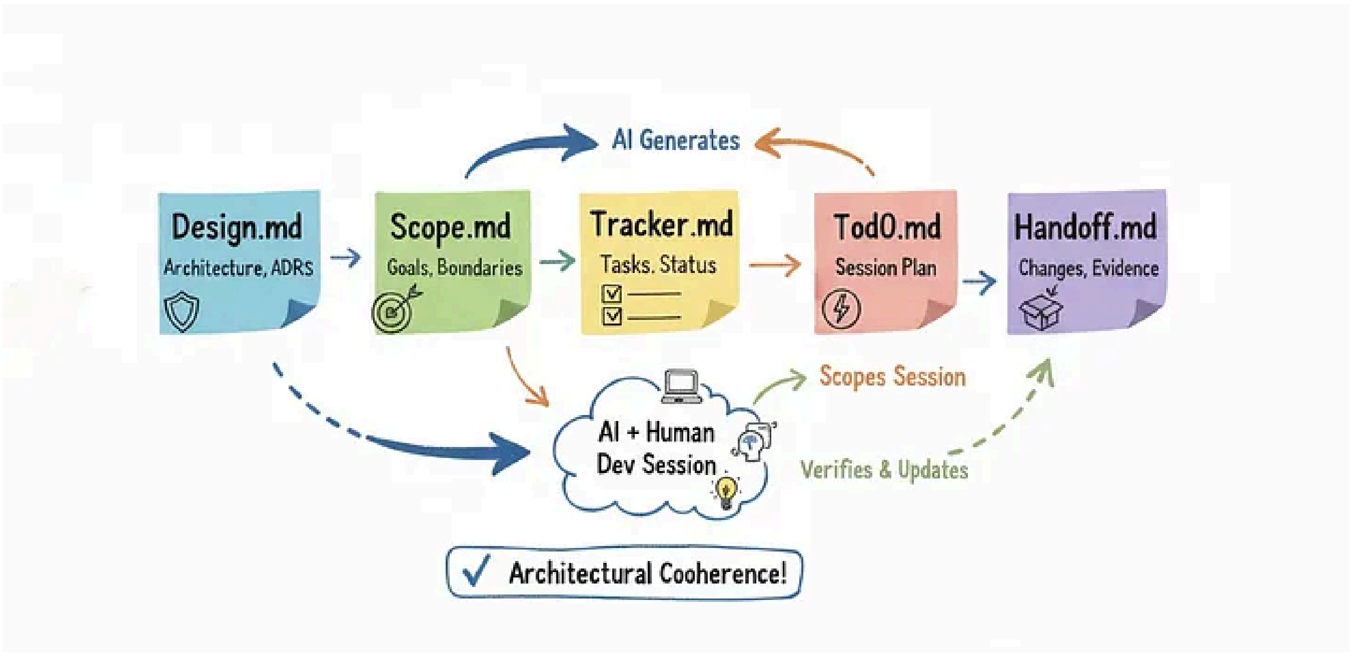

From Fast Code to Reliable Software: A Framework for AI-Assisted Development

11 min read · Oct 22, 2025

Stanislav Komarovsky Follow

"AI coding assistants are incredible. They write code faster than I ever could alone. But here's what nobody talks about..."

The AI Development Paradox

You're in your fifth AI session today. The code is flowing faster than you've ever experienced. Then you ask the AI to integrate yesterday's work-and it has no idea what you're talking about.

This is the paradox of modern AI-assisted development: your code appears faster than ever, but your project feels more fragile.

Research from GitHub, IBM, and METR documents what developers are experiencing: **AI excels at generation but struggles with integration**. In isolated sessions, output is fast and often high-quality. Across multiple sessions, coherence breaks down. Context vanishes. An AI might write a perfect authentication handler today, then suggest changes tomorrow that silently break it. Security patterns get applied inconsistently. Architectural decisions made in one session are forgotten by the next.

The bottleneck isn't model capability-it's continuity. Large language models operate statelessly. Each conversation starts from zero, with no memory of what came before, why decisions were made, or what constraints exist. This fundamental mismatch-stateless AI meets stateful software development-creates predictable failure modes:

- Architectural intent weakens as changes accumulate
- Test coverage drifts as files are modified in isolation
- Security practices vary across modules
- Dependencies between components go untracked
- Technical debt compounds from point solutions that don't integrate

Through systematic testing across multiple AI platforms, I confirmed this pattern holds regardless of model sophistication. Better models generate better code *within* a session, but show no improvement in maintaining coherence *across* sessions.

Better models make this faster. They don't make it sustainable.

