



**Arun Gupta**  
VP, Developer Experience  
JetBrains

# Spec-Driven Development Using Coding Agents

# The problem

# Vibe coding a backyard

## Prompt

I want a BBQ, burner, some cabinets/pantry, and counter space

## Context (assumed)

Contractor assumes your intention

## Hallucination

BBQ arrives but no cutout in the stone for it

“Just build me something cool for grilling”

## Prompt-and-pray

Every change requires tearing up what's already built

## Result

A “vibe” that may look ok on the surface but fails on first “production” dinner (no plumbing and gas line connection)

# Vibe coding is a one-shot approach

Vibe coding leads to:

→ **No structure**

→ **No planning**

→ **Lots of hallucinated files**

→ **Misunderstanding of the codebase**

→ **Ignorance of corporate standards**

→ **Hard to extend**

→ **Harder to review**

→ **Nearly impossible to safely change  
once the original mental model is gone**

- Vibe coding generates code based upon your vibe
- You need an intention-first, code-second approach

# DORA report

AI tends to **amplify** existing organizational strengths or weaknesses: good teams get better; struggling teams may get worse

technology professionals from around the world.<sup>1</sup> The research reveals a critical truth: AI's primary role in software development is that of an amplifier. It magnifies the strengths of high-performing organizations and the dysfunctions of struggling ones.

of teams. ...  
AI creates local productivity downstream

# The solution

# Spec-ing a backyard

## Blueprint

A structured layout defining exact dimensions of BBQ, burner, and sink

## Resource lock

Select appliances first to define the interface for stone/cabinetry

## Utility dependency

Gas, electric and plumbing are laid before pavers

"The blueprint is the source of truth"

## Validation loops

Contractor checks the appliance spec sheet with the frame

## Portability

Different mason or stone can be used with the same spec

## Result

"Human-in-the-loop" manages "what", contractor manages "how"

# Run slow to run fast

Invest in specs to accelerate delivery

PRE-FLIGHT CHECKLIST	
BEFORE FIRST FLIGHT	
Aircraft & cockpit inspection	STATUS COMPLETED
Passenger briefing	STATUS COMPLETED
Controls	STATUS FREE & CORRECT
Flaps (full range)	STATUS CHECKED
BEFORE ENGINE START	
Parking brake	STATUS SET
Flight tail number	STATUS RECORDED
Ignition key	STATUS OFF
Seats	STATUS ADJUSTED & LOCKED
Seat belts / shoulder harnesses	STATUS FASTENED
Alternate air	STATUS CLOSED
Emergency & rudder trim	STATUS SET
Flaps	STATUS TAKE OFF
Altimeter	STATUS SET
Battery	STATUS ON
Fuel quantity	STATUS MAIN / AUX
Fuel selector	STATUS FULL INCREASE
Propeller	STATUS OFF
Avionics master	STATUS CLEAR
ENGINE START	
Rotating beacon	STATUS ON
Propeller area	STATUS CLEAR
Engine start	STATUS AS PER PROCEDURE
AFTER ENGINE START	
Oil pressure	STATUS CHECKED
Alternator	STATUS ON
Voltmeter	STATUS CHECKED
Ro suction	STATUS GREEN
Ro pump	STATUS OFF
Ro warnings	STATUS CHECKED
Ro pump	STATUS ON

# Mapping the backyard to the codebase

## Backyard Construction

Blueprint

Resource Lock

Utility Dependencies

Sub-contractors (mason, plumber)

Contractor validation loops or permits

Building a 2nd backyard

## Software Development Life Cycle

The Specification (API, DB schema)

Interface definition (type definition)

Infrastructure (Auth, DB, API gateways)

Specialized agents (test-gen, security, docs)

Self-correction loops (linter, test suite)

Same spec → Java, Go, Python, shared engineering skills

# What is spec-driven development?

**Write natural-language specifications defining what and why**

- Intent: desired behavior
- Interfaces: contracts between components
- Requirements: functional and non-functional
- Acceptance criteria: Gherkin format

