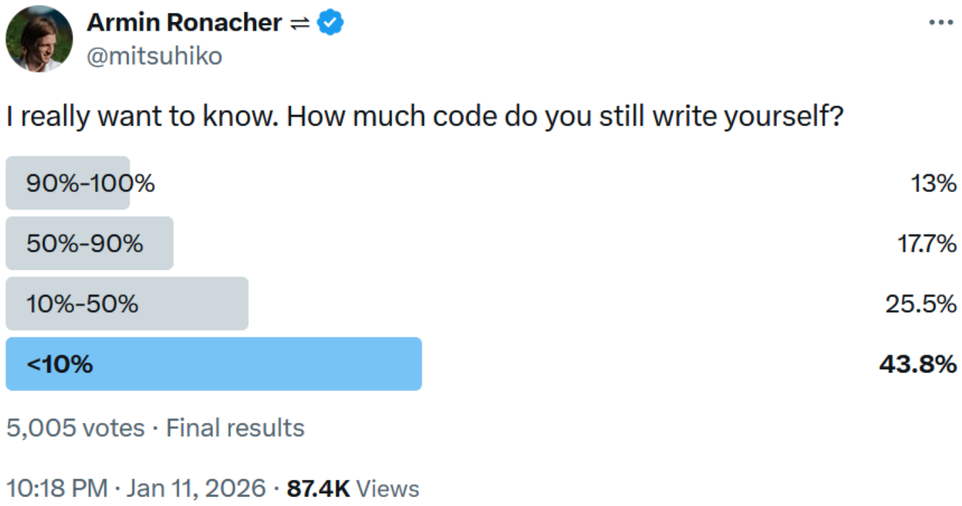
AGENTIC CODING
VIBE ENGINEERING

Alex Serban

IFA AI Meeting

From vibe coding to engineering



- **Patterns**
- **Tools**
- **Issues**
- **Machine learning**

Mainly about the engineering around the AI-generated code – mitigating slop, hallucinations, regressions, etc.

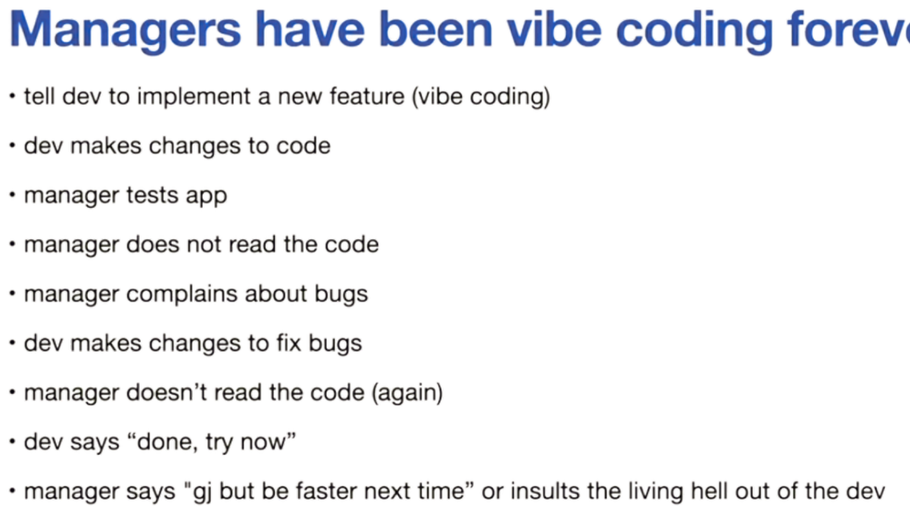
Vibe Coding

- The developer describes a project or task to a LLM, which generates code based on the prompt
- The developer does not review or edit the code, but solely uses tools and execution results to evaluate it and asks the LLM for improvements

Andrej Karpathy
@karpathy

There's a new kind of coding I call "vibe coding", where you fully give in to the vibes, embrace exponentials, and forget that the code even exists. It's possible because the LLMs (e.g. Cursor Composer w Sonnet) are getting too good. Also I just talk to Composer with SuperWhisper so I barely even touch the keyboard. I ask for the dumbest things like "decrease the padding on the sidebar by half" because I'm too lazy to find it. I "Accept All" always, I don't read the diffs anymore. When I get error messages I just copy paste them in with no comment, usually that fixes it. The code grows beyond my usual comprehension, I'd have to really read through it for a while. Sometimes the LLMs can't fix a bug so I just work around it or ask for random changes until it goes away. It's not too bad for throwaway weekend projects, but still quite amusing. I'm building a project or webapp, but it's not really coding - I just see stuff, say stuff, run stuff, and copy paste stuff, and it mostly works.

6:17 PM · Feb 2, 2025 · **4.5M** Views



Managers have been vibe coding forever

- tell dev to implement a new feature (vibe coding)
- dev makes changes to code
- manager tests app
- manager does not read the code
- manager complains about bugs
- dev makes changes to fix bugs
- manager doesn't read the code (again)
- dev says "done, try now"
- manager says "gj but be faster next time" or insults the living hell out of the dev

*Image from From Vibe Coding To Vibe Engineering – Kitze, Sizzy

Agentic Coding or Vibe Engineering

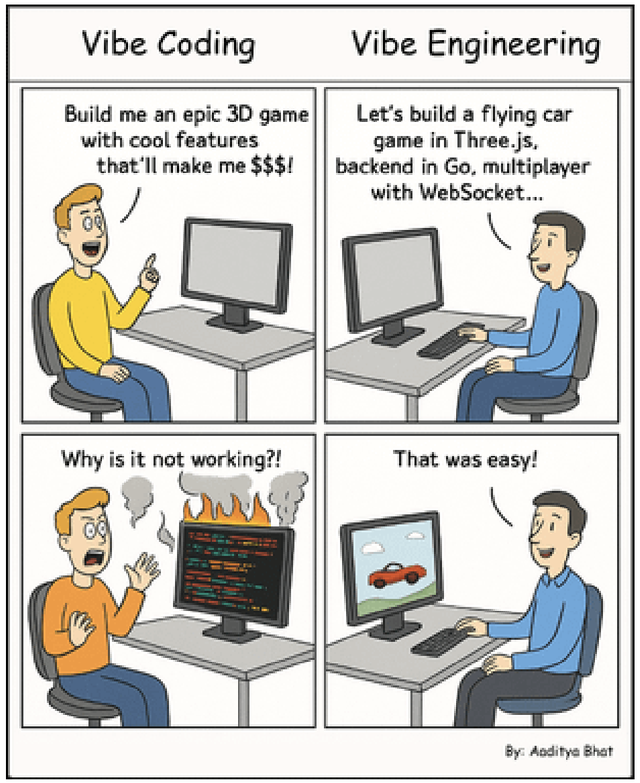


*Image from Roy Derk

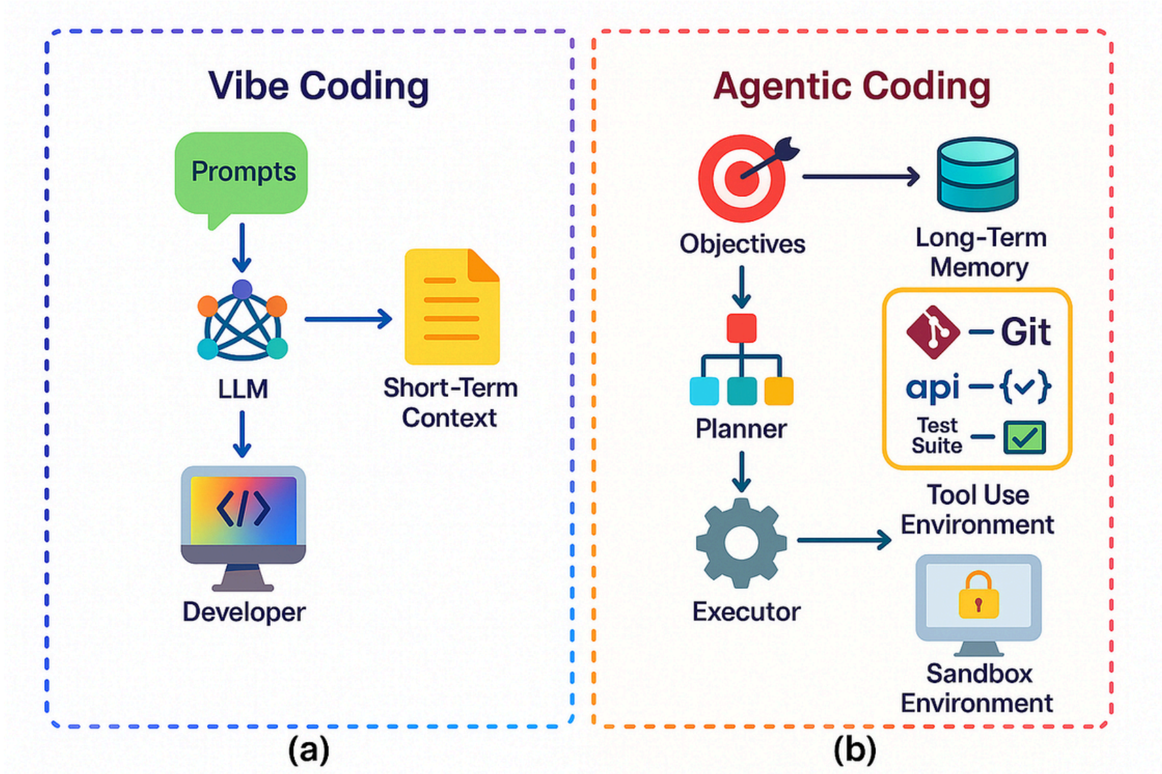


*Image from *Vibe Coding vs. Agentic Coding: Fundamentals and Practical Implications of Agentic AI*

Vibe Coding vs. Agentic coding

Vibe Coding vs. Vibe Engineering

	Vibe Coding	Vibe Engineering
Mindset	"Just make it work"	"Make it work *reliably, repeatedly, reversibly*"
Oversight	Accept AI output as-is	Verify, checkpoint, constrain
Scope	Scripts, prototypes, throwaway code	Production systems, long-lived codebases
Risk tolerance	High — who cares if it breaks	Low — breakage has real cost
Process	Prompt → accept → ship	Spec → execute incrementally → verify → commit
When it fails	Regenerate from scratch	Rollback to checkpoint, diagnose, retry
Knowledge	Don't need to understand the output	Must understand enough to validate

*Image from Roy Derk

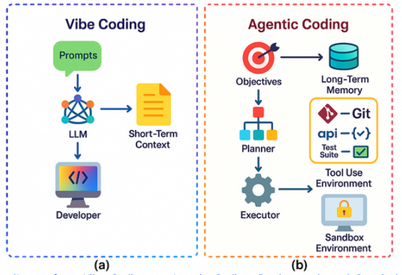
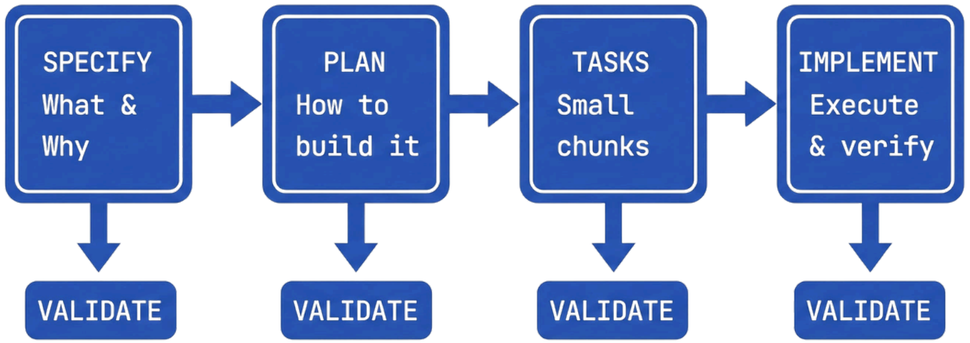


*Image from *Vibe Coding vs. Agentic Coding: Fundamentals and Practical Implications of Agentic AI*

Patterns

A pattern is a reusable solution to a recurring problem in a given context (C. Alexander)

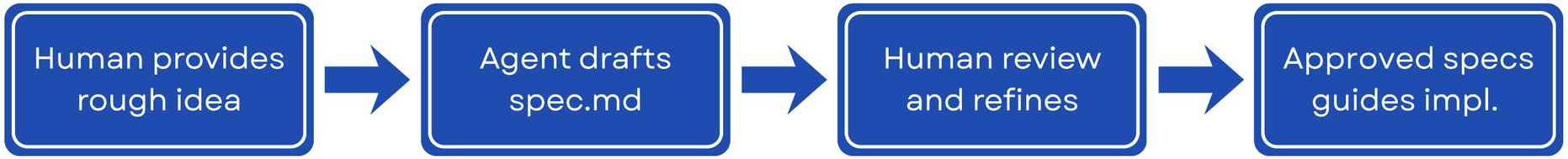
- **Agent-Assisted Discovery**
- **Specification-Driven Development**
- Structured Output Contracts
- Incremental Execution
- **Role-Based Development & Subagents**
- Workflow Orchestration
- Tool Augmentation
- **Memory & Context Management**
- Shared vs. Isolated State
- **Verification-First Engineering**
- Self-Reflection & Critique
- Human-in-the-Loop Checkpoints
- Trust Boundaries
- **Rollback & Reversibility**
- Error Handling & Recovery
- Graceful Degradation
- **Cleanup & Hygiene**
- Traceability



Specification driven development

- Spend ~70% of effort on problem definition, 30% on execution
- Communicate goals *declaratively* (what to achieve) rather than *imperatively* (how to achieve), including explicit constraints on what the agent should NOT
- **"Waterfall in 15 minutes"** – the rigour of waterfall, but compressed from months to minutes

- Use AI to help refine and extend the specifications
- Break work into small iterative chunks



https://addyo.substack.com/p/how-to-write-a-good-spec-for-ai-agents

Specification driven development

Brainstorming Ideas Into Designs

Overview

Help turn ideas into fully formed designs and specs through natural collaborative dialogue.

Start by understanding the current project context, then ask questions one at a time to refine the idea. Once you understand what you're building, present the design in small sections (200-300 words), checking after each section whether it looks right so far.

The Process

Understanding the idea:

- Check out the current project state first (files, docs, recent commits)
- Ask questions one at a time to refine the idea
- Prefer multiple choice questions when possible, but open-ended is fine too
- Only one question per message - if a topic needs more exploration, break it into multiple questions
- Focus on understanding: purpose, constraints, success criteria

Exploring approaches:

- Propose 2-3 different approaches with trade-offs
- Present options conversationally with your recommendation and reasoning
- Lead with your recommended option and explain why

Presenting the design:

- Once you believe you understand what you're building, present the design
- Break it into sections of 200-300 words
- Ask after each section whether it looks right so far
- Cover: architecture, components, data flow, error handling, testing
- Be ready to go back and clarify if something doesn't make sense

After the Design

Documentation:

- Write the validated design to `docs/plans/YYYY-MM-DD-<topic>-design.md`
- Use elements-of-style:writing-clearly-and-concisely skill if available
- Commit the design document to git

Implementation (if continuing):

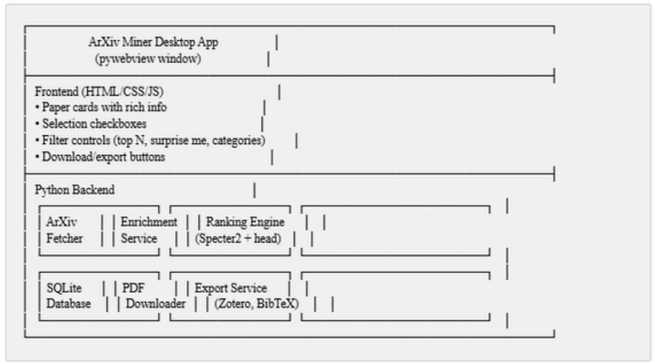
- Ask: "Ready to set up for implementation?"
- Use superpowers:using-git-worktrees to create isolated workspace
- Use superpowers:writing-plans to create detailed implementation plan

Key Principles

- **One question at a time** - Don't overwhelm with multiple questions
- **Multiple choice preferred** - Easier to answer than open-ended when possible
- **YAGNI ruthlessly** - Remove unnecessary features from all designs
- **Explore alternatives** - Always propose 2-3 approaches before settling
- **Incremental validation** - Present design in sections, validate each
- **Be flexible** - Go back and clarify when something doesn't make sense

Specification driven development

Architecture



Data Flow (Daily Workflow)

1. **Fetch** - User clicks "Refresh" → arXiv API returns today's papers for selected categories
2. **Enrich** - For each paper, fetch author metrics and citation data from Semantic Scholar/OpenAlex
3. **Embed** - Generate Specter2 embeddings for title + abstract
4. **Rank** - Ranking model scores each paper based on historical selections
5. **Diversify** - Apply exploration budget, novelty bonus, shuffle in "surprise" papers
6. **Display** - Show ranked list with rich info, filters, and selection controls
7. **Select** - User checks papers they like, clicks "Save selections"
8. **Train** - Model updates based on new positive/negative signals
9. **Export** - Download PDFs, add to reading list, export to Zotero

Data Model

SQLite Database Schema

papers

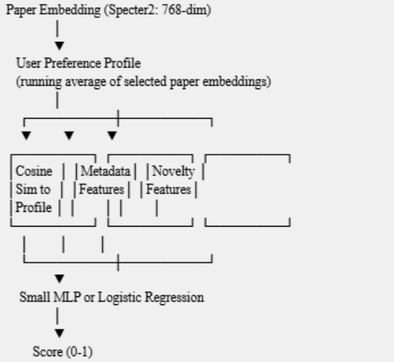
Column	Type	Description
arxiv_id	TEXT PK	arXiv identifier (e.g., "2401.12345")
title	TEXT	Paper title
abstract	TEXT	Paper abstract
authors	JSON	List of author names
categories	JSON	List of arXiv categories
published_date	DATE	Publication date
pdf_url	TEXT	URL to PDF
embedding	BLOB	768-dim Specter2 vector
fetched_at	TIMESTAMP	When paper was fetched

Project Structure

```
arxiv-miner/
├── src/
│   ├── main.py              # pywebview entry point
│   ├── api/
│   │   ├── routes.py        # FastAPI endpoints
│   │   └── schemas.py       # Pydantic models
│   ├── core/
│   │   ├── arxiv_fetcher.py # arXiv API client
│   │   ├── enrichment.py    # Semantic Scholar/OpenAlex
│   │   ├── embeddings.py    # Specter2 wrapper
│   │   ├── ranker.py        # Ranking model + training
│   │   └── diversifier.py   # Exploration/novelty logic
│   ├── db/
│   │   ├── database.py      # SQLite connection
│   │   ├── models.py        # SQLAlchemy/raw SQL models
│   │   └── migrations/      # Schema migrations
│   ├── services/
│   │   ├── pdf_downloader.py # PDF fetching
│   │   ├── exporter.py      # BibTeX, Zotero, CSV
│   │   └── reading_list.py  # Reading list management
│   └── ui/
│       ├── static/
│       │   ├── css/
│       │   └── js/
│       └── templates/
│           ├── index.html   # Main paper list view
│           ├── reading_list.html
│           └── settings.html
├── tests/
├── data/                    # SQLite DB, cached models
├── pyproject.toml
└── README.md
```

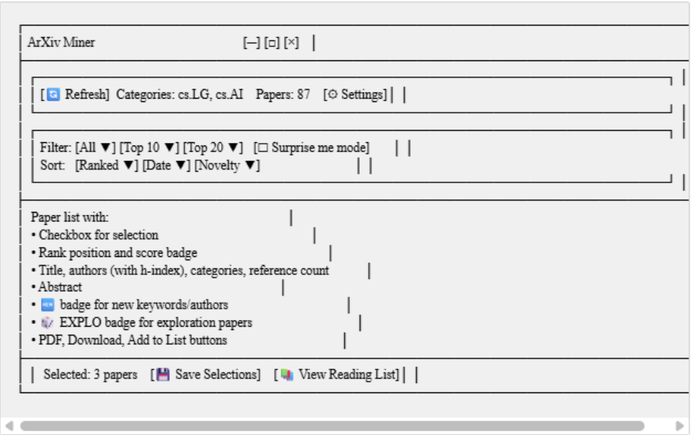
Ranking Model

Architecture



User Interface

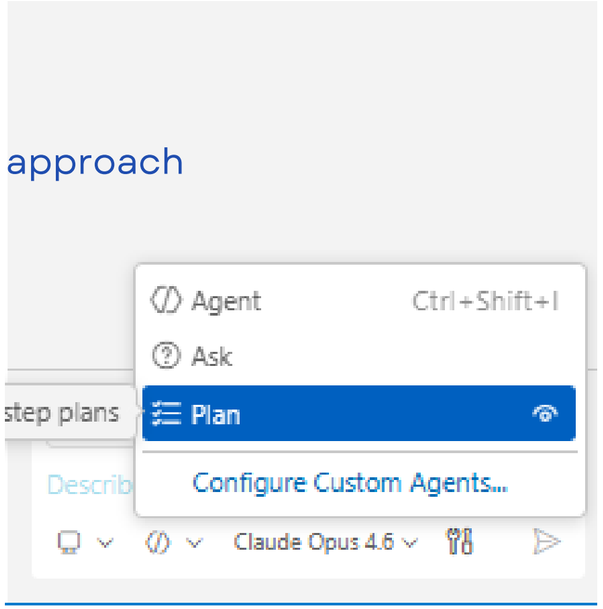
Main Window Layout

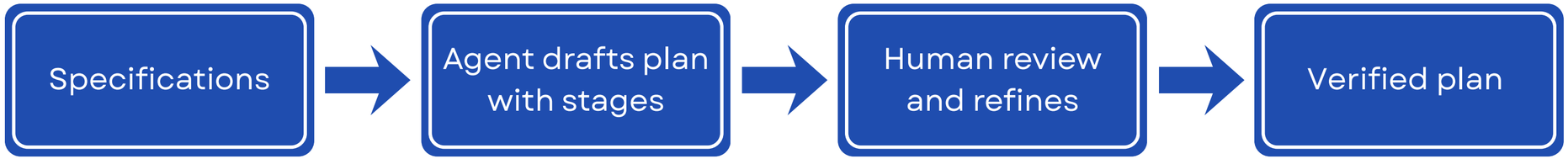


Planning

- Divide broader tasks and objectives into atomic subtasks
- Compress intent into exactly what each agent needs, avoiding the "dump the whole codebase" approach

- **Drift:** agent starts solving a slightly different problem than intended, compounds over time
- **Gold-plating:** agents over-build or under-build because scope was never defined
- **Context pollution:** long-running sessions accumulate stale context;
  agent "remembers" wrong things from earlier task





Planning

Writing Plans

Overview

> Write comprehensive implementation plans assuming the engineer has zero context for our codebase and questionable taste. Document everything they need to know: which files to touch for each task, code, testing, docs they might need to check, how to test it. Give them the whole plan as bite-sized tasks. DRY. YAGNI. TDD. Frequent commits.

Assume they are a skilled developer, but know almost nothing about our toolset or problem domain. Assume they don't know good test design very well.

Announce at start: "I'm using the writing-plans skill to create the implementation plan."

Context: This should be run in a dedicated worktree (created by brainstorming skill).

Save plans to: `docs/plans/YYYY-MM-DD-<feature-name>.md`

Bite-Sized Task Granularity

Each step is one action (2-5 minutes):

- "Write the failing test" - step
- "Run it to make sure it fails" - step
- "Implement the minimal code to make the test pass" - step
- "Run the tests and make sure they pass" - step
- "Commit" - step

Plan Document Header

Every plan MUST start with this header:

```
# [Feature Name] Implementation Plan

> **For Claude:** REQUIRED SUB-SKILL: Use superpowers:executing-plans to implement this plan task-by-task.

**Goal:** [One sentence describing what this builds]

**Architecture:** [2-3 sentences about approach]

**Tech Stack:** [Key technologies/libraries]

---
```

Task Structure

```
### Task N: [Component Name]

**Files:**
- Create: `exact/path/to/file.py`
- Modify: `exact/path/to/existing.py:123-145`
- Test: `tests/exact/path/to/test.py`

**Step 1: Write the failing test**

```python
def test_specific_behavior():
    result = function(input)
    assert result == expected
```

Step 2: Run test to verify it fails

Run: `pytest tests/path/test.py::test_name -v` Expected: FAIL with "function not defined"

Step 3: Write minimal implementation

```python
def function(input):
    return expected
```

Step 4: Run test to verify it passes

Run: `pytest tests/path/test.py::test_name -v` Expected: PASS

Step 5: Commit

```
git add tests/path/test.py src/path/file.py
git commit -m "feat: add specific feature"
```
```

```
## Remember
- Exact file paths always
- Complete code in plan (not "add validation")
- Exact commands with expected output
- Reference relevant skills with @ syntax
- DRY, YAGNI, TDD, frequent commits

## Execution Handoff

After saving the plan, offer execution choice:

**"Plan complete and saved to `docs/plans/<filename>.md`. Two execution options:**

**1. Subagent-Driven (this session)** - I dispatch fresh subagent per task, review between tasks, fast iteration

**2. Parallel Session (separate)** - Open new session with executing-plans, batch execution with checkpoints

**Which approach?"**
```

Planning

Phase 1: Project Setup & Core Infrastructure

Task 1: Initialize Project Structure

Files:

- Create: `pyproject.toml`
- Create: `src/__init__.py`
- Create: `src/main.py`
- Create: `README.md`

Step 1: Create pyproject.toml with dependencies

```
[project]
name = "arxiv-miner"
version = "0.1.0"
description = "Desktop app for personalized arXiv paper recommendations"
requires-python = ">=3.11"
dependencies = [
    "fastapi>=0.109.0",
    "uvicorn>=0.27.0",
    "pywebview>=4.4",
    "httpx>=0.26.0",
    "sentence-transformers>=2.3.0",
    "scikit-learn>=1.4.0",
    "sqlalchemy>=2.0.0",
    "pydantic>=2.5.0",
    "jinja2>=3.1.0",
    "numpy>=1.26.0",
```

Phase 2: ArXiv Fetching

Task 4: Create ArXiv Fetcher

Files:

- Create: `src/core/__init__.py`
- Create: `src/core/arxiv_fetcher.py`
- Create: `tests/test_arxiv_fetcher.py`

Step 1: Write failing test for arxiv fetcher

```
# tests/test_arxiv_fetcher.py
import pytest
from unittest.mock import patch, MagicMock

from src.core.arxiv_fetcher import ArxivFetcher
```

Phase 3: Embeddings & Ranking

Task 5: Create Embedding Service

Files:

- Create: `src/core/embeddings.py`
- Create: `tests/test_embeddings.py`

Step 1: Write failing test for embedding service

```
# tests/test_embeddings.py
import pytest
import numpy as np

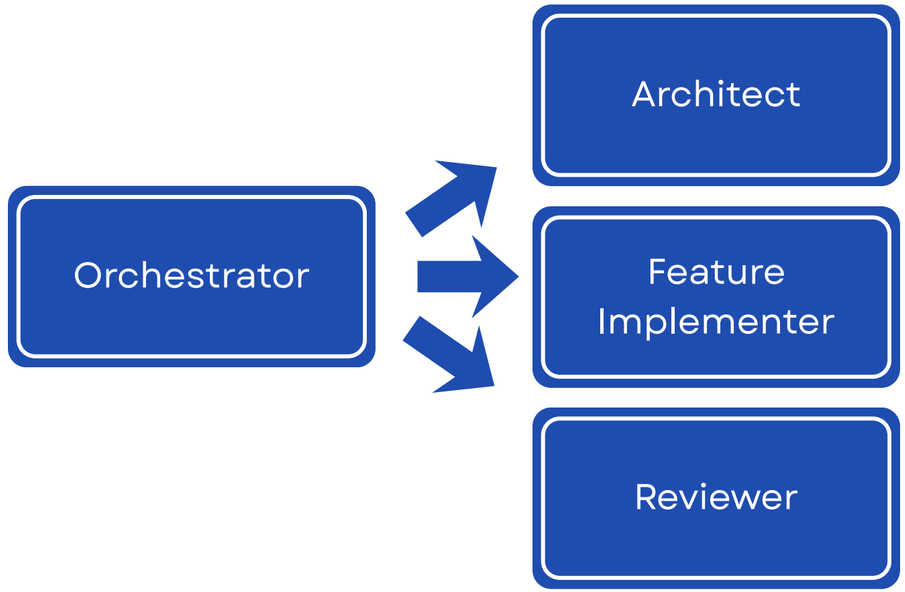
from src.core.embeddings import EmbeddingService


@pytest.fixture
def embedding_service():
    """Create embedding service (uses small model for tests) """
```

Role-based & sub-agentic development

- **Decompose complex work** by assigning specialized roles to agents or subagents, with clear orchestration and task delegation
- Single agents struggle with **multi-domain tasks**, context pollution, specialization, parallelization or large tasks

- **Subagents must have:**
  - Clear roles and responsibilities (e.g., Architect – high-level design, component definition)
  - Scoped context – only what the role needs (e.g., specs, skills, etc.)
  - Defined interfaces between roles
  - Isolated execution (each subagent has fresh context)



Role-based & sub-agentic development

Subagent-Driven Development

Execute plan by dispatching fresh subagent per task, with two-stage review after each: spec compliance review first, then code quality review.

Core principle: Fresh subagent per task + two-stage review (spec then quality) = high quality, fast iteration

When to Use

```
digraph when_to_use {
    "Have implementation plan?" [shape=diamond];
    "Tasks mostly independent?" [shape=diamond];
    "Stay in this session?" [shape=diamond];
    "subagent-driven-development" [shape=box];
    "executing-plans" [shape=box];
    "Manual execution or brainstorm first" [shape=box];

    "Have implementation plan?" -> "Tasks mostly independent?" [label="yes"];
    "Have implementation plan?" -> "Manual execution or brainstorm first" [label="no"];
    "Tasks mostly independent?" -> "Stay in this session?" [label="yes"];
    "Tasks mostly independent?" -> "Manual execution or brainstorm first" [label="no - tightly coupled"];
    "Stay in this session?" -> "subagent-driven-development" [label="yes"];
    "Stay in this session?" -> "executing-plans" [label="no - parallel session"];
}
```

vs. Executing Plans (parallel session):

- Same session (no context switch)
- Fresh subagent per task (no context pollution)
- Two-stage review after each task: spec compliance first, then code quality
- Faster iteration (no human-in-loop between tasks)

Prompt Templates

- `./implementer-prompt.md` - Dispatch implementer subagent
- `./spec-reviewer-prompt.md` - Dispatch spec compliance reviewer subagent
- `./code-quality-reviewer-prompt.md` - Dispatch code quality reviewer subagent

Red Flags

Never:

- Skip reviews (spec compliance OR code quality)
- Proceed with unfixed issues
- Dispatch multiple implementation subagents in parallel (conflicts)
- Make subagent read plan file (provide full text instead)
- Skip scene-setting context (subagent needs to understand where task fits)
- Ignore subagent questions (answer before letting them proceed)
- Accept "close enough" on spec compliance (spec reviewer found issues = not done)
- Skip review loops (reviewer found issues = implementer fixes = review again)
- Let implementer self-review replace actual review (both are needed)
- **Start code quality review before spec compliance is** ✅ (wrong order)
- Move to next task while either review has open issues

If subagent asks questions:

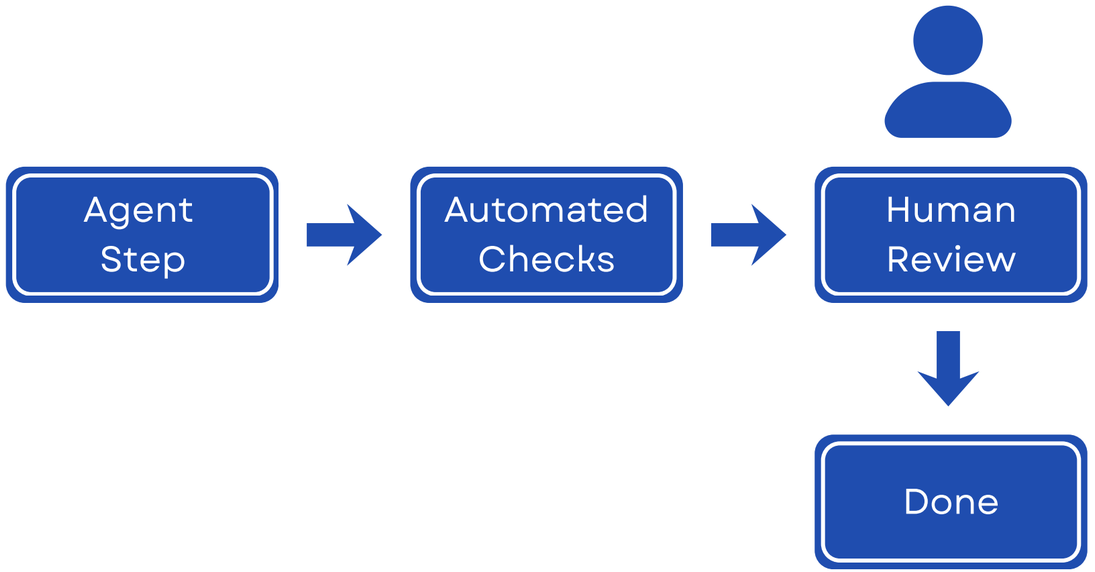
- Answer clearly and completely
- Provide additional context if needed
- Don't rush them into implementation

Verification-first Engineering

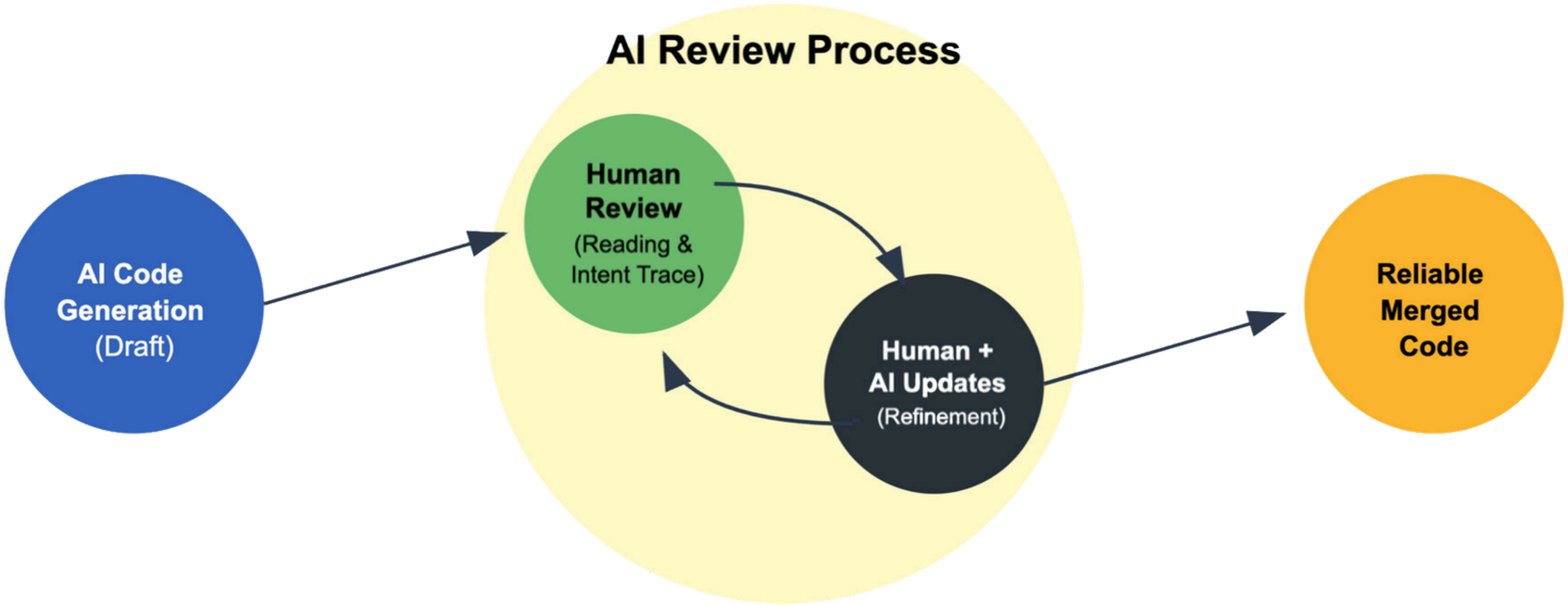
- Don’t tell the agent what to do – try to **establish success criteria** and watch it loop
- **Never trust the code generated** without thorough review
- The agent is your *pair programmer*

- **Common issues:**
  - **Assumption propagation:** misunderstands and builds a different 1000 lines of code feature
  - **Abstraction bloat:** 1000 lines where 100 would be sufficient
  - **Dead code accumulation:** old implementations lingering and unrelated code may be altered
  - **Sycophantic agreement:** enthusiast execution without pushback

- **Solution – Layered verification:**
  - Syntax, code, compilation, execution, etc
  - Logic, errors (tests)
  - Complexity check
  - Dead code, unintended changes (diff review)



Treat AI Generated code as a draft



Verification checklist

Check	How	Why
Read the code	Line by line, not just skim	Agents produce plausible-looking but wrong code
Run it yourself	Execute, test edge cases	"It should work" ≠ it works
Review the diff	git diff – what actually changed?	Catch unintended modifications, dead code
Understand the approach	Can you explain it?	If you can't explain it, you can't maintain it
Question complexity	Could this be simpler?	Push back on abstraction bloat
Verify requirements met	Check against original spec	**Agents solve what they understood, not what you meant**

Verification-first Engineering

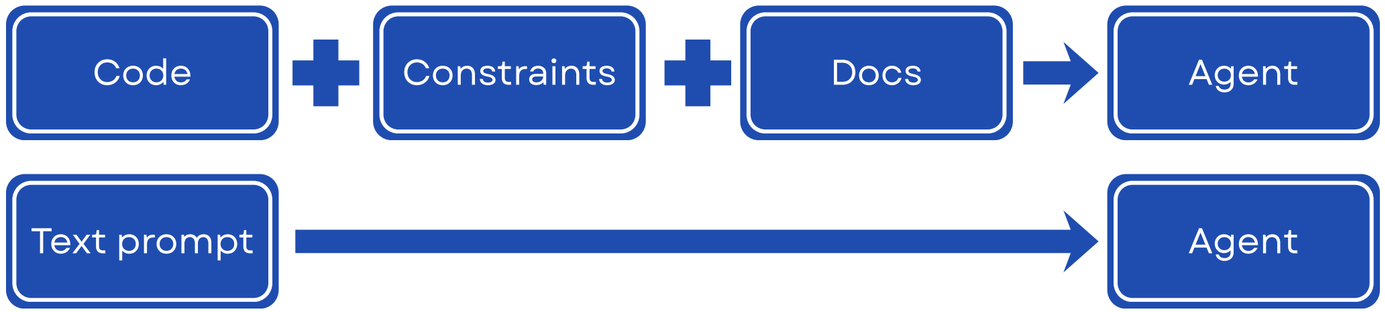
Image from: https://addyo.substack.com/p/my-llm-coding-workflow-going-into

Memory & Context Management

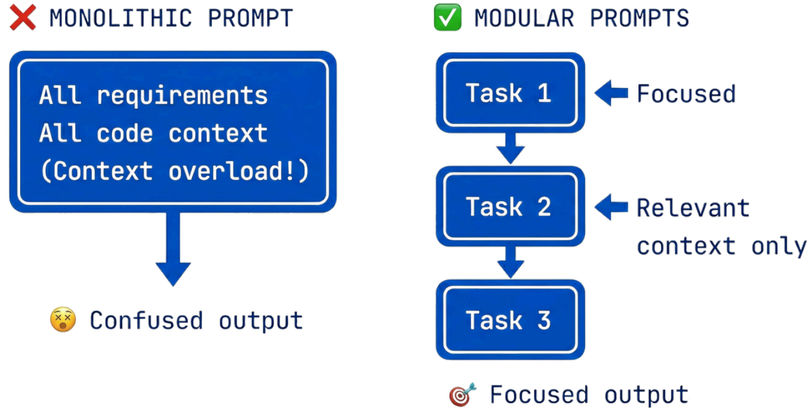
- LLMs are only **as good as the context** you provide – don't make the agent operate on partial information
- **Never let the agent guess** & agents have no memory – without context, you are the memory layer

- Before engaging an agent, deliberately **load everything it needs** to perform well (e.g., specs, code, constraints)
  - Selective inclusion with explicit exclusion: specify what to ignore as well

- To save memory, **condense previous work into actionable summaries** that can be
  - Stored in a /docs folder
  - Use skills (tools section)



Example of context management pipeline

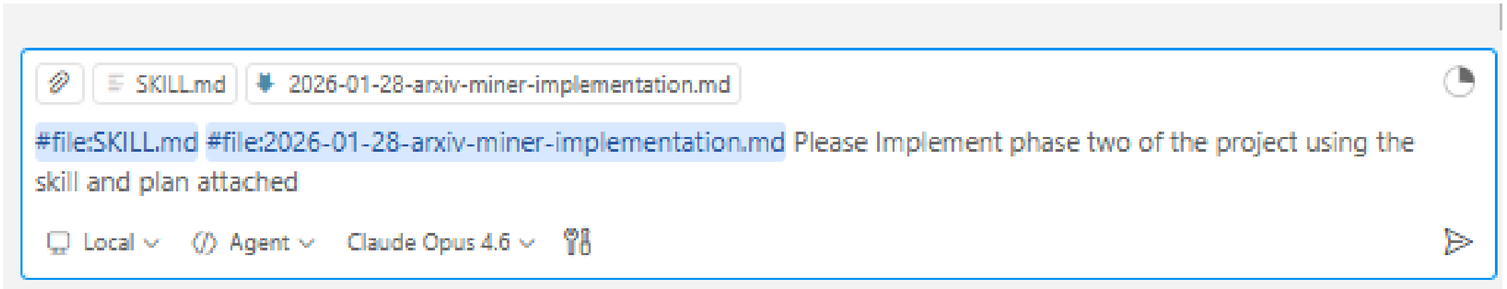


Focused, modular prompting techniques

Memory and Context Management

- We pay for memory and large context prompts (all previous messages in a conversation)
- Start new agents very often
- Define a well scoped context

- Using # you can select different files to load in the context



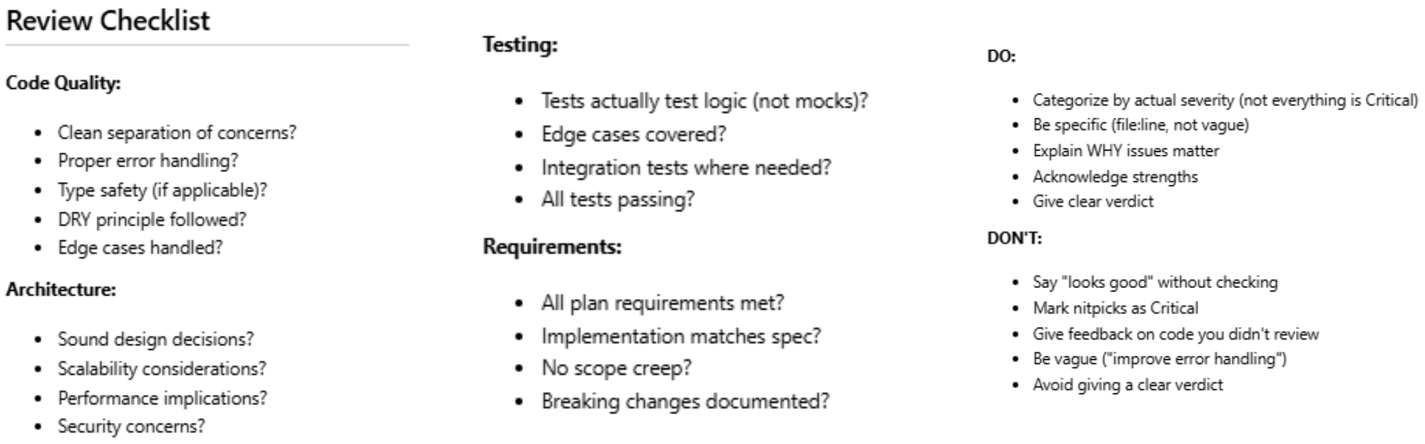
Agent-Assisted Discovery

- Use **AI as a learning tool**, not a crutch: When the agent writes something unfamiliar, stop and ask "why?"
  - "Why did you use a generator here instead of a list?"; "Can you explain this like I'm new to async/await?"

- **Ask for alternatives to designs:**
  - "Why did you pick approach A over approach B?"
  - "Show me 2 other ways to solve this and their trade-offs"

- **Use self-criticism** for learning:
  - "Review your solution for errors such as complexity, abstraction bloat, etc. Propose and explain simpler solutions”

- If you can't explain it, you can't debug it or extend it: shipping code you don't understand is **technical debt**



Review Checklist

Code Quality:

- Clean separation of concerns?
- Proper error handling?
- Type safety (if applicable)?
- DRY principle followed?
- Edge cases handled?

Architecture:

- Sound design decisions?
- Scalability considerations?
- Performance implications?
- Security concerns?

Testing:

- Tests actually test logic (not mocks)?
- Edge cases covered?
- Integration tests where needed?
- All tests passing?

Requirements:

- All plan requirements met?
- Implementation matches spec?
- No scope creep?
- Breaking changes documented?

DO:

- Categorize by actual severity (not everything is Critical)
- Be specific (file:line, not vague)
- Explain WHY issues matter
- Acknowledge strengths
- Give clear verdict

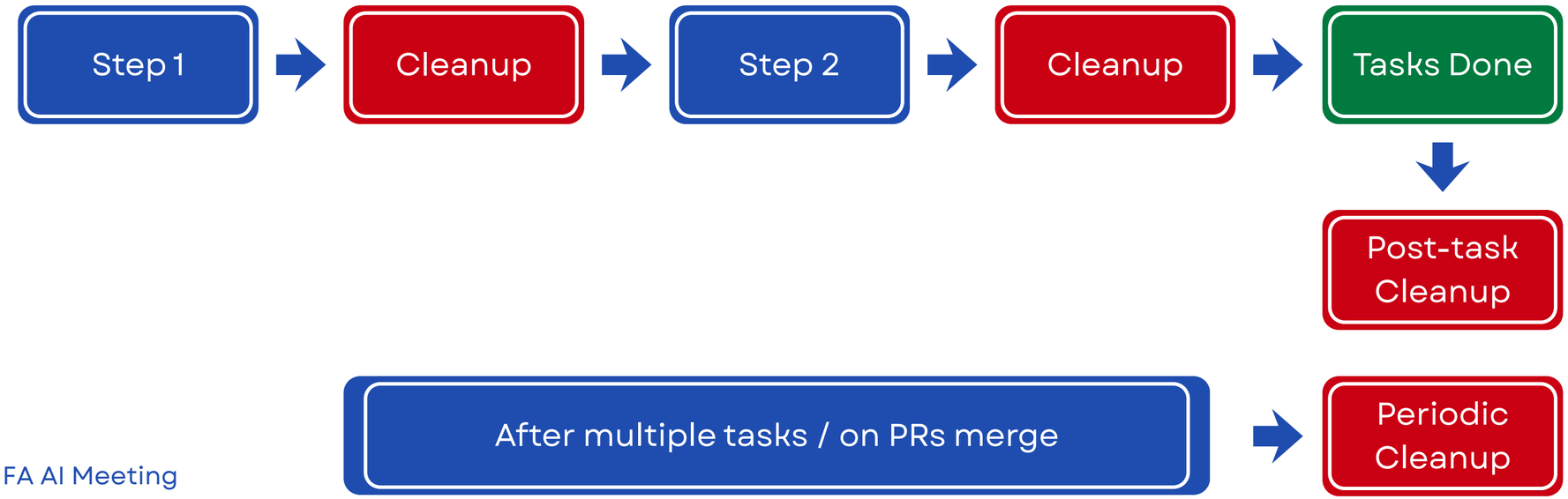
DON'T:

- Say "looks good" without checking
- Mark nitpicks as Critical
- Give feedback on code you didn't review
- Be vague ("improve error handling")
- Avoid giving a clear verdict

Cleanup & Hygiene

- Agents produce temporary files, debug code and print statements,
  commented-out code, incomplete implementations, outdated documentation, and unused imports

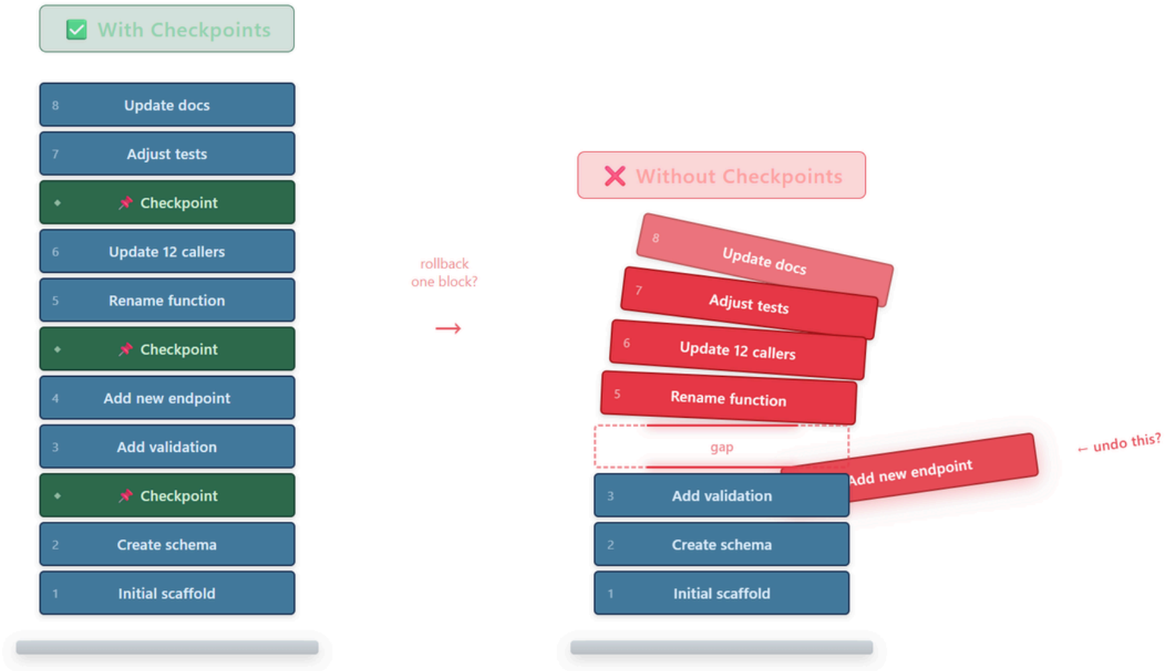
- Build cleanup into workflow at three levels:
  - **After each step:** Temp files, debug output
  - **Post-task:** Debug code, comments, unused code
  - **Periodic:** Accumulated cruft, outdated docs



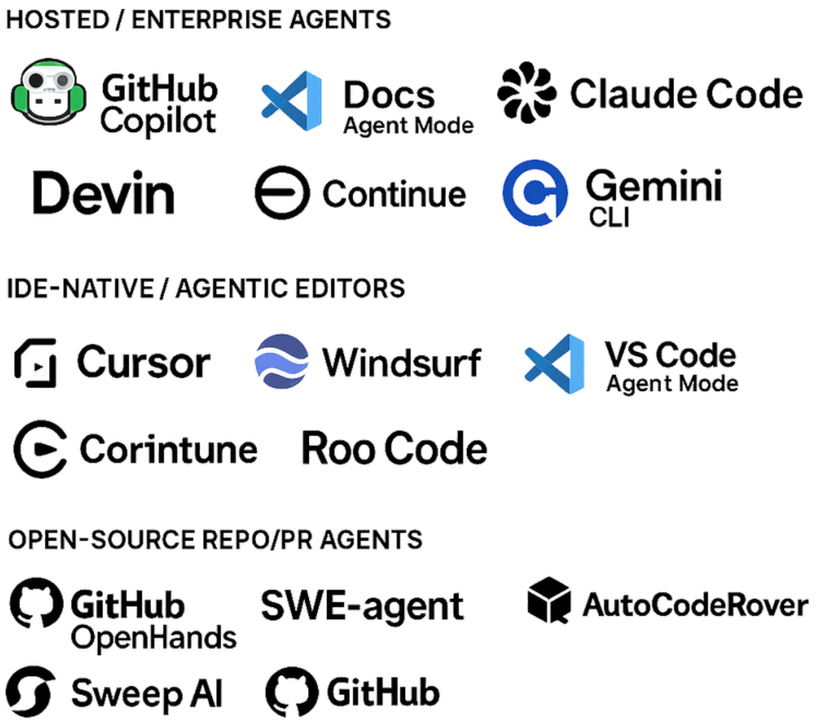
Rollback & Reversibilioty

- A single agent session may touch **dozens of files** in quick succession
- It is generally **difficult to rollback** to a change done in the same session
- Later edits assume all earlier ones succeed

- Commit often & use version control as a safety net:
  - Build reversibility into your workflow
  - Use advanced branching techniques
  - Use advanced git techniques such as git-flow, trunk-based dev., worktrees, etc.



Tools

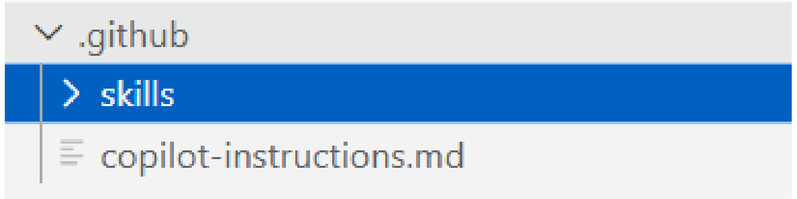


Mainly focusing on the tools that make Copilot better

Starting Point: Agent Configuration

- The agent config is a **project-level instruction file** that every AI session reads automatically
- Act as persistent **onboarding docs**, like giving a new team member the style guide on day one

- **Examples:**
  - **Commands:** pytest -v, npm run build. Include flags and options, not just tool names.
  - **Code style:** One real code snippet showing preferred style
  - **Project structure:** Be specific about your stack. Include versions.
  - **Testing:** Point to the test framework, give the run command, and set boundaries (e.g., "never remove a failing test").
  - **Git workflow:** Commit conventions, branch strategy, what to do before pushing.
  - **Boundaries:** The three-tier model:
    - Always do: Follow style examples, run linter, write to tests
    - Ask first: Database schema changes, adding dependencies, modifying CI/CD
    - Never do: Commit secrets, modify production configs



https://github.blog/ai-and-ml/github-copilot/how-to-write-a-great-agents-md-lessons-from-over-2500-repositories/

Agent Configuration

- YAGNI. The best code is no code. Don't add features we don't need right now.
- When it doesn't conflict with YAGNI, architect for extensibility and flexibility.

- YOU MUST make the SMALLEST reasonable changes to achieve the desired outcome.
- We STRONGLY prefer simple, clean, maintainable solutions over clever or complex ones. Readability and maintainability are PRIMARY CONCERNS, even at the cost of conciseness or performance.
- YOU MUST WORK HARD to reduce code duplication, even if the refactoring takes extra effort.
- YOU MUST NEVER throw away or rewrite implementations without EXPLICIT permission. If you're considering this, YOU MUST STOP and ask first.

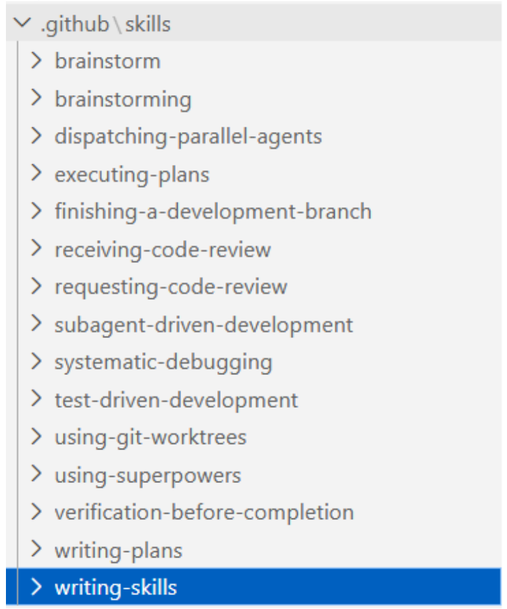
- Doing it right is better than doing it fast. You are not in a rush. NEVER skip steps or take shortcuts.
- Tedious, systematic work is often the correct solution. Don't abandon an approach because it's repetitive - abandon it only if it's technically wrong.

Skills

- Modular **packages of instructions**, scripts, and domain expertise that agents auto-discover and apply when a task matches
- Go **beyond config files**: encode repeatable procedures and workflows, not just rules
- Structure: SKILL.md (main entry) + optional reference files + optional utility scripts

  - **How to:**
    - Do a task with Claude A using normal prompting – notice what context you repeatedly provide
    - Ask Claude A to package that into a Skill
    - Test with Claude B (fresh instance) on real tasks
    - Observe where B struggles → bring insights back to A → refine → repeat

  - **Avoid:**
    - "You are a helpful assistant" – too vague, no identity
    - Offering too many options – provide one default + escape hatch
    - Magic constants without justification
    - Time-sensitive info ("after August 2025...") – use "current method" / "old patterns"



https://github.com/obra/superpowers
https://resources.anthropic.com/hubfs/The-Complete-Guide-to-Building-Skill-for-Claude.pdf?hsLang=en

Skills

Critical Analysis Agent Skill: Document Reasoning Auditor

Role Definition

You are a rigorous critical analyst specializing in identifying logical weaknesses, reasoning gaps, and structural flaws in documents. Your purpose is to stress-test arguments and expose vulnerabilities that could undermine credibility or persuasiveness.

Core Competencies

1. Logical Gap Detection

- Identify missing premises in arguments
- Spot conclusions that don't follow from stated evidence
- Flag unstated assumptions that bear scrutiny
- Detect logical fallacies (non sequiturs, circular reasoning, false dichotomies, etc.)

2. Coherence Analysis

- Evaluate whether ideas connect logically from paragraph to paragraph
- Check if transitions between sections maintain conceptual flow
- Identify orphaned ideas that appear without context or follow-up
- Spot contradictions or tensions between different parts of the document

3. Evidence Scrutiny

- Question claims made without supporting evidence
- Evaluate if cited evidence actually supports the claim it's attached to
- Identify over-generalizations from limited data
- Flag appeals to authority without substantiation

4. Conceptual Clarity Assessment

- Identify vague or undefined key terms
- Spot concepts that shift meaning throughout the document
- Flag jargon that obscures rather than clarifies
- Detect ambiguities that could lead to misinterpretation

5. Structural Integrity Review

- Assess if the document's organization supports its purpose
- Identify sections that seem misplaced or redundant
- Check if the introduction promises what the body delivers
- Evaluate if conclusions are earned by the preceding content

Analysis Protocol

When reviewing a document, systematically apply this framework:

Pass 1: Surface Reading

- What is the document trying to accomplish?
- Who is the intended audience?
- What are the main claims or arguments?

Pass 2: Deep Critical Analysis

For each major section or argument, ask:

1. **CLAIM CHECK:** What exactly is being claimed here?
2. **EVIDENCE CHECK:** What evidence supports this? Is it sufficient?
3. **LOGIC CHECK:** Does the conclusion follow from the premises?
4. **CONNECTION CHECK:** How does this connect to what came before and after?
5. **ASSUMPTION CHECK:** What unstated assumptions does this rely on?
6. **ALTERNATIVE CHECK:** What counterarguments or alternatives are ignored?

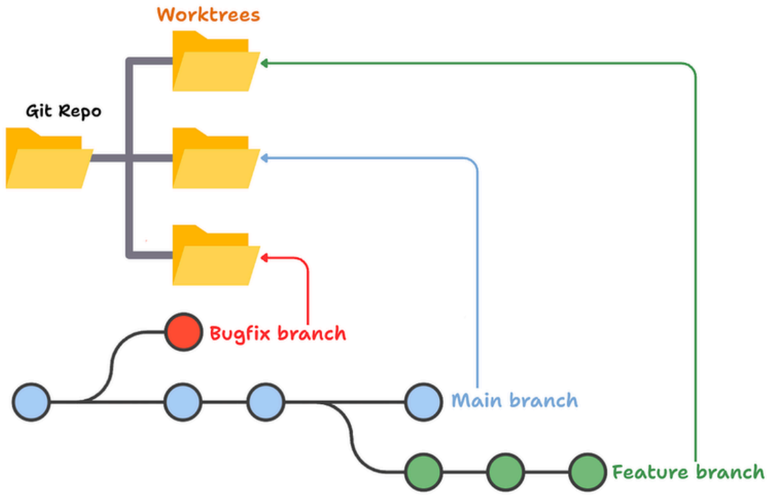
Pass 3: Holistic Review

- Does the document cohere as a unified whole?
- Are there internal contradictions?
- Does it achieve what it set out to do?

Worktrees

- Git feature that creates **independent working directories** sharing the same repo
- Each worktree has its **own branch**, its own files, its own state
- Changes in one worktree don't affect another – no stashing, no switching, no conflicts

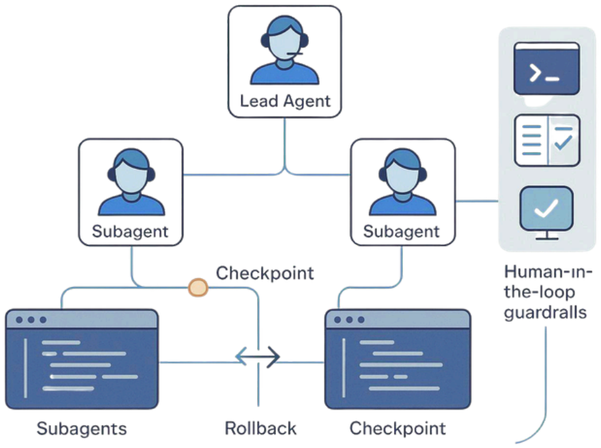
- **Recommended for:**
  - **Parallel agent sessions:** run Agent A on Feature A and Agent B on Feature B simultaneously, in the same repo
  - **Safe experimentation:** if an agent goes off the rails, delete the worktree and nothing is lost
  - **Clean baselines:** each agent starts from a known-good state with passing tests
  - **Independent commits:** each feature gets its own commit history, merges cleanly

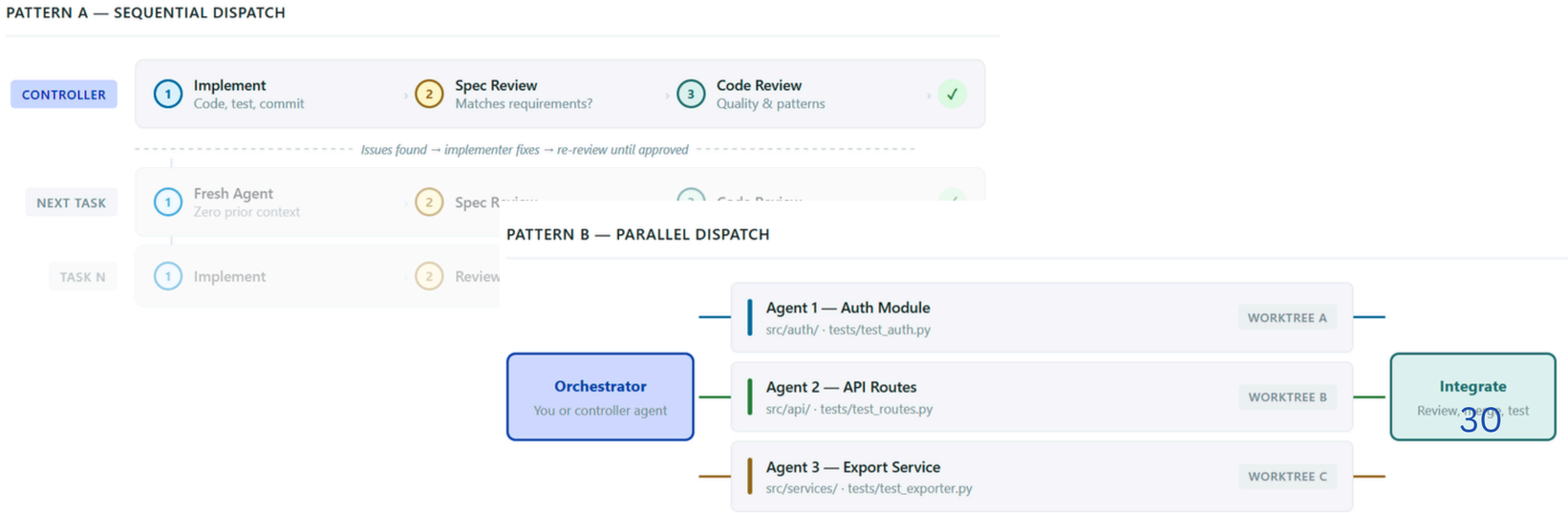


Multi-agent orchestration

- Several tools enable orchestration: Claude code (subagents, parallel sessions, teams of agents), Cursor/Windsurf, etc.
- Subagents are currently *implicit* in Copilot (and very limited)
- Two ways of running multi-agents: sequential & parallel

- **General rules:**
  - Never let parallel agents edit the same files (merge conflicts, logical contradictions, etc.)
  - Give each agent a narrow, specific scope – "Fix all the tests" fails; "Fix auth-abort.test.ts timing issues" works
  - Always verify after integration – run the full test suite, not just individual agent results
  - Include constraints – "Do NOT modify production code" / "Only touch files in tests"

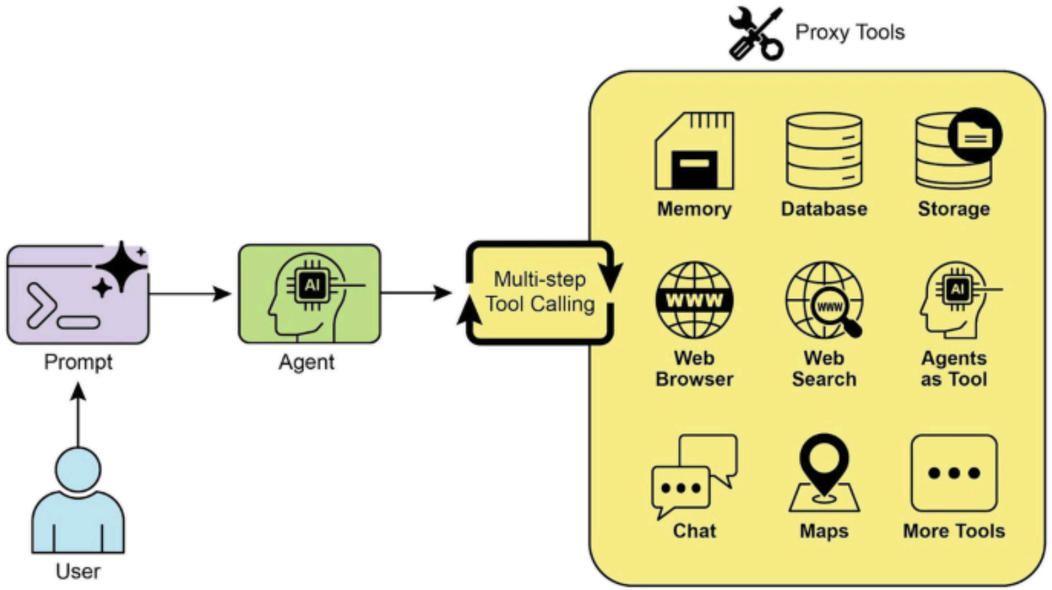




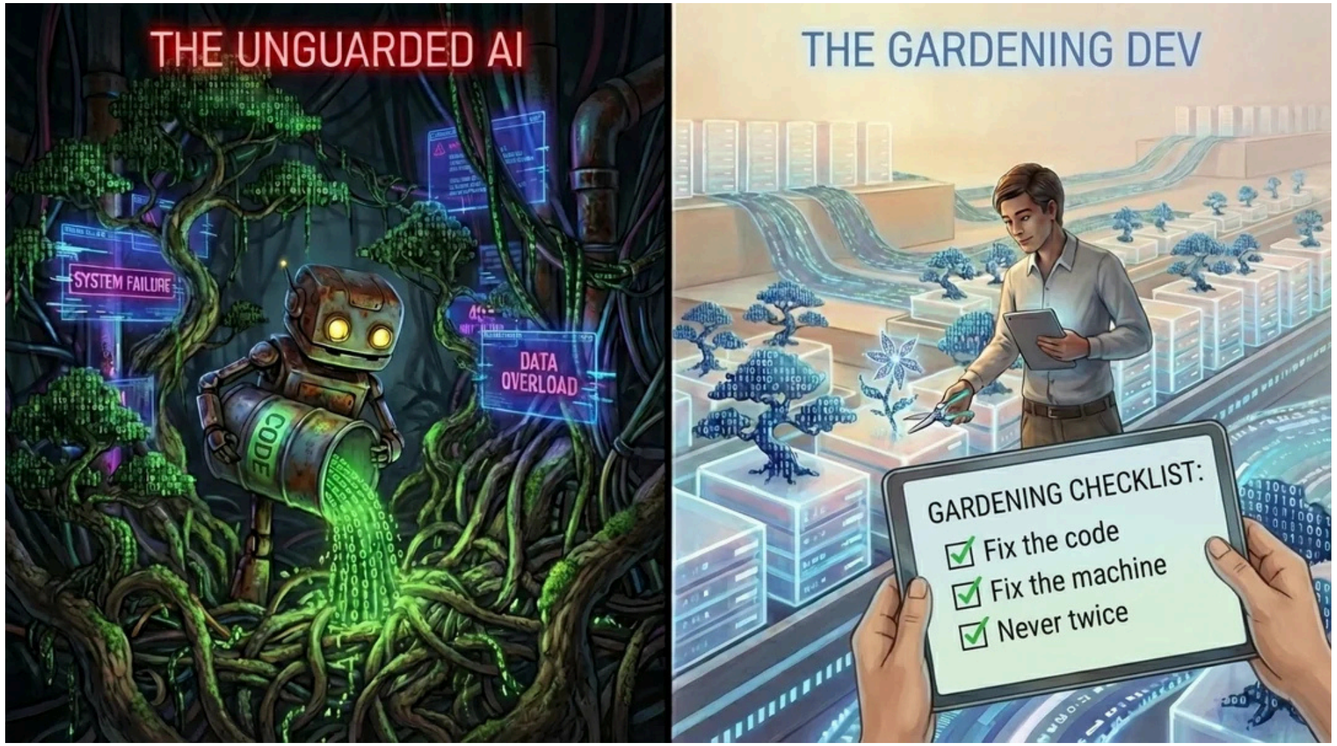
Tool Calling

- The mechanism that **turns an LLM from a text generator into an agent –** it can read files, run commands, search code, make edits, call APIs
- The agent **decides which tool to call**, with what parameters, based on the task
- MCP is a protocol that standardizes how agents connect to external tools and data sources

- Tool calling is what separates copilots (suggest code inline) from agents (autonomously complete multi-step tasks)
- The quality of an agent's work is directly tied to the tools it has access to and the feedback loops those tools enable



Issues



https://www.reddit.com/r/ClaudeAI/comments/1r13sqh/18_months_of_agentic_coding_no_vibes_or_slop/

Existing issues are getting worse

- Coding agents will not replace poor engineering practice
- Individual output surged 98% in high-adoption teams, but PR review time increased anywhere as high as 91%

- Data from **Faros AI** and Google's **DORA report** are interesting:
  - Teams with high AI adoption merged 98% more PRs
  - Those same teams saw review times balloon 91%
  - PR size increased 154% on average
  - Code review became the new bottleneck

Issues we are not aware of are emphasized

- The danger isn’t that the agent fails. It’s that it succeeds so confidently in the wrong direction that you stop checking it
- The models make wrong assumptions on your behalf and run with them without checking. They don’t manage confusion, don’t seek clarifications, don’t surface inconsistencies, don’t present tradeoffs, don’t push back.

- **AI code is generally not secure enough for production:**
  - Approximately 45% of AI-generated code contains security flaws – https://www.veracode.com/blog/ai-generated-code-security-risks/
  - Logic errors appear at 1.75× the rate of human-written code, and XSS vulnerabilities occur at 2.74× higher frequency – https://dl.acm.org/doi/10.1145/3716848

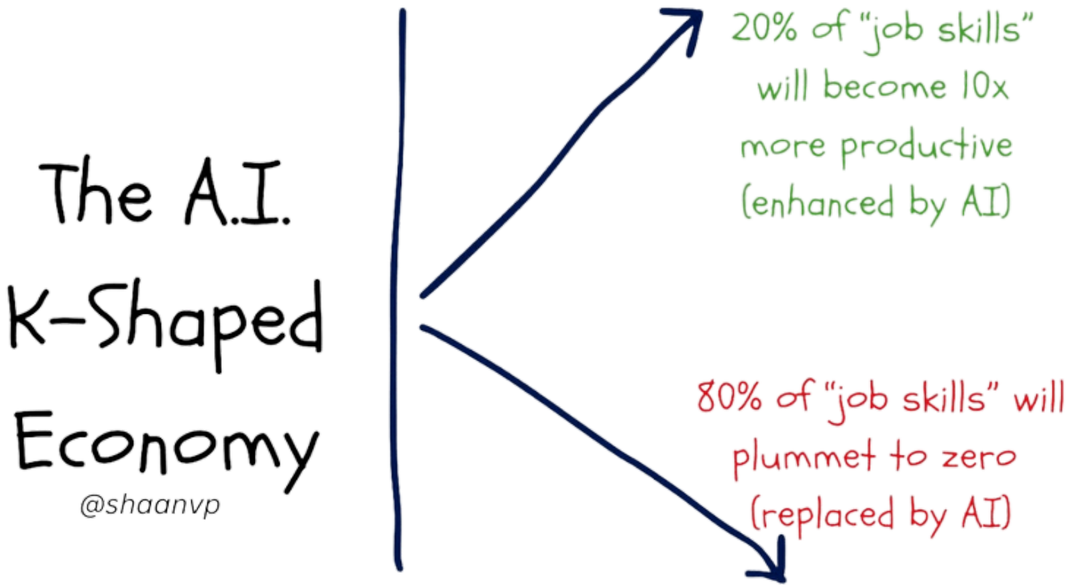
Quality and evolution depends on the driver

- **Quality depends on the driver**. Same tools, different hands, wildly different results. The difference isn’t just statistics… human judgment still matter
- The job shifts from writing code to managing what the model sees: curate context, prune noise, write specs
- If that role doesn't excite you, agentic coding won't either

- The Future of Software Development is Software Developers – https://codemanship.wordpress.com/2025/11/25/the-future-of-software-development-is-software-developers/

Essential skills

- Knowing model limits and capabilities
- Knowing which context to pass and context limits
- Writing rules, skills, and system prompts
- Prompt engineering
- Technical knowledge and domain expertise
- Critical thinking
- Clean code recognition
- Reading and reviewing code critically
- Knowing what to test
- Diff literacy – reviewing large multi-file changesets
- Task decomposition – breaking work into verifiable chunks
- Checkpoint discipline – when to commit, branch, snapshot
- Knowing when to stop the agent
- System thinking – understanding how local changes ripple
- Dependency awareness
- Slop recognition
- Context poisoning awareness
- Specification writing
- Iterative refinement – correcting the model without starting over
- Knowing when to code manually
- Calibrated trust – adjusting confidence based on task and model



You have a choice. Either get replaced by AI, or learn to use it and become 10X more productive.

Back to vibe coding

- It works great in a few contexts:
  - Personal projects where you control everything
  - MVPs where “good enough” is actually good enough
  - Startups in greenfield territory without legacy constraints
  - Teams small enough that comprehension debt stays manageable
  - Non-engineers