Agentic Programming -

A Compiler Case Study

Driving Effective Agentic Programming

Jian Weng

CEMSE, KAUST

Week-1 Session-2

I forget one thing last time

- But I did not want to do it last time, as some of you may not officially enroll the class yet...
- I also found it is super hard to interact with you without knowing you...
- Self-introduction around all of us:
  - Name
  - Background
  - Experience with AI coding tools
  - Expectation from this class

Recap from Last Session

- This is **NOT** a class for vibe coding
- AI is giving you a **team of you**
  - AI is only as good as you are
  - But you save the most bandwidth on communication
- Two key pains of agentic programming:
  1. Deep **human involvement** still needed
  2. How to manage a large codebase **effectively and reliably**?

Today's Focus

- Pain #1 (human involvement) — we will address this later
- To reduce human involvement, we first need to solve:
  - **How to make AI write high-quality code?**
- Once the code quality is reliable, then we can reduce human oversight

Agenda

- **The problem with AI-generated code**
- SDD: Spec/Standard-Driven Development
- DDD: Document-Driven Development
- TDD: Test-Driven Development
- The complete workflow
- Wrap-up

The Problem: Before AI Agents

Each feature request used to take you a whole day:

- **Implement** — half the time
- **Test** — half the time (when buggy); zero (when it works)
- **Document** — "Why do I even do this? I remember everything..."

The goal: **move on faster**

But moving on faster ≠ skipping quality

The Problem: With AI Agents

AI writes code for you, but:

- Maybe the code is **buggy**?
    - → Test it! Tests don't make it 100% correct, but better than nothing
- You didn't write it, you didn't even read it! How do you **understand** it?
    - → Documentation!
    - To my observations: documentation significantly helps AI understand
- Without tests and docs, you're flying blind

Proposed Solution: Three Pillars

- **SDD** — Spec/Standard-Driven Development
- **DDD** — Document-Driven Development
- **TDD** — Test-Driven Development

The standard: **Write docs first, write tests second, write code last**

Agenda

- The problem with AI-generated code
- **SDD: Spec/Standard-Driven Development**
- DDD: Document-Driven Development
- TDD: Test-Driven Development
- The complete workflow
- Wrap-up

SDD: The Spec is the Charter

A **spec** defines how development must proceed:

- Coding standard
- Documentation standard
- Testing standard

The spec enforces:

1. AI reads the **spec of the development flow**
2. AI reads the **spec of the feature request**
3. AI makes a **plan**
4. AI **executes** the plan following the spec

SDD: What Does the Standard Say?

The development standard we adopt:

1. **Document first** — update docs before writing any code
2. **Test second** — write test cases based on the documented interface
3. **Code last** — implement the code to pass the tests

This is the order. Not the other way around.

SDD: Where Does the Spec Live?

Remember `CLAUDE.md` from last session?

`CLAUDE.md` / `AGENTS.md` — only the basics:

- How to **build** the project
- How to **run tests**
- Project overview (what this repo is)

Planning & execution standards live elsewhere:

- `commands/` — encode the development workflow as slash commands
- `rules/` — enforce coding, doc, and testing standards per file

> `CLAUDE.md` = context. Commands & Rules = standards.

SDD: Not Just Claude Code

This layered system is **universal** across AI coding agents:

Claude Code	Codex	Cursor
`CLAUDE.md`	`AGENTS.md`	`AGENTS.md`
`.claude/commands/`	—	—
`.claude/rules/`	—	`.cursor/rules/`

- The names differ, but the **concept is the same**
- We use Claude Code as our example — it was the first AI coding CLI
    - It defined most of these standards
    - Other tools adopted similar patterns

SDD: Beyond CLAUDE.md — Rules, Commands, Skills

Claude Code provides a layered system for encoding standards:

```
.claude/
├── CLAUDE.md                  ← Project-level context & spec
├── rules/                     ← Modular, path-specific instructions
│   ├── code-style.md
│   ├── testing.md
│   └── frontend.md
├── commands/                  ← Custom slash commands (legacy)
│   └── review.md
└── skills/                    ← Modern extensible actions
    └── implement-feature/
        └── SKILL.md
```

