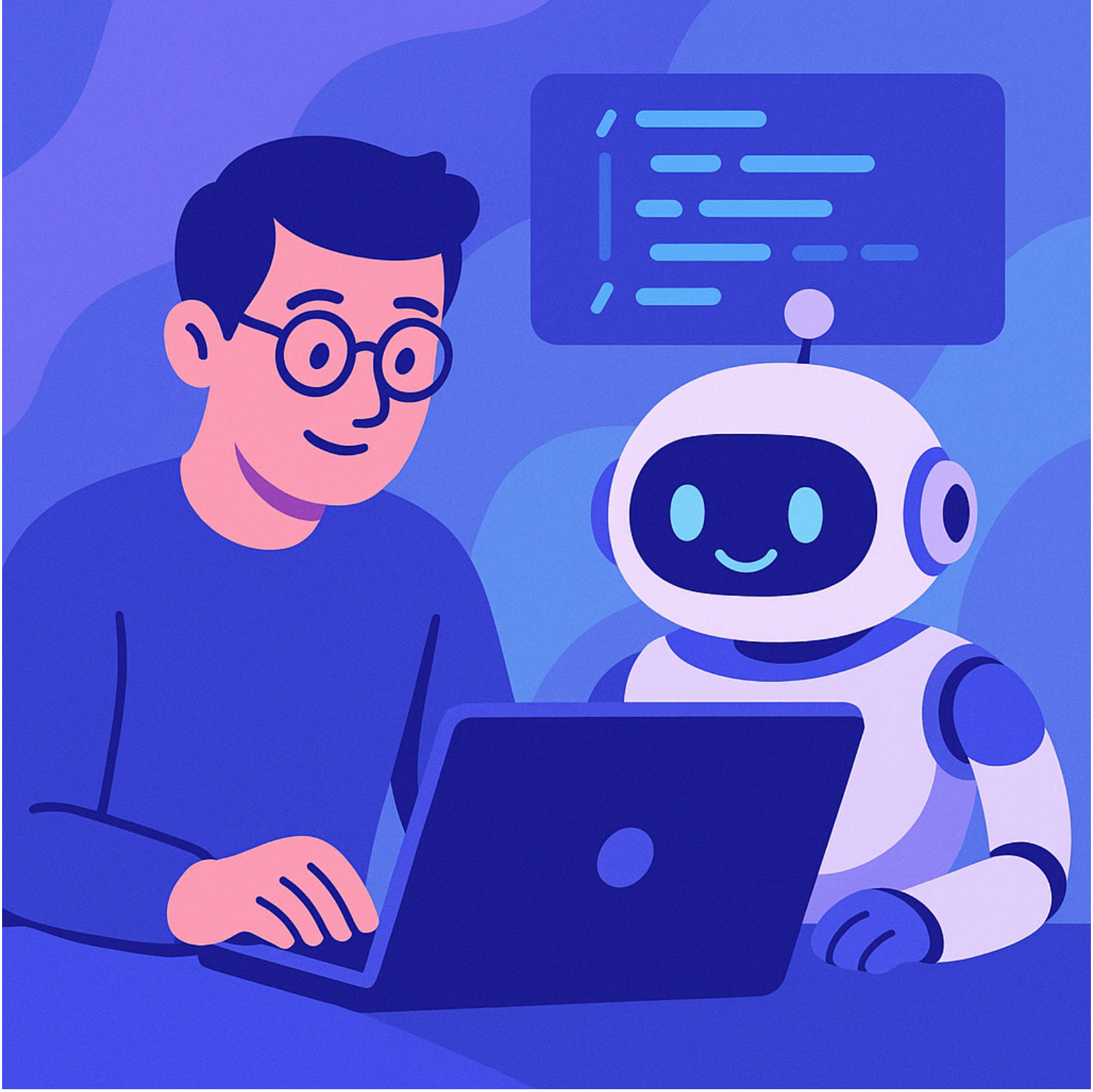
Back to Blogs

What Actually Works: 12 Lessons from AI Pair Programming

June 1, 2025 · 7 min read

AI Coding | Pair Programming | Productivity | Software Engineering



After 6 months of daily AI pair programming across multiple codebases, here's what actually moves the needle. Skip the hype this is what works in practice.

