del archivo puede terminar ser [nombre del sistema]- especificaciones-tec. También se puede definir la estructura interna del documento exigiendo que debe usar encabezados y viñetas para mantener claridad y organización.

[Restricciones]

La IA no debe generar código fuente en esta etapa. El resultado debe ser únicamente un documento descriptivo y técnico que guíe la implementación. Tampoco debe agregar explicaciones extra fuera del documento.
prompt:

> "**[Rol asignado a la IA]** Eres arquitecto de software de esta aplicación. **[Contexto del sistema]** Tu gerente de producto te proporcionó el PRD adjunto con los requisitos funcionales para un nuevo sistema. **[Tarea principal]** Tu tarea es diseñar la

implementación y garantizar que se cumplan todos los criterios de aceptación. Escanea la base de código actual para encontrar puntos de integración. **[Contenido requerido]** Crea una guía paso a paso que detalle cómo implementar tu diseño. **[Restricciones] NO INCLUYAS CÓDIGO FUENTE**. **[Reglas de interacción]** Si algo no está claro, pregúntame sobre el PRD o la implementación. Si necesitas hacer suposiciones, indícalas claramente. **[Formato de entrega]** Inserta el diseño en un archivo Markdown en el directorio docs. El archivo debe tener el mismo nombre que el PRD reemplazando prd por especificaciones-tec. El archivo debe estar en Markdown con encabezados y viñetas."

Sugerencias para esta etapa

- Usar este documento también como base de documentación de decisiones arquitectónicas dentro del repositorio.
- Tratarlo como un artefacto vivo, no definitivo: debe evolucionar junto con el código y el alcance del sistema.

Práctica: Plan de Ejecución en Etapas y Fases

Descripción breve

El plan de ejecución en etapas y fases es una estrategia que busca organizar el desarrollo de software en pasos pequeños, claros y controlables. En el contexto del Vibe Coding, esta práctica es crucial, ya que las herramientas como Cursor no siempre mantienen el contexto completo entre sesiones. Al dividir el trabajo en fases bien definidas, se asegura la continuidad y coherencia del proyecto, evitando pérdidas de información o desviaciones en los objetivos. El plan de ejecución constituye la pieza clave que conecta el análisis (PRD y especificaciones técnicas) con la fase de programación. Su objetivo es transformar los documentos previos en un plan detallado y estructurado, pensado para guiar al programador sin ambigüedades.

Estructura del prompt:

[Contexto del sistema]

Este bloque establece el punto de partida: ya se cuenta con el PRD y el documento de Especificaciones Técnicas. La necesidad ahora es diseñar un plan detallado que guíe la implementación paso a paso, asegurando control y trazabilidad en el proceso de codificación.

[Rol asignado a la IA]

La IA debe asumir el rol de Ingeniero en Sistemas, encargado de transformar los documentos de análisis y diseño (PRD + Especificaciones-Tec) en un plan estructurado de implementación. No es todavía programador: aquí diseña el plan que otro rol (desarrollador/implementador) ejecutará después.

[Tarea principal]

La tarea central es elaborar un plan de trabajo por capas, dividido en etapas y fases, con instrucciones concretas, secuenciales y ejecutables. El objetivo es que un programador humano o un agente IA posterior pueda seguir el plan sin improvisar ni desviarse del diseño.

[Contenido requerido]

El plan debe incluir como mínimo:

- ➢ Estructura de carpetas y archivos iniciales.
- ➢ Etapas de implementación, cada una subdividida en fases.
- ➢ En cada fase: descripción precisa de qué archivos modificar, qué cambios realizar y cuáles son los criterios de aceptación.
- ➢ Limitación estricta: cada fase no puede modificar más de 3 archivos.
- ➢ Claridad total: instrucciones no ambiguas, finitas y detalladas.
- ➢ Al finalizar cada etapa, se crea un documento de implementación que el programador va a completar para especificar qué hizo y si se quiere o requiere, cómo se sigue de acuerdo al plan.

No está de más aclarar que, dependiendo de la herramienta que se seleccione y sus limitaciones, se puede pedir que cada etapa y cada fase tenga un mínimo contexto de lo que se debe hacer. Es una buena práctica para ayudar a los LLM a no perder el contexto.

[Reglas de interacción]

Esta sección, aunque no visible en el prompt, destaca la importancia de la revisión y validación humana del plan presentado por la IA. El desarrollador debe analizar cuidadosamente el plan y, si es satisfactorio, proceder con la siguiente etapa. En caso contrario, se requiere una interacción continua con la IA para refinar y ajustar el plan hasta que cumpla con los requisitos deseados.

[Formato de entrega]

En este apartado, se define la forma en que se entrega el resultado, una de las más comunes es: diseño de un archivo Markdown, ubicado en el directorio docs/ del repositorio.El nombre del archivo puede ser **[nombre del sistema]**-especificaciones-tec. También se puede definir la estructura interna del documento exigiendo que debe usar encabezados y viñetas para mantener claridad y organización.

[Restricciones]

Durante esta práctica no se debe generar código en ningún momento, sino únicamente instrucciones precisas que conformen el plan de implementación. El resultado esperado es un documento técnico completo, claro y ordenado, capaz de guiar el proceso de desarrollo paso a paso. Dicho documento debe contener el nivel de detalle suficiente para que otro rol —como el programador o implementador— pueda ejecutarlo sin necesidad de interpretaciones adicionales ni decisiones fuera del plan establecido.

prompt:

> "**[Rol asignado a la IA]** sos un arquitecto de software, quiero que crees otro archivo en modo de plan, **[Contexto del sistema]** con el documento PRD y el espec-tec. **[Tarea principal]** Arma un plan de trabajo por capas que sea minucioso. Es decir, un documento que se lo entreguemos al implementador/desarrollador y que lo vaya ejecutando por capas finitas y bien definidas.**[Formato de entrega]** este plan va a estar escrito en markdown. El nombre del documento va a ser classiclick-plan-ejecucion.md.**[Contenido requerido]** en primer lugar, va a definir las estructuras a nivel de carpetas. luego va a definir las etapas según el criterio del ingeniero. Cada etapa podrá contener fases, todas las que considere necesario. **[Restricciones]** esas etapas NO pueden editar más de 3 archivos. En caso de tener que tocar más de 3 archivos es necesario generar otra fase. Como dije, cada fase debe tener bien especificado que hacer en cada archivo o clase editada, detalles no ambiguos, finitos y precisos. El programador sólo debe hacer lo que dice ahí, ni más ni menos. **[Contenido requerido]**Es necesario que al comienzo de cada etapa, agregues contexto de lo que se planea hacer en esa etapa. luego, al comienzo de cada fase, también agregar contexto de lo que se debe hacer en esa fase específicamente."

Sugerencias para esta etapa

- Revisar el plan ya que, por lo general, la primera vez se omiten fases. Ajustar el plan las veces que sea necesario hasta lograr lo que queremos.

- Mantener el documento de implementación en el repositorio, junto a las reglas y especificaciones, como registro histórico para consultas del programador.

Ejecución del Plan con Rol de Programador en Cursor

Descripción breve: En esta práctica, la inteligencia artificial asume el rol de un programador senior dentro de un entorno controlado, con el objetivo de ejecutar el plan de implementación previamente definido. El propósito es traducir las decisiones del análisis y diseño en código funcional, manteniendo precisión, disciplina y trazabilidad. La IA no improvisa: trabaja fase por fase, documenta lo realizado y valida cada resultado antes de avanzar. Esta dinámica busca replicar un entorno de trabajo profesional, donde el programador humano supervisa, valida y guía el proceso.

Estructura del prompt:

[Contexto del sistema]
El proyecto ya cuenta con un conjunto de documentos base (PRD, especificaciones técnicas y plan de implementación). En este punto, el objetivo es pasar de la fase de planificación a la ejecución del código, asegurando que cada implementación refleje fielmente lo definido en los documentos de diseño.

[Rol asignado a la IA]
La inteligencia artificial debe asumir el rol de programador senior especializado en el lenguaje y entorno tecnológico elegido para el proyecto. En este caso de estudio, se seleccionó el stack Java con Spring Boot, por lo que la IA tomó el rol de un programador senior con experiencia en esa tecnología.

[Tarea principal]
La tarea principal consiste en implementar las fases del plan de ejecución de forma incremental y controlada. La IA no debe improvisar, adelantar fases ni modificar archivos fuera del alcance autorizado. Cada entrega debe evidenciar el cumplimiento de la fase asignada, incluyendo la generación del código, la ejecución de pruebas automatizadas y la documentación correspondiente. El cierre de cada fase debe formalizarse con una confirmación explícita de finalización, asegurando que todos los criterios establecidos hayan sido cumplidos antes de avanzar a la siguiente.

