ChatGPT

Executable Specifications: Flawless One-Shot Code Generation

Executable Specifications represent a paradigm shift in how we communicate requirements to Large Language Models (LLMs) for coding tasks. Instead of traditional descriptive docs or vague prompts, we provide the LLM with a **rigorous, executable blueprint** of the software. The LLM's job is then purely **translational** – converting this precise specification into target language code – rather than interpretive or creative. By eliminating ambiguity and mirroring the strictness of formal languages, this approach aims to enable LLMs to produce **flawless, one-shot code** on the first attempt.

The Philosophy: From Narrative to Blueprint

Conventional "prompt engineering" often fails because natural language is too imprecise for complex programming tasks. As one developer noted, *"Perhaps the problem with vibe-coding is trying to do it in English?"* – machines need formal, structured input for reliable output [1]. Executable Specifications embrace this insight. They treat the spec not as a prose document open to interpretation, but as a **project skeleton**: a combination of code-like definitions, tests, and unambiguous instructions. In essence, the specification itself is **as formal and checkable as code**, just one level higher in abstraction. The LLM, acting like a sophisticated compiler, **"compiles" the specification into actual code.**

By using the language of Test-Driven Development (TDD) and precise acceptance criteria, we communicate in a way that's as strict and verifiable as the code we want. Research has already shown that providing LLMs with test cases alongside problem descriptions dramatically improves success rates in code generation [2]. Executable Specifications leverage this by baking the tests and requirements directly into the prompt. The result is a deterministic blueprint that directs the LLM step-by-step, minimizing creativity and maximizing fidelity to requirements.

The Specification Suite Structure

To manage complexity and fit within LLM context windows, the overall specification is organized into a **multi-layered suite**. Each layer serves a distinct purpose, from broad philosophy down to detailed test cases:

1. **L1: Philosophy & Constraints (`constraints.md`)** – *The Rulebook.* This top layer codifies the high-level principles and global constraints of the project. It defines the architectural philosophy (e.g. emphasize TDD, favor functional style, avoid global state), any overarching constraints (max 500 lines per file, performance targets), allowed or forbidden libraries and patterns, and general style guidelines. L1 is essentially the LLM's governing constitution for the project.

2. **L2: Architecture & Data Model (`architecture.md`)** – *The Blueprint.* This section provides system-wide design overviews. It might include architecture diagrams (using Mermaid for instance),

descriptions of how components interact, and data models. All data structures, database schemas (SQL DDL), and an **exhaustive error hierarchy** (every error `enum` or code used across modules) are defined here. By listing *every* error variant and data field up front, the LLM will know exactly what types and errors to use, leaving no guesswork in data modeling.

3. **L3: Module Specifications** (e.g. `modules/message_service.md`) – *The Executable Logic.* This is the core of the methodology. Each module or feature gets its own spec document detailing **exactly what to implement** in that module. For every function or method, the spec follows a TDD cycle format (explained below) of **STUB -> RED -> GREEN -> REFACTOR**. This includes the function signature, its behavior contract, a suite of tests (unit, property-based, integration) written in the target language, and clear step-by-step implementation guidance. Essentially, L3 provides the LLM with both the *questions* and the *answers* – it outlines how the function should behave and even hints at the approach, so the LLM can proceed to coding with high confidence.

4. **L4: User Journeys (**`user_journeys.md`**)** – *The Validation Layer.* Finally, the spec suite includes end-to-end scenarios and acceptance tests from a user's perspective. These are high-level integration tests or even pseudo-code for browser/E2E tests (e.g. Playwright scripts) that demonstrate the system's features in action. They serve as a final verification that when all modules are assembled, the system meets the real-world use cases. L4 ensures the LLM doesn't implement modules in isolation but with an understanding of how everything connects to satisfy user requirements.

