6.4. Spec Driven Development

- 01. Introduction
- 02. From Programmer to Architect
  - 01. Skills an AI programmer needs
  - 02. Collaboration forces understanding
  - 03. From programmers to architects
- 03. Generalization
  - 01. Spec Driven Development Generalization

01. Introduction

The Core Tension: Specs vs. Vibes

Spec-driven development and "vibe coding" represent fundamentally different philosophies, and the tension between them will likely define AI coding's evolution rather than one winning outright.

Vibe coding (exploratory, intuition-led, iterative tinkering) excels at:

- Discovery and novel solutions when requirements are unclear
- Creative problem-solving and architectural innovation
- Rapid prototyping when you don't know what you want yet

Spec-driven development excels at:

- Reproducibility and maintainability
- Collaboration across teams and AI systems
- Quality assurance and regression prevention
- Knowledge transfer and documentation

The future likely isn't purely spec-driven—it's hybrid and context-dependent. Early-stage projects thrive with vibe coding; mature products need specifications.

Your Framework: Points of Agreement and Nuance

On Human Intent (Point 2)

You're correct that humans will always need to communicate intent to AI, but this doesn't automatically mean formal specifications. Intent can be communicated through:

- Natural language conversations

- Example-driven development (showing the AI what you want via test cases)
- Interactive refinement loops
- Visual prototypes and mockups

The *format* of specification matters. A formal swagger spec is different from "make a dark mode toggle that persists to localStorage." Both communicate intent; one is more structured.

On Market Saturation and Standardization (Points 3-5)

This is your strongest insight. You're describing a natural consolidation pattern we've already seen in other domains:

Phase	Characteristic	Example
Explosion	Thousands of solutions to the same problem	Early web frameworks (Rails, Django, Flask, Express, etc.)
Consolidation	Market settles on 2-3 dominant patterns	React dominates frontend; Django/FastAPI dominate Python backends
Commodification	Standard stacks become the default	WordPress for blogs; Django for MVPs
Parameterization	Differentiation shifts to configuration, not code	Theme systems, plugin ecosystems, feature flags

Your hypothesis: once Docusaurus, Jellyfin, and similar projects become the de facto standard, AI coding shifts from "build new backends" to "configure and customize existing ones."

This is plausible. When this happens:

- Spec-driven development becomes essential because you're not writing code—you're declaring configuration against a known schema
- The specifications become *domain-specific languages* (DSLs) or configuration files, not prose
- Example: YAML for Kubernetes, Terraform for infrastructure, Tailwind config for styling

---

On Collaborative AI Development (Point 6)

This is the critical insight for spec-driven development's future. Multiple agents (human or AI) coordinating on a codebase requires:

1. Unambiguous specifications – without them, agent A and agent B create conflicting code
2. Modular contracts – each component must define its inputs, outputs, and side effects clearly
3. Version control and merging – conflict resolution becomes impossible without shared understanding

Conversely, a single developer + single AI can succeed with vaguer intent because they share context through conversation history.

Collaboration naturally drives toward specs. As AI coding scales to team-sized projects, spec-driven development becomes not optional—it's the only scalable approach.

Where Spec-Driven Development Becomes the Default

Spec-driven development will dominate in:

1. Back-end systems – Your point 5 directly applies here. Once databases, APIs, and authentication patterns standardize, backends become declarative specifications against frameworks like FastAPI, Django, or Supabase.
2. Regulated industries – Healthcare, finance, legal. Compliance requirements force specifications into the codebase anyway. AI-assisted coding here is already spec-driven.
3. Large-scale projects – The more developers/agents involved, the more specifications are non-negotiable.
4. Maintenance-heavy codebases – Legacy systems, widespread libraries, frameworks. The cost of ambiguity is too high.

Where Vibe Coding Persists

Vibe coding will remain viable in:

1. Personal projects and prototypes – Solo developers can afford to iterate loosely.
2. Research and exploration – When the problem space is novel, specifications are premature.
3. Front-end and UX – Design, visual polish, and user experience often benefit from experimentation rather than upfront specs.
4. Early-stage startups – When requirements change weekly, formal specs become anchors rather than anchors.

The Real Future: Spec-Driven as Infrastructure, Not Philosophy

I'd refine your thesis slightly: Spec-driven development won't be "the future" because developers choose it—it will be the future because infrastructure makes it inevitable.

Think of it this way:

- Today, a developer can write spaghetti code because the cost of coordination is low (solo project)
- Tomorrow, when you're collaborating with 2 other humans, 3 AI agents, and a linter, the cost of ambiguity becomes unbearable
- The market will pressure toward frameworks and tools that *enforce* specifications (TypeScript, contracts, schema validation)
- Those tools make spec-driven development the path of least resistance

Your point about standardized open-source codebases (Docusaurus, Jellyfin) is key here. Once the ecosystem converges on a few "canonical" backends, the differentiation isn't in code structure—it's in *specs and configuration*. An AI coding a new feature for Jellyfin doesn't write backend logic; it writes a feature spec that generates code from a template.

The Open Question

One thing your framework doesn't fully address: What happens to the long tail of custom, domain-specific code?

Not everything can be Docusaurus or Jellyfin. Some businesses have genuinely unique requirements (custom pricing logic, proprietary algorithms, niche workflows). In those domains:

- Specs become *more* important because the AI can't rely on commodity patterns
- But they also become *harder* to write because there's less precedent

The question is: Will AI improve at spec inference faster than humans improve at spec writing? If AI can examine your existing codebase and *infer* specs retroactively, then vibe coding doesn't die—it just gets post-processed into specs. If not, humans will need to learn to write better specs.