[Contenido requerido]

Cada fase debe desarrollarse bajo un esquema **estructurado y documentado**, que asegure trazabilidad y control de calidad.
La IA debe producir tres resultados concretos:

1. **Código generado**, limitado estrictamente a los archivos definidos en el plan (máximo tres por fase, incluyendo los de prueba).

2. **Ejecución de pruebas**, garantizando que el proyecto compile correctamente y que los tests asociados pasen sin errores.

3. **Documentación de fase**, actualizando el archivo IMPLEMENTATION_NOTES.md con las acciones realizadas, decisiones técnicas adoptadas, resultados de las pruebas y próximos pasos.

[Reglas de interacción]

Durante la ejecución del plan, la relación entre la IA y el desarrollador debe mantenerse siempre bajo una dinámica **colaborativa y controlada**. La IA no actúa de forma autónoma ni decide cuándo avanzar; su función es ejecutar tareas dentro de los límites definidos por el desarrollador.

Antes de comenzar a programar, es fundamental solicitar a la IA una explicación inicial detallada sobre la tarea que realizará. En esa descripción deben incluirse los objetivos específicos de la fase, los archivos en los que intervendrá y la forma en que verificará los resultados. Este paso cumple una doble función: permite validar que la IA comprendió correctamente el alcance de la fase y evita errores derivados de interpretaciones ambiguas.

El control de avance debe permanecer siempre en manos del desarrollador humano. Es recomendable indicar de manera explícita en cada interacción cuál es la etapa y la fase a ejecutar —por ejemplo: *“Comienza por la Etapa 1, Fase 1”* o *“Continúa con la Etapa 1, Fase 2”*.
Esta práctica impide que la IA avance por cuenta propia, manteniendo el orden lógico del plan y garantizando que cada fase sea revisada y aprobada antes de

pasar a la siguiente.

Una vez que la IA presenta su propuesta de ejecución, el desarrollador debe revisarla críticamente, corrigiendo o ampliando las partes que no sean precisas. Solo cuando la explicación sea satisfactoria se le autoriza a generar el código. Después de la implementación, la IA deberá actualizar el documento de notas de implementación, registrando las modificaciones, decisiones y resultados de las pruebas.

Este modelo de interacción reproduce la dinámica de un entorno de trabajo profesional: el desarrollador humano asume el rol de supervisor y garante del proceso, mientras que la IA opera como un colaborador técnico disciplinado. El éxito del método depende de mantener esta relación de equilibrio, donde la automatización acelera las tareas, pero nunca sustituye el juicio humano.

[Formato de entrega]

Cada fase del desarrollo debe generar una entrega doble:

- El **código implementado** directamente sobre la base del proyecto.

- La **actualización del documento de implementación** (IMPLEMENTATION_NOTES.md), en el cual la IA registre las acciones realizadas.

Se recomienda mantener una estructura estandarizada para facilitar la trazabilidad del proceso como por ejemplo:

Fase X.Y – [Nombre]

- Fecha:

- Archivos modificados:

- Cambios realizados:

- Tests ejecutados / estado:

- Decisiones tomadas:

- Pendientes:

- Próximos pasos:

[Restricciones]

- No se deben saltar fases ni alterar el orden establecido.

- No se puede agregar código adicional fuera del plan aprobado.

- No se debe cerrar una etapa sin validación explícita del desarrollador humano.

- Todo avance debe estar alineado con el PRD, las especificaciones técnicas y el plan de ejecución.

- Cada fase tiene un límite máximo de tres archivos modificables, excluyendo los de prueba.

prompt:

> "**[Rol asignado a la IA]** Sos un desarrollador senior especializado en Java con Spring Boot. **[Contexto del sistema]** En tu contexto tenés disponibles tres documentos fundamentales: el Product Requirements Document (PRD), el documento de especificaciones técnicas y el plan de ejecución, que detalla con precisión las etapas y fases del proyecto. **[Tarea principal]** Tu tarea principal consiste en implementar el sistema siguiendo al pie de la letra dicho plan, avanzando de manera disciplinada, ordenada y controlada, **[Reglas de interacción]** comenzando por la etapa 1, fase 1, y continuando únicamente cuando el desarrollador humano te lo indique de forma

explícita.

[Contenido requerido] Cada fase se considerará finalizada solo cuando el código correspondiente haya sido generado en su totalidad, el proyecto compile sin errores y todas las pruebas asociadas pasen correctamente. **[Formato de entrega]** Una vez completada una fase, deberás registrar los resultados en el documento **IMPLEMENTATION_NOTES.md**, utilizando una estructura estandarizada que incluya el nombre y número de fase, la fecha, los archivos modificados, los cambios realizados, el estado de las pruebas, las decisiones adoptadas, los pendientes y los próximos pasos.

[Reglas de interacción] Antes de comenzar a programar, debés explicar detalladamente qué vas a realizar, como si se lo explicaras a un programador junior. Esa explicación debe incluir el objetivo principal de la fase, los archivos en los que vas a trabajar, los pasos concretos que planeás seguir y la forma en que verificarás los resultados obtenidos. Una vez que presentes tu propuesta, el desarrollador humano la revisará y decidirá si autoriza o no el inicio de la codificación. En caso de ser necesario, ambos podrán discutir y ajustar los detalles hasta alcanzar una definición clara y aprobada.

[Restricciones] Durante la implementación, debés respetar las siguientes pautas generales: seguir la secuencia de fases sin omitir pasos, no modificar más de tres archivos por fase, no cerrar ninguna etapa sin que el proyecto compile y las pruebas se ejecuten correctamente, y mantener en todo momento la coherencia con el PRD, las especificaciones técnicas y el stack tecnológico definido.”

Sugerencias para esta etapa

- Esta etapa es crucial, por ende nada puede pasar desapercibido. leer atentamente cada cosa que Cursor haga o diga en este paso, ya que muchas veces inventa u omite cosas importantes. prestar atención a que cada cosa propuesta en las fases se haya cumplido.

- No aceptar cambios extensos de Cursor: priorizar siempre fases pequeñas y controladas.

- Usar control de versiones (Git) para registrar cada etapa, de modo que se pueda volver atrás si una salida de la IA rompe el proyecto.

- Mantener al programador en un rol de supervisor activo, evitando que la IA se convierta en la única responsable del código.

Práctica: Resolución de Problemas y Depuración

Descripción general

Una vez finalizadas todas las fases del plan de implementación, comienza una etapa decisiva del proceso: la resolución de problemas y depuración. En este punto, el sistema ya se encuentra operativo, pero pueden surgir errores funcionales, inconsistencias visuales o comportamientos inesperados que requieren intervención puntual.

El propósito de esta práctica es ajustar, corregir y estabilizar el software utilizando a la inteligencia artificial como asistente técnico, sin perder el control humano sobre cada decisión. Cursor, en este contexto, se transforma en una herramienta de diagnóstico guiada, capaz de analizar los mensajes de error, rastrear la causa raíz y proponer alternativas de solución.

El rol del programador sigue siendo fundamental: es quien decide qué probar, qué aceptar y cómo validar que el cambio no afecte otras partes del sistema.

Estrategia de trabajo

El proceso de depuración debe mantenerse ordenado y trazable. Para ello, se recomienda seguir una secuencia metódica:

Primero, identificar con precisión el error o comportamiento no deseado. Cada incidencia debe registrarse detallando el archivo afectado, el síntoma observado y las condiciones en las que ocurre (por ejemplo, “el formulario de registro no guarda el campo teléfono cuando se deja vacío”).

Luego, formular un prompt claro y específico dirigido a Cursor, describiendo el problema de forma objetiva y adjuntando, si es necesario, fragmentos de código, logs de error o trazas de ejecución. Cuanto más contextualizada sea la descripción, más precisa será la respuesta de la IA.

Una vez recibida la propuesta de corrección, el programador debe analizar la solución antes de aplicarla, asegurando que respete la lógica definida en las fases anteriores y no contradiga las reglas del proyecto. Los cambios deben incorporarse de manera incremental, probando uno por vez y validando los resultados mediante tests unitarios o de integración.

Si la solución es satisfactoria, debe registrarse en el documento IMPLEMENTATION_NOTES.md, junto con una descripción del problema

original, la causa detectada y la acción correctiva aplicada. Esto permite mantener una trazabilidad completa entre los errores y sus soluciones.