Each layer feeds into the next. L1 and L2 provide context and boundaries so that L3 module specs are consistent in style and design. The L3 specs then ensure that any code the LLM writes will automatically pass L4 journeys. This modular breakdown also helps keep prompts within token limits – you can supply the LLM with just the relevant slice (e.g. one module's spec plus the global constraints) when generating a specific piece of the system, rather than the entire project spec at once.

L3 Module Specifications: The TDD Cycle Methodology

Within the module specifications (L3), every function follows a standardized **TDD cycle** format. The goal is to leave no aspect of the function's required behavior or structure unspecified. The cycle is broken into four prescribed stages, each serving a specific purpose in guiding the LLM:

1. STUB (Interface Contract)

This stage defines the **function signature and its contract** in complete detail. It's essentially a copy-pasteable interface or stub that the LLM will eventually flesh out. It includes the function prototype (or method signature) with all parameters and return types, and rich documentation for each element. Crucially, it also lists any side effects, invariants, and constraints right next to the signature in comments. The idea is to fix *all* the outward-facing decisions here – naming, types, error handling, etc. – so the LLM does not have to design them.

For example, a stub in a Rust module spec might look like this, defining a message creation function with idempotency (deduplication) requirements:

```rust
pub trait MessageService: Send + Sync {
    /// Creates a message with deduplication (Critical Gap #1).
    ///
    /// **Side Effects:**
    /// 1. Inserts a new row into the `messages` table.
    /// 2. Updates `room.last_message_at` timestamp.
    /// 3. Broadcasts a `NewMessage` event to room participants.
    ///
    /// **Invariant:**
    /// - `content` must be 1-10000 characters of sanitized HTML (no scripts).
    /// - `client_message_id` is a UUID used for idempotency; calling this twice
with the same value should not create duplicate messages.
    async fn create_message_with_deduplication(
        &self,
        content: String,           // Invariant: length 1-10000 chars, already
sanitized
        room_id: RoomId,
        creator_id: UserId,
        client_message_id: Uuid,   // Used for deduplication (idempotency key)
    ) -> Result<Message<Persisted>, MessageError>;
}
```

In this stub, nothing is left implicit. The trait method's purpose is clear ("Creates a message with deduplication"), and it enumerates exactly what side effects it must have. All input constraints are described (content length, format, etc.), and the idempotency behavior is hinted by the `client_message_id` parameter. By laying out the interface and expectations, we ensure the LLM doesn't, for example, introduce additional parameters, pick different naming, or forget a side effect. **The stub is the contract**: the LLM's implementation must conform to it exactly.

(Note: This approach builds on the initial design outlines you may have in earlier docs or in `tasks.md`. If an earlier phase defined some function signatures or invariants, those are directly used in the stub to maintain consistency.)

2. RED (Behavioral Specification – Failing Tests)

In TDD, one writes tests before the code. The **RED** stage provides a suite of failing test cases that define the function's expected behavior under various conditions. Instead of a vague description like "should handle duplicates correctly," we give the LLM actual **executable test code** that it must make pass. These tests act as a concrete definition of "Done" for the function – if the LLM's generated code passes all these tests (turning them green), we consider the behavior correct.

There are typically multiple layers of tests included:

- **Unit Tests:** Cover each important scenario, edge case, and error condition in isolation. For our example, unit tests would include cases like creating a message normally, attempting to create with

a duplicate `client_message_id`, content validation errors, and unauthorized user attempts. Each test spells out the expected result or error.

- **Property-Based Tests (using `proptest` or similar):** These define more abstract invariants or properties that should always hold, using random data. For instance, a property test might verify that calling `create_message_with_deduplication` twice with the same IDs never produces two distinct messages (ensuring true idempotency across arbitrary inputs). This helps catch any corner cases not covered by manual examples.

- **Integration Tests (or stubs thereof):** If the function interacts with external systems (database, message bus, etc.), the spec can include integration test setups or at least the outline of them. This might involve using in-memory databases or mock services to assert that, say, the database was actually updated or an event was actually broadcast when the function is called. These tests ensure the function works in the context of the larger system boundaries.