Bottom Line

Spec-driven development is likely the long-term future of AI coding, but not because it's philosophically superior—because it's pragmatically necessary for scale. Your reasoning is sound: standardization → configuration → specifications. The real wildcard is whether AI systems will be *good enough* at inferring intent from vague language that formal specs remain optional. I suspect they will be for simple cases, but complex collaborative projects will always need them.

02. From Programmer to Architect

This chapter describes how programmers need to become more focused on system design and architect instead of being good in one skill of the software stack.

01. Skills an AI programmer needs

Skills that will set programmers apart

As programming becomes more AI-assisted, the soft and strategic skills will become even more important. Here are some **future-proof skills**:

- **System Design Thinking:** Knowing how to architect reliable, secure, and scalable systems will always be in demand.
- **Problem Decomposition:** The ability to break down complex challenges into solvable chunks.
- **Human-AI Interaction:** Prompt engineering, instruction tuning, and feedback loop design.
- **Ethical Reasoning:** Anticipating the long-term consequences of deploying models and tools into real-world ecosystems.

In short, **programmers of the future must be great thinkers and designers**, not just code typists.

The Uncomfortable Truth

Most current programming education and bootcamps are training Tier 1 coders. They teach "how to build a to-do app" or "how to use React," not "how systems fail" or "how to think about correctness."

When AI can do Tier 1 work, those graduates won't be in demand. But graduates who spent 4 years building mental models of computation, algorithms, systems architecture, and trade-offs? They'll be leading teams.

The future requires that people who code understand programming deeply. Not just because it's philosophically nice, but because collaboration—with humans and AI—demands it.

Your framing is exactly right: Understanding isn't a luxury skill for seniors. It's the foundational requirement for anyone working on collaborative projects. Everything else is tooling.

02. Collaboration forces understanding

Collaborative Projects as a Forcing Function

Your point about collaborative projects is crucial: collaboration with other intelligent agents (human or AI) is the killer application that forces understanding.

Why? Because:

1. You can't debug via conversation alone – You need to read code and understand it deeply
2. You can't trust without verification – You need to know *why* a solution works, not just that it compiles
3. You can't design collectively without shared mental models – Specs without understanding are just word salad
4. You can't negotiate tradeoffs without comprehension – "Should we optimize for speed or memory?" requires understanding the implications

Ironically, AI making coding easier makes understanding *more* critical, not less. Because now the friction isn't "can I write this?"—it's "do I understand what I'm writing and why?"

03. From programmers to architects

The Harsh Implication for the Industry

Your point has a sobering corollary: A significant portion of current programmers will become obsolete, not because AI is so good, but because they were never really architects—they were syntax-execution machines.

Consider:

- A developer who learned to code by copy-pasting Stack Overflow answers → replaceable by AI
- A developer who can write CRUD endpoints but doesn't understand database normalization, query optimization, or transaction semantics → replaceable by AI
- A developer who treats code as "stuff that works" rather than "a system with invariants and constraints" → replaceable by AI

But developers who understand systems deeply will become *more* valuable, because:

1. They can design the specs that AI needs
2. They can validate AI's work
3. They can architect for constraints (scalability, security, maintainability)
4. They can collaborate effectively with other developers and AI agents

---

The Collaboration Problem You Identified

Here's where understanding becomes *non-negotiable*:

In a collaborative project with humans and AI agents, someone has to hold the shared mental model of the system. That model includes:

- How components interact
- What invariants must hold (data consistency, performance bounds, security properties)
- Why certain tradeoffs were made
- Where the system is fragile

Without understanding, you can't maintain this model. You can't communicate specs effectively because you don't know what's actually important. You can't review AI-generated code because you don't know what "correct" looks like. You can't resolve merge conflicts or design changes because you don't grasp the reasoning.

Understanding is what makes collaboration possible. This is why open-source projects with great documentation and clear architecture attract contributors, while chaotic codebases repel them—even if both are technically functional.

What "Learning Programming" Actually Means (Going Forward)

The content of programming education will shift:

What Won't Matter As Much

- Syntax memorization
- Language-specific idioms and patterns
- Boilerplate construction
- Debugging by trial-and-error

What Becomes Essential

- Data structures and their tradeoffs – Why use a B-tree over a hash table? When does O(n log n) fail you?
- Systems thinking – How do components interact? Where are bottlenecks, races, and failure modes?
- Constraints and invariants – What *must* be true for this system to work? What could break it?
- Specification and design – How do you communicate intent unambiguously to other agents (human or AI)?

- Correctness verification – How do you know the code actually does what you think?
- Trade-off analysis – Given multiple possible implementations, which fits your constraints (performance, maintainability, scalability)?

In short: from "how to write code" to "how to think about code."

The Real Skill Hierarchy Emerging

I'd propose a three-tier system:

Tier	Capability	AI Replaceability
Level 1: Coder	Can write syntactically correct code given clear instructions	✗ Highly replaceable by AI
Level 2: Engineer	Can design systems, spec features, review implementation, debug issues	~ Partially augmented by AI
Level 3: Architect	Can design for unknown future requirements, identify systemic risks, mentor and guide, innovate	✓ Irreplaceable (AI is a tool)

The market will compress Level 1 wages or eliminate Level 1 roles entirely. Level 2 and Level 3 developers will become scarcer and more valuable.

03. Generalization

01. Spec Driven Development Generalization

If Spec driven development works for digital software then it can also be used for any other field of research. This could be physics, chemistry, biology and what not.