PromptShelf.AI
Powered by TALAVERA SOLUTIONS

Más allá de la Ola del Codigo

Desarrollo Asistido por LLM Determinista

Por Edgar Joya, VP Desarrollo Full Stack (COE)
Gabe Arce, Talavera Solutions CEO

El problema con la “Ola del Codigo”

Durante los últimos meses, he estado intentando descubrir cómo trabajar de manera efectiva con la inteligencia artificial. Los LLM han sido excepcionales para transformar mis ideas medio desarrolladas en realidad; solo el éter sabe cuánta curiosidad impulsada por el TDAH ha quedado satisfecha. Pero, cuando se trata de software de producción real, como la mayoría de nosotros, me he visto atrapado en un ciclo interminable de descubrimiento: saltando entre fragmentos de código, explorando nuevas ideas y probando distintos caminos para crear software.

El ciclo de retroalimentación de iterar una y otra vez con un sistema de matrices neuronales —esperando que cada generación te acerque un poco más a lo que realmente necesitas— empieza a sentirse como jugar en un casino, esperando que aparezca ese bloque de código mágico que te dé el resultado deseado. Después de un tiempo, se vuelve agotador. Sinceramente, me he divertido más viendo desfilar actualizaciones de paquetes de brew o apt.

A esto algunos lo llaman “vibe coding”: desarrollo guiado por la intuición y con los dedos cruzados, esperando que el código generado sea realmente correcto. El código parece estar bien, los patrones parecen razonables, pero no hay una forma concreta de verificar su corrección más allá de “compila y no se bloquea de inmediato.”

El problema no es el LLM, sino la falta de r**estricciones y contexto.** Cuando le pides a un LLM que “cree un endpoint para el procesamiento de órdenes”, claro, te generará código. Pero ¿realmente coincide con lo que necesitabas? ¿Maneja los casos límite? ¿Sobrevivirá a la próxima refactorización? ¡Quién sabe! Se siente como una partida de ruleta rusa.

Entrando: Desarrollo Guiado por Pruebas (TDD) con Copilot Chat.

Esto fue lo que cambió todo para mí:**¿Qué pasaría si las pruebas le dijeran a Copilot exactamente qué construir?**

En lugar de este caos:

Yo: "Crea un endpoint para mejorar prompts."
Copilot: *genera 200 líneas de código*
Me: "¡Se ve bien!" *presiona guardar con nerviosismo*

Empecé a hacer esto:

Archivo de características :
"Cuando hago un POST con un prompt vago a /api/v1/enhance, debería obtener una versión mejorada."
Prueba: FALLA — "No se encontró el endpoint en /api/v1/enhance."
Copilot: genera un endpoint mínimo
Terminal: La salida de la prueba muestra "Falta el campo enhanced_prompt en la respuesta."
Copilot: agrega enhanced_prompt
Prueba: PASA

Diagrama Mermaid: Arquitectura Nivel 1 (Básica)
Archivo de características: "Cuando hago un POST con un prompt vago a **/api/v1/enhance,** obtengo una versión mejorada."
Terminal: FAIL - FALLA — "No se encontró el endpoint en /api/v1/enhance."
Copilot: sugiere un endpoint mínimo basado en el diagrama + salida de prueba
Terminal: FALLA — "Falta el campo enhanced_prompt en la respuesta."
Copilot: agrega el campo enhanced_prompt
Terminal: PASA

Mi Flujo de Trabajo: Uso GitHub Copilot en VS Code con bucles de retroalimentación desde la terminal.
Las pruebas se ejecutan en la terminal, Copilot ve la salida, los archivos de características y los diagramas Mermaid, y genera código que satisface las restricciones.
No se necesita prompting manual al LLM.

El Patrón de Desarrollo dirigido por características

Fase 0: Visualiza Antes de Programar (con Arquitectura Progresiva)

Antes de escribir cualquier archivo de características o pruebas, empiezo con diagramas. Pero aquí está la parte crítica: **no documentes tus sueños, documenta tu camino.**

En lugar de crear un solo diagrama hermoso que muestre tu futura arquitectura de microservicios, crea **tres diagramas progresivos** que coincidan con la realidad:

Nivel 1: Básico (Lo que Construyes Primero)

Usuario → API Endpoint → enhance_prompt() → LLM (Gemini) → Base de Datos

Cuándo: Primera implementación, MVP, probar el concepto

Características:

- Lógica en línea en controladores/vistas
- Funciones, no servicios
- Estucturas de dato simples (diccionarios, modelos basicos)
- Llamadas directas a LLM

Ejemplo: Un endpoint simple para mejorar prompts que toma prompts vagos y los hace específicos.

Nivel 2: Intermedio (Próximas 2-4 Funciones)

Usuario → API → Funciones Mejoraxas → LLM → Base de Datos (+ nuevos modelos)
↓
Capa de Vaiación
↓
Lógica de Dominio (chequeo de calidad, versionado)

Cuándo: Añadiendo funciones, manejando casos límite, guardando historial

Características:

- Nuevos modelos (PromptVersion, EnhancementHistory)
- Validación y manejo de errores
- Lógica específica del dominio
- Aún monolítico, pero organizado

Ejemplo: Añadir versionado de prompts, puntuación de calidad y comparación entre prompts originales/mejorados.

Nivel 3: Avanzado (Escala/Fase de Producción)

Usuario →
API Gateway → Servicio de Autenticación → Servicio de Mejoras
→ Servicio LLM
→ Servicio de Memoria
→ Servicio de Análisis
→ Capa deCache
→ Cluster Base de Datos

Cúando: Escalar a miles de usuarios, múltiples equipos, necesidad de despliegues separados

Características:

- Arquitectura de microservicios
- Bases de datos separadas por servicio
- Colas de mensajes
- Orquestación compleja

Ejemplo: Sistema completo de producción con servicios dedicados para cache, limitación de tasa, análisis y enrutamiento multimodelo de LLM.

¿Por qué crear los tres diagramas desde el inicio?

Cuando creas los tres diagramas por adelantado:

1. **Diagrama Nivel 1** mantiene tu implementación inicial simple (sin sobre ingeniería)
2. **Diagrama Nivel 2** muestra tu siguiente paso (camino claro de crecimiento)
3. **Diagrama Nivel 3** documenta tu visión (alineación del equipo, pero no inmediata)

Crítico: Al implementar una función, dale a Copilot Chat el diagrama del nivel correcto. ¡No le des el diagrama avanzado de microservicios cuando estás construyendo lo básico!
Formato Preferido: Uso diagramas Mermaid en markdown; son versionables, se renderizan en GitHub/VS Code y son fáciles de actualizar conforme evoluciona el sistema.