SDD: Anatomy of Rules, Commands, and Skills

All of them share the same two-part structure:

```
---
  - description: When does this apply?
  - argument-hint: What arguments does it take?
  - tools: What tools can it use?

---
(prompt: natural language instructions)
  - What should the AI actually do?
  - Step-by-step workflow
```

Front matter = machine-readable config (scope, triggers, options)

Prompt = human-readable instructions (the actual standard)

SDD: Rules — Path-Specific Standards

Rules are markdown files in `.claude/rules/` that apply **conditionally**:

```
# .claude/rules/testing.md
---
paths:
  - "src/**/*.cc"
  - "src/**/*.h"
---

# Testing Rules
- Every public function must have a corresponding test
- Test files are named <module>_test.cc
- Use GoogleTest framework
- Run tests with: make test
```

Rules with `paths` only activate when AI touches matching files.

SDD: Commands — Reusable Workflows

Commands are single-file slash commands in `.claude/commands/`:

```
# .claude/commands/implement.md

Read the feature request in $ARGUMENTS.
Follow this workflow:
1. Read docs/ to understand the current design
2. Update the relevant documentation first
3. Write test stubs that compile but fail
4. Implement the code to pass all tests
5. Run `make test` to verify
```

Usage: `/implement "add string literal support to lexer"`

> Commands encode your **development workflow** as a reusable action

SDD: Skills — The Modern Standard

Skills are directories in `.claude/skills/` with richer capabilities:

```
# .claude/skills/document-module/SKILL.md
---
name: document-module
description: Generate module documentation
argument-hint: "[module path]"
---

For the module at $ARGUMENTS, generate a README.md with:
1. **Purpose** — what this module does
2. **Public Interface** — exported functions and classes
3. **Internal Helpers** — private functions
4. **Data Structures** — types, enums, structs
```

Usage: Shoulbe be automatic when determined to be needed.

SDD: Commands vs Skills

Think of them in C terms:

- **Command** = a **function call**
  - You invoke it explicitly: `/implement "add string literals"`
  - You pass arguments in, it runs a workflow
  - It is an **entry point** you control
- **Skill** = a **macro**
  - It expands automatically when the AI determines it's needed
  - Skills **cannot call commands** — by design
  - This keeps skills framework-agnostic and composable

SDD: The Context Cost

Everything has a cost — **descriptions are loaded at session start**:

- Every `CLAUDE.md`, every rule, every skill `description`
  → fed into the context **before you even type**
- This is your **startup overhead**

Implications:

- Write descriptions **concisely** — every token counts
- Too many verbose rules = bloated context = less room for actual work
- Think of it as: your spec is **always in memory**

> Good specs are short and precise. Bad specs waste your context window.

Anti-pattern: Vague Spec

What happens when the spec is too vague?

```
# Bad CLAUDE.md
- Write good code
- Add tests when needed
- Document important things
```

AI interprets "good", "when needed", "important" **differently every time**

Result: inconsistent code, missing tests, sparse docs

> Be specific. The spec is a **contract**, not a suggestion.

Agenda

- The problem with AI-generated code
- SDD: Spec/Standard-Driven Development
- **DDD: Document-Driven Development**
- TDD: Test-Driven Development
- The complete workflow
- Wrap-up

DDD: Document as a Source Tree

In the AI era, documentation is not an afterthought.

Two levels of documentation:

- `docs/` — **architecture-level**: project overview, module relationships, design decisions
- Next to each source file — **file-level**: interfaces, helpers, data structures

Each `.cc` / `.h` file has a companion `.md` **right beside it**.

DDD: Example — Doc Source Tree

```
project/
├── docs/
│   └── architecture.md      ← Project architecture & module relationships
├── src/
│   ├── lexer/
│   │   ├── lexer.h
│   │   ├── lexer.cc
│   │   └── lexer.md         ← Public API, helpers, data structures
│   ├── parser/
│   │   ├── parser.h
│   │   ├── parser.cc
│   │   └── parser.md        ← Public API, helpers, data structures
│   └── codegen/
│       ├── codegen.h
│       ├── codegen.cc
│       └── codegen.md        ← Public API, helpers, data structures
│
```

`docs/` = the big picture. `src/*.md` = the details, right next to the code.