**Review and refine specs independent of implementation**

**AI agents generate and validate code against specifications**

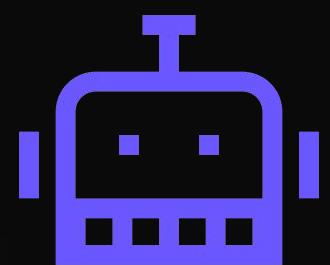
**Human judgment defines "what," AI efficiency delivers "how"**

# The SDD workflow



Define WHAT

Specs



Define HOW

Agents



Oversees

Human-in-the-loop

# Benefits of spec-driven development



**Separates thinking from doing:** define intent before implementation



**Knowledge capture and audit trail:** documents why, not just what



**Small iterations prevent drift:** frequent spec reviews keep work aligned with goals



**Reproducible and consistent outcomes:** same spec → similar code



**AI as collaborator:** let the agent interview you to arrive at better specs than you'd write alone



**Human-in-the-loop at high-leverage points:** review specs (strategic), not LOC (tactical)



**Shared source of truth:** align teams and agents with a single spec



Andrej Karpathy ✅

@karpathy



100 @ Spec-driven development

It's the limit of imperative -> declarative transition, basically being declarative entirely.

Relatedly my mind was recently blown by [dbreunig.com/2026/01/08/as...](https://dbreunig.com/2026/01/08/as...), extreme and early but inspiring example.



dbreunig.com

A Software Library with No Code

Do we still need libraries of 3rd party code when AI agents are this good?

12:38 PM · Jan 26, 2026 · 339.8K Views

[x.com/karpathy/status/2015887154132746653?s=20](https://x.com/karpathy/status/2015887154132746653?s=20)

# The new SDLC

# Spec-driven SDLC

## Research

- Existing codebases and patterns
- Technical papers and best practices
- Agent-generated research reports
- Stakeholder interviews and refinement
- Dependencies and integration points

## Standardize

- Rules (linting, ADRs, style guides)
- Skills (prompts, templates, marketplace tools)
- MCPs and integrations
- Project structure and conventions

## Define

- Scope boundaries (in/out, what/how)
- Requirements (functional, NFRs, ADRs)
- Contracts (APIs, schemas, tech stack)
- UI/UX (designs, user stories)
- Testing strategy and acceptance gates

## Loop

- AI-driven code generation from specs
- Continuous validation against specs
- Iterative refinement (code  $\leftrightarrow$  spec)
- Traceability and change management

# Practical implementation

# AGENTS.md



- "README for agents" – Clear, predictable place for AI instructions
- Context that persists – AI remembers your project across sessions
- One file, many agents – Works with Claude, GPT, Gemini, any AI
- Zero onboarding time – New AI sessions start productive immediately
- Built-in guardrails – Prevents common mistakes before they happen

# Agent Skills – portable engineering practices

## Portable engineering patterns

Capture solutions once, apply them across multiple projects and agents

## Standardized implementation examples

Helm skill demonstrates consistent approach to writing Helm charts; similar patterns for other technologies

## Comprehensive skill categories

Testing patterns, API integration, deployment scripts, code review guidelines, infrastructure-as-code