Below is a sample of how a **unit test** might be presented for the deduplication logic, followed by a simple property test stub:

```
// RED: Unit Test for deduplication idempotency
#[tokio::test]
async fn test_dedup_returns_existing_message_and_preserves_content() {
    let fixture = setup_test_fixture().await;
    let client_id = Uuid::new_v4();

    // First call creates a new message
    let msg1 = fixture.service
        .create_message_with_deduplication("Original content", room_id,
user_id, client_id)
        .await
        .expect("First call should succeed");

    // Second call with *same* client_id but different content tries to create
duplicate
    let msg2 = fixture.service
        .create_message_with_deduplication("Different content", room_id,
user_id, client_id)
        .await
        .expect("Second call with duplicate client_id should also succeed");

    // The second call should not create a new message, but return the existing
one
    assert_eq!(msg1.id, msg2.id, "Both calls should return the same Message (no
duplicate created)");
    assert_eq!(msg2.content, "Original content", "Content should remain as in
the first call, ignoring subsequent different content");
```

```
    // Verify DB state: there should only be one message in the DB for that
client_message_id (not shown here)
    // ...
}

// RED: Property Test Invariant (simplified example)
proptest! {
    #[test]
    fn prop_deduplication_is_idempotent(unique_content in any::<String>()) {
        let fixture = setup_test_fixture();
        let client_id = Uuid::new_v4();
        let _ =
fixture.service.create_message_with_deduplication(unique_content.clone(),
room_id, user_id, client_id);
        let result2 =
fixture.service.create_message_with_deduplication("different".to_string(),
room_id, user_id, client_id);
        prop_assert!(result2.is_ok());
        prop_assert_eq!(result2.unwrap().content, unique_content);
    }
}
```

From these tests, the LLM can **deduce the exact intended behavior**: the function should create a new message on first call, return the existing message on second call with the same ID, ignore any different content on duplicates, and never insert two rows for the same `client_message_id`. We aren't just saying "it should be idempotent" – we're demonstrating it with an executable example. Likewise, you would include tests for validation (e.g. content too long returns a `MessageError::Validation`) or authorization (wrong user returns `MessageError::Unauthorized`), each illustrating the expected error variant and no side-effects in those cases.

Providing these tests has a dual benefit: **(a)** The LLM can use them as guidance while writing code (it "knows" what output will make the tests pass), and **(b)** once code is generated, we can run the same tests to verify correctness. This aligns with findings that giving LLMs test cases improves code accuracy [2] – essentially, the tests become a part of the spec.

3. GREEN (Implementation Guidance & Logic)

The GREEN stage is where we guide the LLM on how to implement the function to make the RED tests pass. Rather than leaving the LLM to invent the algorithm from scratch, we provide an **outline or hints of the solution strategy**. However, to avoid ambiguity, we prefer structured formats over prose pseudocode. Two powerful techniques used here are **Decision Tables** and **Algorithmic Step Lists**, supplemented by occasional diagrams for clarity.

- **Decision Tables:** For complex conditional logic, a decision table lays out all possible combinations of key conditions and the required outcomes. This acts like a truth table or flowchart in text form. It's unambiguous and exhaustive. The LLM can essentially translate each row into an `if/else` branch or pattern match in code.