DDD: What Goes in a Module Doc?

```markdown
# Lexer Module

## Purpose
Tokenizes source code into a stream of tokens.

## Public Interface
- `Lexer(std::string source)` — constructor
- `Token nextToken()` — returns next token, advances cursor
- `std::vector<Token> tokenize()` — tokenizes entire source

## Internal Helpers
- `skipWhitespace()` — advances past whitespace
- `readIdentifier()` — reads an identifier token
- `readNumber()` — reads a numeric literal

## Data Structures
- `Token { TokenType type; std::string value; int line; }`
- `enum TokenType { IDENT, NUMBER, PLUS, ... }`
```

DDD: Who Writes the Docs?

- AI can help you write them
- You can write them manually
- Either way, **docs must exist before code changes**

Why document first?

- Updating the doc = **designing the interface**
- Like C's separation of `.h` (header) and `.c` (implementation)
- Once the interface is documented, you can write stubs and tests

> Docs save AI's chain-of-thought when understanding your code later

DDD: The Workflow

When a new feature request comes in:

1. **Update the doc** — add/modify the interface description
2. Now you have a clear picture of:
   - What functions exist
   - What parameters they take
   - What they return
3. The doc becomes the **blueprint** for stubs and tests

> This is not extra work — this IS the design phase

Anti-pattern: Code First, Doc Later

What actually happens when you say "I'll document later":

1. Write code → ship it → move on
2. Next feature comes in → "I'll read the code to understand"
3. AI reads 2000 lines → gets confused → hallucinates
4. You debug AI's hallucination → more time wasted than documenting

With AI agents, the cost of missing docs is amplified

- You don't just slow down yourself
- You slow down every future AI interaction with that code

Agenda

- The problem with AI-generated code
- SDD: Spec/Standard-Driven Development
- DDD: Document-Driven Development
- **TDD: Test-Driven Development**
- The complete workflow
- Wrap-up

Traditional TDD vs Agent TDD

Traditional TDD (human writes everything):

1. Human writes a failing test
2. Human writes code to pass the test
3. Human refactors
4. Repeat

Agent TDD (human designs, AI implements):

1. Human designs the interface (via docs)
2. Human (or AI) writes stubs + failing tests
3. **AI implements code to pass the tests**

Why Agent TDD is More Powerful

Traditional TDD:

- Human must context-switch between test-writing and implementation
- Temptation to skip tests when "the code obviously works"

Agent TDD:

- Clear separation of concerns
    - **You**: design interface, write tests (the spec of correctness)
    - **AI**: implement until tests pass
- Tests are **not optional** — they are the AI's success criterion
- No more "it obviously works" — prove it

TDD: Step by Step

After docs define the interfaces:

1. Write **empty function stubs** that match the documented interface
2. Write **test cases** that call these stubs
3. Tests **compile** — but they **fail** (stubs return wrong/no values)
4. Now tell AI: **make the tests pass**

```cpp
// stub — compiles, but tests will fail
Token Lexer::nextToken() {
    return Token{TokenType::UNKNOWN, "", 0}; // TODO
}
```

TDD: Example — Test Cases

```
TEST(LexerTest, SingleNumber) {
    Lexer lexer("42");
    Token tok = lexer.nextToken();
    EXPECT_EQ(tok.type, TokenType::NUMBER);
    EXPECT_EQ(tok.value, "42");
}

TEST(LexerTest, SimplePlus) {
    Lexer lexer("1 + 2");
    auto tokens = lexer.tokenize();
    ASSERT_EQ(tokens.size(), 3);
    EXPECT_EQ(tokens[0].type, TokenType::NUMBER);
    EXPECT_EQ(tokens[1].type, TokenType::PLUS);
    EXPECT_EQ(tokens[2].type, TokenType::NUMBER);
}
```

TDD: The Prompt to AI

Now you tell the agent:

> Run `make test`. Some tests are failing.
> Read the documentation in `docs/lexer/README.md` to understand the expected behavior.
> Implement the code in `src/lexer/lexer.cc` to make all tests pass.