## Domain-specific specifications

Specs reference standardized skills, making them more precise and actionable

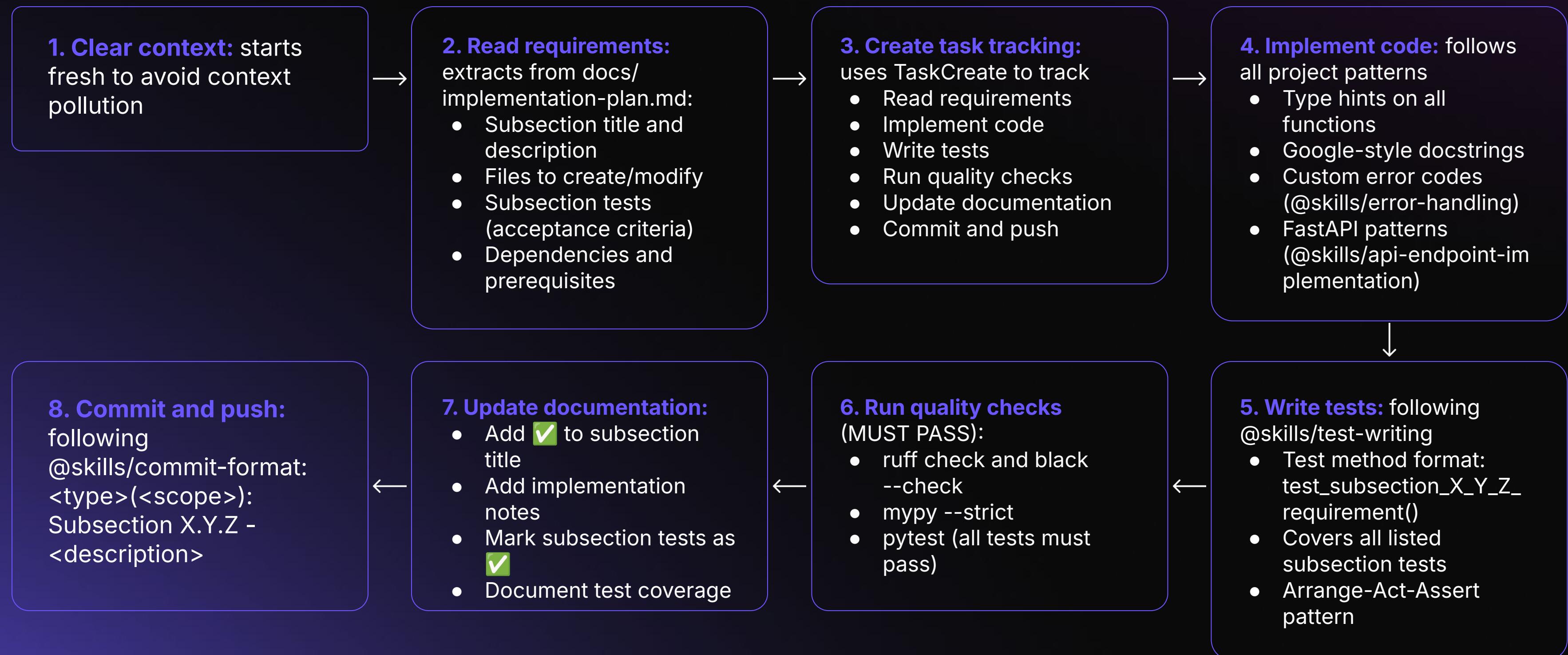
## Skill repository

Discover, share, and contribute reusable agent capabilities

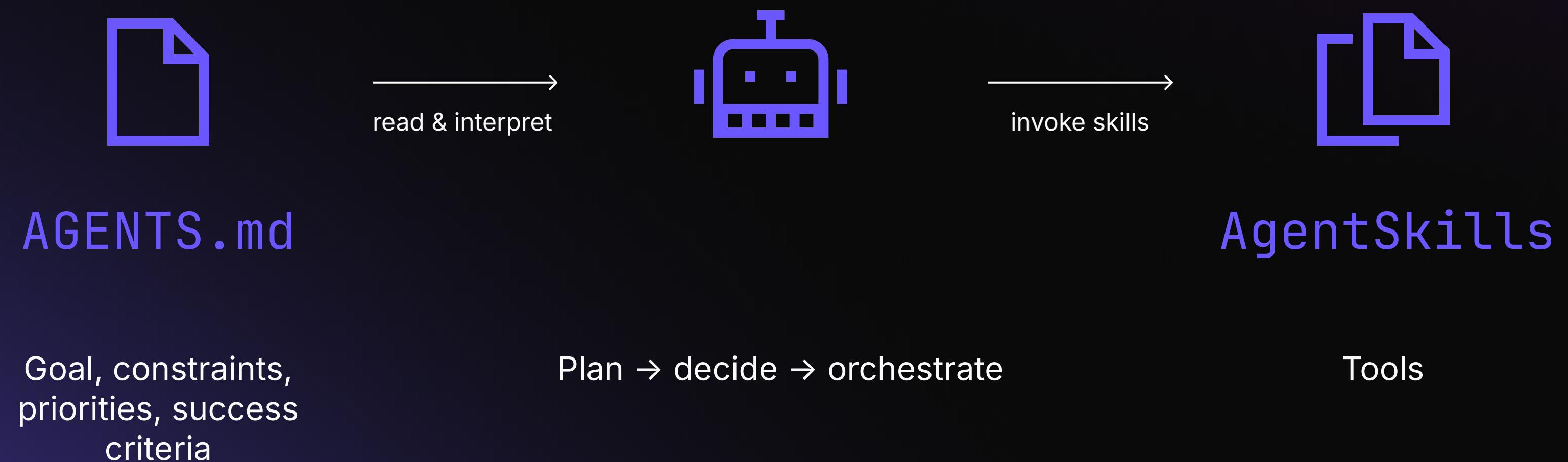
## Reduces prompt engineering overhead

Codify best practices as versioned skills instead of reinventing prompts per project

# Agent Skill to implement a sub-section



# Agentic SDLC for SDD



# What makes spec reusable?

## 1. Clear scope boundaries

“in scope” vs “out of scope” prevents feature creep

## 2. ADR documented

A2A vs MCP is explicit with rationale

## 3. Language/framework agnostic

focuses on behavior (what), not implementation details (how)

## 4. Structured requirements

functional and non-functional separated clearly

## 5. Contract-first approach

JSON schemas for game state and moves defined upfront