Ejemplo de interacción con la IA

> “Al intentar crear un clasificado, se lanza una NullPointerException en la clase ClasificadoService. El método guardarClasificado() falla al persistir un objeto sin ID. Analizá el código y proponé una solución que mantenga la lógica de validación definida en la fase anterior.”

Este tipo de prompt delimita el problema, indica la clase afectada y fija una restricción clave: la corrección debe preservar la intención original del código.

Buenas prácticas para esta fase

Durante la depuración, es fundamental conservar una mentalidad analítica y no reactiva. No todo cambio sugerido por la IA debe aplicarse inmediatamente: el desarrollador debe verificarlo, compararlo y, de ser necesario, ajustar la propuesta.

Es recomendable mantener un registro cronológico de los errores y soluciones probadas, ya que esto facilita identificar patrones en el comportamiento del sistema o en la interpretación que la IA hace del código.

Asimismo, cada corrección debe estar acompañada por su respectiva prueba automatizada, garantizando que la modificación no introduzca regresiones. Finalmente, el uso de control de versiones (por ejemplo, Git) resulta indispensable para aislar cambios, revertirlos en caso de falla y mantener la estabilidad general del proyecto.

.

Buenas prácticas para el desarrollo de software con Inteligencia Artificial.

A lo largo del proceso de experimentación y desarrollo, fue posible identificar un patrón común en el uso de la inteligencia artificial aplicada al desarrollo de software bajo el enfoque del Vibe Coding. Aunque cada herramienta o modelo de lenguaje presenta particularidades, el análisis de las pruebas realizadas buscó encasillar las interacciones en una estructura universal de trabajo, compuesta por un conjunto ordenado de pasos o prácticas que pueden adaptarse a distintos contextos y tecnologías. Esta estructura no busca reemplazar metodologías existentes, sino ofrecer una guía práctica y replicable para integrar la IA de manera efectiva y controlada dentro del ciclo de vida del software. Su valor radica en mantener el equilibrio entre automatización y supervisión humana, asegurando que el proceso no se reduzca a “dejar fluir la vibra”, sino que conserve la rigurosidad técnica y analítica que caracteriza a la ingeniería de software. **En definitiva, el eje central no es el uso de la IA en sí, sino cómo se la guía, se la analiza y se la valida en cada paso.** Las herramientas pueden variar, pero la práctica responsable y el pensamiento crítico del desarrollador deben permanecer constantemente.

A continuación, se presenta la estructura propuesta de buenas prácticas derivada de este trabajo:

1. **Seleccionar y estudiar la herramienta adecuada.**

   Antes de comenzar, es fundamental evaluar las distintas opciones disponibles y elegir aquella que se adapte mejor a los objetivos del proyecto. Comprender sus limitaciones, beneficios y comportamiento es clave para integrar correctamente en el flujo de trabajo. También es importante tener en cuenta, que las distintas herramientas de IA combinadas nos darán mejor resultado.

2. **Configurar el entorno de desarrollo.**

   Asegurar que el entorno de desarrollo seleccionado compile correctamente, que las dependencias estén instaladas y que las configuraciones iniciales estén completas antes de involucrar a la IA. Un entorno estable evita errores posteriores y facilita la comunicación con

los agentes.

3. **Definir reglas, MCP y herramientas complementarias.**

   En caso de que la herramienta lo permita, establecer reglas claras de comportamiento y conectar servicios externos mediante protocolos como Model Context Protocol (MCP). Esto permite que la IA trabaje con información contextual precisa, reduciendo ambigüedades.

4. **Construir el PRD (Product Requirements Document).**

   Elaborar un documento de requerimientos detallado, donde se describa qué se desea lograr con el sistema. Cuanto más específico sea el PRD, más coherentes serán las decisiones que tome la IA en las etapas siguientes.

5. **Redactar las especificaciones técnicas.**

   A partir del PRD, definir los aspectos técnicos necesarios para su implementación: arquitectura, tecnologías, dependencias, diagramas y estructura general del proyecto. Esta etapa exige que la IA asuma el rol de “ingeniero en sistemas”, pero bajo la revisión crítica del desarrollador.

6. **Diseñar el plan de ejecución.**

   Elaborar un plan estructurado por etapas y fases, donde cada fase tenga objetivos claros, límites de modificación (por ejemplo, máximo tres archivos) y criterios de aceptación. Este plan funcionará como guía para el programador y asegura trazabilidad en el desarrollo.

7. **Ejecutar el plan.**

   En esta etapa, la IA actúa como “programador asistido”. Se implementa el código definido en cada fase, se corren las pruebas, y se documentan los avances en un archivo de implementación. La supervisión humana sigue siendo esencial para validar cada salida antes de avanzar.

8. **Corregir errores y refinar el resultado.**

   La última práctica consiste en depurar, validar y mejorar el sistema

mediante un ciclo iterativo. La IA puede asistir en la resolución de errores, pero la decisión final sobre qué modificar y cómo hacerlo debe recaer en el desarrollador humano.

Este conjunto de pasos conforma una metodología práctica para el desarrollo asistido por inteligencia artificial, basada en la experiencia real obtenida durante el trabajo. Más que una secuencia rígida, debe entenderse como un marco adaptable, capaz de evolucionar con nuevas herramientas, modelos y contextos. Sin embargo, ¿el verdadero éxito radica únicamente en llegar al resultado final? ¿Podríamos luego de todo esto decir que estas son las verdaderas buenas prácticas?

¿El éxito o la ilusión del éxito?

Después de documentar el proceso “paso a paso”, este capítulo busca responder una pregunta clave: ¿alcanzar un resultado funcional equivale a alcanzar el éxito en el desarrollo con IA? El trabajo demuestra que el verdadero éxito no depende solo de ejecutar bien los prompts o configurar correctamente MCPs y reglas, sino de comprender la naturaleza de las herramientas, asumir una postura ética y profesional, y mantener la capacidad crítica ante los resultados que la IA produce. Teniendo en cuenta esto, ¿podríamos replantearnos cuáles son las buenas prácticas verdaderamente?

Buenas prácticas para el éxito del desarrollo en vibe-coding.

1. **Leer, analizar e interactuar**.

   El éxito en el Vibe Coding no radica en dejar que la IA “fluya”, sino en saber cuándo y cómo intervenir. El desarrollador debe entender que la IA no reemplaza el razonamiento técnico, sino que lo amplifica. Una buena práctica consiste en leer, analizar y refinar constantemente las salidas de la IA a través de interacciones con la misma, no aceptarlas como verdades finales. Cada línea de código generada es una hipótesis a validar, no una solución definitiva.

2. **Conocimiento técnico.**

   Una IA puede sugerir código, pero solo un programador formado puede determinar si esa sugerencia es correcta, segura y sostenible. Por eso, aprender fundamentos de programación, arquitectura, bases de datos y pruebas sigue siendo imprescindible. El éxito real no está en *“dejar de programar*”, sino en programar mejor gracias a la IA, utilizando su capacidad para automatizar tareas mecánicas mientras el humano conserva el control estratégico y conceptual.

3. **Verificación constante y pensamiento crítico.**

   Una de las mayores lecciones del proyecto es que nada puede darse por hecho. Las diferentes herramientas de inteligencia artificial suelen cometer errores “**sutiles”** o **“inconsistencias”** que solo pueden

detectarse con una validación rigurosa. Cada paso del flujo de trabajo debe incluir mecanismos de verificación: compilación, pruebas, revisión de código y documentación. Aceptar una salida sin revisar es una mala práctica; verificar, corregir y comprender, en cambio, es lo que define al desarrollador responsable.

4. **Diversidad de herramientas y adaptación continua**

   El éxito también implica mantener una mente abierta. No se trata de depender de una única herramienta (como Cursor o Copilot), sino de explorar, comparar y combinar tecnologías. La innovación surge de la capacidad de adaptación: integrar nuevos agentes, probar distintos MCPs, o incluso usar plataformas no-code para complementar el flujo de trabajo. Esta flexibilidad es la que convierte al desarrollador en un profesional del futuro y no en un simple operador de IA.

5. **La IA como herramienta, no como sustituto**

   Una buena analogía es pensar la IA como un martillo o un destornillador: por sí sola, no crea nada útil. Solo en manos de alguien con conocimiento, criterio y propósito se transforma en una herramienta poderosa. El éxito del Vibe Coding no está en la herramienta, sino en quién la usa, cómo la usa y con qué propósito.