For example, for `create_message_with_deduplication`, the spec might include a decision table like this:

Decision Table – `create_message_with_deduplication` outcomes:

Case	`client_message_id` already exists?	User authorized?	Content valid?	Outcome (Result)	Side Effects
C1	Yes	*Don't care*	*Don't care*	**Return** `Ok(existing_message)` (found by dedup key)	*No new side effects* (no DB write)
C2	No	**No**	*N/A*	**Return** `Err(MessageError::Unauthorized)`	*None* (reject before any write)
C3	No	Yes	**No**	**Return** `Err(MessageError::Validation)`	*None* (invalid input, no write)
C4	No	Yes	Yes	**INSERT** new message, **Return** `Ok(new_message)`	**Yes** (DB write, update room timestamp, broadcast event)

This table enumerates every relevant condition combination. For instance, C1 captures the idempotency behavior (if the client ID exists, short-circuit and return the existing message, doing nothing new). C2 and C3 cover authorization and validation failures (ensuring those checks happen *before* any insert). C4 is the normal success path. An LLM given this table can translate it almost directly into code with clear if/else or match logic. The precision here prevents mistakes like forgetting an auth check or performing actions in the wrong order.

- **Algorithmic Steps:** For sequential logic or algorithms, the spec can list steps in order. This is like high-level pseudocode but in numbered form. It might say, for example: *"1. Validate the* `content` *length and format; if invalid, return Validation error. 2. Check user's permission to post in the room; if not allowed, return Unauthorized. 3. Attempt to insert the new message into the database via a* `WriteCommand` *to the DB handler. 4. If insertion fails due to a duplicate key, retrieve the existing message (idempotent result) and return it; otherwise, on success, broadcast the event and update the room timestamp, then return the new message."* Each step aligns with parts of the decision table. This gives the LLM a narrative of the flow without leaving the low-level details to chance.

- **Visualizations:** When appropriate, the spec might include diagrams (written in Mermaid or other text-based formats) to illustrate flows, especially for multi-component interactions or async processes. For example, if there's a complex sequence of events (user sends message -> service -> database -> websocket broadcast to clients), a Mermaid sequence diagram could show the timeline. While an LLM might not literally "execute" a diagram, including it reinforces the intended sequence and can help the developer (or anyone reading the spec) verify completeness.

By the end of the GREEN stage, the LLM is equipped with a *clear plan* for implementation. At this point in the spec, there should be zero ambiguity about what the code should do. The model doesn't need to be clever; it just needs to faithfully **write code that follows the provided blueprint** and makes all the RED tests pass.

4. REFACTOR (Constraints, Patterns, and Imperfections)

The final stage acknowledges that real-world software has non-functional requirements and sometimes intentional imperfections for simplicity. Here we specify things that go beyond the basic logic: performance considerations, coding patterns to use or avoid, and acceptable deviations from ideal behavior.

- **Non-functional Constraints & Optimizations:** If certain performance optimizations are expected, we note them. For instance, *"This function will be called frequently, so ensure to use a DB index on* `(room_id, client_message_id)` *for the deduplication check."* While the LLM might not handle index creation, the code it generates (like a SQL query or ORM usage) can be influenced by knowing about indexes or expected Big-O behavior. We also list any global constraints that apply (like memory limits or concurrency considerations) as they pertain to this function.

- **Forbidden Patterns (Anti-Patterns):** It's crucial to call out pitfalls to avoid. LLMs might inadvertently use common but wrong approaches unless explicitly told not to. In our running example, we *forbid* doing a separate SELECT query to check for duplicates before insertion, because that can introduce a race condition (TOCTOU bug). Instead, the spec insists on relying on the database's unique constraint to catch duplicates atomically. By stating **"DO NOT perform a pre-check query for client_message_id; always attempt the insert and handle a duplicate key error"**, we prevent the LLM from implementing a known anti-pattern. Other forbidden list items could include things like not using certain insecure functions, avoiding recursion due to stack depth, etc., depending on project constraints.

- **Defining Acceptable Imperfections:** Interestingly, sometimes we *allow* a controlled level of imperfection to avoid over-engineering. The spec should define these explicitly so the LLM doesn't attempt to "fix" something that we consider an acceptable limitation. The prompt snippet refers to "Rails-Equivalent Imperfection," meaning behavior that is slightly sloppy but tolerable (in the way Ruby on Rails might choose convention over perfect accuracy for the sake of simplicity). In our example, presence tracking had a 60-second heartbeat delay – so a user might appear online for up to 60 seconds after disconnecting. We accept this inaccuracy to avoid a complex consensus system for real-time presence.