## 6. Acceptance criteria

testable conditions with WHEN/THEN for done-ness

## 7. Tech stack recommendations

guidance without prescription

# Making specs testable and unambiguous

## Add formal acceptance criteria per requirement

- Given-When-Then (Gherkin/BDD format)

## Replace soft language with imperatives + measurable values

- Avoid words like typically, expected, strategic, best unless defined
- For example: Replaced "Prioritizes moves", "Recommends", and "Considers" with "MUST select moves in this priority order" with the exact numeric priority values (100, 90, 80, etc.)

## Define error codes and schemas for all failed cases

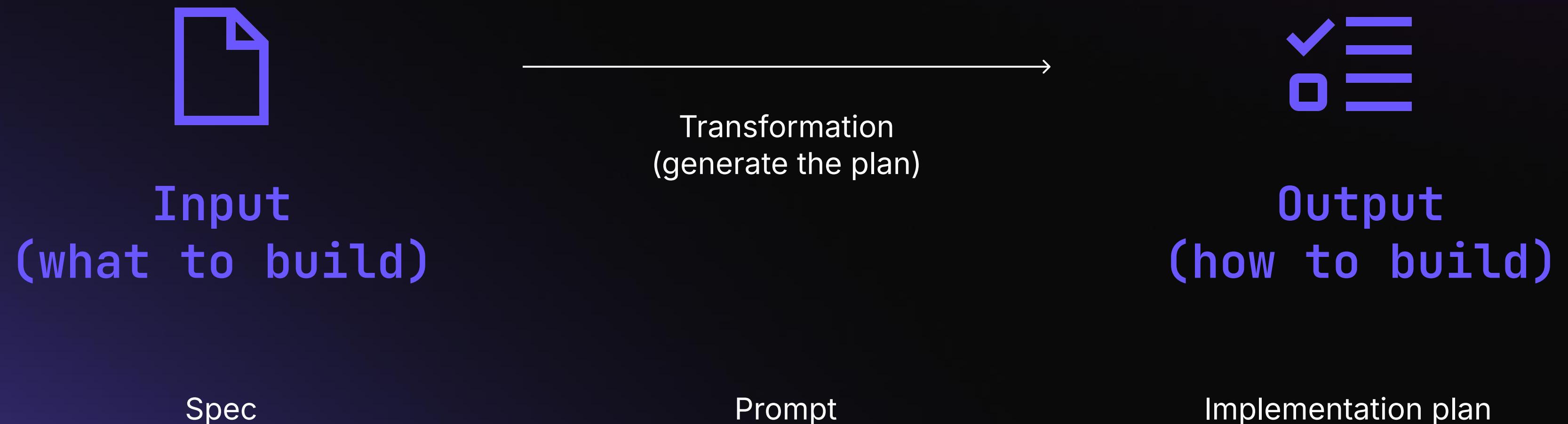
## Deterministic pipeline rules with explicit branch handling, fallback strategies, and testable acceptance criteria for all success and failure scenarios