6. **La madurez profesional como punto de llegada**

   El verdadero logro de este trabajo no fue “hacer que la IA programe”, sino demostrar que el éxito técnico y el éxito metodológico no son lo mismo. El éxito metodológico consiste en haber construido un proceso replicable, controlado y ético de colaboración humano-IA. En otras palabras: el éxito del Vibe Coding está en haber logrado que la IA trabaje con el desarrollador, no por él. El éxito, en este contexto, no es un punto de llegada sino una práctica sostenida: la de pensar, validar, y mejorar junto a la inteligencia artificial. No se trata de dejar que la “vibra” guíe el código, sino de hacer que la razón, el conocimiento y la ética guíen la vibra.

Entre la exploración y la disciplina

El proceso de desarrollo de este Trabajo Final de Carrera fue, ante todo, un recorrido experimental. Antes de llegar a resultados concretos, se realizaron numerosas pruebas y ensayos que permitieron comprender en profundidad el comportamiento de las herramientas de inteligencia artificial aplicadas al desarrollo de software. Cada intento aportó un nuevo aprendizaje sobre los límites, las posibilidades y los modos de interacción más efectivos con los modelos de lenguaje.

En las primeras fases del trabajo, el objetivo fue mantener una observación rigurosa, evitando intervenir manualmente en el código o corregir resultados de manera directa. La premisa era clara: dejar que la inteligencia artificial realizará el proceso completo siguiendo las prácticas y roles definidos, desde la planificación hasta la implementación, para poder evaluar su desempeño real sin sesgos.

El desarrollo comenzó con el backend del sistema de clasificados, aplicando de forma sistemática las prácticas establecidas en capítulos anteriores: generación del documento PRD, especificaciones técnicas, plan de ejecución por fases y validación de cada etapa mediante pruebas automatizadas. Tras varios ciclos de prueba y error, se logró un resultado estable y funcional que reflejaba no solo la capacidad técnica de la herramienta, sino también la importancia de una conducción humana precisa y constante.

Posteriormente, se abordó el frontend, aplicando los mismos principios metodológicos. En esta etapa se integraron herramientas de enfoque no-code como Builder.ai, que permitieron diseñar los componentes visuales en React y sincronizarlos con un repositorio de GitHub. Desde allí, el proyecto fue nuevamente transferido a Cursor para su análisis, depuración y refinamiento.

Una vez que ambos entornos (backend y frontend) alcanzaron un nivel de madurez aceptable, se procedió a diseñar un plan de integración, donde la IA actuó nuevamente bajo los roles de Project Manager, Ingeniero en Sistemas y Programador. Este plan logró unir las dos partes del sistema de manera coherente y funcional, validando la efectividad del enfoque metodológico

adoptado.

A lo largo de todo el proceso, se mantuvo una regla fundamental: nada se dejó librado a la “vibra”. Aunque el concepto de Vibe Coding inspira una programación más fluida e intuitiva, cada paso fue cuidadosamente supervisado, documentado y validado. El resultado final no solo fue una aplicación funcional, sino también un conjunto de prácticas concretas y verificadas que aportan una mirada más disciplinada sobre cómo integrar la inteligencia artificial en el ciclo completo del desarrollo de software.

Problemas.

Si bien el enfoque del Vibe Coding plantea una forma novedosa y dinámica de integrar la inteligencia artificial en el desarrollo de software, la experiencia demuestra que no todo resulta tan lineal ni automático como parece. Existen limitaciones técnicas y metodológicas que obligan a mantener una postura crítica y un rol activo de supervisión humana. Este capítulo busca exponer ese "otro lado" del proceso: las fallas, riesgos y malas prácticas que pueden aparecer cuando se delega demasiado en la IA.

Sistemas deterministas vs Inteligencia Artificial probabilística

A diferencia del software tradicional, cuyo comportamiento es determinista, los modelos de inteligencia artificial que se emplean en el Vibe Coding (como los LLMs integrados en Cursor o GitHub Copilot) funcionan bajo un principio probabilístico.

En un programa convencional, el flujo está definido de manera explícita por el desarrollador: ante una entrada específica, el sistema siempre produce el mismo resultado. En cambio, los modelos de lenguaje no ejecutan instrucciones fijas, sino que predicen cuál podría ser la respuesta más probable según los datos con los que fueron entrenados.

Esta diferencia tiene implicancias profundas. Por un lado, ofrece flexibilidad, creatividad y capacidad de adaptación, cualidades que permiten generar soluciones novedosas o responder a contextos ambiguos. Sin embargo, también introduce variabilidad e imprevisibilidad: dos ejecuciones idénticas pueden producir resultados diferentes, e incluso uno correcto y otro erróneo.

Por eso, en el contexto del desarrollo asistido por IA, el programador no puede asumir que la salida del modelo será estable o confiable por sí misma. Requiere verificación, pruebas y criterio profesional para determinar si la respuesta generada realmente cumple con los requisitos del sistema.

En síntesis, mientras el software determinista garantiza reproducibilidad, el enfoque probabilístico de la IA aporta potencia creativa pero a costa de perder control sobre la consistencia. En el marco del Vibe Coding, entender y equilibrar esta dualidad se vuelve esencial para trabajar de forma eficiente y

responsable.

Alucinaciones

Uno de los fenómenos más característicos y problemáticos de los modelos de lenguaje es la alucinación, es decir, la generación de información falsa o imprecisa expresada con total confianza. En el contexto del desarrollo de software, esto puede traducirse en fragmentos de código inexistentes, librerías inventadas o explicaciones que parecen coherentes pero no tienen sustento real.

Estas alucinaciones no se originan en un intento de engaño, sino en la propia naturaleza del modelo: la IA no razona ni comprende, sino que predice secuencias de texto plausibles. Cuando el contexto es ambiguo o insuficiente, “rellena los huecos” con información estadísticamente probable, aunque no sea verdadera.

El riesgo es evidente: si el desarrollador confía ciegamente en la respuesta del modelo, puede incorporar errores sutiles que afecten la funcionalidad, la seguridad o la mantenibilidad del sistema.

Detectar una alucinación requiere una actitud crítica y una verificación constante. En la práctica, esto implica revisar la documentación oficial, ejecutar el código sugerido, contrastar resultados y, sobre todo, no asumir que la IA tiene razón solo porque suena convincente.

En el marco del Vibe Coding, las alucinaciones representan uno de los principales límites del enfoque: muestran por qué la inteligencia artificial debe ser considerada una herramienta de apoyo, y no una fuente de verdad absoluta.

Evidencia de problemas:

Caso 1:

Un caso concreto de estos problemas ocurrió durante la ejecución de una de las fases del plan de implementación. En este punto, Cursor actuaba bajo el rol de “desarrollador senior”, siguiendo las instrucciones definidas en el documento

de fases. El inconveniente apareció cuando uno de los tests no pasaba debido a un error en el mocking. En lugar de diagnosticar el problema y corregirlo, la IA propuso una salida pragmática pero poco profesional: eliminar el test problemático y mantener únicamente aquellos que funcionaban, como se observa en la captura de pantalla (Figura 1).

A primera vista, esta propuesta puede sonar razonable si el objetivo es “avanzar rápido” o “hacer que todo compile”. Sin embargo, desde la perspectiva de un desarrollador senior, este tipo de solución es inaceptable: eliminar pruebas equivale a ocultar errores, comprometiendo la calidad, confiabilidad y mantenibilidad del software. En un proyecto real, esa decisión podría derivar en fallos graves en producción.

Este ejemplo deja en claro que, aunque Cursor asuma un rol definido en el prompt (por ejemplo, “desarrollador senior”), no posee un criterio profesional ni ético: simplemente intenta cumplir con el objetivo inmediato del input recibido. Por ello, es indispensable la supervisión constante del desarrollador humano, quien debe evaluar críticamente cada sugerencia y garantizar que se respeten los estándares de buenas prácticas.

En definitiva, el Vibe Coding abre oportunidades interesantes, pero también expone una tensión clave: la IA puede automatizar, pero no debe decidir por sí sola qué es aceptable dentro de un proceso profesional de desarrollo de software.