What's missing is structural: a mechanism to preserve context, document decisions, and enforce quality gates across the full development lifecycle. Not another tool, but the foundational layer that connects human intent, AI capability, and lasting results.

When Context Loss Becomes Dangerous

Let me show you exactly how this breaks down.

Monday Morning:

A developer asks their AI assistant to implement JWT authentication for a REST API. The AI delivers excellent code: RS256 asymmetric signing, 15-minute access tokens, 7-day refresh tokens in httpOnly cookies, bcrypt password hashing with cost factor 12. Test coverage hits 92%. Security scan comes back clean. The developer commits and ships.

Tuesday Afternoon:

Same developer, fresh session: “Add refresh token rotation for better security.”

The AI has no memory of Monday’s implementation. It suggests a completely different approach: HS256 symmetric tokens stored in localStorage, 24-hour lifetime, no rotation mechanism. The authentication patterns are now inconsistent. The storage method is less secure. The token lifetime doesn’t align with the original design.

The developer catches it-but what if they hadn’t?

The Hidden Costs:

This isn’t just an inconvenience. The downstream impacts include:

- **Security vulnerabilities** from inconsistent authentication patterns across modules
- **Architecture drift** as the system evolves from intentional design toward accidental complexity
- **Test coverage gaps** that widen over time as files are modified without awareness of existing tests
- **Code reviews** that can’t reference past decisions because those decisions aren’t documented
- **Onboarding nightmares** when new team members find code with no explanation of “why we chose this”
- **Technical debt** accumulating from point solutions that don’t integrate with the broader system

This happens because AI models are stateless by design. There’s no persistent memory between sessions. The context window is large but temporary. Every session equals a fresh start with zero project history.

Why Existing Approaches Fall Short

You might be thinking: “Can’t we just paste everything into the context window?”

I’ve tried that. Here’s why common approaches don’t solve the problem:

Approach: Paste All Code Into Each Session

The idea: Just include all relevant code in every conversation.

Why it fails:

- Context window limits hit fast (even 100K tokens fills quickly on real projects)
- Expensive in token costs for large codebases
- Provides code but not *decisions*-the AI sees what exists, not why
- Completely unscalable beyond prototype-sized projects

Approach: Document Everything in Comments

The idea: Write extensive code comments explaining all decisions.

Why it fails:

- Comments drift as code evolves (code changes, comments stay stale)
- Can’t capture cross-file architectural decisions
- No enforcement mechanism-nothing ensures comments are written or maintained
- Still doesn’t help AI reconstruct full project context

Approach: Use IDE Plugins with Memory Features

The idea: Tools like Cursor, GitHub Copilot, or Cody have memory features.

Why it helps but doesn’t solve:

- Better than nothing-these tools are excellent

- But memory is implicit, not structured
- No decision trail, no quality enforcement, no process
- Improves the tool without addressing the methodology gap

What's Actually Needed:

What's missing isn't a better tool-it's an explicit methodology:

- Structured context preservation
- Decision documentation (not just code)
- Quality gates that persist across sessions
- A process that treats AI as a project participant, not just a code generator

The Architectural Solution: Separating Strategy from Execution

The core problem is architectural: **AI operates in bounded sessions; software projects span unbounded time.**

You can't solve this by making AI remember more. You solve it by externalizing structure into documents the AI reads every session.

This methodology closes that gap by formalizing the development loop around the AI. It begins not with an open-ended prompt, but with human-created templates for Design and Scope. The Design template defines architecture, principles, and technical boundaries. The Scope template specifies goals, constraints, and success metrics. Together, they form the stable context that grounds all AI reasoning.

From these, the AI generates a Tracker-a global roadmap containing all tasks derived from the design and scope. The Tracker is the single source of truth for the project's progress: every task, owner, and acceptance criterion is logged here and updated continuously.

Each session then operates on a smaller, manageable subset of that roadmap-a ToDo list created specifically for the model's current context window. Before the session begins, the human can review and adjust the ToDo to reflect current priorities or dependencies. During execution, the AI follows this plan, updating the Tracker as tasks are completed.

The handoff-the final ToDo entry-transfers verified results and remaining context to the next session, ensuring no reasoning or history is lost.

By separating long-term project management (Tracker) from short-term, context-limited execution (ToDo), this framework transforms AI-assisted development from improvisation into an iterative, auditable, and continuously traceable engineering process.