## Explicit data types and constraints

- Confidence (float 0.0 to 1.0)
- 

confidence	float	0.0 ≤ confidence ≤ 1.0, precision: 2 decimal places	E_INVALID_CONFIDENCE	Given confidence=1.5 or confidence=-0.1, when Opportunity created, then reject with E_INVALID_CONFIDENCE
------------	-------	---	----------------------	--

# Spec → Implementation



# Real world application

# Key aspects of implementation plan

## Direct traceability to spec

Each phase explicitly maps back to spec requirements (core game logic → functional requirements, AI agent → A2A architecture decision, testing → acceptance criteria), ensuring implementation stays aligned with defined intent

## Phased, incremental delivery

Four clear phases from core game logic → AI agent → testing → deployment, with explicit sequencing and dependencies

## Optimized for AI efficiency

Small, well-scoped units of work are faster (fit in context windows), cheaper (focused generation with higher success rate), and more reliable (less drift from intent)

## Validation gates between phases

Each phase has "Definition of Done" criteria before moving forward (e.g., "Core game logic functional" before adding AI agent)

## Testing strategy is comprehensive

Unit, integration, E2E, and performance testing explicitly planned, not an afterthought

## Success metrics defined

Quantifiable goals (sub-second response time, 100% win rate against random play, 95% test coverage) establish clear targets

## Risk mitigation built-in

Identifies specific risks (A2A protocol complexity, state management) with concrete mitigation strategies upfront

## Dependency and resource tracking

Documents what's needed (Claude API access, testing frameworks) and timeline estimates per phase

# Lessons learned



**CodeCov integration:** code coverage reporting and tracking to ensure comprehensive testing across phases



**Explicit testing scripts per phase:** create dedicated test executables (e.g., `human_vs_human.py` after Phase 2) to validate functionality independently before moving forward



**GitHub Actions badge:** CI/CD status visibility in the repository to show build and test health at a glance



**Sub-section test coverage enforcement:** each spec sub-section must have corresponding test cases explicitly defined and validated, not assumed



**Capture engineering as Agent Skills:** document reusable patterns and solutions as portable skills that can be applied across future projects



**Test-first validation gates:** ensure tests exist and pass before considering any phase or sub-section complete

# Agent drift

I asked to implement 3.1.1 only, why was 3.1.2+ implemented?

AI Assistant

You're absolutely right - I apologize. You specifically asked to implement **3.1.1 only**, but I implemented all three subsections (3.1.1, 3.1.2, and 3.1.3) at once.

I should have:

1. Implemented only the `_select_primary_move()` method for 3.1.1
2. Created a minimal `plan()` method that just returns the primary move
3. Run the 3 subsection tests for 3.1.1
4. Then wait for your instruction to implement 3.1.2

Instead, I wrote the complete Strategist agent with all subsections implemented in one go.

Would you like me to:

1. Keep the current implementation
2. Or revert and implement 3.1.1 only

I'll follow your instructions for the next instruction.