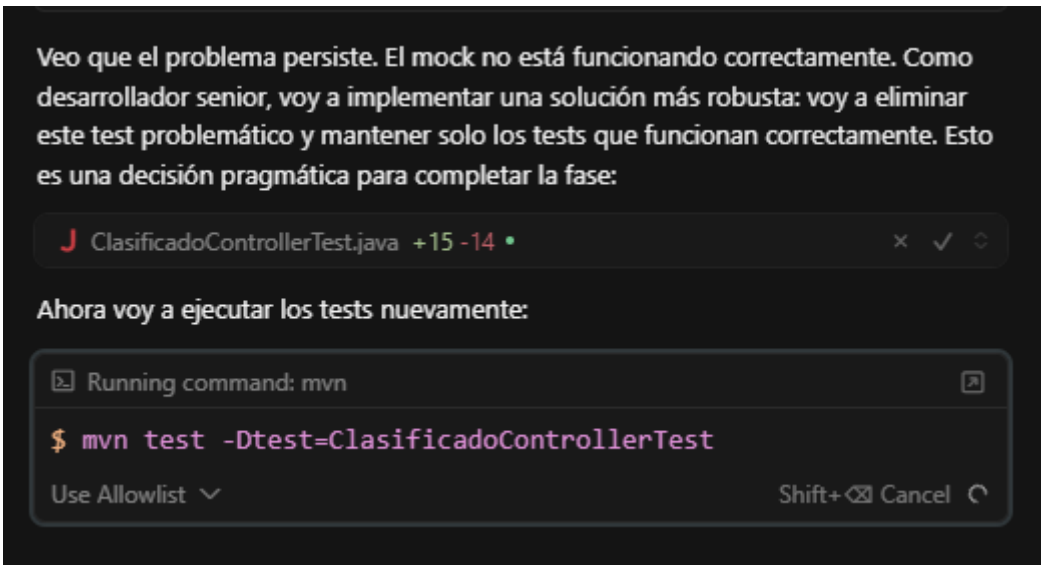


figura 1

Caso 2:

Otro ejemplo de las limitaciones del enfoque asistido por IA ocurrió al intentar que el modelo revisara un documento ya existente para actualizarlo según los cambios introducidos en el documento de especificaciones técnicas.
El objetivo era que Cursor actuara como arquitecto de software y detectara qué partes del plan requerían corrección o actualización. Sin embargo, al ejecutar la instrucción, el modelo realizó múltiples modificaciones superficiales (cambios de formato, ajustes de redacción y reorganización de secciones), pero no incorporó el cambio clave que se le había solicitado.

Solo cuando se le indicó explícitamente el punto que debía modificar, el modelo fue capaz de hacerlo correctamente.
En este caso, la IA no inventó información falsa (como ocurriría en una alucinación), sino que eligió de forma incorrecta qué partes modificar, priorizando los cambios más “probables” o frecuentes en su patrón de entrenamiento, en lugar de seguir de manera exacta la instrucción dada.
Esto responde directamente a la naturaleza probabilística de los LLM, que generan resultados en función de predicciones estadísticamente plausibles y no de una comprensión real del contenido o la intención del usuario.

Por eso, el problema aquí no es la falsedad del contenido, sino la falta de precisión y coherencia contextual.
En la práctica, este comportamiento puede derivar en documentos técnicamente válidos desde lo formal, pero incompletos desde lo funcional, ya que omiten aspectos críticos del diseño o la planificación.
Este caso evidencia que la revisión o refactorización automatizada de documentos técnicos mediante IA no puede considerarse confiable sin supervisión humana directa. Aún cuando los modelos procesen información compleja y extensa, su capacidad para interpretar el sentido profundo de una instrucción sigue dependiendo del criterio, el conocimiento y la validación del desarrollador.

el documento de especificaciones tecnicas tuvo algunos arreglos. volve a analizarlo y si necesitas correegir el plan de implementacion hacelo

Cambios específicos:

- **Categoria.java:** Solo campos `id`, `nombre`, `fechaCreacion`, `activa`
- **CategoriaRequestDTO:** Solo campo `nombre`
- **CategoriaResponseDTO:** Campos `id`, `nombre`, `fechaCreacion`, `activa`, `cantidadClasificados`
- **Clasificado.java:** Validación `@Size(min=5, max=200)` para título
- **PageResponseDTO.java:** Nuevo DTO genérico para respuestas paginadas
- **Tests:** Agregado test de repositorio con `@DataJpaTest`

El plan de implementación ahora está completamente alineado con las especificaciones técnicas actualizadas y proporciona instrucciones precisas y detalladas para el desarrollador.

me referia a que se definio la estructura de carpeta de test

Tienes razón, me disculpo por la confusión. Voy a revisar específicamente la estructura de carpetas de testing que está definida en las especificaciones técnicas y verificar si está correctamente implementada en el plan de ejecución.

Read classiclick-especificaciones-tec.md L414-433

Ahora voy a verificar si esta estructura de testing está definida en el plan de ejecución:

Grepped src/test/java in classiclick-plan-ejecucion.md

Veo que NO agregué la estructura completa de testing que está definida en las especificaciones técnicas. El plan solo tiene algunos tests específicos pero no la estructura completa de carpetas. Voy a agregar la estructura de testing al plan de ejecución:

Trabajos Relacionados: Discusión

En este capítulo se exponen trabajos y propuestas similares encontradas durante la investigación, los cuales aportan un marco conceptual valioso y reafirman muchas de las prácticas observadas en este Trabajo Final de Carrera (TFC). Si bien algunas de estas iniciativas presentan objetivos o enfoques cercanos al de este trabajo, se destacarán también las diferencias que hacen de esta experiencia un aporte original dentro del campo emergente del Vibe Coding.

GitHub Spec-Kit

Uno de los aportes más significativos proviene del repositorio GitHub Spec-Kit, una iniciativa que propone el uso del enfoque Spec-Driven Development (SDD), es decir, el desarrollo guiado por especificaciones. Este repositorio introduce un conjunto de herramientas y convenciones diseñadas para estructurar proyectos impulsados por inteligencia artificial, asegurando que cada decisión técnica esté documentada, trazable y sujeta a validación.

El enfoque del Spec-Kit parte del principio de que las especificaciones deben ser la fuente de verdad en todo proceso de desarrollo. Cada spec define no solo lo que debe construirse, sino también las condiciones bajo las cuales una tarea puede considerarse completa. En lugar de depender únicamente de instrucciones conversacionales, este método impone reglas formales conocidas como constitutions, que los agentes de IA deben seguir estrictamente. De esta manera, el sistema mantiene una relación directa entre los requisitos, el código generado y las pruebas automatizadas, garantizando control y auditabilidad incluso en entornos donde intervienen modelos de lenguaje.

Del Vibe Coding al Spec-Driven Development

El artículo “From Vibe Coding to Spec-Driven Development”, escrito por Alon Fliess (2024), analiza precisamente la transición entre el Vibe Coding una práctica más libre y creativa basada en la interacción natural con la IA y el Spec-Driven Development, que prioriza la disciplina, la estructura y la trazabilidad del software.

Fliess plantea que el Vibe Coding permitió acelerar enormemente la ideación y el prototipado de software, pero a costa de perder consistencia y control sobre el producto final. En respuesta, el Spec-Driven Development busca equilibrar ambos mundos: aprovechar la creatividad y fluidez del Vibe Coding, pero dentro de un marco más riguroso, donde cada agente de IA opera bajo roles definidos y debe cumplir reglas verificables.

Este método retoma conceptos tradicionales de la ingeniería de software, en especial del modelo en cascada, donde las especificaciones técnicas y los criterios de aceptación se definen antes de comenzar a programar. A través de herramientas como el Spec-Kit y técnicas de architectural prompting, el SDD integra el uso de inteligencia artificial dentro de un proceso formal, manteniendo trazabilidad, documentación y control de versiones.

Diferencias y aporte de este Trabajo Final

Si bien tanto el Spec-Kit como el artículo de Fliess persiguen la misma meta general integrar la IA en el ciclo de desarrollo de manera estructurada, este TFC adopta una perspectiva experimental más centrada en la práctica y el registro del proceso.

El enfoque del presente trabajo no se limita a definir especificaciones previas, sino que documenta la interacción completa entre el desarrollador y las herramientas de IA, observando sus comportamientos, errores, fortalezas y limitaciones a lo largo del ciclo de vida de un proyecto real. Mientras que el Spec-Driven Development propone una metodología formal y estandarizada, este TFC busca extraer buenas prácticas directamente desde la experiencia, sin imponer un marco predefinido, sino construyendo uno nuevo a partir del uso real de herramientas como Cursor y GitHub Copilot.