To illustrate, the spec might include a **test that encodes this expected imperfection**, showing that the system is allowed to be a little wrong within certain bounds:

```rust
// Rails-Equivalent Imperfection: Presence tracking delay tolerance (Limitation
#4)
// We expect a disconnected user to be marked offline within 60 seconds.

#[tokio::test]
async fn test_presence_cleanup_occurs_within_ttl_window() {
    let tracker = PresenceTracker::new();
    tracker.set_online(UserId(1));
    tracker.disconnect(UserId(1));

    // Immediately after disconnect, user might still be reported as present.
    assert!(tracker.is_present(UserId(1)), "User should still be present
immediately after disconnect");

    // After the 60s TTL window (with some tolerance), the user should no longer
be present.
    tokio::time::sleep(Duration::from_secs(65)).await;
    assert!(!tracker.is_present(UserId(1)),
"User should be absent after the TTL window expires");
}
```

By providing such a test, we signal to the LLM that it *should not* try to make the presence system perfectly real-time – a slight delay is expected and acceptable, but it must eventually resolve within the given window. This level of detail prevents the AI from optimising or altering things that it shouldn't, and it documents the intentional trade-off for human readers.

After REFACTOR, the specification for the function is essentially complete. We have described what to do (Stub), how to verify it (Red tests), how to do it (Green guidance), and any tweaks or cautions (Refactor). Following this format for every function in a module yields a comprehensive module spec. An LLM that ingests this spec should be able to produce the implementation **straight through**, passing all tests, respecting all constraints, and avoiding forbidden approaches. The code, in theory, will be correct by construction.

Verification and Definition of Done

Having the LLM generate code from these specs is only part of the story. We also need to **verify** that the result truly meets our standards. The specification suite therefore includes a **Verification Harness** (sometimes documented in a `verification.md` or CI configuration notes) which defines the criteria for success. Essentially, this is the *"Definition of Done"* for the entire project, and it typically includes:

1. **Static Analysis Cleanliness:** The generated code must pass format and lint checks. For example, in a Rust project, we would require `cargo fmt --check` and `cargo clippy -- -D warnings` to run without errors or warnings. This ensures the code adheres to style guidelines and has no obvious code smells or lint violations out of the gate.

2. **Unit and Property Tests (L3 TDD Harness) Pass:** All the tests provided in the module specs (unit tests and property-based tests written under the `tests` or module sections) must pass 100%. This is the immediate measure of functional correctness for each module.

3. **Integration Tests Pass:** Any integration tests (broader tests that might span multiple modules or test interactions with external components) should also pass. These might be in a separate `tests` directory or marked differently in the project (for example, Rust integration tests vs. unit tests). A command like `cargo test --test integration` would be expected to succeed with 100% pass rate.

4. **End-to-End (E2E) Tests Pass:** If user journey tests or end-to-end scenarios are automated (e.g. using Playwright, Selenium, or API contract tests), those should all pass when the system is run. For instance, `npm run test:e2e` might execute a suite of browser tests that simulate a user using the app – all features demonstrated in `user_journeys.md` should work as specified. Passing these means the system likely meets the real-world requirements.

5. **Manual Constraint Review:** Finally, a human (or an additional automated checker if available) reviews the code against the L1 constraints and architecture expectations. This includes verifying things that are hard to test automatically – e.g., ensuring no file exceeds the 500-line limit, ensuring that no disallowed functions or libraries were used (if not caught by lint), and that the overall design boundaries were respected. It might also involve looking at performance metrics or simple load-testing to ensure non-functional requirements are met.