The agent has:

- A **clear goal** (tests pass)
- A **clear reference** (documentation)
- A **clear scope** (specific files)

TDD: Leveraging Agent Persistence

A key insight about AI agents:

- All agents are **persistent** — they keep iterating toward a goal
- If the goal is not met, they will try again and again
- We **leverage** this property:
  - Give the agent a **clear, verifiable target** (all tests pass)
  - Let it grind until it succeeds

> The agent won't complain. It won't get tired.
>
> It will keep fixing bugs until the tests are green.

Anti-pattern: No Tests, Just "Looks Right"

Without tests, the AI agent has no feedback loop:

1. AI writes code
2. You read it → "looks right" → ship it
3. Bug surfaces in production → back to square one

Or worse:

1. AI writes code
2. You don't read it → ship it
3. Bug surfaces → you don't even know where to look

> Tests are the **minimum viable verification**.

Anti-pattern: Writing Tests After Code

Why not write code first, then add tests?

- AI writes code → it "works" → you write tests that match the code
- **Circular reasoning**: tests verify what the code does, not what it should do
- You end up testing the implementation, not the specification

The correct order:

1. Spec defines **what should happen**
2. Tests encode **what should happen**
3. Code makes it happen

> Tests written after code are just a rubber stamp.

Agenda

- The problem with AI-generated code
- SDD: Spec/Standard-Driven Development
- DDD: Document-Driven Development
- TDD: Test-Driven Development
- **The complete workflow**
- Wrap-up

The Complete Workflow

```
Feature
Request
   ↓
Update Docs      ← Design the interface
   ↓
Write Stubs      ← Empty functions, correct signatures
   ↓
Write Tests      ← Tests compile, but fail
   ↓
AI Implements    ← Agent iterates until tests pass
   ↓
All Pass
```

Walkthrough: Adding a Feature

Let's say we want to add **string literal support** to our lexer.

Step 1: Update docs

```
## Public Interface (updated)
- `Token nextToken()` - now also handles string literals

## Data Structures (updated)
- `enum TokenType { ..., STRING, ... }`

## Behavior
- String literals are enclosed in double quotes: "hello"
- Escape sequences supported: \n, \t, \\, \"
```

Walkthrough: Stubs and Tests

Step 2: Write stub

```
// lexer.cc — string case added, returns UNKNOWN for now
case '"':
    return Token{TokenType::UNKNOWN, "", line_};
```

Step 3: Write tests

```
TEST(LexerTest, StringLiteral) {
    Lexer lexer("\"hello\"");
    Token tok = lexer.nextToken();
    EXPECT_EQ(tok.type, TokenType::STRING);
    EXPECT_EQ(tok.value, "hello");
}

TEST(LexerTest, StringEscape) {
```

Walkthrough: AI Implements

Step 4: Prompt the agent

> The string literal tests in `lexer_test.cc` are failing.
> Read `docs/lexer/README.md` for the expected behavior.
> Implement string literal tokenization in `lexer.cc`.
> Run `make test` to verify.

What happens:

1. Agent reads the doc → understands escape sequences
2. Agent implements `readString()` helper
3. Runs tests → some fail (forgot `\"` escape)
4. Fixes → runs again → all pass

Why This Works

- **Docs** give the agent context without guessing
- **Stubs** define the exact interface to implement
- **Tests** provide a clear, automated success criterion
- **Agent persistence** means it will keep trying until it works
- **You** stay in control of design, but delegate implementation

Putting It All Together: CLAUDE.md

Your `CLAUDE.md` ties everything together:

```
# My Compiler Project

## Build & Test
- Build: make
- Test: make test

## Development Standard
1. Update docs in docs/ BEFORE writing code
2. Write test cases that compile but fail
3. Implement code to make tests pass

## Documentation Standard
- Each module has docs/<module>/README.md
- Document: purpose, public API, helpers, data structures
```

Wrap-up

- High-quality AI code requires structure: **SDD + DDD + TDD**
- Document first → Test second → Code last
- Documentation mirrors your source tree
- Tests give agents a clear, verifiable goal
- Agents are persistent — leverage that property
- `CLAUDE.md` encodes both context and standards

Next session

- We will put this into practice
- Setting up the spec and documentation for our compiler project