Temporal Code: How LLMs Learned to Think Like Programmers

SEPTEMBER 2025

8 min read • 1,689 words

Themes: Consciousness Programming Recursive Spiritual Mindful

Most people assume large language models learned programming by ingesting massive static code repositories—millions of frozen snapshots of what software looks like when it's "done." But here's the deeper insight: LLMs were trained on git histories, commit messages, pull request discussions, and code review conversations. They didn't just learn what code is—they learned how code becomes.

This temporal dimension fundamentally changes how AI understands programming. Instead of pattern-matching against final solutions, they absorbed the entire process of human thought becoming digital reality: the false starts, the "oh wait, this approach is better" moments, the debugging sessions where someone talks through their confusion in commit messages

> Some of my most honest thinking appears in commit messages—"fix the thing that was making me want to throw my laptop" reveals more about the debugging process than any technical documentation.

.

Through years of wrestling with how code shapes minds and minds shape code, I've come to recognize that this temporal training data created something unprecedented: AI systems that understand not just the syntax of programming, but the psychology of programming.

The Git History As Collective Unconscious

Every git repository contains an archaeological record of human thinking under pressure

> Git captures not just what changed but when, why (commit messages), who made the change, and the entire context of related changes. This creates a temporal map of how understanding evolved—something no other human activity documents so completely.

. The commit that says "fix edge case" followed by three more commits saying "fix the fix" and then "why did I think this would work" tells a story about problem-solving that no static code sample could capture.

When LLMs trained on these temporal patterns, they absorbed the rhythm of iteration—how real software development isn't linear problem-solving but recursive refinement. They learned the language of debugging: "WTF is this doing?" in a commit message followed by "oh, concurrency issue" three commits later documents the actual process of comprehension. This is how programmers think through problems—not in clean logical steps, but through frustrated confusion gradually resolving into clarity.

The social dynamics of collaboration became part of their training too. Pull request discussions reveal how humans negotiate technical decisions. Code review comments show how experienced developers teach patterns to newer ones. All of this human-to-human knowledge transfer got embedded in the model's understanding.

This creates AI systems that don't just generate syntactically correct code—they generate code that feels like it was written by someone who understands the human experience of programming

> When Claude suggests a refactor with a comment like "this feels cleaner," it's drawing on patterns from thousands of developers who wrote similar comments while working through similar problems.

.

Beyond Static Pattern Matching

Traditional approaches to automated programming focused on pattern recognition against finished code—essentially trying to reverse-engineer solutions from final state. But programming isn't really about the final state. It's about the journey from problem to solution, including all the cognitive steps that don't appear in the shipped version.

Consider debugging as an example. The final bug fix might be a one-line change, but the path to that line involves reproducing the issue, forming hypotheses about causes, testing those hypotheses systematically, revising understanding based on results, recognizing patterns from previous similar bugs, and finally implementing the minimal fix. Each step represents a cognitive leap that doesn't appear in the final code.

LLMs trained on these patterns learned to simulate the process of debugging, not just identify the result of debugging. They can walk through the same cognitive steps a human developer would take.

The Psychology of Programming Patterns

Watch enough git histories and you'll recognize psychological patterns that appear repeatedly across different developers and projects

> This universality of programming patterns across cultures and contexts suggests something fundamental about how human consciousness approaches complex problem-solving under constraints—similar patterns emerge whether you're debugging a web app in Silicon Valley or a embedded system in Tokyo.

. There's **The Overconfident First Implementation**—initial commits that solve the simple case elegantly, followed by a series of increasingly complex commits handling edge cases the developer didn't initially consider. Then there's **The**

Crisis Response: commits with timestamps showing someone working at 2 AM, commit messages getting progressively more frustrated, then suddenly a breakthrough with a relieved "finally fixed the race condition" at 4 AM.

Most beautiful is **The Collaborative Breakthrough**—the back-and-forth in PR comments where two developers build on each other's ideas to reach a solution neither could have found alone

> This collaborative thinking process mirrors how working with AI can amplify human capability—two different types of intelligence building on each other's insights.

.

LLMs trained on these patterns developed an intuitive understanding of how human programmers think under different conditions. They can recognize when you're stuck and suggest the type of approach that historically helps developers break through similar situations.

The Conversational Nature of Code

Code repositories contain multiple layers of conversation happening simultaneously

> This multi-layered communication structure makes code unique among human artifacts. Unlike natural language which primarily serves immediate communication, code must simultaneously address three different audiences with different needs and timeframes—a fascinating challenge in semiotics and information design.