Only when **all** of the above are satisfied do we consider the code "flawless" and the spec fully realized. Essentially, if the LLM's output code passes every test and check without modifications, we have achieved the one-shot goal. The specification is so complete that it has driven the creation of a working system with zero iterations of bug-fixing. Any failure in these verification steps would indicate a gap in the spec (or a rare lapse by the LLM), meaning we'd refine the spec and regenerate rather than debugging the code manually.

Benefits and Challenges of this Approach

Crafting Executable Specifications is an intensive process, but it carries significant benefits:

- **Ambiguity Elimination:** The primary benefit is removing guesswork. The LLM doesn't have to "figure out" what the developer meant – it's told explicitly. This leads to far higher success rates on first try. Including tests as part of the prompt has been shown to increase solution correctness in studies [2]. By extension, a full specification of interfaces and logic should let the LLM achieve near 100% accuracy if it follows the plan.

- **One-Shot Correctness:** When done right, this approach can save the considerable time normally spent in the edit-compile-run-debug loop. Instead of the LLM producing code that we then manually test and fix repeatedly, we aim for it to produce code that passes all tests immediately. This is essentially shifting the effort left – we invest more in writing the spec and tests up front, so that little to no time is spent debugging later. It's akin to extreme TDD: we handle all the "think work" in spec form, and the coding is just mechanical translation.

- **Specification as Documentation:** The resulting spec suite serves as excellent project documentation. Future maintainers (whether human or AI) can read the specs to understand *why* the code does what it does, not just how. The tests and decision tables explain the intended behavior. This is something traditional code generation often lacks – here the documentation is built-in and executable.

- **Separation of Concerns – Design vs. Implementation:** Humans define the *what* and *why* in a very structured way, and the AI handles the *how* at the code level. This can leverage the respective strengths of each: humans excel at understanding requirements and spotting edge cases, while AIs are tireless in writing boilerplate, following patterns, and even performing complex transformations as long as the instructions are clear. It's a bit like having a junior programmer who never deviates from the senior architect's outline. The architect (spec writer) ensures the solution is correct; the AI handles syntax and repetitive details.

- **Cross-Language Potential:** In theory, the same high-level spec could be used to generate implementations in multiple languages (by altering the code examples/tests accordingly). For instance, one could maintain a language-agnostic decision table and English descriptions, but provide test cases in Rust for one project and in TypeScript for another. The core logic is consistent, reducing discrepancies between different implementations of the same service.

However, this methodology also comes with challenges and costs:

- **Spec Authoring Effort:** Writing a truly exhaustive spec is a **significant upfront investment**. It requires considerable skill and foresight – essentially, the spec writer must think of every edge case, every error condition, every needed test. This can be as difficult as writing the code itself, if not more so in some cases. In scenarios where requirements are unclear or evolving, drafting such a detailed spec could be impractical. It demands a shift in mindset to spending more time in design and testing on paper before any code exists.

- **Thoroughness is Critical:** If the spec misses a scenario, the LLM might produce code that fails in that case without anyone knowing. Or if the tests are not comprehensive, the LLM might pass all given tests but still have a bug. In the Medium experiment "LLM + TDD = NBG," an incomplete test suite led the LLM to produce a parser that missed certain token types and had subtle bugs [3]. This underscores that **the spec is only as good as the thoroughness of its author**. Executable Specifications mitigate this by encouraging property tests and exhaustive enumerations of behavior, but humans must still anticipate the tricky cases.

- **Context Window Limitations:** While we modularize the spec to fit into context windows, very large projects might still strain the limits of an LLM's memory. You may have to generate code in pieces and ensure the pieces integrate. If an LLM can't see the whole picture at once, there's a risk of integration issues (though L4 tests can catch inconsistencies). Advances in LLM context size will alleviate this, but it's a factor to consider. Careful ordering of generation and perhaps iterative refinement on a module-by-module basis might be needed for truly massive codebases.