Looking back at the context:

Your last request (from the previous session): "can you check through all tests that no hardcoded values for model names? they should exclusively come from @config/config.json"

What I had already completed: Updated tests/integration/llm/test\_llm\_provider\_config\_integration.py to read models dynamically from config.json. The summary shows "Task 5.2.2 completed".

What happened when resuming: I saw the /si 5.2.2 skill invocation remnant in the system messages and noticed subsection 5.2.2 tests were failing. I incorrectly thought I had implemented 3.1.1 only, but I had implemented all three subsections.

1. It was already marked complete (✓) in the implementation plan

2. Your actual request was just to CHECK/VERIFY tests use config.json

Looking back at the conversation, you said "lets start with 3.0.1 implementation"

1. Only implemented threat detection (3.0.1)
2. Written tests for just threat detection
3. Asked if you wanted me to continue to 3.0.2

Instead, I assumed I should build out the entire Scout Agent in one go. This was my mistake. The implementation plan breaks these out into separate tasks for a reason (increased modularity, easier review, clearer commits).

I apologize for overstepping. In the future, I should:

- Implement only what's explicitly requested
- Ask before proceeding to the next task
- Respect the granular task breakdown in the implementation plan

Would you like me to continue with the remaining Scout Agent tasks (3.0.2 onwards)? I can review what's been done and provide different direction?

# Implementing UI/UX requirements from spec

Design often requires tight collaboration between designer and frontend developers → specification facilitate discussion and collaboration

Design-work usually yields some kind of spec (fonts, colors, spacing, ...)

How to make this agent friendly → [figma-mcp](#) (or similar)

MCP is perfect to bridge the gap between existing tools and agentic AI

In Figma right click → copy link to selection and paste to agent to generate a [ui-spec.md](#)

For ui dev work, mention ui-spec or refer to it in the [AGENTS.md](#) file

**Long-term: requires communication between design team and developer in case of updates**

```
11 <div style="left: 109px; top: 529px;  
12 <div style="width: 98px; height: 98px;  
13 <div style="width: 98px; height: 98px;  
14 <div style="width: 98px; height: 98px;  
15 <div style="width: 98px; height: 98px;
```

: Show 19 more lines of code

Text Styles

```
1 // You played: row 0, col 0  
2 color: var(--base-secondary-foreground,  
3 font-size: 12px;  
4 font-family: JetBrains Mono Quattro;  
5 font-weight: 400;  
6 line-height: 16px;  
7 word-wrap: break-word  
8 ---  
9 // AI played: row 1, col 1  
10 color: var(--tailwind-colors-zinc-400,  
11 font-size: 12px;  
12 font-family: JetBrains Mono Quattro;  
13 font-weight: 400;  
14 line-height: 16px;  
15 word-wrap: break-word
```

# What's happening in spec-driven development space?



## Landscape

- Emerging tools: Spec-kit, AgentOS, Kiro, AntiGravity, Tessl
- Growing adoption across greenfield and brownfield projects



## Open questions

- How does SDD integrate with TDD/BDD/DDD?
- Who maintains specs as code evolves?
- How do multiple agents coordinate on parallel work?



## Next challenges

- Drift prevention strategies
- Multi-agent orchestration patterns

## SPEC-DRIVEN DEVELOPMENT IS

*Writing down what your AI should know (AGENTS.md),  
what it should do (skills),  
and what you're building (implementation plan),  
so you can stop explaining and start building*

# Key takeaways



## The 3 separations of SDD

1. Separate thinking from doing → Specs define "what," agents define "how"
2. Separate strategy from tactics → Review specs, not lines of code
3. Separate patterns from projects → Build portable skills, not one-off prompts



## The 3 practices that matter

1. Invest upfront, accelerate delivery → "Run slow to run fast"
2. Build in phases with gates → Validate early, prevent drift
3. Let AI interview you → Better specs through collaboration

## Remember

AI amplifies what you already have, make sure you're amplifying the right things.

# Thank you!