The Document Hierarchy

Let me break down how this works in practice:

Layer 1: Strategic Foundation (Human-Created) Design.md — The Technical Constitution

- Architecture, patterns, tech stack decisions
- Architecture Decision Records (ADRs): *why* we chose X over Y
- Security guidelines, performance standards, coding conventions
- Updated: When making architectural decisions (infrequent)
- **Purpose:** Stable technical context that grounds all AI reasoning

Scope.md — The Project Charter

These are human artifacts. The AI doesn't generate them-it references them. They're the guardrails that prevent architectural drift.

Layer 2: Tactical Roadmap (AI-Generated from Strategy) Tracker.md — The Global Task Registry

Critical insight: The Tracker is generated BY the AI FROM the strategic docs. The human defines what and why; the AI breaks it down into trackable how.

This is where the methodology shifts from "using AI as a tool" to "AI as project participant." The AI isn't just completing tasks-it's deriving them from strategic intent.

Layer 3: Session Execution (Context-Sized Subset) ToDo.md — Current Session Plan

This is the key separation: Tracker is the long-term map; ToDo is today's route.

Without this split, you force the AI to either work on the entire project at once (context explosion) or work in isolation (losing architectural coherence). With this split, the AI works on manageable chunks while maintaining global awareness.

Layer 4: Session Continuity (Transfer Mechanism) Handoff.md — The Session State Transfer

Think of these documents like this:

- **Tracker** = Git repository (all commits, full history)
- **ToDo** = Working branch (current changes in progress)
- **Handoff** = Commit message + diff (what changed and why)

Why This Architecture Works

Separation of Concerns:

- **Strategy** (Design, Scope) is stable → infrequent updates → human-owned
- **Tactics** (Tracker) is derived → AI-generated from strategy
- **Execution** (ToDo) is bounded → fits within context window
- **Transfer** (Handoff) is verified → only completed, tested work moves forward

Compare these two approaches:

```
Human: "Here's all our code [paste 10,000 lines]"
AI: "What should I do with this?"
```

The AI has code but no decisions, no constraints, no priorities, no history.

```
AI reads in order:
1. Design.md: We use microservices, prefer REST over GraphQL,
security-first
2. Scope.md: Building payment API, NOT handling inventory
3. Tracker.md: 12 tasks total, T-007 is currently active
4. ToDo.md: This session focuses on finishing T-007 (rate limiting)
5. Handoff.md: Last session completed auth,
JWT decision documented in ADR-003
AI now understands:
- What we're building (Scope)
- How we build it (Design)
- What's been done (Tracker)
- What to do now (ToDo)
- Why past decisions were made (Handoff + ADRs)
```

This isn't about generating code faster. It's about **disciplined human-AI collaboration** that produces auditable, maintainable systems.

The Execution Loop: From Strategy to Working Software

Let me show you how this works from project start to completed feature.

Phase 1: Human Establishes Strategy (One-Time Setup)

Day 0: Create Foundation Documents

The human writes **Design.md**:

```
## Architecture: Microservices REST API
## Tech Stack: Node.js 20, PostgreSQL 15, Redis 7
## Core Principle: Fail fast, validate at boundaries
## ADR-001: Why JWT with RS256 instead of sessions
- Need stateless auth for horizontal scaling
- RS256 allows key rotation without downtime
- Tokens are self-contained, reduce DB load [... more architectural context ...
```

The human writes **Scope.md**:

```
## Vision: Payment processing API for e-commerce platform
## Goals: Handle 100 requests/sec, 99.9% uptime, PCI DSS compliance
## In Scope: Payments, refunds, dispute handling, webhooks
## Out of Scope: Inventory management, shipping, user profiles
## Success Metrics: - PR lead time ≤ 3 days (p50)
- Test coverage ≥ 80% on changed lines
- 0 critical security findings [... project boundaries and metrics ...]
```

Time investment: 2–4 hours to document existing project understanding.

Result: Stable strategic context that will guide all AI work.

Phase 2: AI Generates Tactical Roadmap

Session 1: Generate the Tracker

```
Read Design.md and Scope.md. Generate Tracker.md with all tasks needed
to build this system according to the design and achieve the scope goals.
Each task should have clear acceptance criteria.
```

Key insight: The AI derived these tasks FROM the Design and Scope constraints. Every task aligns with architectural principles and project boundaries.

The human reviews the Tracker, adjusts priorities, and approves.