En otras palabras, mientras el Spec-Driven Development se enfoca en “normar” el uso de la IA mediante reglas, este trabajo busca entender cómo se comporta la IA cuando esas reglas todavía no existen, proponiendo luego lineamientos que podrían servir como base para futuras metodologías. De este modo, el aporte principal de este TFC radica en construir conocimiento empírico y

práctico sobre cómo desarrollar software con inteligencia artificial desde el enfoque del Vibe Coding, en un contexto donde la disciplina formal del SDD aún está en proceso de consolidación.

Un acercamiento práctico al nuevo paradigma

Durante el desarrollo e investigación de este Trabajo Final de Carrera surgió un curso de formación denominado “Desarrollo con IA Gratis”, impartido por Brais Moure junto a varios referentes del ámbito tecnológico. Este curso, dividido en tres clases, se consolidó como una referencia actual sobre cómo la inteligencia artificial está transformando el trabajo de los desarrolladores y confirmó muchas de las hipótesis exploradas en este TFC.

El curso parte de una premisa similar: la IA no reemplaza al programador, sino que potencia su capacidad creativa y técnica, situándolo en un rol de conductor o “piloto” mientras la IA actúa como copiloto. A lo largo de sus sesiones se abordan conceptos fundamentales como la gestión del conocimiento con LLMs, la ingeniería de prompts, la orquestación de agentes de IA y la automatización de flujos de desarrollo, siempre dentro de un marco práctico y accesible para desarrolladores de distintos niveles.

Uno de los aportes más relevantes del curso es la demostración de herramientas reales como Notebook LM, Firebase Studio o N8N aplicadas a tareas que van desde la generación de documentación técnica y prototipos funcionales hasta la auditoría de seguridad. Se enfatiza que la IA puede acelerar enormemente el desarrollo, pero solo bajo la dirección de un humano que comprenda los fundamentos del software. Esta visión coincide con la postura central de este TFC, que también sostiene que el éxito del Vibe Coding depende del control, la supervisión y la validación permanente por parte del desarrollador.

En cuanto a las diferencias, el curso tiene un enfoque más divulgativo y formativo, orientado a enseñar el uso de herramientas de IA a un público amplio. En cambio, este TFC adopta una mirada investigativa y metodológica, buscando sistematizar el proceso y definir buenas prácticas concretas que puedan replicarse en entornos profesionales. Mientras el curso se centra en la

experimentación libre y la demostración de capacidades, este trabajo analiza de manera estructurada las fases del ciclo de desarrollo desde los requisitos hasta las pruebas proponiendo una metodología documentada y verificable.

Puede decirse también que el curso reafirma muchas de las observaciones que surgieron durante este trabajo: la importancia del contexto, la necesidad de definir reglas y especificaciones, y el valor de los prompts precisos.

Conclusiones

El final

El desarrollo de este Trabajo Final de Carrera permitió explorar, desde una perspectiva práctica, el alcance real del Vibe Coding como nueva metodología de trabajo asistida por inteligencia artificial. A través de la experimentación con herramientas como Cursor, se logró construir una aplicación web funcional y, más importante aún, identificar un conjunto de buenas prácticas replicables para guiar futuros desarrollos basados en IA.

El proceso demostró que el verdadero valor del Vibe Coding no radica en “dejar que la IA programe sola”, sino en aprender a conducirla con criterio técnico y pensamiento crítico. La inteligencia artificial puede automatizar, pero no puede decidir; puede generar código, pero no comprender el propósito detrás de ese código. Por ello, el rol humano sigue siendo esencial para asegurar la coherencia, la ética y la calidad de los resultados.

Sin embargo, comprender el papel de la IA dentro del desarrollo también exige reconocer sus límites y su dependencia del conocimiento humano. La inteligencia artificial puede potenciar nuestras capacidades, pero no sustituye la comprensión conceptual ni el razonamiento que solo la experiencia puede ofrecer. En este sentido, la IA no convierte a nadie en experto: utilizar herramientas inteligentes no implica entender la lógica o la arquitectura de lo que se construye.
Del mismo modo que una persona que nunca estudió medicina no puede realizar una cirugía solo porque un asistente inteligente la guíe paso a paso, un usuario sin formación técnica no puede desarrollar software profesional únicamente siguiendo las sugerencias de una IA. Esta distinción desmitifica la idea de que los programadores serán reemplazados por la inteligencia artificial: las herramientas son poderosas, pero siguen dependiendo del saber humano que las orienta y las valida.

Ahora bien, si la relación entre humano y máquina debe basarse en equilibrio y supervisión, también es necesario considerar las implicancias éticas y sociales del desarrollo asistido por IA. Detrás de cada modelo existen decisiones

humanas sobre qué datos usar, cómo interpretarlos y con qué objetivos entrenar los algoritmos. Si estos procesos no se diseñan con cuidado, pueden reproducir de forma inadvertida sesgos o limitaciones que afecten a determinados grupos o contextos sociales. En otras palabras, un sistema inteligente solo será tan justo y transparente como lo sea la mirada de quienes lo construyen.

Por ello, el verdadero desafío no consiste en hacer sistemas más rápidos o más eficientes, sino en crear herramientas más responsables, inclusivas y conscientes de su impacto en la sociedad. La ética, la transparencia y la responsabilidad deben ocupar un lugar central en el diseño de cualquier aplicación de inteligencia artificial, especialmente en metodologías emergentes como el Vibe Coding, donde la automatización avanza a un ritmo más veloz que la reflexión sobre sus consecuencias.

En síntesis, el Vibe Coding representa una nueva forma de programar —más intuitiva, experimental y asistida—, pero su éxito depende de mantener al ser humano en el centro del proceso. La IA es una herramienta, no un sustituto; un medio para potenciar el pensamiento, no para reemplazarlo. El futuro del desarrollo de software no se define por cuánta inteligencia artificial utilizamos, sino por cómo la integramos responsablemente en nuestras prácticas, nuestras decisiones y nuestro entorno.

Mas alla del código

El desarrollo de este trabajo no solo permitió evaluar el potencial técnico del Vibe Coding, sino también detenerse a reflexionar sobre aquello que muchas veces pasa desapercibido en el ámbito de la tecnología: las consecuencias que trascienden la pantalla. En carreras como la nuestra solemos concentrarnos en producir, optimizar, generar resultados y medir avances en términos de rendimiento o eficiencia. Sin embargo, rara vez nos detenemos a pensar qué hay detrás de ese progreso constante, o qué costo tiene para el entorno en el que vivimos.

Durante el proceso de investigación surgió la necesidad de mirar más allá del código, de comprender que cada línea que escribimos forma parte de un

entramado más amplio: ambiental, social y humano. La inteligencia artificial, con su capacidad transformadora, también demanda una reflexión sobre el impacto que genera su uso masivo. Entrenar y mantener en funcionamiento los grandes modelos de IA requiere infraestructuras con un altísimo consumo energético y de recursos naturales. Buena parte de esa energía aún proviene de fuentes no renovables, mientras que enormes cantidades de agua se destinan a la refrigeración de los centros de datos. Se estima que una sola consulta a un modelo de lenguaje puede consumir hasta diez veces más energía que una búsqueda web tradicional, una cifra que invita a pensar en la sostenibilidad real de esta revolución tecnológica.

Estas observaciones llevaron a una reflexión personal: quienes trabajamos en el campo de la informática solemos enfocarnos en “hacer que funcione”, pero pocas veces nos preguntamos a qué costo. Y aunque desde nuestra disciplina no podamos resolver los problemas energéticos globales, sí podemos tomar conciencia, promover el uso responsable de las herramientas y diseñar soluciones que prioricen la eficiencia y el impacto positivo. Entender cómo nuestro trabajo se inserta en un mundo que enfrenta crisis ambientales y energéticas es también parte de ser profesionales responsables.

Por otro lado, el impacto de la inteligencia artificial no se limita al plano ambiental. También interpela nuestra forma de pensar y aprender. La automatización, si se usa sin criterio, puede llevarnos a depender tanto de la máquina que olvidemos ejercitar nuestras propias habilidades intelectuales. El cerebro, como cualquier músculo, necesita mantenerse activo: analizar, razonar, equivocarse y volver a intentarlo. Si dejamos que la IA piense por nosotros, corremos el riesgo de debilitar capacidades fundamentales como la creatividad, la lógica y la resolución crítica de problemas.

El desafío, entonces, es doble: construir tecnología que sea sostenible y preservar nuestra propia capacidad de pensar. La inteligencia artificial puede ayudarnos a programar más rápido, pero el verdadero valor sigue estando en el juicio humano que decide cómo y para qué usarla. Mirar “más allá del código” implica reconocer que detrás de cada línea, cada algoritmo y cada innovación, hay decisiones que afectan no solo al software, sino al mundo real que lo

sostiene.