- **Over-constraining vs. Under-constraining:** Striking the right balance in spec detail is tricky. If you over-specify (giving every tiny implementation detail), you might as well write the code yourself. If you under-specify, the LLM might make wrong choices. The goal is to specify the *what* and *why*

comprehensively, without necessarily dictating trivial *how* (e.g., you wouldn't need to tell the LLM to use a `for` loop vs a `while` loop unless it matters). In practice, the provided tests and performance constraints should guide those low-level choices naturally.

- **Maintenance of Specs and Code:** Once the code is generated and in use, any change in requirements means the spec and tests need updating (as they would in any TDD process). One has to decide whether to regenerate code via the LLM or hand-edit it. Ideally, one would go back to the spec, adjust the desired behavior or add a new test, and then have the LLM regenerate or modify the code accordingly. This could introduce its own form of tech debt: the spec has to remain the single source of truth. If a quick fix is made in code without updating the spec/tests, the whole paradigm breaks down. Therefore, discipline is required to treat the spec suite as a living artifact alongside the code.

- **Limitations of LLM reasoning:** While LLMs are powerful, they might still occasionally misinterpret a complex instruction or produce imperfect code for a tricky algorithm (especially if the spec has any gaps). Ideally, the tests catch these. In a worst-case scenario, if the LLM output fails some tests, a developer might need to step in to either refine the prompt or fix the code. That said, with each spec refinement, the process gets closer to one-shot success. It's an iterative loop of improving the spec rather than debugging code – which keeps us in a high-level problem-solving mode rather than low-level fix mode.

In summary, **Executable Specifications** present an exciting approach to AI-assisted development. They essentially treat the LLM as an obedient implementer of a design, rather than a partner to brainstorm with. By being extremely explicit about what we want (much like writing a formal requirements document that doubles as a test suite), we harness the LLM's strengths (speed, consistency) and avoid its weaknesses (tendency to hallucinate or make inconsistent decisions). It's a methodology that demands rigorous thought up front – very much in the spirit of TDD and formal methods – but promises a big payoff: code that works correctly on the first try, with minimal human correction.

My perspective: This approach has the potential to **redefine the developer's role** in AI-assisted coding. Instead of writing and debugging the final code, a developer using Executable Specs spends time writing the blueprint and the tests, essentially doing the "thinking work" and letting the AI handle the "typing work." It's akin to writing assembly instructions for furniture rather than building it yourself. When the instructions are crystal clear, the furniture (code) comes out just as envisioned. Given the evidence that tests improve LLM output and considering how this method addresses many pitfalls identified in naive "prompt the AI with a vague spec" attempts, I believe this is a promising direction. It does require a high level of diligence – not every team or project will be ready to invest in such thorough specs – but for safety-critical or very complex systems, it could drastically reduce bugs and integration time. It's also a great way to document and enforce best practices (since the spec explicitly lists them).

In a way, we're coming full circle to the idea that **good software is built on good specifications**. We're just now leveraging AI as a tool to execute those specifications. If one-shot flawless code generation is the goal, then turning specs into something *executable* (tests and formalized rules) is not just ideal, it's arguably necessary. Natural language alone won't cut it for programming; we need the precision of executable specs to speak effectively with our new AI coding assistants. With this methodology, we set the stage for LLMs to truly become reliable coders, and we as developers become architects and test writers – the curators of requirements and quality.

Overall, I think this is a **powerful idea** that, while challenging to implement fully, addresses the core issues that have made AI-generated code unreliable in the past. It's an exciting blend of classic software engineering wisdom (TDD, design by contract, clear specs) with cutting-edge AI capabilities, and it could very well be a blueprint for how software will be developed in the future.

---

1 3 LLM + TDD = NBG. What happens when you ask an LLM to... | by Jules May | Aug, 2025 | Medium
https://julesmay.medium.com/llm-tdd-nbg-449bf9ab12e9

2 [2402.13521] Test-Driven Development for Code Generation
https://arxiv.org/abs/2402.13521