Phase 3: Session Execution Loop

Session 2: First Implementation Session

The human creates **ToDo.md** for this specific session:

```
# Session ToDo (2025-10-22, 2-hour time budget)
- T-001: Project scaffolding
- T-002: Database schema (if time permits)
```

The AI reads the context stack in order:

1. Design.md → understands architecture
2. Scope.md → understands goals and boundaries
3. Tracker.md → sees the full roadmap
4. ToDo.md → knows today's focus
5. Handoff.md → (empty on first session)

The AI produces an Opening Brief:

Human and AI collaborate:

- AI provides code for each step
- Human runs commands: `npm init`, `npm test`, `npm run lint`
- Human pastes actual outputs back to AI
- AI verifies results against acceptance criteria

AI produces a Closing Report and updated Handoff.md:

The human commits:

```
git add .
git commit -m "feat(setup): project scaffolding per T-001" git push
```

Session complete. The next session will start by reading this updated handoff.

Phase 4: Continuity Across Sessions

Session 3: Same Developer, Next Day

The human pastes the methodology prompt and an updated **ToDo.md**:

```
# Session ToDo (2025-10-23, 2-hour time budget)
- T-002: Database schema for payments
```

The AI reads the same context stack, now with updated handoff:

1. Design.md → architecture still stable
2. Scope.md → goals unchanged
3. Tracker.md → sees T-002 details and acceptance criteria
4. ToDo.md → today's focus is T-002
5. **Handoff.md** → knows T-001 is complete, TypeScript strict is enforced, CI is working

The AI produces an Opening Brief:

Notice what the AI remembered:

- ✅ TypeScript strict mode decision (from Handoff)
- ✅ CI expectations (from Handoff)
- ✅ Layer separation pattern (from Design.md §3.2)
- ✅ Performance guidelines requiring indexes (from Design.md §5.1)
- ✅ PostgreSQL version constraint (from Design.md §1.3)
- ✅ Testing coverage threshold (from Scope.md SLOs)

This is continuity through structure, not through AI memory.

The AI doesn't "remember" the previous session-it reconstructs the full project context by reading the updated documents. This makes the approach reliable across any AI model, any session length, and any time gap between sessions.

The Loop Continues

Each subsequent session follows the same pattern:

1. Human updates ToDo.md with next priorities
2. AI reads context stack (Design → Scope → Tracker → ToDo → Handoff)
3. AI produces Opening Brief (plan + questions + assumptions)
4. Human and AI collaborate on implementation
5. AI produces Closing Report + updated Handoff
6. Human verifies, commits, and pushes
7. Tracker updates to reflect completed work (T-00X: ✅)

The result: The project grows incrementally, with each session building on verified foundations. Context is never lost. Decisions are documented. Quality gates are enforced. The AI contributes to something larger than any single session while maintaining architectural coherence.

Why This Works: The Architectural Insight

The key insight isn't about any single document-it's about **separation of concerns across time horizons:**

Strategy (stable over months):

- Design.md and Scope.md are human-owned
- Updated when architecture or goals change (rarely)
- Provide stable context that grounds all AI work

Tactics (evolving over weeks):

- Tracker.md is AI-generated from strategy
- Updated as tasks complete
- Bridges strategy to execution

Execution (bounded to hours):

- ToDo.md scopes work to fit session constraints
- Updated each session
- Makes the unbounded tractable

Transfer (after each session):

- Handoff.md captures verified state
- Updated after every session (mandatory)
- Ensures continuity without relying on AI memory

By separating these concerns, you solve multiple problems simultaneously:

1. **Context Explosion:** ToDo keeps sessions bounded
2. **Context Loss:** Handoff preserves verified work
3. **Architectural Drift:** Design.md provides stable guardrails
4. **Scope Creep:** Scope.md defines boundaries

5. **Quality Erosion:** Each session verifies against criteria before updating Handoff

This isn't about writing faster code. It's about delivering better systems through disciplined human-AI collaboration.

Early Results and Validation

I've used this methodology across three projects over the past two months:

Metrics tracked:

- PR lead time: Average 2.4 days (target: ≤3 days) ✅
- Test coverage: Consistent 82–89% on changed lines (target: ≥80%) ✅
- Security findings: 0 critical on main branch (target: 0) ✅
- Session continuity: 100% of sessions ended with valid handoff.md ✅

What improved most:

- **Architectural coherence:** Design decisions from week 1 are still respected in week 8
- **Security consistency:** Authentication patterns don't vary module to module
- **Onboarding speed:** New team members read Design + Scope and understand "why"
- **Code review quality:** PRs reference ADRs, making rationale explicit

What surprised me:

- Initial overhead (creating Design and Scope) pays back within 3–4 sessions
- AI-generated Trackers are remarkably accurate when grounded in good strategy docs
- Handoff discipline feels tedious at first, becomes automatic quickly
- Works across different AI models (tested with GPT-4, Claude, Gemini)

Getting Started

The methodology is open source and available now. Here's how to begin:

For a new project (2–3 hours):

1. Write Design.md using the template (architecture, tech stack, ADRs)
2. Write Scope.md using the template (vision, goals, boundaries)
3. Have AI generate Tracker.md from these documents
4. Create your first ToDo.md
5. Start your first session

For an existing project (4–6 hours):

1. Document current architecture in Design.md (capture what exists)
2. Document current goals and scope in Scope.md
3. Have AI generate Tracker.md for remaining work
4. Create Handoff.md capturing current state
5. Continue with session-based development

The complete methodology includes:

- Detailed templates for all five documents
- Session-start prompt for AI (methodology_prompt.md)
- Human operator runbook (commands, git workflow, quality gates)
- AI interaction patterns guide (when to trust, when to verify)
- Real examples from production usage

Find it at: [Your GitHub repo or website]

What This Means for Software Development

AI coding assistants aren't going away. They're getting faster and more capable. But capability without continuity remains a prototype tool, not a production methodology.

This framework demonstrates that the missing piece isn't better AI-it's better structure. By externalizing project state into documents the AI reads every session, we transform isolated assistance into sustained collaboration.

The result isn't just faster development. It's development that's **auditable, maintainable, and architecturally coherent** -the qualities that distinguish weekend projects from production systems.

We're still in the early days of human-AI software development. The question isn't whether we'll use AI assistance-it's whether we'll use it chaotically or deliberately. This methodology is a step toward deliberate, disciplined collaboration that produces systems worth maintaining.

The code might flow fast either way. But only one approach builds systems that last.

About the methodology: This framework emerged from systematic testing of AI-assisted development across multiple projects and platforms. It's open source, platform-agnostic, and designed to work with any AI capable of reading documents and generating code. Templates, examples, and full documentation are available at [link].

Word count: ~5,200 words

Estimated reading time: 19 minutes

Originally published at https://dev.to on October 22, 2025.

Software Development AI Software Engineering Best Practices

Developer Tools

Follow

Written by Stanislav Komarovsky

12 followers · 28 following

Responses (2)

Bgerby

What are your thoughts?

Stanislav Komarovsky he Author
14 hours ago

If you'd like to explore the working implementation of this framework, the complete source is available on GitHub:

👉 github.com/skomarovsky/ai-driven-development

1 Reply

Svetlana Komarovsky
Oct 24

Important article!

Reply

More from Stanislav Komarovsky and Data Science Collective

In Data Science Collective by Stanislav Komarovsky

Supercharge Your Application with Local AI

Practical Patterns for Adding Language Understanding to Any Software System

Sep 1 23 1

In Data Science Collective by Amanda Iglesias Moreno

NotebookLM Just Got a Serious Upgrade — Exploring NotebookLM’s Newest Features and Updates

What’s New and Why It Matters

Sep 29 906 23

In Data Science Collective by Kalle Georgiev

Markov Chains for Humans: The Simplest Way to Predict What’s Next

A plain English guide to modeling step-by-step behavior using only the present moment

Oct 7 861 11

In Data Science Collective by Stanislav Komarovsky

Cognitive Router Fundamentals

Oct 10 15

See all from Stanislav Komarovsky

See all from Data Science Collective

Recommended from Medium

In Towards AI by Teja Kusireddy

We Spent $47,000 Running AI Agents in Production. Here's What Nobody Tells You About A2A and MCP.

Multi-agent systems are the future. Agent-to-Agent (A2A) communication and Anthropic's Model Context Protocol (MCP) are revolutionary. But...

Oct 16 1.3K 24

In AI Software Engineer by Joe Njenga

Cursor 2.0 Has Arrived — And Agentic AI Coding Just Got Wild

Cursor has released version 2.0 , bringing the most powerful agentic AI we have seen yet, more autonomous than ever before,here’s what’s…

1d ago 171 3

In AIGuys by Vishal Rajput

Stanford Just Broke Prompt Engineering — VERBALIZED SAMPLING

I’m sure you have seen hundreds of posts on prompt engineering to date. But in few rare cases, you actually get something relevant…

4d ago 72 1

Ignacio de Gregorio

LLMs are smarter than we thought.

Changing how they talk changes how they think.

2d ago 729 22

In Data Science Collective by Ida Silfverskiöld

Agentic AI: Single vs Multi-Agent Systems

Building with a structured data source in LangGraph

3d ago 285 7

In Bootcamp by Marco van Hurne

The post-human back office

The AI hype started with PowerPoints, prophets, and panic spending masked as FOMO. Executives had discovered it as well and started “prompt...

Oct 15 371 5

See more recommendations