Spec-kit

The Future of Spec-Driven Development

Agenda

1. The Current State of AI Coding
2. The Problems with Current Methods
3. What is Spec-driven Development?
4. What is Spec-kit
5. Anatomy of a Spec-kit
6. The Workflow
7. Comparison with Other Methods
8. Benefits & Challenges
9. Future Outlook

The Current State of AI Coding

- **Chat-based (ChatGPT, Claude):**
  - Conversational, good for exploration.
  - "Copy-paste" workflow.
- **Autocomplete (Copilot, Codeium):**
  - "Ghost text" in your editor.
  - Fast, tactical, line-by-line assistance.
- **Agentic (Codex, Github Copilot, Claude Code, Gemini CLI):**
  - Autonomous, but often struggles with complex, unguided tasks.

The Problem: Context Drift

- **The "Goldfish Memory" Effect:**
  - Long chat threads lose context.
  - AI forgets the database schema you mentioned 20 messages ago.
- **Fragmented Knowledge:**
  - Your IDE knows your files, but the Chatbot might not.
  - "Hallucinations" occur when context is missing.

The Problem: Prompt Engineering Fatigue

- **The "Blank Page" Syndrome:**
  - Staring at a cursor, trying to craft the *perfect* prompt.
- **Inconsistency:**
  - Asking the same question twice yields different code styles.
- **Lack of Standards:**
  - Hard to enforce team coding standards via simple prompts.

Enter: Spec-Driven Development

Definition:

A software approach where a clear **specification** is written first and serves as the **single source of truth**.

Benefits:

- **Higher Quality:** Reduces ambiguity and catches errors early.
- **Faster Scaling:** Enables parallel work and automated code generation.

Challenges:

- **High Upfront Cost:** Requires discipline to write specs before coding.
- **Maintenance Overhead:** The spec must always be kept in sync with the code.

The Spec-kit: Spec-Driven Development in Practice

The **Spec-kit** is the practical toolkit that makes Spec-Driven Development actionable. It's how you apply the theory.

How it connects:

- **The "Spec" is your input:** Your natural language prompt *is* the specification.
- **The "Kit" is the process:** The `/speckit.*` commands provide a structured workflow.
- **"Driven" is the AI's job:** The AI takes your spec and drives the implementation from it.

This turns the abstract principles of SDD into a concrete, repeatable process.

Anatomy of a Spec-kit (1/4)

Requirements

- **User Stories:** "As a user, I want to..."
- **Acceptance Criteria:** "Verify that..."
- **Business Logic:** Rules for validation, calculations, and workflows.
- **Format:** Markdown, Gherkin syntax.

Anatomy of a Spec-kit (2/4)

API doc & Schemas

- **Interfaces:** The "hard" constraints. "explain"
- **OpenAPI / Swagger:** Defines endpoints, inputs, and outputs strictly.
- **GraphQL Schemas:** Defines the data graph.
- **Protobuf / gRPC:** For microservices.

Anatomy of a Spec-kit (3/4)

Data Models

- **ER Diagrams:** Visual or text-based (Mermaid.js/PlantUML/DBML).
- **SQL Schemas:** `CREATE TABLE` statements.
- **JSON Schemas:** Structure of NoSQL documents.
- Prevents AI from inventing fields that don't exist.

Anatomy of a Spec-kit (4/4)

Design & Constraints

- **Style:** Colors, spacing, typography (JSON/CSS variables).
- **Component Library:** List of available UI components (don't reinvent the button).
- **Tech Stack:** "Use React 18, TypeScript, Tailwind CSS".
- **Linting Rules:** `.eslintrc`, `.prettierrc`.

The Workflow: Step 1 - Define

Human Role: Architect & Product Owner.

- Write the User Stories.
- Define the Data Model.
- Sketch the API Contract.
- *No coding yet.*

The Workflow: Step 2 - Assemble

Human Role: Curator.

- Gather these documents into a folder (the `spec-kit`).
- Ensure documents don't contradict each other.
- This kit becomes the **Prompt Context**.

The Workflow: Step 3 - Generate

AI Role: Builder.

- Feed the Spec-kit to the AI (LLM).
- Request: "Implement the feature defined in `story-101.md` using the schema in `db-schema.sql`."
- AI generates the code, tests, and documentation.

The Workflow: Step 4 - Verify & Iterate

Human Role: Reviewer.

- Review the generated code against the Spec-kit.
- Run tests (which the AI also wrote based on the specs).
- **Refinement:** If code is wrong, *update the Spec-kit*, not just the code.
- **Refinement:** If code is wrong, *update the Spec-kit*, not just the code.