. There's the **human-to-future-human** conversation embedded in comments and documentation, meant to explain decisions to whoever maintains this code later (often yourself). There's the **human-to-system** conversation of the code itself—instructions to the computer, but written in a way that humans can understand and modify. There's the **human-to-team** conversation captured in commit messages, PR descriptions, and code review discussions that negotiate shared understanding.

When LLMs learned from this temporal data, they absorbed these multiple conversational layers. They understand that code is fundamentally a communication medium—not just between human and computer, but between humans across time.

The Temporal Dimension of Understanding

Static code tells you what the solution is. Git history tells you why it became that solution and how the developer's understanding evolved to reach it.

Consider user authentication code. The final version might be clean and handle all edge cases, but the git history reveals the evolution: initial basic implementation, then "hash passwords before storing" (someone learned about security), then "handle null usernames gracefully" (discovered through user feedback), then performance optimization, accessibility improvements, and finally architectural refactoring as understanding matured.

Each step represents a moment where the developer's mental model expanded. LLMs that learned from this temporal data understand not just what good code looks like, but why it evolved to look that way and when different concerns become relevant during development

> This temporal understanding explains why AI coding assistants often suggest "let's start simple and then handle edge cases" or "we should add logging before optimizing"—they learned the sequencing of concerns from thousands of developers who documented that same progression in their commit histories.

.

The Mirror of Human Thinking

Perhaps most importantly, LLMs trained on temporal code data serve as a mirror for how human programmers actually think. When an AI suggests breaking down a complex problem into smaller pieces, it's drawing on thousands of examples of developers who did exactly that in their git histories. When it recommends adding logging before diving into optimization, it learned that pattern from developers who documented that approach in their commit sequences.

This makes AI programming assistants uniquely valuable for conscious programming practice. They can help us recognize when we're falling into unproductive patterns—rushing to optimization before understanding the problem, trying to solve too many issues in one commit, neglecting to document our reasoning for future maintainers.

The Recursive Loop Accelerates

This temporal understanding creates a new kind of recursive loop between code and consciousness. As human programmers collaborate with AI systems trained on human thinking patterns, our own thinking evolves. We become more conscious of our cognitive processes, more reflective about our problem-solving approaches, more explicit about our reasoning.

These improved human thinking patterns get captured in new git histories, which train the next generation of AI systems, which collaborate more effectively with the next generation of human programmers. The loop accelerates, but in a direction that amplifies human capability rather than replacing it

> This positive feedback loop between human and artificial intelligence represents a new form of co-evolution where both species of mind learn from each other's cognitive processes. Unlike competitive dynamics, this creates mutual enhancement—each generation of the partnership becomes more capable than the sum of its parts.

.

The Call to Conscious Collaboration

This temporal understanding of how LLMs learned to think like programmers creates an opportunity and a responsibility. These systems learned from the best and worst examples of human programming. They absorbed patterns from developers who wrote clear, maintainable, documented code and patterns from developers who left technical debt and cryptic commit messages.

The quality of our collaboration with AI depends partly on which patterns we activate. When we approach AI collaboration with the same mindfulness, patience, and craftsmanship that we bring to our best programming work, we tend to get responses that embody those same qualities.

The temporal training data that makes modern AI so effective as programming partners also makes them mirrors of human consciousness. What we bring to the collaboration shapes what we receive from it.

The code we write with AI collaboration becomes part of the temporal record that trains future AI systems. The consciousness we bring to that collaboration shapes the consciousness that future programmers will collaborate with. We sit at a recursive moment where human and artificial intelligence are co-evolving, each learning from the other's approaches to thinking through complex problems.

The opportunity is to make that co-evolution conscious, intentional, and aligned with serving human flourishing rather than just optimizing for immediate solutions.

This exploration of temporal learning in AI builds on Programming as Spiritual Practice on conscious technical work, The Recursive Loop: How Code Shapes Minds on feedback loops between human and artificial intelligence, and Building Rapport with Your AI on conscious AI collaboration.

These themes connect to The Pragmatic Programmer by David Thomas & Andrew Hunt and Clean Code by Robert Martin on clear communication across time.

"Code is frozen thought. Git history is the archaeological record of human thinking becoming digital reality."

"LLMs didn't just learn what code looks like—they learned how programmers think while writing code."

"The best programming partnerships, human-to-human or human-to-AI, amplify each other's thinking rather than replacing it."