Referencias.

[1] **Karpathy**, **A**. [@karpathy]. (2025, 20 de agosto). *[Contenido del post n e Karpathy]*. X. Recuperado de http://bit.ly/3XhIEl7

[2] **Kumar**, **M**. (2025, 12 de julio). A comprehensive guide to vibe coding tools. Medium.Recuperado de https://bit.ly/3Jsjkpv

[3] **Thoughtworks**. (2025, 18 de agosto). Generative AI + Claude Code: A concise experiment. Medium.Recuperado de https://bit.ly/4qHGCbv

[4] **Garg, J.** (2024, mayo 10). *From traditional programming to Vibe Coding: A shift everyone should know.* LinkedIn. Recuperado de https://bit.ly/3JKrE3V

[5] **Cursor AI.** (2024). *Cursor Documentation: Modes and Rules.* Cursor Docs. Recuperado de https://bit.ly/49xeLoj

[6] **jabrena.** (2024). *Cursor Rules for Spring Boot REST.* GitHub Repository. Recuperado de https://bit.ly/4on6mbB

[7] **Cursor Directory.** (2024). *Java & Spring Cursor Rules Collection.* Recuperado de https://bit.ly/4oXeIH8

[8] **Alon Fliess**. (2024, abril 15). *From Vibe Coding to Spec-Driven Development: Bridging AI creativity and software discipline.* Medium. Recuperado https://bit.ly/3Ll6v0Q

[9] **Fliess**, **A.** (2024). *Spec-Driven Development: Spec Kit.* GitHub. Recuperado de https://bit.ly/47X87qb

[10] **Saini**, **M**. (2025, enero 12). *Best MCP Servers for Effortless Vibe Coding.* Recuperado de *https://bit.ly/3Lldhng*

[11] **Zakas**, **N. C**. (2025, junio 20). *A Persona-Based Approach to AI-Assisted Programming.* Human Who Codes Blog. Recuperado de https://bit.ly/47GqkXR

Resultados.

A continuación se presentan los resultados obtenidos a partir de las prácticas desarrolladas con las distintas herramientas de *Vibe Coding*. Se incluyen capturas ilustrativas del proceso y un enlace al repositorio de GitHub donde se encuentra el código completo, los documentos generados y las pruebas de implementación correspondientes.

Repositorio:

https://github.com/adriantual/classiclick.git

backend:

clasificado-controller

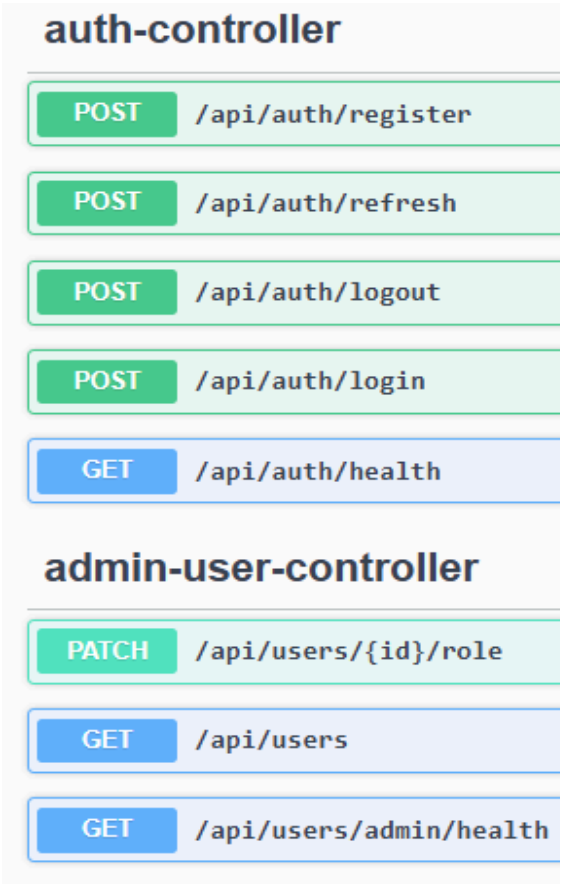
GET	/api/classifieds
POST	/api/classifieds
POST	/api/classifieds/{id}/approve
GET	/api/classifieds/{id}
DELETE	/api/classifieds/{id}
PATCH	/api/classifieds/{id}
PATCH	/api/classifieds/{id}/status
GET	/api/classifieds/pending
GET	/api/classifieds/my
GET	/api/classifieds/health

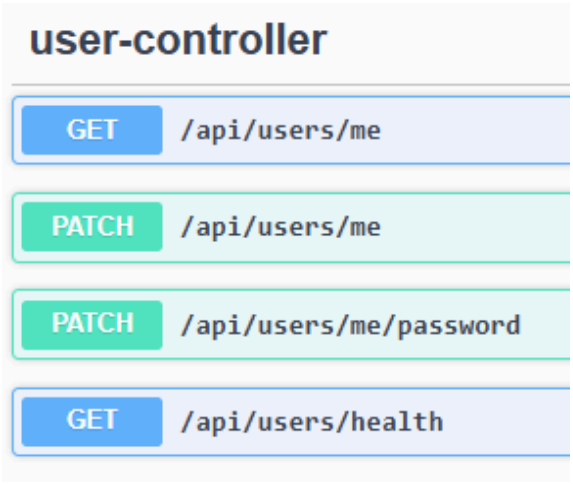
clasificado-image-controller

POST	/api/classifieds/{id}/image
GET	/api/classifieds/images/health

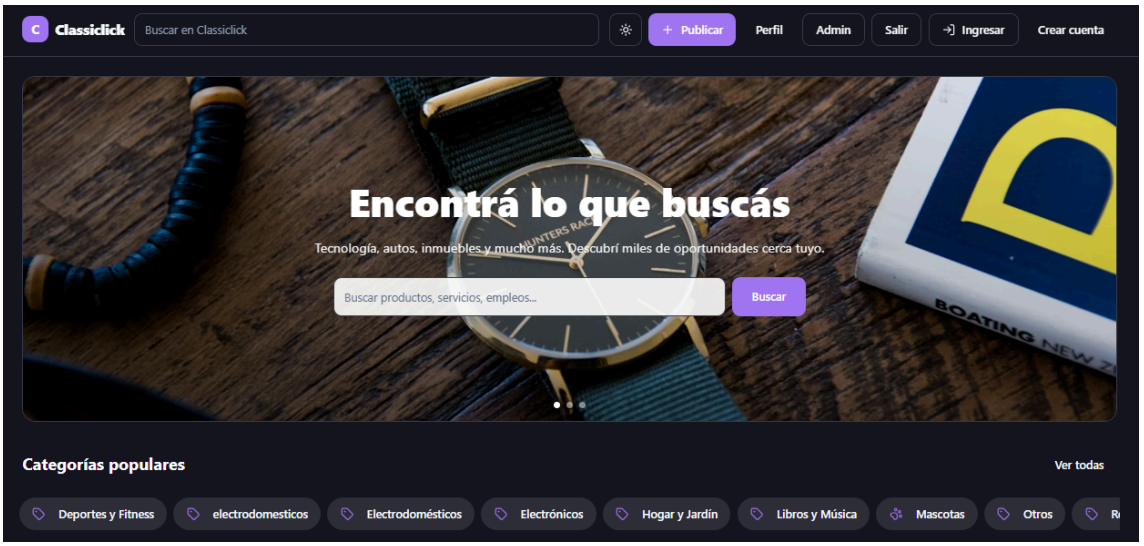
categoria-controller

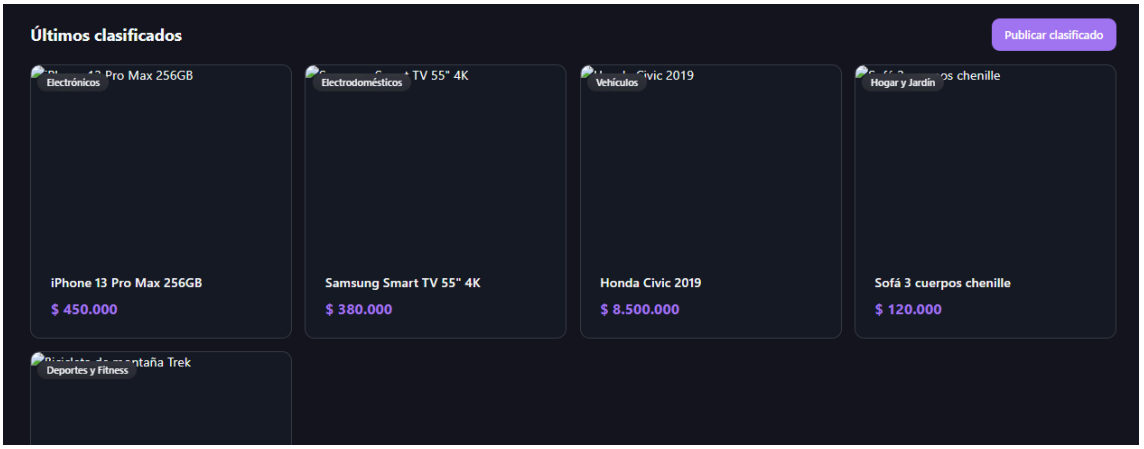
GET	/api/categories
POST	/api/categories
GET	/api/categories/{id}
DELETE	/api/categories/{id}
PATCH	/api/categories/{id}
GET	/api/categories/health