TL;DR

Planning & Process:

- Write a plan first, let AI critique it before coding
- Use edit-test loops: write failing test → AI fixes → repeat

- Commit small, frequent changes for readable diffs

Prompt Engineering:

- Keep prompts short and specific context bloat kills accuracy
- Ask for step-by-step reasoning before code
- Use file references (@path/file.rs:42-88) not code dumps

Context Management:

- Re-index your project after major changes to avoid hallucinations
- Use tools like gitingest.com for codebase summaries
- Use Context7 MCP to stay synced with latest documentation
- Treat AI output like junior dev PRs review everything

What Doesn't Work:

- Dumping entire codebases into prompts
- Expecting AI to understand implicit requirements
- Trusting AI with security-critical code without review

---

1. Start With a Written Plan (Seriously, Do This First)

Ask your AI to draft a **Markdown plan** of the feature you're building. Then make it better:

1. **Ask clarifying questions** about edge cases
2. **Have it critique its own plan** for gaps
3. **Regenerate an improved version**

Save the final plan as `instructions.md` and reference it in every prompt. This single step eliminates 80% of "the AI got confused halfway through" moments.

Real example:

```
Write a plan for adding rate limiting to our API. Include:
- Which endpoints need protection
- Storage mechanism for rate data
- Error responses and status codes
- Integration points with existing middleware

Now critique this plan. What did you miss?
```

---

2. Master the Edit-Test Loop

This is TDD but with an AI doing the implementation:

1. **Ask AI to write a failing test** that captures exactly what you want
2. **Review the test yourself** - make sure it tests the right behavior
3. **Then tell the AI: "Make this test pass"**
4. **Let the AI iterate** - it can run tests and fix failures automatically

The key is reviewing the test before implementation. A bad test will lead to code that passes the wrong requirements.

---

3. Demand Step-by-Step Reasoning

Add this to your prompts:

```
Explain your approach step-by-step before writing any code.
```

You'll catch wrong assumptions before they become wrong code. AI models that think out loud make fewer stupid mistakes.

---

4. Stop Dumping Context, Start Curating It

Large projects break AI attention. Here's how to fix it:

Use gitingest.com for Codebase Summaries

1. Go to gitingest.com
2. Enter your repo URL (or replace "github.com" with "gitingest.com" in any GitHub URL)
3. Download the generated text summary
4. Reference this instead of copy-pasting files

Instead of: Pasting 10 files into your prompt
Do this: "See attached codebase_summary.txt for project structure"

For Documentation: Use Context7 MCP or Alternatives for Live Docs

Context7 MCP keeps AI synced with the latest documentation by presenting the "Most Current Page" of your docs.

When to use: When your docs change frequently, reference the MCP connection rather than pasting outdated snippets each time.

---

5. Version Control Is Your Safety Net

- **Commit granularly** with `git add -p` so diffs stay readable
- **Never let uncommitted changes pile up**: clean git state makes it easier to isolate AI-introduced bugs and rollback cleanly
- **Use meaningful commit messages**: they help AI understand change context

---

6. Keep Prompts Laser-Focused

Bad: "Here's my entire codebase. Why doesn't authentication work?"

Good: "`@src/auth.rs` line 85 panics on `None` when JWT is malformed. Fix this and add proper error handling."

Specific problems get specific solutions. Vague problems get hallucinations.

Use your code's terminology in prompts: reference the exact identifiers from your codebase, not generic business terms. For example, call `createOrder()` and `processRefund()` instead of 'place order' or 'issue refund', or use `UserEntity` rather than 'account'. This precision helps the AI apply the correct abstractions and avoids mismatches between your domain language and code.

---

7. Re-Index After Big Changes

If you're using AI tools with project indexing, rebuild the index after major refactors. Out-of-date indexes are why AI "can't find" functions that definitely exist.

Most tools auto-index, but force a refresh when things seem off.

---

8. Use File References, Not Copy-Paste

Most AI editors support references like `@src/database.rs`. Use them instead of pasting code blocks.

Benefits:

- AI sees the current file state, not a stale snapshot
- Smaller token usage = better accuracy
- Less prompt clutter

Note: Syntax varies by tool (Forge uses `@`, some use `#`, etc.)

---

9. Let AI Write Tests, But You Write the Specs

Tell the AI exactly what to test:

```
For the new `validate_email` function, write tests for:
- Valid email formats (basic cases)
- Invalid formats (no @, multiple @, empty string)
- Edge cases (very long domains, unicode characters)
- Return value format (should be Result<(), ValidationError>)
```

AI is good at generating test boilerplate once you specify the cases.

---

10. Debug with Diagnostic Reports

When stuck, ask for a systematic breakdown:

```
Generate a diagnostic report:
1. List all files modified in our last session
2. Explain the role of each file in the current feature
3. Identify why the current error is occurring
4. Propose 3 different debugging approaches
```

This forces the AI to think systematically instead of guess-and-check.

---

11. Set Clear Style Guidelines

Give your AI a brief system prompt:

```
Code style rules:
- Use explicit error handling, no unwraps in production code
- Include docstrings for public functions
```

```
- Prefer composition over inheritance
- Keep functions under 50 lines
- Use `pretty_assertions` in test
- Be explicit about lifetimes in Rust
- Use `anyhow::Result` for error handling in services and repositories.
- Create domain errors using `thiserror`.
- Never implement `From` for converting domain errors, manually convert them
```

Consistent rules = consistent code quality.

---

12. Review Everything Like a Senior Engineer

Treat every AI change like a junior developer's PR:

Security Review:

- Check for injection vulnerabilities
- Verify input validation
- Look for hardcoded secrets

Performance Review:

- Watch for N+1 queries
- Check algorithm complexity
- Look for unnecessary allocations

Correctness Review:

- Test edge cases manually
- Verify error handling
- Check for off-by-one errors

The AI is smart but not wise. Your experience matters.

What Doesn't Work (Learn From My Mistakes)

The "Magic Prompt" Fallacy

There's no perfect prompt that makes AI never make mistakes. Better workflows beat better prompts.

Expecting Mind-Reading

AI can't infer requirements you haven't stated. "Make it production-ready" means nothing without specifics.

Trusting AI with Architecture Decisions

AI is great at implementing your design but terrible at high-level system design. You architect, AI implements.

Ignoring Domain-Specific Context

AI doesn't know your business logic, deployment constraints, or team conventions unless you tell it.

Controversial Take: AI Pair Programming Is Better Than Human Pair Programming

For most implementation tasks.

AI doesn't get tired, doesn't have ego, doesn't argue about code style, and doesn't judge your googling habits. It's like having a junior developer with infinite patience and perfect memory.

But it also doesn't catch logic errors, doesn't understand business context, and doesn't push back on bad ideas. You still need humans for the hard stuff.

Final Reality Check

AI coding tools can significantly boost productivity, but only if you use them systematically. The engineers seeing massive gains aren't using magic prompts they're using disciplined workflows.

Plan first, test everything, review like your production system depends on it (because it does), and remember: the AI is your intern, not your architect.

The future of coding isn't human vs AI it's humans with AI vs humans without it. Choose your side wisely.

Posted By

Forge
The Forge Team

Older post
First Experience Coding with DeepSeek-R1-0528

Recent Blog Posts

May 31, 2025

What Actually Works: 12 Lessons from AI Pair…

Field-tested practices for productive AI-assisted development. Real lessons from 6…

Forge

May 29, 2025

First Experience Coding with DeepSeek-R1-0528

I spent time testing DeepSeek-R1-0528's impressive capabilities and challenging latency via OpenRoute…

Forge

May 25, 2025

May 22, 2025

Claude 4 vs Gemini 2.5 Pro: A Developer's Deep Dive…

After extensive testing with real-world coding challenges, I compared Claude Sonnet 4 and…

Forge

Claude 4 First Impressions: A Developer's Perspective

Claude 4 achieves 72.7% on SWE-bench Verified, surpassing OpenAI's latest models. After 24…

Forge