From Theory to Practice: The Commands

Now, let's see how this workflow translates into concrete commands you can use in your AI assistant.

Command: `/speckit.constitution`

- **Purpose:** To establish the project's foundational rules and guidelines. This is the "law" the AI must follow.
- **Input:** A natural language prompt describing your desired standards (e.g., code quality, testing, UX consistency).
- **Output:** A `constitution.md` file in the `.specify/memory/` directory.
- **Where:** Use this once at the very beginning of your project, right after `specify init`.

Command: `/speckit.specify`

- **Purpose:** To define the functional requirements of what you want to build.
- **Input:** A detailed, high-level description of the feature or app. Focus on the **WHAT** and **WHY**, not the technology.
- **Output:** A new `spec.md` file inside a feature-specific folder (e.g., `specs/001-photo-album/`), containing user stories and requirements.
- **Where:** Use this in your AI chat after the constitution is set.

Command: `/speckit.clarify`

- **Purpose:** To let the AI ask questions and fill in gaps in the specification *before* planning.
- **Input:** Just the command. The AI will then prompt you with questions.
- **Output:** An updated `spec.md` with a new "Clarifications" section containing your answers.
- **Where:** Use this immediately after `/speckit.specify` to ensure the requirements are solid.

Command: `/speckit.plan`

- **Purpose:** To create a detailed technical implementation plan.
- **Input:** A prompt specifying your desired tech stack, architecture, and libraries (e.g., "Use Vite, vanilla JS, and a local SQLite database").
- **Output:** Technical documents like `plan.md`, `data-model.md`, and API contracts within your feature's spec folder.
- **Where:** Use this after the specification has been created and clarified.

Command: `/speckit.tasks`

- **Purpose:** To automatically break down the technical plan into a detailed, actionable checklist.
- **Input:** Just the command. It reads the `plan.md` that already exists.
- **Output:** A `tasks.md` file in your feature's spec folder, with an ordered list of implementation steps.
- **Where:** Use this after the technical plan is finalized.

Command: `/speckit.implement`

- **Purpose:** To execute the generated task list and build the application.
- **Input:** Just the command. The AI will read `tasks.md` and start building.
- **Output:** The final source code for your application. The AI will run local commands (`npm`, `dotnet`, etc.) as needed.
- **Where:** This is the final step. Use it when your plan and tasks are ready for execution.

Optional Commands

- **/speckit.analyze:**
  - **Purpose:** Checks for consistency across all your spec files.
  - **When:** Run after `/speckit.tasks` to catch errors before implementation.
- **/speckit.checklist:**
  - **Purpose:** Generates a quality checklist to validate that the spec is clear and complete.
  - **When:** Run after `/speckit.specify` to act like "unit tests for your English."

Comparison: Spec-kit vs. Others

Feature	Chat-driven	Autocomplete	Spec-kit
Input	Natural Language	Code Context	Structured Specs
Scope	Snippets	Lines / Blocks	Modules / Features
Context	Ephemeral	Local (File)	**Persistent (Kit)**
Consistency	Low	Medium	**High**

Why is it Better? (Quality)

- **Deterministic Outputs:**
  - Strict inputs lead to strict outputs.
- **Reduced Hallucination:**
  - AI is grounded by the provided schemas.
- **"Shift Left":**
  - Bugs are caught in the *design phase* (conflicting specs) before code is written.

Why is it Better? (Scalability)

- **Batch Generation:**
  - Generate Frontend, Backend, and Tests simultaneously using the same shared Spec-kit.
- **Onboarding:**
  - New developers (and new AI agents) can read the Spec-kit to understand the system immediately.

Challenges & Limitations

- **Upfront Investment:**
  - Requires writing specs *before* coding. (The "Lazy Developer" problem).
- **Maintenance:**
  - The Spec-kit must be kept in sync with the code.
- **Complexity:**
  - Over-engineering the kit can lead to diminishing returns.

Future Outlook

- **Auto-generated Kits:**
  - AI analyzing existing code to *create* the Spec-kit for you.
- **IDE Integration:**
  - "Spec-driven" IDEs where the spec is a first-class citizen.
- **Continuous Validation:**
  - CI/CD pipelines that check if code matches the Spec-kit.

Conclusion

Spec-kit Development moves us from "Coding with AI" to **"Architecting for AI"**.

It empowers developers to focus on the **WHAT** (The Specification) and lets the AI handle the **HOW** (The Implementation).

The code is temporary. The Specification is eternal.