Ejemplo Práctico de Diagrama de Secuencia (Nivel 1 - Mermaid)

Antes de cualquier código, dibujo la interaccion exacta usando la sintaxis Mermaid:

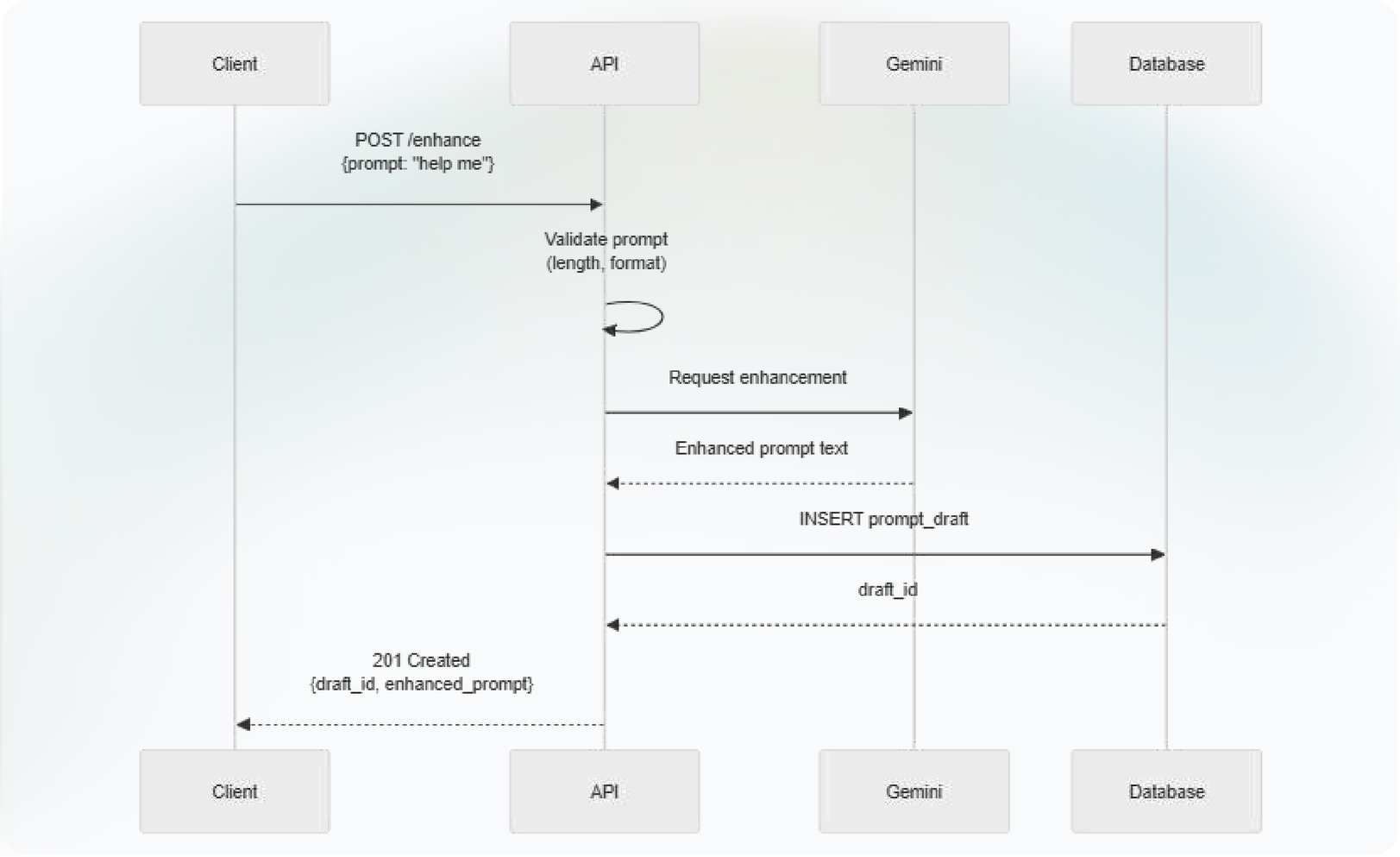


Diagrama 1

Este diagrama se convierte en mi “contrato” que las pruebas harán cumplir. Lo guardo como docs/sequence-diagrams/promptenhancement-basic.md para que Copilot pueda referenciarlo.

Resumen del Flujo Nivel 1:

Usuario → API Endpoint → enhance_prompt() → Gemini → Base de Datos

Cuándo: Primera implementación, MVP, prueba del concepto

Características:

- Lógica en línea en controladores/vistas
- Funciones, no servicios
- Estructuras simples de datos
- Mínimas abstracciones

Ejemplo: "Necesito un endpoint API que mejore promptsts"

- Un solo PromptController
- Una función enhance_prompt()
- Llama directamente a la API Gemini
- Guarda en la tabla PromptDraft

¿Por qué Mermaid?

- Versión controlada como texto plano
- Copilot puede leerlo como ontexto
- Fácil de actualizar a medida que evoluciona la arquitectura

Nivel 2: Intermedio (Próximas 2-4 funciones)

Usuario → API → Funciones Mejoradas → Lógica de Negocio → Servicio Externo → Base de Datos

Cándo: Agregar funciones relacionadas, ver como surgen patrones

Características:

- Funciones de lógica de negocio extraídas
- Modelos adicionales para relaciones de datos
- Archivos de configuración para reglas del dominio
- Aún monolítico

Ejemplo: "Ahora necesito comparación de versiones, seguimiento de cambios y preservación de contexto"

- Añadir funciones compare_versions(), track_changes()
- Añadir modelo PromptVersion para auditoría
- Añadir enhancement_rules.json para reglas de negocio
- Todo en el mismo código base

Nivel 3: Avanzado (Cuando Realmente Lo Necesitas)

Usuario → API Endpoint → enhance_prompt() → Gemini → Base de Datos

Cuándo: Problemas reales de escala, múltiples equipos, cuellos de botella de rendimiento

Características:

- Servicios separados con límites claros
- Arquitectura basada en eventos
- Componentes dedicados (cache, colas)
- Listo para microservicios

Ejemplo: "Procesamos 10,000 mejoras por hora y tenemos múltiples equipos"

- Servicio de Mejora separado
- Bus de eventos para procesamiento asíncrono
- Servicio de Contexto para preferencias de usuario
- Bases de datos múltiples

Regla Crítica: Creas los tres diagramas desde el inicio, pero solo implementas el **Nivel 1.** Avanza a Nivel 2 cuando tengas problemas reales que el Nivel 1 no pueda resolver. Avanza a Nivel 3 cuando el Nivel 2 falle.

Por Qué Importa la Arquitectura Progresiva

Sin ella:

Yo: "Construye una API para mejorar prompts"
Copilot: Crea arquitectura de microservicios con event sourcing
Yo: "Demasiado complejo para un MVP"
Copilot: Retrocede a algo demasiado simple
Yo: "Esto no soportará contexto de conversación"
Copilot: Confundido, genera código inconsistente

Cn elat:

Yo: "Construye mejora de prompts. Aquí está el diagrama Nivel 1"
Copilot: Crea implementación limpia basada en funciones
Prueba: Todas pasan [Luego...]
Yo: "Agrega contexto de conversación y seguimiento de versiones. Aquí está el diagrama Nivel 2."
Copilot: Añade funciones mejoradas y modelo de eventos
Prueba: Siguen pasando

La Magia: las pruebas BDD funcionan en los tres niveles. Mismo contrato API. Mismo comportamiento. Solo cambia la complejidad interna.

Diagramas de Secuencia para Cada Nivel (Mermaid)

Crea diagramas Mermaid mostrando la secuencia exacta de operaciones para cada nivel:

Nivel 1 (Básico) - Flujo Directo Simple:

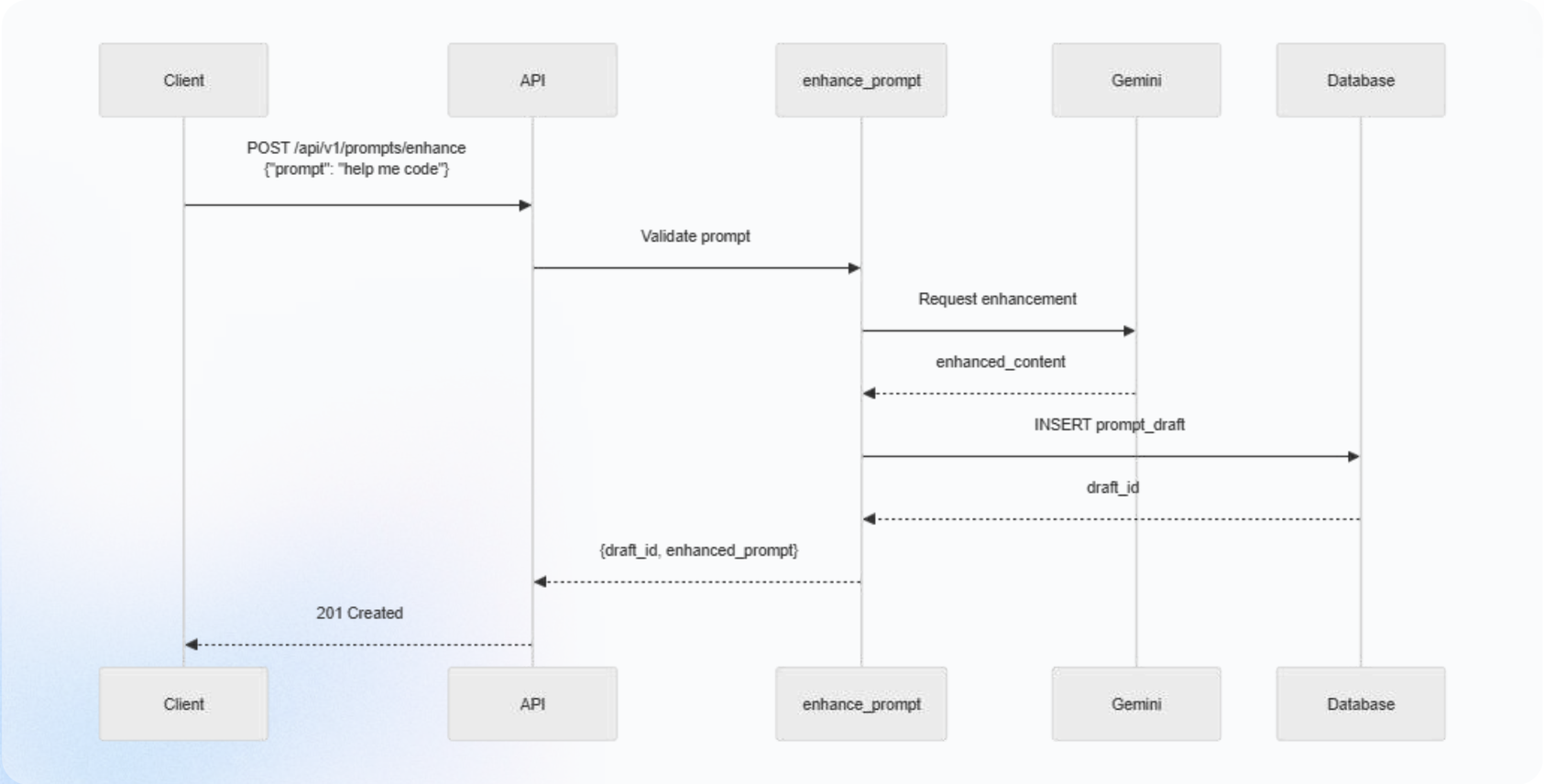


Diagrama 2

Nivel 2 (Intermedio) - Añadir Capa de Lógica de Negocio:

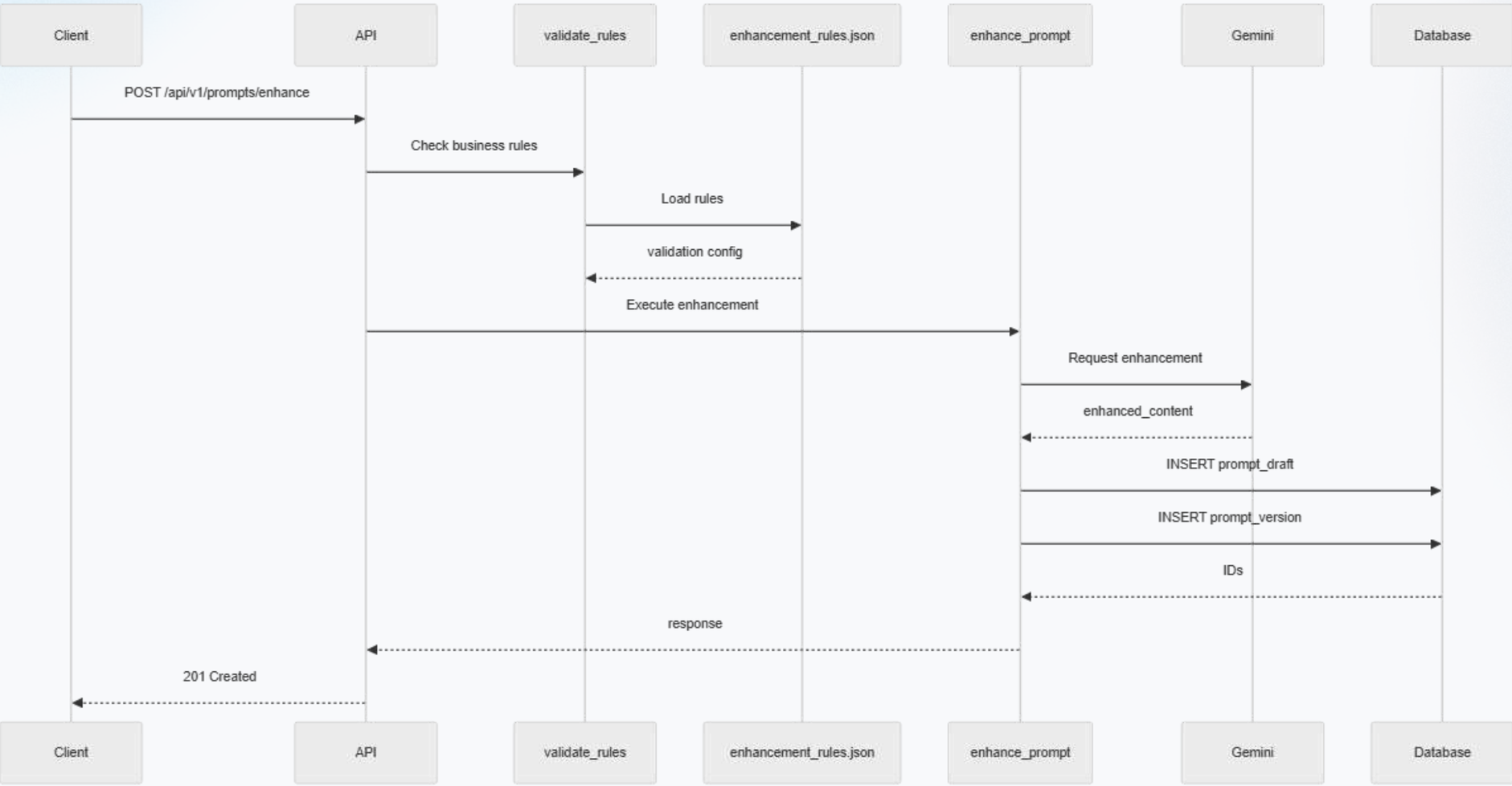


Diagrama 3

Nivel 3 (Avanzado) - Microservicios Basados en Eventos:

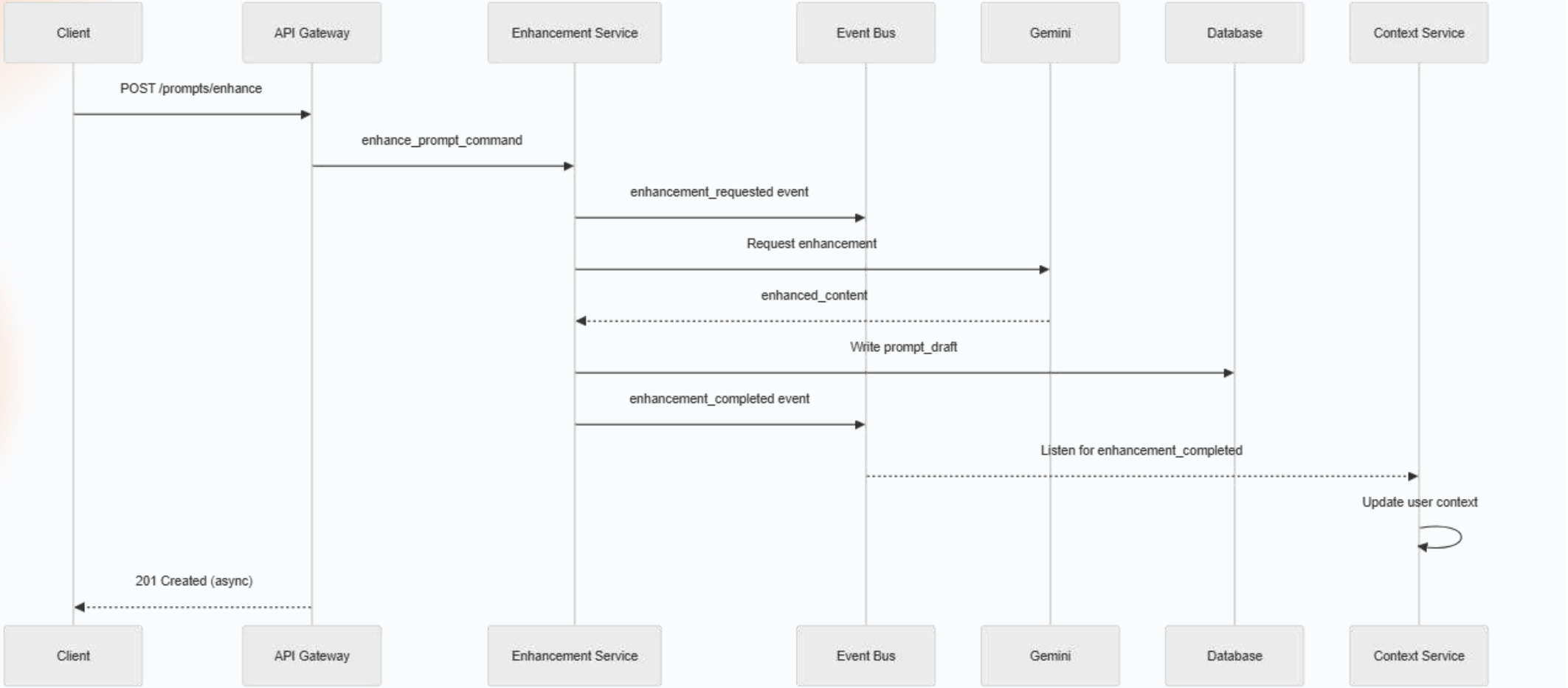


Diagrama 4

Insight: El cliente recibe la misma respuesta (201 Created con detalles del prompt) sin importar el nivel implementado. Las pruebas no cambian, solo la complejidad interna.

Pro Tip: Guarda estos diagramas como archivos separados:

- docs/sequence-diagrams/prompt-enhancement-basic.md
- docs/sequence-diagrams/prompt-enhancement-intermediate.md
- docs/sequence-diagrams/prompt-enhancement-advanced.md

Luego referencia el correcto al trabajar con Copilot!

Fase 1: Escribir Requerimientos Legibles para Humanos-Readable Requirements

Escenario: Mejorar un prompt vago
Dado que tengo un prompt vago "ayudame a escribir código"
Cuando solicito una mejora a través de la API
Luego el estado de la respuesta debe ser 200
Y la respuesta debe contener prompt_mejorado
Y el prompt mejorado debe ser más específico que el original
Y el prompt mejorado debe ser al menos tres veces más largo

Punto Clave:
Las personas sin conocimientos técnicos pueden leer esto. Tu gerente de proyecto puede validarlo. Tu diseñador puede entenderlo. Y se relaciona directamente con el diagrama secuencial que dibujaste.

Fase 2: Crear Pruebas Fallidas

```
@given("the prompt enhancement API is running and
accessible")
def step_impl_api_running(context):
    endpoint_path = "/api/v1/prompts/enhance"
    try:
        resolve(endpoint_path)
    except Resolver404 as exc:
        raise AssertionError(
            f"Endpoint not found at ' {endpoint_path}'. "
            "Implement the endpoint following the Level 1
(Basic)
            diagram."
        ) from exc
```

Punto Clave: La prueba **verifica** que tu implementación exista para la lógica de negocio principal. Haz mock a dependencias externas (APIs LLM, servicios de email, bases externas), pero verifica tus propias rutas de código. El mensaje de error apunta al nivel arquitectónico correcto.

Cuando hacer el Mock:
- APIs externas (Gemini, OpenAI, SendGrid)
- Servicios de terceros que no controlas
- Operaciones lentas (I/O de archivos, llamadas de red) en pruebas unitarias
- Antes de configurar CI/CD

Cuándo usar la Implementación Real:
- Tu propia lógica de negocio
- Operaciones de base de datos (usa base de datos de prueba)
- Flujos centrales de aplicación
- Pruebas de integración con servicios realess

Fase 3: Deja que las Pruebas Dirijan la Generación de Código con Copilot

Ahora cuando pides a Copilot Chat que implemente algo, tienes:

1. **Requisitos claros** (el archivo feature — nada de "constrúyeme algo" vago)
2. **Mensajes de error específicos** (exactamente qué está fallando y dónde, según la salida del terminal)
3. **Criterios de éxito explícitos** (la prueba pasa o no pasa, sin puntos intermedios)
4. **El nivel arquitectónico correcto** (el diagrama Básico, no tu sueño de microservicios del futuro)

Mi Flujo de Trabajo: Ejecuto las pruebas en el terminal de VS Code y luego pido ayuda a Copilot Chat, usando menciones # para proporcionar contexto:

El test falló. Por favor implementa el endpoint siguiendo el diagrama Básico.
#terminalLastCommand
#file:features/prompt_enhancement.feature
#file:docs/sequence-diagrams/prompt-enhancement-basic.md
#file:api/urls.py
#file:api/views.py

La salida de Copilot ya no es una apuesta - esta **limitada por afirmaciones concretas y una complejidad adecuada.** Nada de vibras de tragamonedas. Nada de soluciones sobreingenierizadas.
Que le das a Copilot (via # meciones):

#terminalLastCommand - Salida del terminal con un fallo específico
#file:features/*.feature - Tu archivo de caracteristícas con requisitos claros
#file:docs/sequence-diagrams/*.md - El diagrama Mermaid en el nivel arquitectónico correcto
#file:api/*.py - Tus patrones de códigos existentes

La Magia: Las sugerencias de Copilot están limitadas por:

- El fallo de la prueba (#terminalLastCommand – sabe exactamente qué está roto)
- El diagrama (#file: – sigue el nivel arquitectónico correcto)
- Las convenciones de tu código base (#file: – se ajusta a tu estilo)
- El ciclo de retroalimentación del terminal (cada iteración es más específica)

Ejemplo de cómo proporcionar contexto a Copilot:

Por favor, corrige la prueba que está fallando.
#terminalLastCommand

El terminal podría mostrar:

$ behave features/prompt_enhancement.feature
FALLO: Endpoint no encontrado en '/api/v1/prompts/enhance'.
Implemente el endpoint siguiendo el diagrama del nivel 1 (Basico).
Agregue la ruta a api/urls.py con el siguiente path: 'prompts/enhance'

Con este contexto, Copilot sugiere la implementación mínima exacta necesaria. Sin sobreingeniería. Sin adivinar

Idea Clave: La combinación de:

- Archivos feature BDD (requisitos)
- Salida de pruebas del terminal (retroalimentación inmediata)
- Diagramas Mermaid (restricciones arquitectónicas)
- Copilot (generación de código)

... crea un bucle de desarrollo determinista. El terminal te dice qué está roto, el diagrama te dice cómo arreglarlo, y Copilot genera el código que satisface ambos.

Por qué esto cambió realmente mi forma de trabajar

1. **Resultados deterministas (¡por fin!)**

```
# La “Vibra” de Codigo
assert response.data # ¡Esperemos que haya datos!
# Codificación guiada por pruebas
assert "session_id" in response.data
assert uuid.UUID(response.data["session_id"])
assert len(response.data["message"]) > 20
```

Sin ambigüedad. El UUID es válido o no. El mensaje es útil o es demasiado corto. Hecho.

2. Código autodocumentado (que realmente ayuda)

Tus archivos feature se convierten en documentación viva que cualquiera puede leer:

```
# Esto es LA especificación
Escenario: Exportar sesión con historial de conversación
Cuando solicite exportar la sesión
Luego la exportación debe contener el mensaje final
Y la exportación debe incluir el historial de conversación
Y la exportación debe proporcionar instrucciones de uso
```

Los product managers pueden escribirlos. Los diseñadores pueden validarlos. QA puede rastrearlos. El LLM puede implementarlos. Por fin, todos hablan el mismo idioma.

3. Refactorización sin miedo

Seis meses después, necesitas refactorizar la lógica de sesión. Con la codificación “por vibra”, estás sudando: ¿y si rompes algo oculto?

Con desarrollo guiado por pruebas:

```
$ make test
Todas las prueban pasan
p
*refactoriza agresivamente mientras pone música a todo volumen*

$ make test
Todasl las pruebas siguen pasando
```

ISi las pruebas pasan, el comportamiento es correcto. Eso es todo. Sin ansiedad, sin “voy a probar manualmente 47 escenarios”.

4. Copilot Chat como agente restringido (no generador del caos)

Piensa en Copilot como un desarrollador junior brillante pero demasiado entusiasta:

- **Sin pruebas:**“¡Construye algo genial!” → Se desborda, crea algo interesante pero probablemente no lo que necesitabas.
- **Con pruebas + retroalimentación del terminal:** “Haz que esta prueba específica pase.” → Enfocado, verificable, resuelve tu problema real.

Las pruebas son los rieles de seguridad. La salida del terminal es el bucle de retroalimentación. Canalizan todo ese poder generativo para resolver tu problema exacto, no para crear proyectos artísticos.

Ejemplo del mundo real: cómo lo hago ahora

Déjame guiarte por cómo implemento una funcionalidad en este momento.

Paso 0: Crea el diagrama de secuencia

Antes de tocar código, dibujo la interacción usando sintaxis Mermaid.

El Diagrama de Secuencia

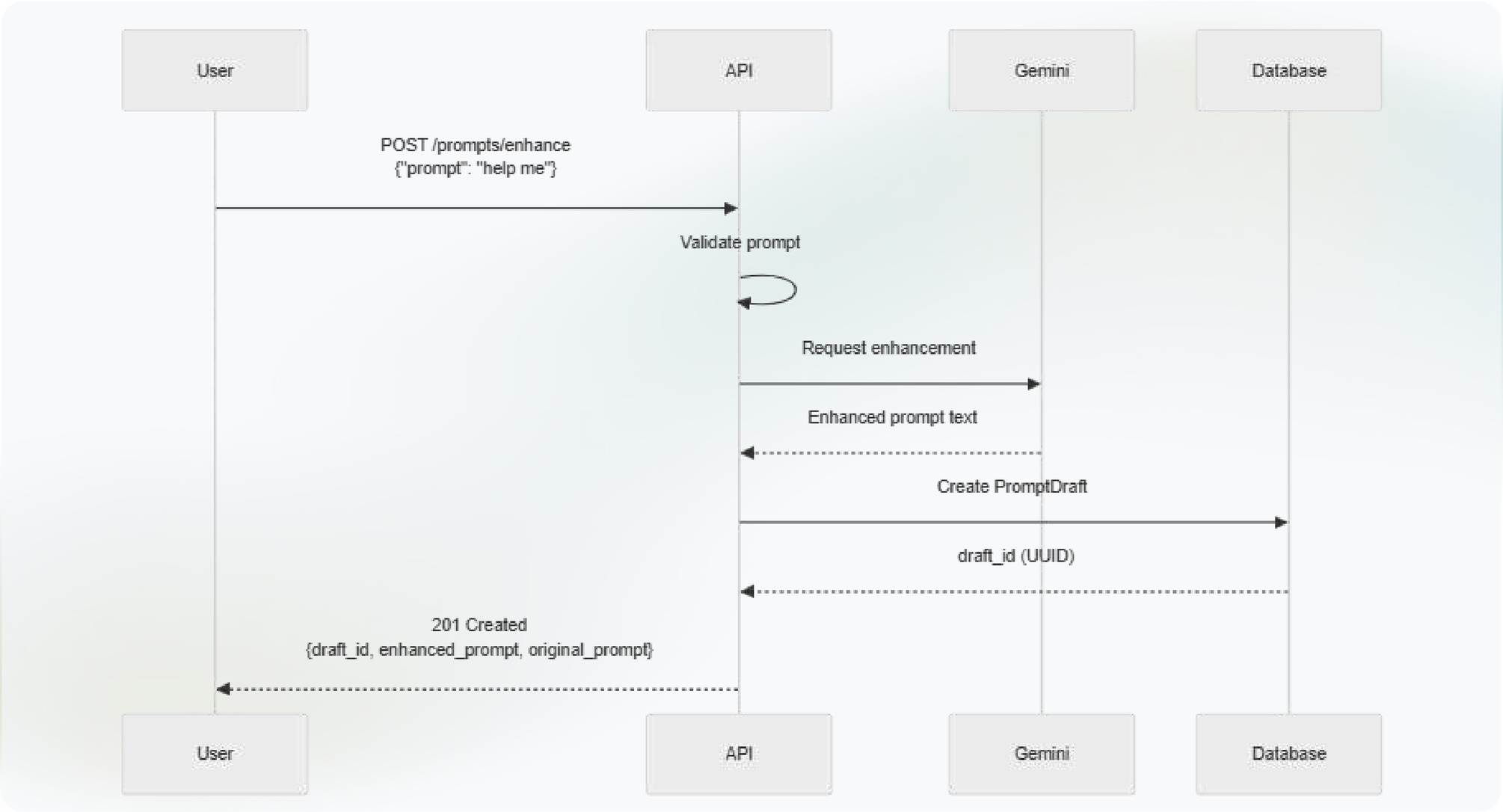


Diagrama 5

Este diagrama se convierte en mi “contrato” que las pruebas harán cumplir. Sin adivinar.

Paso 1: Archivo Feature (Lógica de negocio)

Escenario: Mejorar un prompt vago
Dando Tengo un prompt vago "ayúdame"
Cuando Envío un prompt para mejorarlo
Luego el estado de la respuesta debe ser 201
Y la respuesta debe contener un draft_id
Y la respuesta debe contener un enhanced_prompt
Y el borrador del mensaje debe registrarse en la base de datos

Nota: Cada paso Then se mapea directamente a algo en el diagrama de secuencia. Todo está conectado.

Paso 2: Ejecuta las pruebas (míralas fallar)

```
$ behave features/prompt_enhancement.feature

FALLO: endpoint no encontrado para la mejora del Pormpt en '/api/v1/prompts/enhance'.
Por favor, implementa el endpoint en Django. Agrega la ruta en api/urls.py con la ruta: 'prompts/enhance'.
```

Perfecto. Mensaje de error claro. Sin ambigüedad sobre lo que debe suceder a continuación.

Paso 3: Dale a Copilot el contexto necesario

Mi flujo de trabajo real en VS Code:

1. **Ejecuto la prueba** en el terminal integrado para ver el fallo.
2. **Abro Copilot** y le pido ayuda para arreglar la prueba.
3. **Uso menciones #** para agregar contexto explícito:

- #file:prompt_enhancement.feature - Requisitos del feature
- #file:docs/sequence-diagrams/prompt-enhancement-basic.md - Diagrama arquitectónico
- #terminalLastCommand - Salida del test que falló
- #file:api/urls.py and #file:api/views.py -Archivos a editar

Ejemplo de prompt en Copilot

La prueba falló. Por favor, implementa el endpoint faltante:

#terminalLastCommand
#file:features/prompt_enhancement.feature
#file:docs/sequence-diagrams/prompt-enhancement-basic.md
#file:api/urls.py
#file:api/views.py

Ahora Copilot tiene:

- **Salida de terminal** (via #terminalLastCommand) mostrando el fallo específico
- **Archivo de Características** (via #file:) con requisitos claros
- **Diagrama de Secuencia** (via #file:) mostrando la arquitectura
- **Codigo Existente** (via #file:) para entender los patrones y realizar modificaciones

Genera código que aborda directamente el fallo, siguiendo la arquitectura del diagrama.

El flujo de trabajo es iterativo:

- Escribe feature → Ejecuta test → Observa fallo → Pide ayuda a Copilot (con contexto) → Aplica cambios → Ejecuta test otra vez
- Cada salida del terminal aporta retroalimentación más específica.
- Usa #terminalLastCommand para darle a Copilot los últimos resultados de prueba.
- Las pruebas pasan cuando todos los requisitos se cumplen.

Step 4: Copilot Generates Minimal Code

```
# api/urls.py
urlpatterns = [
    path("prompts/enhance",
    PromptEnhanceView.as_view()), ]

# api/views.py
class PromptEnhanceView(APIView):
    def post(self, request):
        draft_id = uuid.uuid4()
        # Minimal processing - just enough to pass the
        test
        return Response({
            "draft_id": str(draft_id),
            "enhanced_prompt": "Enhanced version of
            the prompt", "original_prompt":
            request.data.get("prompt")
        }, status=201)
```

Míralo: Mínimo, enfocado, hace exactamente lo que la prueba requiere. Nada más.

Paso 5: Ejecuta las pruebas otra vez

```
$ behave features/prompt_enhancement.feature
```

- Endpoint encontrado
- Devuelve 201
- Contiene session_id
- session_id es un UUID válido
- Contiene mensaje de saludo
- El saludo es útil (contiene palabras clave: "ayúdame", "construyé)

¡Todos los escenarios pasan!

Esa sensación nunca envejece.
Paso 6: Itera cuando necesites más
Agrega más requisitos al archivo feature:
```gherkin
Y la respuesta debe incluir sugerencias accionables.
```

Ejecuta las pruebas en la terminal. Fallan:

```
FALLO: La respuesta no contiene el campo 'suggestions'.
```

Copia esa salida de la terminal en Copilot Chat. Lo arregla. Repite hasta que todo esté en verde.

Los Beneficios Realmente se Acumulan.

Para equipos (Acabo de empezar a liderar uno, con dolores de crecimiento incluidos)
Dirijo un equipo de cuatro desarrolladores (incluyéndome a mí), y he visto de primera mano dónde el código generado por LLM es un cambio radical y dónde simplemente falla.

- **Comprensión compartida:** Los archivos de funcionalidades + diagramas progresivos se convierten en la única fuente de verdad.
- **Mejores revisiones de código:** Revisa primero los tests y comprueba si la implementación coincide con el diagrama del nivel actual.
- **Incorporación más fácil:** Los nuevos desarrolladores pueden seguir los requisitos → diagramas adecuados al nivel → tests → código.
- **Camino de progresión claro:** Todos saben que estamos en Nivel 1, trabajando hacia el Nivel 2, con el Nivel 3 documentado para el futuro.
- **Estándares consistentes:** Los tests aplican patrones de manera uniforme en todo el código del equipo.

Cuando todos trabajan con LLMs, tener los tests como barreras de seguridad significa que todos estamos construyendo hacia la misma especificación, no cuatro interpretaciones diferentes.

Para desarrolladores individuales

- **Confianza en el código generado por IA:** Sabes que es correcto porque los tests lo confirman.
- **Iteración más rápida:** Mensajes de fallo claros = sin adivinanzas sobre qué salió mal.
- **Mucho menos debugging:** Detecta problemas de inmediato, no a las 2 a.m. en producción.

Para el flujo de trabajo con Copilot

- **Requisitos precisos:** Nada de “hazme una API” → caja misteriosa de código.
- **Progreso incremental:** Cada fallo en un test desde la terminal es una tarea específica y resoluble.
- **Validación instantánea:** Cada sugerencia de Copilot se verifica de inmediato en la terminal.
- **Bucle de feedback estrecho**: Terminal → Copilot Chat → Código → Terminal.

Objeciones Comunes (y por qué están equivocadas)

- **“¡Escribir tests toma demasiado tiempo!”:** Ya estás escribiendo tests — están solo en tu cabeza. Hacerlos explícitos detecta bugs antes y hace que el LLM sea mucho más efectivo. Además, ¿honestamente? Escribir un test que falle es más rápido que depurar código misterioso a las 11 p.m.
- **“¡Mis requisitos cambian demasiado rápido para esto!”**: ¡Incluso mejor! Cuando los requisitos cambian, actualiza primero el archivo de funcionalidades. Los tests fallan, mostrando exactamente qué necesita actualizarse. Luego el LLM arregla la implementación. Acabas de convertir el caos en una checklist clara.
- **“¡Esto no funciona para programación exploratoria!”**: ¡Cierto! Para prototipos y spikes, experimenta a tu gusto. Yo también lo hago. Pero cuando es hora de pasar a producción: tests o fuera.

Empezando (En realidad, bastante sencillo)

1. **Elige un framework de tests**

- **Python:** Behave (BDD) + pytest
- **JavaScript:** Cucumber + Jest
- **Ruby:** RSpec + Cucumber
- **.NET:** SpecFlow + xUnit

No lo compliques. Elige uno y listo.

2. Diagrama primero, código después
Antes de escribir cualquier código o test, crea **diagramas progresivos en Mermaid** que reflejen tu recorrido de desarrollo:

Diagrama básico (documenta lo que tienes ahora):

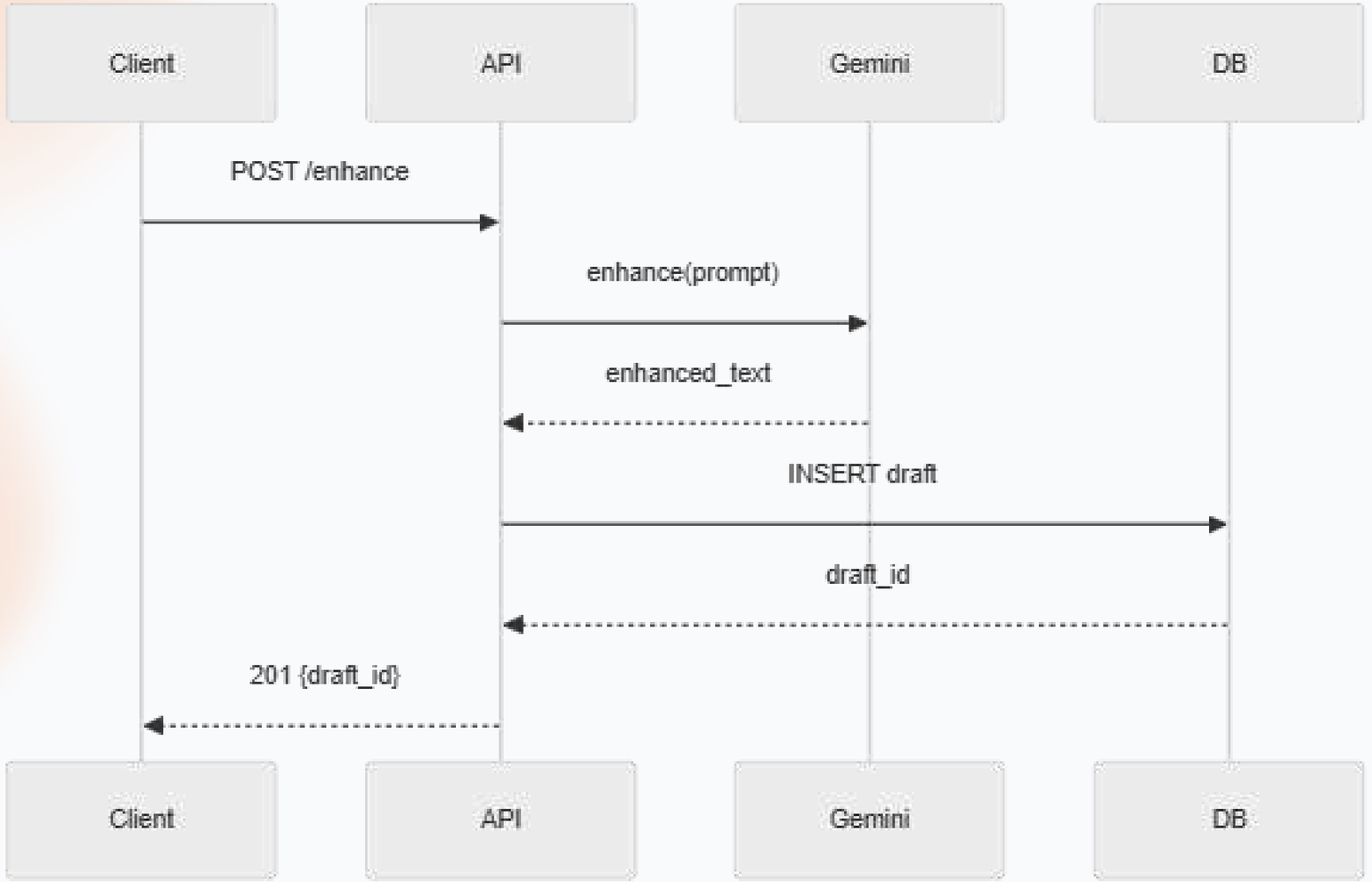


Diagrama 6

Diagrama intermedio (siguientes 2-3 funcionalidades):

- Agregar capa de validación, nuevos modelos (PromptVersion)
- Funciones mejoradas con contexto
- Aún monolítico

Diagrama avanzado (objetivo futuro):

- Separar servicios si es necesario
- Arquitectura basada en eventos
- Para cuando realmente escales

¿Por qué Mermaid?

- Controlable por versiones (texto plano en markdown)
- Se renderiza en GitHub / VS Code / GitLab
- Fácil de actualizar a medida que el sistema evoluciona
- Copilot lo entiende como contexto
-

Crea los tres niveles desde el principio en docs/sequence-diagrams/, pero implementa **solo un nivel a la vez**

3. Escribe un archivo de funcionalidades

Comienza pequeño. Un escenario, 5-10 pasos que correspondan a tu diagrama Mermaid. Etiquétalo con el nivel de implementación:

```
@level-basic @conversation
Escenario: Construcción interactiva de prompts
    # Implementar con arquitectura básica
```

No intentes describir toda tu aplicación desde el primer día.

4. Ejecuta los tests y deja que la salida de la terminal guíe a Copilot

Escribe definiciones de pasos que verifiquen el comportamiento real. Luego, sigue este flujo de trabajo:

- **Ejecuta el test** → La terminal muestra el fallo específico
- **Abre el archivo** → Copilot ve la salida de la terminal + archivo de funcionalidades + diagrama
- **Empieza a escribir** → Copilot sugiere la implementación
- **Ejecuta el test nuevamente** → Feedback más específico
- **Itera** → Hasta que los tests pasen

Ejemplo de desarrollo guiado por la terminal:

```
# Primera Ejecución
$ behave features/prompt_enhancement.feature
FALLO: Endpoint no encontrado en
'/api/v1/prompts/enhance'

# Abra urls.py, Copilot sugiere la ruta
# Ejecuta de nuevo
$ behave features/prompt_enhancement.feature
FAIL: PromptEnhanceView not found

# Abra views.py, Copilot sugiere la vista
# Ejecuta de nuevo
$ behave features/prompt_enhancement.feature
FALLO: Falta campo  'enhanced_prompt' en la respuesta

# Actualice la vista, Copilot sugiere el campo
# Ejecuta de nuevo
$ behave features/prompt_enhancement.feature
1 Escenario pasado
```

Cada fallo de test es una **retroalimentación inmediata** que Copilot utiliza para afinar sus sugerencias.

5. Simula dependencias externas, prueba tu lógica

```
# Bueno: Verifica el comportamiento, simula la API
externa
@when('mejoro un prompt')
def step_impl(context):
    con mock.patch('gemini_client.generate') como
        mock_generate:
        mock_generate.return_value = {'content':
        'Enhanced prompt text'}
        context.result =
enhance_prompt(prompt="ayúdame")

    assert mock_generate.called # Verifique que
    nuestro códico llamé a Gemini
    assert context.result.enhanced # Verifica que
    nuestra lógica funcionó

# Malo: Simula todo, no prueba nada
@when('mejoro un prompt')
def step_impl(context):
    context.result = mock.Mock(enhanced='Some text')
    # Solo estamos fingiendo
```

Usa # Menciones para darle contexto a Copilot

Copilot Chat no ve todo automáticamente: necesitas indicarle explícitamente qué revisar usando # menciones

Contexto esencial (usa # para referenciar):

- #terminalLastCommand – La salida del test más reciente
- #file:features/*.feature – Archivo de funcionalidades con los requisitos
- #file:docs/sequence-diagrams/*.md – Diagrama Mermaid de la arquitectura

Ejemplo de flujo de trabajo:

1. Ejecuta test: behave features/calculator.feature
2. El test falla con un error específico
3. Abre Copilot Chat y escribe un prompt usando las referencias # para dar contexto:

> Arregla el test que falla.
> #terminalLastCommand
> #file:features/calculator.feature
> #file:calculator.py

4. Copilot genera la solución basada en el contexto que proporcionaste
5.Aplica los cambios y ejecuta el test nuevamente

Cuanto más específico sea el contexto que proporciones mediante # menciones, mejores serán las sugerencias de Copilot.

7. Itera hasta que todo pase
Cada fallo es un problema específico y resoluble:

- Ejecuta el test → La terminal muestra qué está roto
- Pregunta a Copilot Chat usando #terminalLastCommand → Ve el error exacto
- Referencia tu diagrama Mermaid con #file: → Sabe cómo arreglarlo
- Copilot genera el código
- Ejecuta el test nuevamente → Feedback más específico

Ejecuta tests con frecuencia (después de cada pequeño cambio). Cuanto más rápido sea el ciclo de retroalimentación, más determinista será el desarrollo.
Clave: Usa #terminalLastCommand en Copilot Chat después de cada ejecución de test para darle retroalimentación fresca.

8. Equilibrar tests unitarios e integrales
Usa la herramienta adecuada para cada caso:

Tests Unitarios (rápidos, dependencias simuladas):

- Prueban la lógica de negocio de forma aislada
- Simulan APIs externas, bases de datos, I/O lento
- Se ejecutan en cada guardado de archivo
- Ideales para ciclos TDD de rojo-verde-refactor

Tests de Integración (más lentos, dependencias reales):

- Prueban interacciones reales entre servicios
- Usan base de datos de prueba, APIs de staging
- Se ejecutan antes de commits o en CI/CD
- Verifican que los flujos de extremo a extremo funcionen

Tests BDD de Funcionalidad (conductuales, simulación estratégica):

- Se enfocan en el comportamiento visible para el usuario
- Simulan lo que no controlas (Gemini, SendGrid)
- Usan implementación real para tu propio código
- Documentan los requisitos en formato legible por humanos

El objetivo no es “nunca usar mocks”, sino s**imulación estratégica** que aísle lo que estás probando sin ocultar bugs reales.

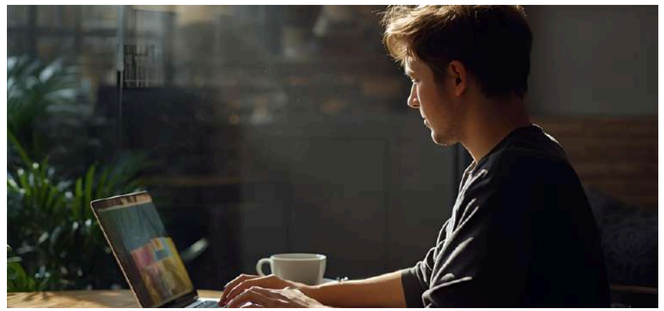
El Futuro es Determinista (por fin)

A medida que los asistentes de programación con IA como Copilot se vuelven más poderosos, la brecha entre “lo que puedo imaginar” y “lo que puedo construir” sigue reduciéndose. Pero eso solo es útil si podemos **verificar** lo que construimos.

El desarrollo guiado por funcionalidades convierte a Copilot Chat de un “generador de código que podría estar bien” en una “implementación que es **demostrablemente correcta**”.

No se trata de desconfiar de la IA, sino de **confiar, pero verificar**. Reagan tenía razón en algo.

¿Y la verificación a la velocidad de la generación de Copilot con retroalimentación de la terminal? Ahí es cuando las cosas se ponen realmente interesantes. Ahí es cuando dejas de apostar y empiezas a entregar.



Pruébalo Tú Mismo (en serio, hazlo)

Aquí tienes el ejemplo más simple para empezar:

```
# features/calculator.feature
Escenario: Sumar dos números
Cuando sumo 2 y 3
Luego el resultado debe ser 5
```

```
# features/steps/calculator_steps.py
@when('sumar {a:d} y {b:d}')
def step_impl(contexto, a, b):
    contexto.resultado = sumar(a, b) # Esto fallará, la función aún no existe

@then('el resultado deberia ser {esperado:d}')
def step_impl(contexto, esperado):
    assert contexto.resultado == esperado
```

Ejecuta el test en la terminal de VS Code. Observa que falla con un mensaje claro. Abre Copilot Chat y pídele que arregle el test, usando #terminalLastCommand para mostrarle el error. Copilot implementará la función add(). Ejecuta el test nuevamente y observa cómo pasa correctamente.

REjecuta el test en la terminal de VS Code. Observa que falla con un mensaje claro. Abre Copilot Chat y pídele que arregle el test, usando #terminalLastCommand para mostrarle el error. Copilot implementará la función add(). Ejecuta el test nuevamente y observa cómo pasa correctamente.

Ahora escala eso a toda tu aplicación. Eso es todo. Ese es el juego completo.

Conclusión: No más Programación al Estilo Ruleta Rusa

"Vibe coding" con LLMs es divertido, rápido y a veces incluso funciona. Pero para software de producción —código que necesita ser mantenido, extendido, depurado a las 3 a.m. y realmente confiable— necesitamos algo mejor.

El desarrollo guiado por tests con arquitectura progresiva nos ofrece:

- Requisitos claros: archivos de funcionalidades que los humanos pueden leer
- Restricciones específicas: aserciones de test que no mienten
- Corrección verificable: tests que pasan = código que funciona
- Documentación viva: los tests son la especificación, no una wiki polvorienta
- Seguridad al refactorizar: los tests detectan regresiones al instante
- Complejidad progresiva: arquitectura que coincide con tu recorrido real: Básico → Intermedio → Avanzado
- Soluciones del tamaño correcto: los LLMs generan código con exactamente el nivel de complejidad que necesitas

El resultado: resultados deterministas con Copilot Chat al nivel de complejidad adecuado. No más máquinas tragamonedas.

No más cruzar los dedos. No más "funcionó en mi máquina" seguido de incendios en producción. Y lo más importante: no más enredos sobre-diseñados ni hacks subdesarrollados.

Documentas tu realidad (Básico), tu siguiente paso (Intermedio) y tu meta futura (Avanzado). Copilot sabe en cuál estás trabajando a partir del diagrama que le muestras. Tus pruebas funcionan en los tres niveles. Pasas de un nivel a otro cuando enfrentas problemas reales, no arquitecturas aspiracionales.

El flujo de trabajo: **archivos de funciones → diagramas Mermaid → retroalimentación de pruebas en la terminal → Copilot Chat (con menciones #) → repetir.**

Los "vibes" son opcionales. Las pruebas no lo son. ¿El ciclo de retroalimentación en la terminal? Esencial. ¿Las menciones # en Copilot Chat? Así es como le das el contexto correcto. ¿Y la arquitectura progresiva? Eso es lo que te mantiene cuerdo.

Y honestamente... una vez que te acostumbras a este flujo de trabajo —pruebas que verifican, diagramas que coinciden con la realidad y una arquitectura que crece contigo— volver a "vibe coding" se siente como intentar navegar con los ojos cerrados. Claro, eventualmente podrías llegar a tu destino... pero, ¿para qué querrías?

¿Interesado en optimizar tu proceso de desarrollo con IA?

Ya sea que estés explorando nuevos enfoques o buscando perfeccionar tu flujo de trabajo actual, estamos aquí para ayudarte.

Contáctanos para una consulta o colaboración, y hablemos sobre cómo podemos impulsar tus proyectos hacia adelante.

Contáctanos hoy mismo para comenzar:

- edgarjoya@talaverasolutions.com
- gabriel@talaverasolutions.com

Sitio Web: www.Promptshelf.ai

Instagram: @PromptShelf.ai

X: @PromptShelfai

Reddit: www.Promptshelf.ai