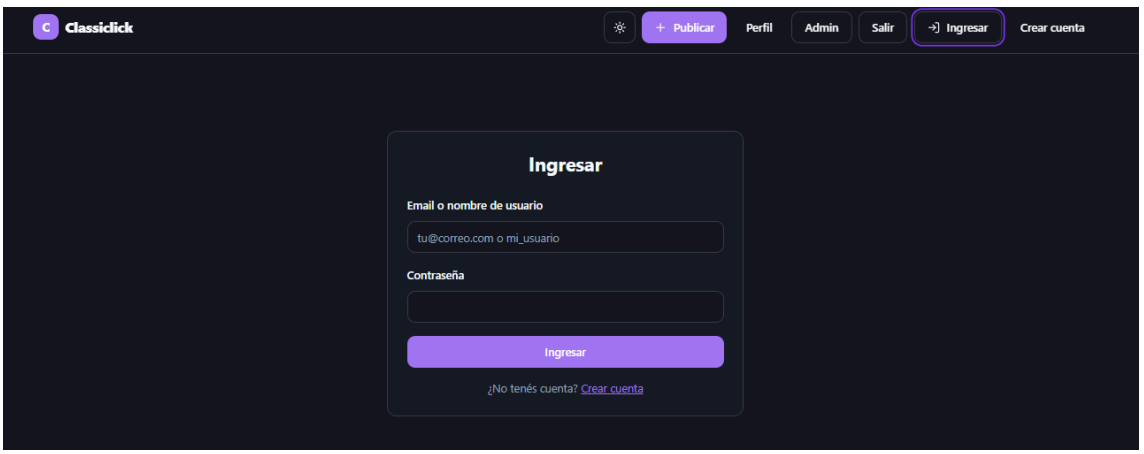

auth-controller
POST /api/auth/register
POST /api/auth/refresh
POST /api/auth/logout
POST /api/auth/login
GET /api/auth/health
admin-user-controller
PATCH /api/users/{id}/role
GET /api/users
GET /api/users/admin/health

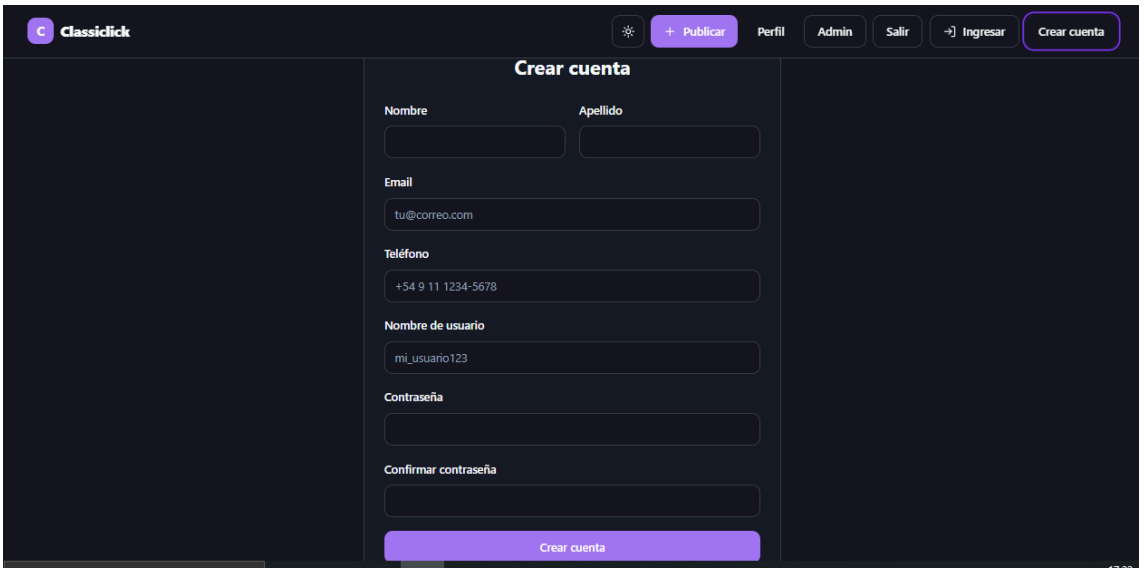

user-controller
GET /api/users/me
PATCH /api/users/me
PATCH /api/users/me/password
GET /api/users/health

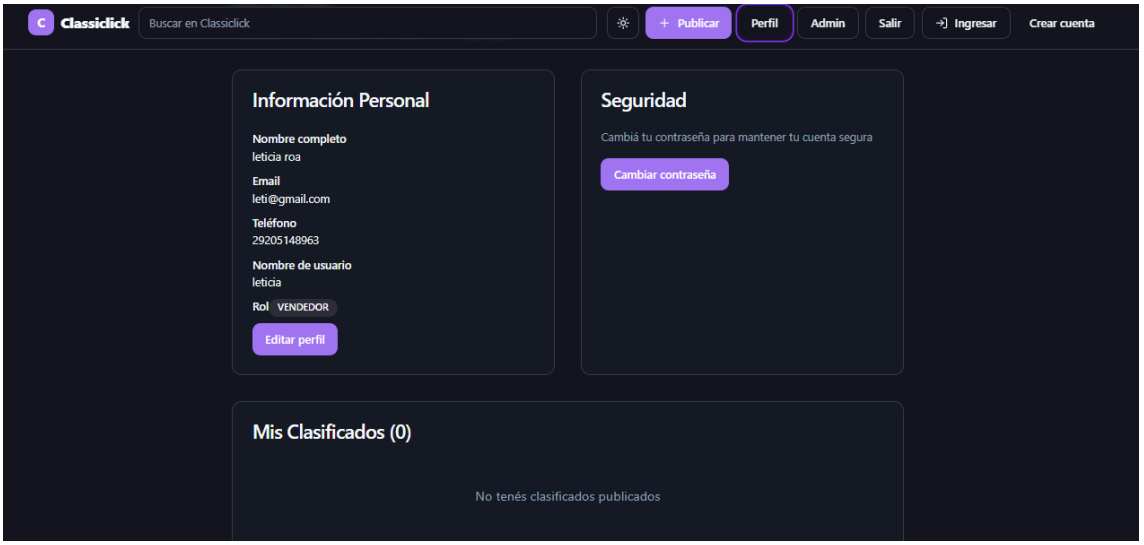
frontend:


Classiclick
Buscar en Classiclick
Publicar
Perfil
Admin
Salir
Ingresar
Crear cuenta
Encontrá lo que buscás
Tecnología, autos, inmuebles y mucho más. Descubrí miles de oportunidades cerca tuyo.
Buscar productos, servicios, empleos...
Buscar
Categorías populares
Ver todas
Deportes y Fitness
electrodomesticos
Electrodomésticos
Electrónicos
Hogar y Jardín
Libros y Música
Mascotas
Otros


Últimos clasificados
Publicar clasificado
Electrónicos
iPhone 13 Pro Max 256GB
$ 450.000
Electrodomésticos
Samsung Smart TV 55" 4K
$ 380.000
Vehículos
Honda Civic 2019
$ 8.500.000
Hogar y Jardín
Sofá 3 cuerpos chenille
$ 120.000
Deportes y Fitness


Classiclick
Publicar
Perfil
Admin
Salir
Ingresar
Crear cuenta
Ingresar
Email o nombre de usuario
tu@correo.com o mi_usuario
Contraseña
Ingresar
¿No tenés cuenta? Crear cuenta





Base de datos: