PromptShelf.AI
Powered by TALAVERA SOLUTIONS

Beyond Vibe Coding

Making LLM Assisted Development Deterministic

By Edgar Joya, VP Full Stack Development COE
Gabe Arce, Talavera Solutions CEO

The Problem with "Vibe Coding"

Over the last couple of months, I've been trying to determine how to work effectively with AI. LLMs have been exceptional at transforming my half-engineered ideas into reality—only the ether knows how much ADHD driven curiosity has been satisfied. But for real-world production software, like most of us, I've been caught in an endless cycle of discovery: bouncing between code snippets, exploring new ideas, and testing different pathways to create software.

The feedback loop of iterating back and forth with a system of neural matrices—hoping each generation brings you closer to what you actually need—starts to feel like playing slots at a casino, waiting for that magical block of code to give you the outcome you wanted. After a while, it gets exhausting. I've genuinely had more fun watching brew or apt package updates scroll by.

This is what some call "vibe coding" - development driven by intuition and crossing your fingers that the generated code is actually correct. The code looks right, the patterns seem reasonable, but there's no concrete way to verify correctness beyond "it compiles and doesn't immediately crash."

The problem isn't the LLM - it's the **lack of constraints** and context. When you ask an LLM to "create an order processing endpoint," sure, you'll get code. But does it actually match what you needed? Does it handle edge cases? Will it survive the next refactor? Who knows! feels like a round of Russian roulette.

Enter: Test-Driven Development with Copilot Chat

Here's the thing that changed everything for me: **What if the tests told Copilot exactly what to build?**

Instead of this mess:

Me: "Create a prompt enhancement endpoint"
Copilot: *generates 200 lines of code*
Me: "Looks good!" *nervously hits save*

I started doing this:

Feature File: "When I POST a vague prompt to /api/v1/enhance, I should get an enhanced version"
Test: FAIL - "Endpoint not found at /api/v1/enhance"
Copilot Chat: *generates minimal endpoint*
Terminal: Test output shows "Response missing 'enhanced_prompt' field"
Copilot Chat: *adds enhanced_prompt*
Test: PASS

Mermaid Diagram: Level 1 (Basic) architecture
Feature File: "When I POST a vague prompt to **/api/v1/enhance, I get enhanced version"**
Terminal: FAIL - "Endpoint not found at /api/v1/enhance"
Copilot: *suggests minimal endpoint based on diagram + test output*
Terminal: FAIL - "Response missing 'enhanced_prompt' field"
Copilot: *adds enhanced_prompt field*
Terminal: PASS

My Workflow: I use GitHub Copilot in VS Code with terminal feedback loops. Tests run in the terminal, Copilot sees the output + feature files + Mermaid diagrams, and generates code that satisfies the constraints. No manual LLM prompting needed.

The Feature-Driven Development Pattern

Phase 0: Visualize Before You Code (with Progressive Architecture)

Before writing any feature files or tests, I start with diagrams. But here's the critical part: **don't document your dreams, document your journey.**

Instead of creating one beautiful diagram showing your future microservices architecture, create **three progressive diagrams** that match reality:

Level 1: Basic (What You're Building First)

User → API Endpoint → enhance_prompt() → LLM (Gemini) → Database

When: First implementation, MVP, proving the concept

Characteristics:

- Inline logic in controllers/views
- Functions, not services
- Simple data structures (dicts, basic models)
- Direct LLM calls

Example: A simple prompt enhancement endpoint that takes vague prompts and makes them specific

Level 2: Intermediate (Next 2-4 Features)

User → API → Enhanced Functions → LLM → Database (+ new models)
↓
Validation Layer
↓
Domain Logic (quality checks, versioning)

When: Adding features, handling edge cases, storing history

Characteristics:

- New models (PromptVersion, EnhancementHistory)
- Validation and error handling
- Domain-specific logic
- Still monolithic, but organized

Example: Add prompt versioning, quality scoring, and comparison between original//enhanced prompts.

Level 3: Advanced (Future/Production Scale)

```
User →
        API Gateway → Auth Service → Enhancement
Service
                                                → LLM Service
                                                → Memory Service
                                                → Analytics Service
                                                → Cache Layer
                                                → Database Cluster
```

When: Scaling to thousands of users, multiple teams, need separate deployments

Characteristics:

- Microservices architecture
- Separate databases per service
- Message queues
- Complex orchestration

Example: Full production system with dedicated services for caching, rate limiting, analytics, and multimodel LLM routing.

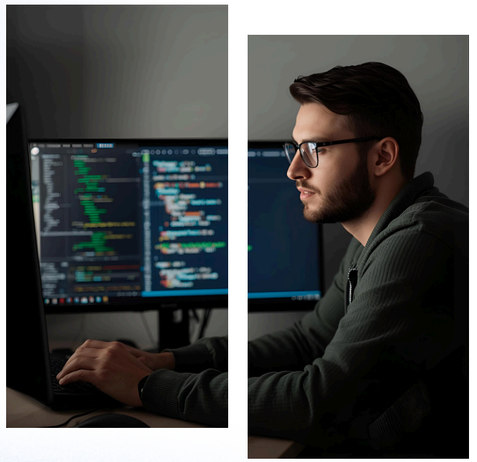
Why Three Levels Matter

When you create all three diagrams upfront:

1. **Level 1 diagram** keeps your initial implementation simple (no over-engineering)
2. **Level 2 diagram** shows your next step (clear growth path)
3. **Level 3 diagram** documents your vision (team alignment, but not immediate)

Critical: When implementing a feature, give Copilot Chat the correct level diagram. Don't give it the Advanced microservices diagram when you're building Basic!

Preferred Format: I use Mermaid diagrams in markdown - they're version-controllable, render in GitHub/VS Code, and easy to update as the system evolves.



Practical Sequence Diagram Example (Level 1 - Mermaid)

Before any code, I draw the exact interaction flow using Mermaid syntax:

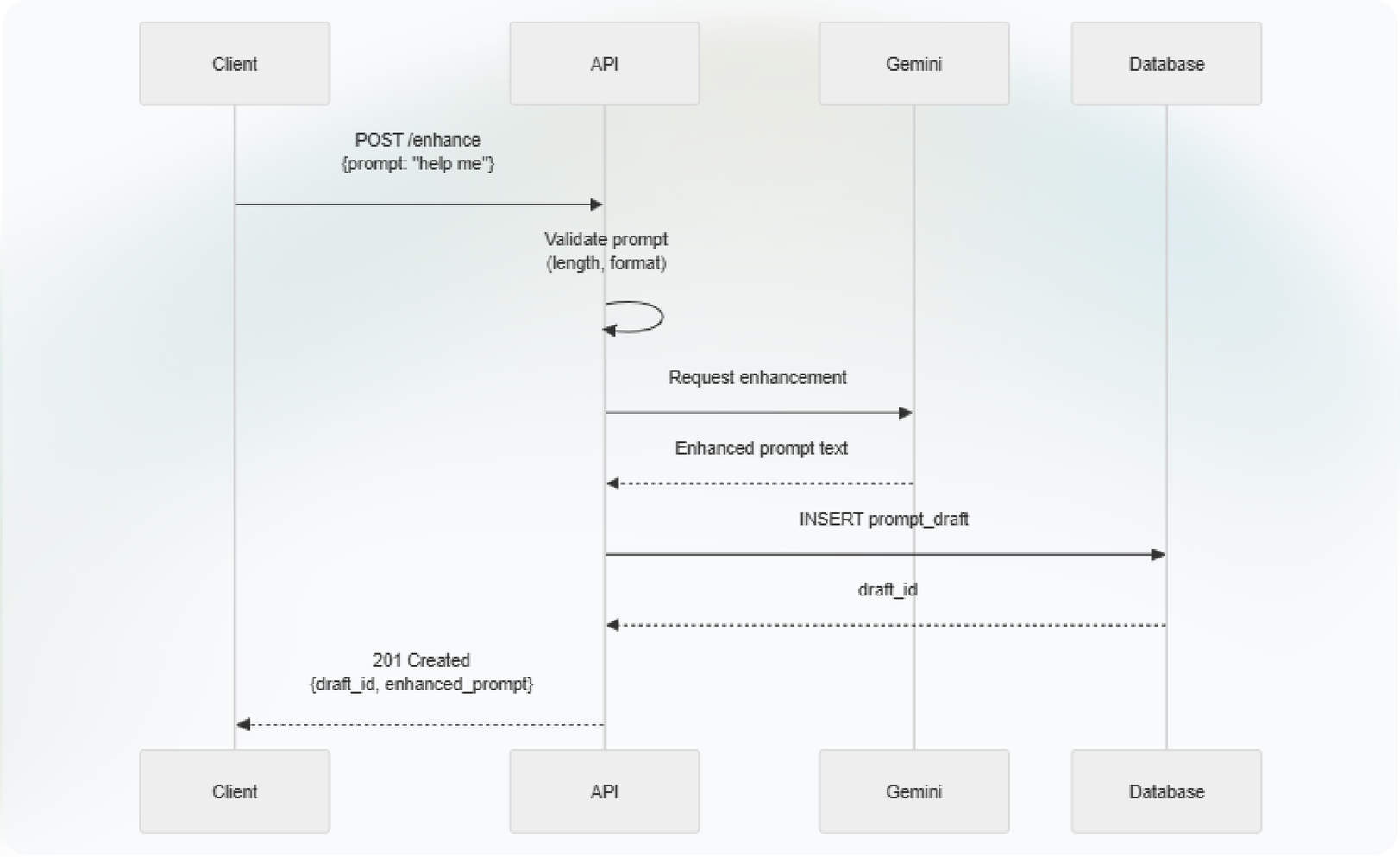


Diagram 1

This diagram becomes my "contract" that tests will enforce. I save it as docs/sequence-diagrams/promptenhancement-basic.md so Copilot can reference it.

Level 1 Flow Summary:

User → API Endpoint → enhance_prompt() → Gemini → Database

When: First implementation, MVP, proving the concept

Characteristics:

- Inline logic in controllers/views
- Functions, not services
- Simple data structures (dicts, basic models)
- Minimal abstractions

Example: "I need an API endpoint that enhances prompts"

- One PromptController
- One enhance_prompt() function
- Calls Gemini API directly
- Saves to PromptDraft table

Why Mermaid?

- Version-controlled as plain text
- Copilot can read it as context
- Easy to update as architecture evolves

Level 2: Intermediate (Next 2-4 Features)

User → API → Enhanced Functions → Business Logic → External Service → Database

When: Adding related features, seeing patterns emerge

Characteristics:

- Extracted business logic functions
- Additional models for data relationships
- Configuration files for domain knowledge
- Still monolithic

Example: "Now I need version comparison, change tracking, and context preservation"

- Add compare_versions() , track_changes() functions
- Add PromptVersion model for audit trail
- Add enhancement_rules.json for business logic
- Still in same codebase

Level 3: Advanced (When You Actually Need It)

User → API Endpoint → enhance_prompt() → Gemini → Database

When: Real scaling problems, multiple teams, performance bottlenecks

Characteristics:

- Separate services with clear boundaries
- Event-driven architecture
- Dedicated components (caching, queuing)
- Microservices-ready

Example: "We process 10k enhancements/hour and need multiple teams"

- Separate Enhancement
- Service Event Bus for async processing
- Context Service handles user preferences
- Multiple databases

Critical Rule: You create all three diagrams upfront, but **only implement Level 1**. Move to Level 2 when you have real problems Level 1 can't solve. Move to Level 3 when Level 2 breaks.

Why Progressive Architecture Matters

Without it:

Me: "Build a prompt enhancement API"
Copilot: *Creates microservices architecture with event sourcing*
Me: "Too complex for MVP"
Copilot: *Backs off to something too simple*
Me: "This won't support conversation context"
Copilot: *Confused, generates inconsistent code*

With it:

Me: "Build prompt enhancement. Here's the Level 1 diagram."
Copilot Chat: *Creates clean function-based implementation*
Tests: All passing [Later...]
Me: "Add conversation context and version tracking. Here's the Level 2 diagram."
Copilot Chat: *Adds enhanced functions and event model*
Tests: Still all passing

The Magic: Your BDD tests work across all three levels. Same API contract. Same behavior. Different implementation complexity.

Sequence Diagrams for Each Level (Mermaid)

For each level, create Mermaid diagrams showing the exact sequence of operations:

Level 1 (Basic) - Simple Direct Flow:

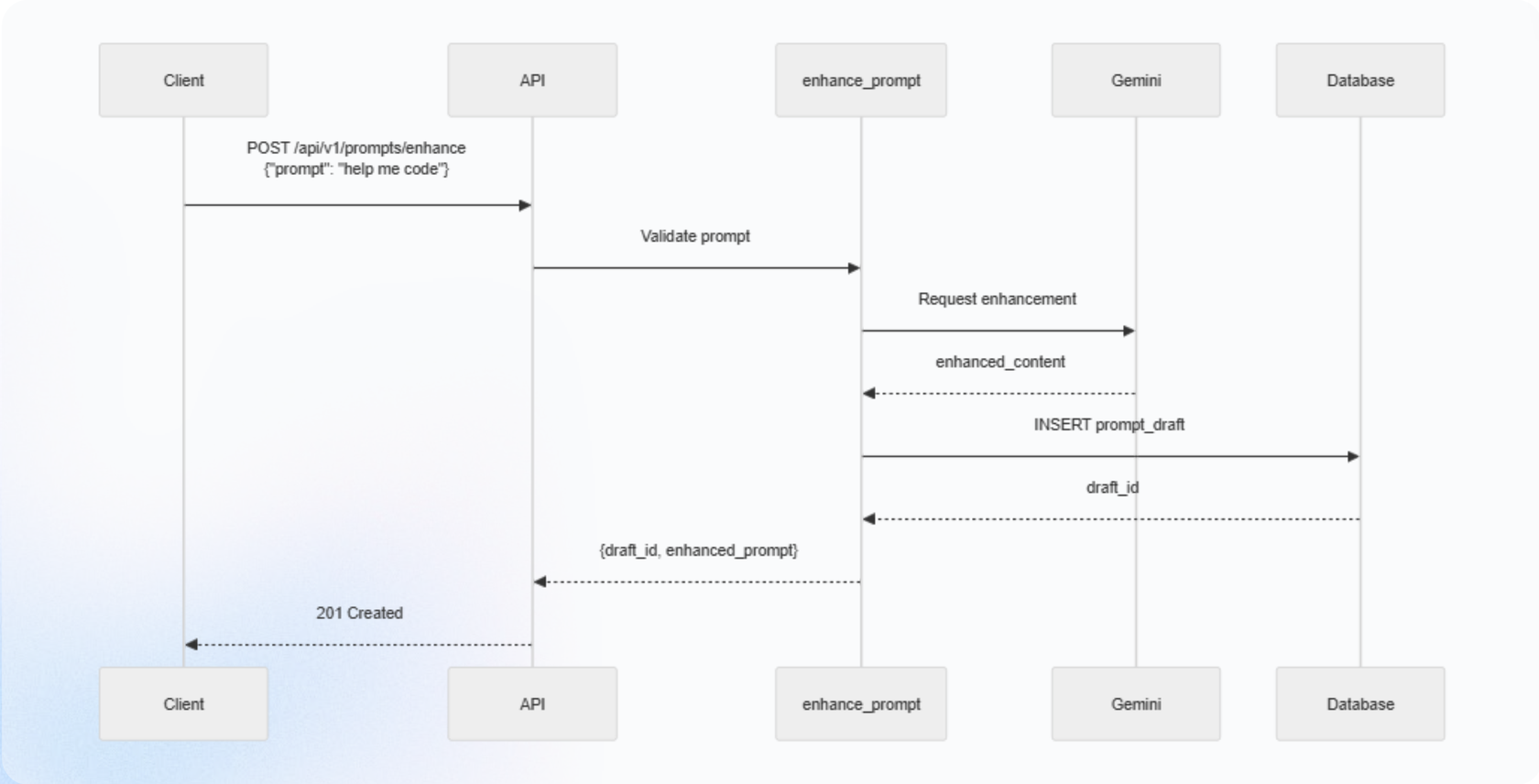


Diagram 2

Level 2 (Intermediate) - Add Business Logic Layer:

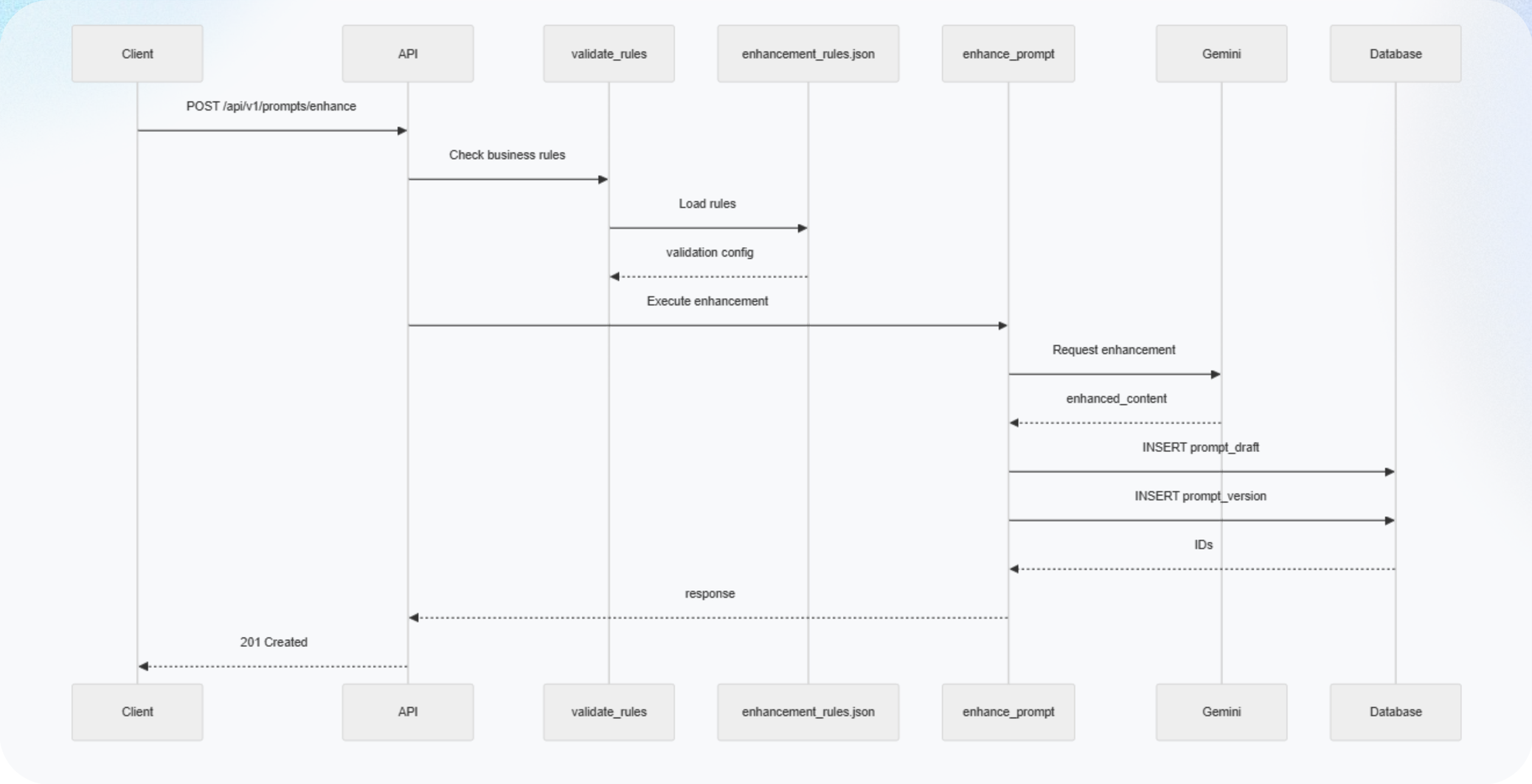


Diagram 3

Level 3 (Advanced) - Event-Driven Microservices:

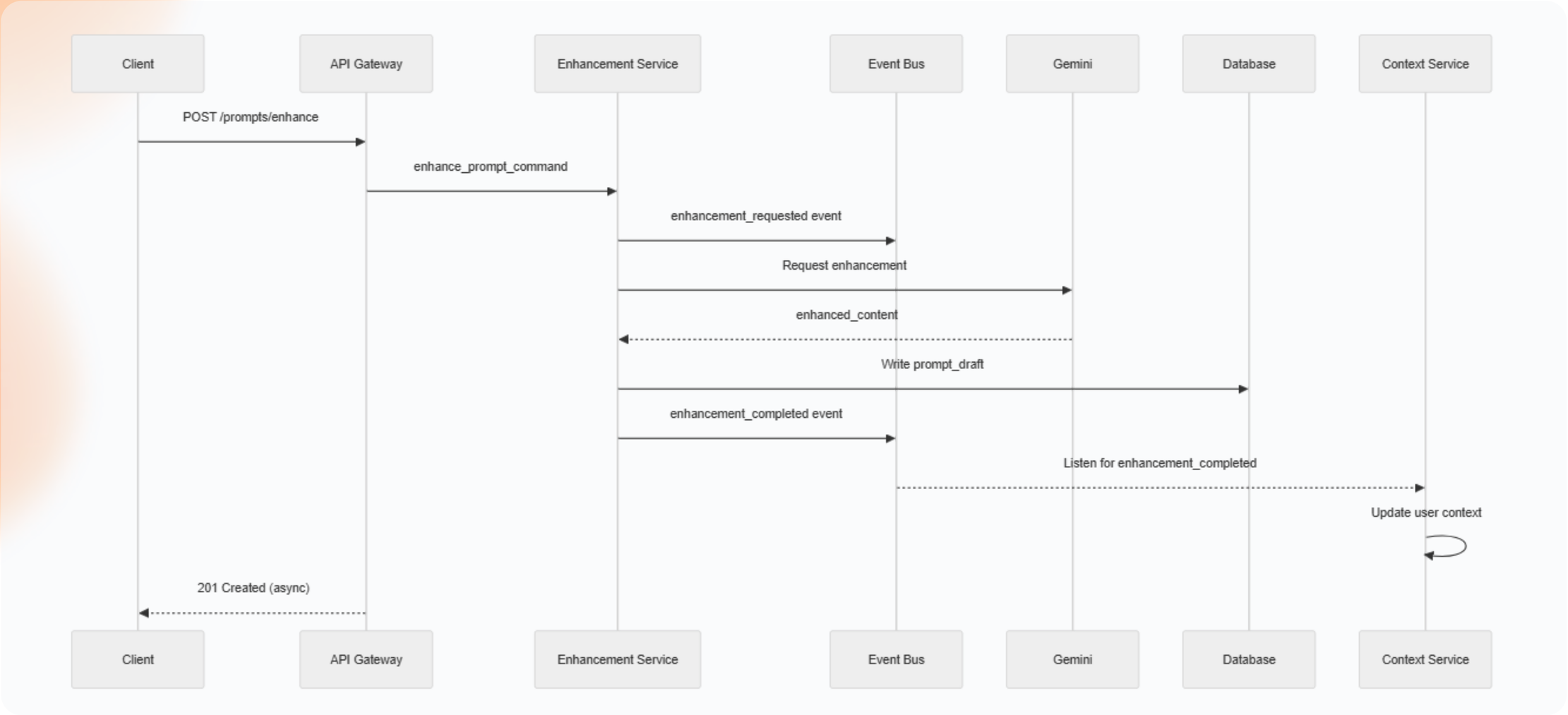


Diagram 4

Key Insight: The client gets the same response (201 Created with prompt details) regardless of which level you've implemented. Tests don't change. Only the internal complexity changes.

Pro Tip: Save these as separate files:

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

Then reference the appropriate one when working with Copilot Chat!

Phase 1: Write Human-Readable Requirements

Scenario: Enhance a vague prompt
Given I have a vague prompt "help me write code"
When I request enhancement via the API
Then the response status should be 200
And the response should contain an enhanced_prompt
And the enhanced prompt should be more specific than the original
And the enhanced prompt should be at least 3x longer

Key Point:
Non-technical people can actually read this. Your PM can validate it. Your designer can understand it. And it maps directly to the sequential diagram you drew.

Non-technical people can read this. Your PM can validate it. Your designer can understand it. And it maps to whichever level diagram you're currently implementing.

Phase 2: Create Failing Tests

```
@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
```

Key Point: The test **actually checks** that your implementation exists for core business logic. Mock external dependencies (LLM APIs, email services, external databases) but verify your own code paths exist. And the failure message points to the right architectural level.

When to Mock:
- External APIs (Gemini, OpenAI, SendGrid)
- Third-party services you don't control
- Slow operations (file I/O, network calls) in unit tests
- Before CI/CD integration is set up

When to Use Real Implementation:
- Your own business logic
- Database operations (use test database)
- Core application flows
- Integration tests with actual service interactions

Phase 3: Let Tests Drive Copilot Chat Code Generation

Now when you ask Copilot Chat to implement something, you've got:

1. **Clear requirements** (the feature file - no more vague "build me a thing")
2. **Specific failure messages** (exactly what's broken and where from terminal output)
3. **Explicit success criteria** (the test either passes or it doesn't)
4. **The right architectural level** (Basic diagram, not your future microservices dream)

My workflow: I run the tests in the VS Code terminal, then ask Copilot Chat to help, using # mentions to provide context:

> The test failed. Please implement the endpoint following the Basic diagram.
> #terminalLastCommand
> #file:features/prompt_enhancement.feature
> #file:docs/sequence-diagrams/prompt-enhancement-basic.md
> #file:api/urls.py
> #file:api/views.py

Copilot's output is no longer gambling - it's **constrained by concrete assertions and appropriate complexity.** No more slot machine vibes. No more over-engineered solutions.

What you give Copilot Chat (via # mentions):

> #terminalLastCommand - Terminal output with specific failure
> #file:features/*.feature - Your feature file with clear requirements
> #file:docs/sequence-diagrams/*.md - The Mermaid diagram at the correct architectural level
> #file:api/*.py - Your existing code patterns

The magic: Copilot's suggestions are constrained by:

- The test failure (via *#terminalLastCommand* - it knows exactly what's broken)
- The diagram (via *#file*: - it follows the right architecture level)
- Your codebase conventions (via *#file*: - it matches your style)
- The terminal feedback loop (each iteration gets more specific)

Example of providing context to Copilot Chat:

> Please fix the failing test.
> #terminalLastCommand

The terminal output might show:

> $ behave features/prompt_enhancement.feature
>
> FAIL: Endpoint not found at '/api/v1/prompts/enhance'.
> Implement the endpoint following the Level 1 (Basic) diagram.
> Add the route to api/urls.py with path: 'prompts/enhance'

With this context, Copilot suggests the exact minimal implementation needed. No over-engineering. No guessing.

Key Insight: The combination of:

- BDD feature files (requirements)
- Terminal test output (immediate feedback)
- Mermaid diagrams (architecture constraints)
- Copilot (code generation)

...creates a deterministic development loop. The terminal tells you what's broken, the diagram tells you how to fix it, and Copilot generates the code that satisfies both.

Why This Actually Changed How I Work

1. Deterministic Outcomes (Finally!)

```
# Vibe coding
assert response.data # Hope there's data!
# Test-driven
assert "session_id" in response.data
assert uuid.UUID(response.data["session_id"])
assert len(response.data["message"]) > 20
```

No ambiguity. The UUID is either valid or it's not. The message is either helpful or it's too short. Done.

2. Self-Documenting Code (That Actually Helps)

Your feature files become living documentation that people can actually read:

```
# This IS the spec
Scenario: Export session with conversation history
When I request to export the session
Then the export should contain the final prompt
And the export should include conversation history
And the export should provide usage instruct
```

Product managers can write these. Designers can validate them. QA can actually trace them. The LLM can implement them. Everyone's speaking the same language for once.

3. Refactoring Without Fear

Six months later, you need to refactor the session logic. With traditional vibe coding, you're sweating bullets - what if you break something obscure?

With test-driven development:

```
$ make test
All tests passing

*refactors aggressively while blasting music*

$ make test
All tests still passing
```

If the tests pass, the behavior is correct. That's it. No anxiety, no "let me just manually test 47 different scenarios."

4. Copilot Chat as a Constrained Agent (Not a Chaos Generator)

Think of Copilot like a brilliant but overeager junior developer:

- **Without tests:** "Build something cool!" → Goes wild, creates something interesting but probably not what you needed
- **With tests + terminal feedback:** "Make this specific test pass" → Focused, verifiable, actually solves your problem

The tests are guard rails. The terminal output is the feedback loop. They channel all that generative power into solving your exact problem, not creating art projects.

Real-World Example: How I Actually Do This

Let me walk you through implementing a feature the way I do it now.

Step 0: Create the Sequence Diagram

Before touching any code, I draw out the interaction using Mermaid syntax:

The Sequence Diagram

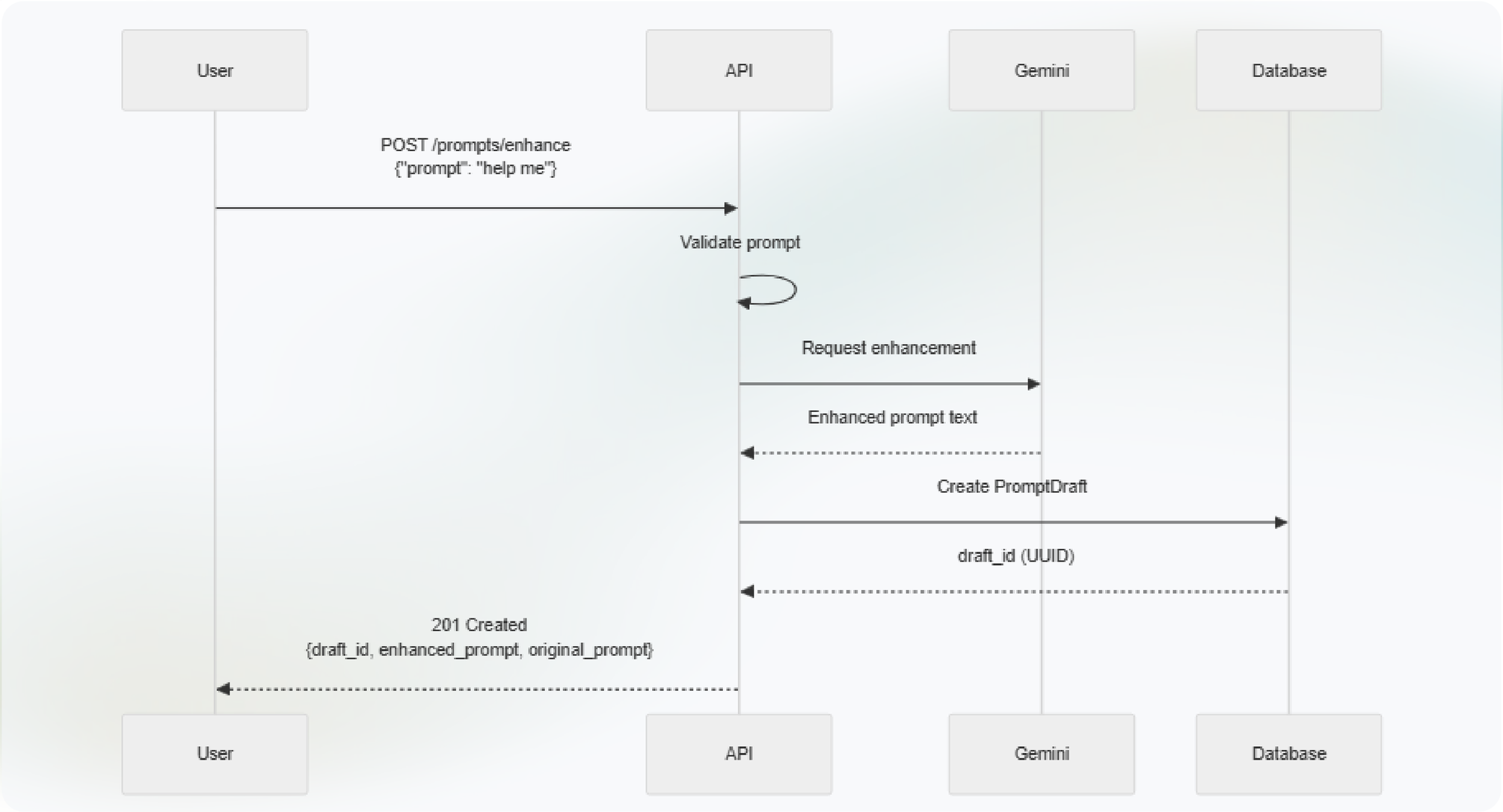


Diagram 5

This diagram becomes my "contract" that the tests will enforce. No more guessing

Step 1: Feature File (Business Logic)

Scenario: Enhance a vague prompt
Given I have a vague prompt "help me"
When I submit the prompt for enhancement
Then the response status should be 201
And the response should contain a draft_id
And the response should contain an enhanced_prompt
And the prompt draft should be recorded in the database

Note: Each Then step maps directly to something in the sequence diagram. Everything connects.

Step 2: Run Tests (Watch Them Fail)

```
$ behave features/prompt_enhancement.feature

FAIL: Prompt enhancement endpoint not found at
'/api/v1/prompts/enhance'.
Please implement the endpoint in Django.
Add the route to api/urls.py with path: 'prompts/enhance'
```

Perfect. Clear failure message. No ambiguity about what needs to happen next.

Step 3: Give Copilot Chat the Context It Needs

Here's my actual workflow in VS Code:

1. **Run the test** in the integrated terminal to see the failure
2. **Open Copilot Chat** and ask it to help fix the failing test
3. **Use # mentions** to explicitly add context:

- #file:prompt_enhancement.feature - The feature requirements
- #file:docs/sequence-diagrams/prompt-enhancement-basic.md - The architecture diagram
- #terminalLastCommand - The test output showing what failed
- #file:api/urls.py and #file:api/views.py - Files I want it to edit

Example Copilot Chat prompt:

```
The test failed. Please implement the missing
endpoint.

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

Copilot Chat now has:

- **Terminal output** (via #terminalLastCommand) showing the specific failure
- **Feature file** (via #file:) with clear requirements
- **Sequence diagram** (via #file:) showing the architecture
- **Existing code** (via #file:) to understand patterns and make edits

It generates code that directly addresses the test failure, following the diagram's architecture

The workflow is iterative:

- Write feature → Run test → See failure → Ask Copilot Chat (with context) → Apply changes → Run test again
- Each terminal output provides more specific feedback
- Use #terminalLastCommand to give Copilot the latest test results
- Tests pass when all requirements are satisfied

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)
```

Look at that. Minimal, focused, does exactly what the test requires. No extra bells and whistles.

Step 5: Run Tests Again

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

- Endpoint found
- Returns 201
- Contains session_id
- session_id is valid UUID
- Contains greeting message
- Greeting is helpful (contains keywords: "help", "build")

All scenarios passing!

This feeling never gets old.
Step 6: Iterate When You Need More
dd more requirements to the feature file:
```gherkin
And the response should include actionable suggestions
```

Run tests in terminal. They fail:

```
FAIL: Response missing 'suggestions' field
```

Copy that terminal output to Copilot Chat. It fixes it. Repeat until everything's green.

The Benefits Actually Compound

For Teams (I Just Started Leading One, Growing Pains Included)

I lead a team of four developers (Including myself), and I've seen firsthand where LLM-generated code is a game-changer and where it falls flat on its face.

- **Shared understanding:** Feature files + progressive diagrams become the single source of truth
- **Better code reviews:** Review the tests first, check if implementation matches the current level diagram
- **Easier onboarding:** New devs can trace requirements → level-appropriate diagrams → tests → code
- **Clear progression path:** Everyone knows we're at Level 1, working toward Level 2, with Level 3 documented for the future
- **Consistent standards:** The tests enforce patterns across the whole team's code

When everyone's working with LLMs, having tests as guard rails means we're all building toward the same spec, not four different interpretations.

For Solo Deveopers

- **Confidence in AI-generated code:** You know it's correct because tests say so
- **Faster iteration:** Clear failure messages = no guessing what went wrong
- **Way less debugging:** Catch issues immediately, not at 2am in production

For Copilot Chat Workflow

- **Precise requirements:** No more "build me an API" → mystery box of code
- **Incremental progress:** Each test failure from terminal is one specific, solvable task
- **Instant validation:** Every Copilot suggestion gets immediately verified in terminal
- **Tight feedback loop:** Terminal → Copilot Chat → Code → Terminal

Common Pushback (And Why It's Wrong)

- **Writing tests takes too long!":** You're already writing tests - they're just in your head. Making them explicit catches bugs earlier and makes the LLM way more effective. Plus, honestly? Writing a failing test is faster than debugging mystery code at 11pm.
- **"My requirements change too fast for this!":** Even better! When requirements change, update the feature file first. Tests fail, showing you exactly what needs updating. Then the LLM fixes the implementation. You just turned chaos into a clear checklist.
- **"This doesn't work for exploratory coding!":** True! For prototypes and spikes, vibe away. I still do. But when it's time to ship to production? Tests or GTFO.

Getting Started (Actually Pretty Simple)

1. Pick a Testing Framework

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

Don't overthink this. Pick one and go

2. Diagram First, Code Later
Before writing any code or tests, create **progressive Mermaid diagrams** that match your development journey:

Basic Diagram (document what you have now)**:**

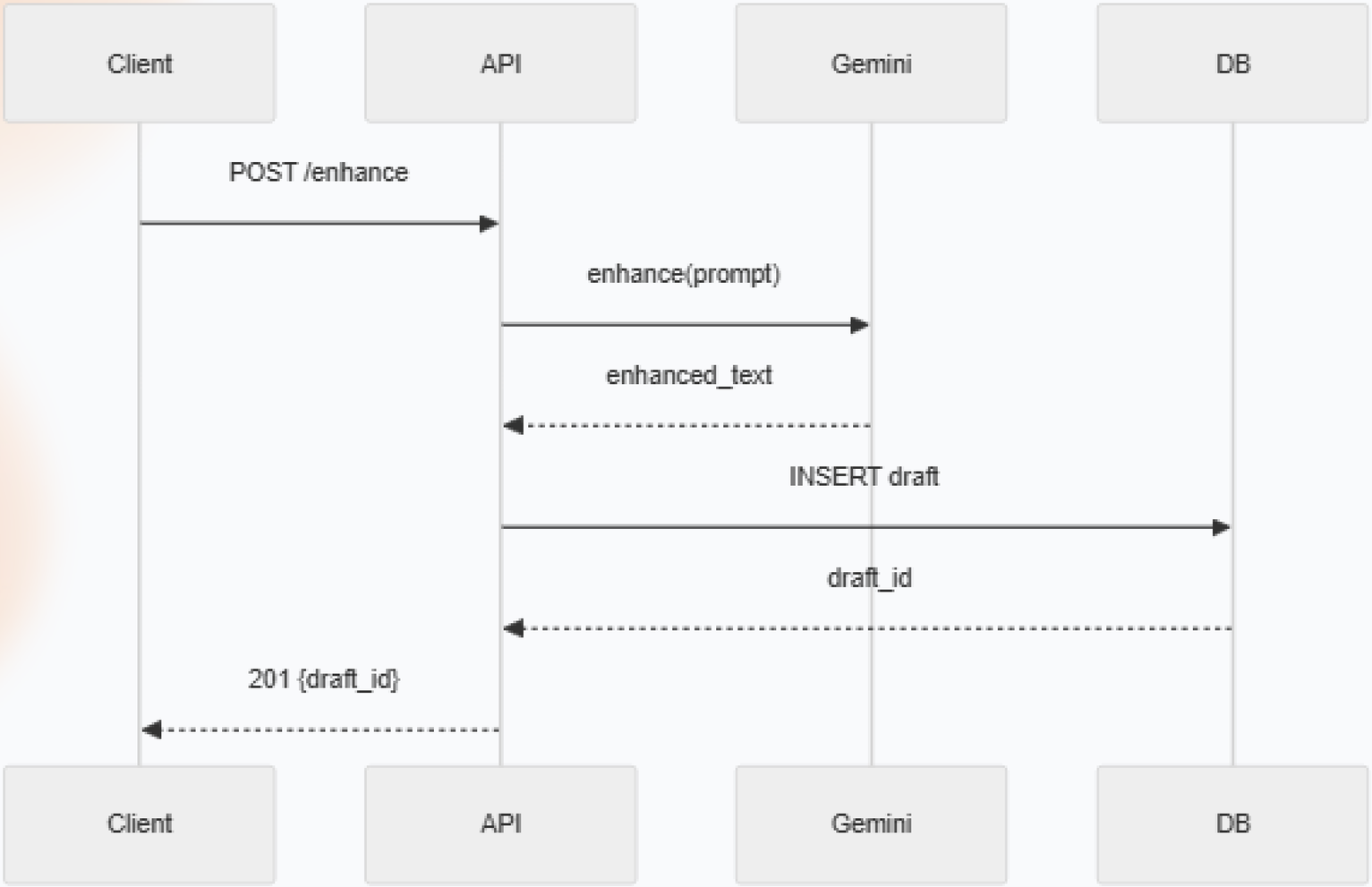


Diagram 6

Intermediate Diagram (next 2-3 features):

- Add validation layer, new models (PromptVersion)
- Enhanced functions with context
- Still monolithic

Advanced Diagram (future goal):

- Separate services if needed
- Event-driven architecture
- When you actually scale

Why Mermaid?

- Version-controllable (plain text in markdown)
- Renders in GitHub/VS Code/GitLab
- Easy to update as system evolves
- Copilot understands it as context

Create all three levels upfront in docs/sequence-diagrams/ , but only implement one level at a time.

3. Write One Feature File

Start small. One scenario, 5-10 steps that map to your Mermaid diagram. Tag it with the implementation level:

```
@level-basic @conversation
Scenario: Interactive prompt building
    # Implement with basic architecture
```

Don't try to describe your entire app on day one.

4. Run Tests, Let Terminal Output Guide Copilot

Write step definitions that check real behavior. Then let the workflow guide you:

1. **Run test** → Terminal shows specific failure
2. **Open file** → Copilot sees terminal output + feature file + diagram
3. **Start typing** → Copilot suggests implementation
4. **Run test again** → More specific feedback
5. **Iterate** → Until tests pass

Example of terminal-driven development:

```
# First run
$ behave features/prompt_enhancement.feature
FAIL: Endpoint not found at '/api/v1/prompts/enhance'

# Open urls.py, Copilot suggests the route
# Run again
$ behave features/prompt_enhancement.feature
FAIL: PromptEnhanceView not found

# Open views.py, Copilot suggests the view
# Run again
$ behave features/prompt_enhancement.feature
FAIL: Response missing 'enhanced_prompt' field

# Update view, Copilot suggests the field
# Run again
$ behave features/prompt_enhancement.feature
1 scenario passed
```

Each test failure is **immediate feedback** that Copilot uses to refine its suggestions

5. Mock External Dependencies, Test Your Logic

```
# Good: Checks behavior, mocks external API
@when('I enhance a prompt')
def step_impl(context):
    with mock.patch('gemini_client.generate') as
        mock_generate:
        mock_generate.return_value = {'content':
        'Enhanced prompt text'}
        context.result =
enhance_prompt(prompt="help
        me")

    assert mock_generate.called # Verify our code
    called Gemini
    assert context.result.enhanced # Verify our
logic
    worked

# Bad: Mocks everything, tests nothing
@when('I enhance a prompt')
def step_impl(context):
    context.result = mock.Mock(enhanced='Some
text')
    # Just pretending
```

6. Use # Mentions to Give Copilot Chat Context

Copilot Chat doesn't automatically see everything - you need to explicitly tell it what to look at using # mentions:

Essential context (use # to reference):

- #terminalLastCommand - The most recent test output
- #file:features/*.feature - Feature file with requirements
- #file:docs/sequence-diagrams/*.md - Mermaid diagram for architecture
- #file: for files you're editing (urls.py, views.py, models.py)

Example workflow:

1. Run test: behave features/calculator.feature
2. Test fails with specific error
3. Open Copilot Chat and prompt:

> Fix the failing test.
> #terminalLastCommand
> #file:features/calculator.feature
> #file:calculator.py

4. Copilot generates fix based on the context you provided
5. Apply the changes and run test again

The more specific context you provide via # mentions, the better Copilot's suggestions

7. Iterate Until Green

Each failure is a specific, solvable problem:

- Run test → Terminal shows what's broken
- Ask Copilot Chat with #terminalLastCommand → It sees the exact error
- Reference your Mermaid diagram with #file: → It knows how to fix it
- Copilot generates the code
- Run test again → More specific feedback

Run tests frequently (after each small change). The faster the feedback loop, the more deterministic the development.

The key: Use #terminalLastCommand in Copilot Chat after every test run to give it fresh feedback.

8. Balance Unit and Integration Tests

Use the right tool for the job:

Unit Tests (fast, mocked dependencies):

- Test business logic in isolation
- Mock external APIs, databases, slow I/O
- Run on every file save
- Great for TDD red-green-refactor cycles

Integration Tests (slower, real dependencies):

- Test actual service interactions
- Use test database, staging APIs
- Run before commits or in CI/CD
- Verify end-to-end flows work

BDD Feature Tests (behavioral, strategic mocking):

- Focus on user-facing behavior
- Mock what you don't control (Gemini, SendGrid)
- Use real implementation for your code
- Document requirements in human-readable format

The goal isn't "no mocks ever" - it's **strategic mocking** that isolates what you're testing without hiding real bugs.

The Future Is Deterministic (Finally)

As AI coding assistants like Copilot get more powerful, the gap between "what I can imagine" and "what I can build" keeps shrinking. But that's only useful if we can **verify** what we build.

Feature-driven development turns Copilot Chat from "code generator that might be right" into "implementation that is provably correct."

It's not about distrusting AI - it's about **trusting, but verifying**. Reagan was onto something.

And verification at the speed of Copilot generation with terminal feedback? That's when things get really interesting. That's when you stop gambling and start shipping.

Try It Yourself (Seriously, Do This)

Here's the absolute simplest example to get started:

```
# features/calculator.feature
Scenario: Add two numbers
When I add 2 and 3
Then the result should be 5 
```

```
# features/steps/calculator_steps.py
@when('I add {a:d} and {b:d}')
def step_impl(context, a, b):
    context.result = add(a, b) # This will fail - function
    doesn't exist yet

@then('the result should be {expected:d}')
def step_impl(context, expected):
    assert context.result == expected
```

Run the test in your VS Code terminal. Watch it fail with a clear message. Open Copilot Chat and ask it to fix the test, using #terminalLastCommand to show it the error. Copilot implements add() . Run the test again. Watch it pass.

Run the test in your VS Code terminal. Watch it fail with a clear message. Open Copilot Chat and ask it to fix the test, using #terminalLastCommand to show it the error. Copilot implements add() . Run the test again. Watch it pass.

Now scale that to your entire application. That's it. That's the whole game.

Conclusion: No More Russian Roulette Coding

"Vibe coding" with LLMs is fun, fast, and sometimes even works. But for production software - code that needs to be maintained, extended, debugged at 3am, and actually trusted - we need something better.

Test-driven development with progressive architecture gives us:

- Clear requirements (feature files that humans can read)
- Specific constraints (test assertions that don't lie)
- Verifiable correctness (tests pass = working code)
- Living documentation (tests are the spec, not some dusty wiki)
- Refactoring safety (tests catch regressions instantly)
- Progressive complexity (architecture that matches your actual journey: Basic → Intermediate → Advanced)
- Right-sized solutions (LLMs generate code at exactly the complexity level you need)

The result? **Deterministic outcomes from Copilot Chat at the right complexity level**. No more slot machines. No more crossing fingers. No more "it worked on my machine" followed by production fires. And critically - no more over-engineered messes or under-built hacks.

You document your reality (Basic), your next step (Intermediate), and your future goal (Advanced). Copilot knows which one you're working on from the diagram you show it. Your tests work across all three. You move between levels when you have real problems, not aspirational architecture.

The workflow: **Feature files → Mermaid diagrams → Terminal test feedback → Copilot Chat (with # mentions) → Repeat.**

The vibes are optional. The tests are not. The terminal feedback loop? Essential. The # mentions in Copilot Chat? That's how you give it the right context. And the progressive architecture? That's what keeps you sane.

And honestly? Once you get used to this workflow - tests that verify, diagrams that match reality, and architecture that grows with you - going back to vibe coding feels like trying to navigate with your eyes closed. Sure, you might get where you're going eventually... but why would you want to?

Interested in optimizing your development process with AI?

Whether you're exploring new approaches or looking to refine your current workflow, we're here to assist.

Reach out for a consultation or collaboration, and let's discuss how we can help drive your projects forward.

Contact us today to get started:

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

Website: www.Promptshelf.ai

Instagram: @PromptShelf.ai

X: @PromptShelfai

Reddit: www.Promptshelf.ai