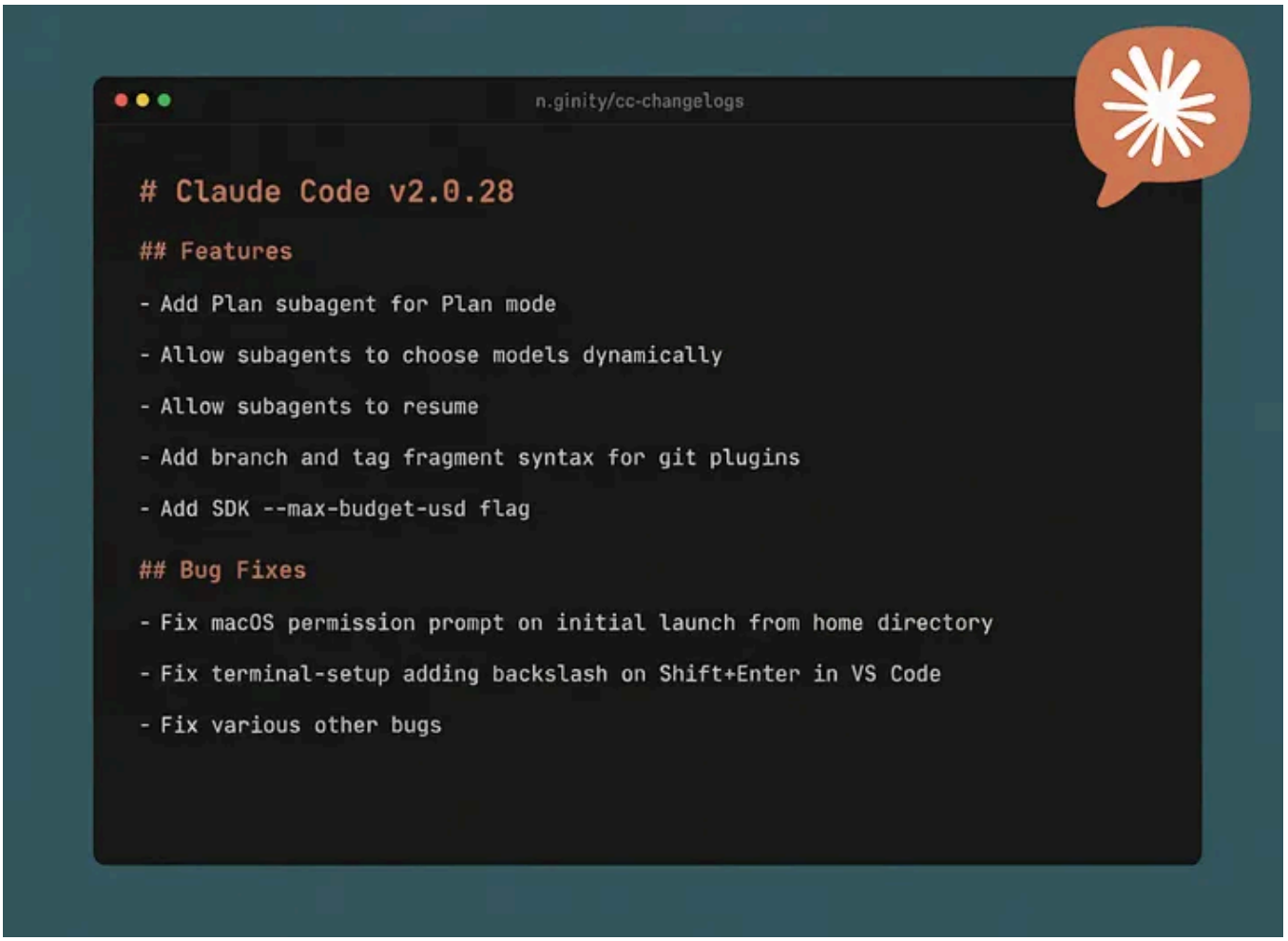
Member-only story

Claude Code v2.0.28: Specialized Subagents and the Architecture of Production-Ready Agentic Development

How dedicated planning agents, resumable context, and intelligent model routing transform autonomous coding from experimental to production-grade

12 min read · 1 day ago

Reza Rezvani Following



When Your AI Agent Needs Its Own Architect

Picture this: Your autonomous coding agent spends three hours refactoring a legacy authentication system. It touches dozens of files, updates import paths, modifies database schemas. You review the pull request and realize it implemented OAuth 1.0a instead of OAuth 2.0 — because somewhere around hour two, it lost track of the original requirements.

I've debugged enough of these *"almost right"* refactors to know the pattern. General-purpose agents drift. They start with clear intent, then get buried in implementation details. By the time they surface, the plan that started in their context window is buried under test failures, dependency conflicts, and exploratory dead-ends.

Claude Code v2.0.28 addresses this with a deceptively simple idea: **give the planning phase its own dedicated agent**. The implications ripple through everything — from how we structure complex workflows to what becomes possible with multi-agent architectures.

From Assistant to Autonomous Engineer: The 9-Month Technical Evolution of Claude Code

alirezarezvani.medium.com

. . .

The Core Problem: Context Pollution in Agentic Workflows

Before we examine the solution, let's understand why autonomous agents struggle with complex, multi-step tasks.

Modern large language models handle single, well-defined problems exceptionally well. Ask them to write a specific function, explain a code pattern, or generate tests for an isolated module — they excel. But chain multiple operations together, and something interesting happens.

The context window becomes a junk drawer.

Planning discussions mix with implementation details. Exploratory code that didn't work stays in memory. Error messages from failed attempts clutter the conversation. The agent's attention gets distributed across *"what we're trying to do"* (planning) and *"what we're currently doing"* (execution).

This manifests in three ways:

Requirement drift: Initial goals blur as implementation progresses. An agent might start refactoring for OAuth 2.0 but slip into OAuth 1.0a because that's what the existing codebase patterns suggested.

Replanning overhead: When a blocker emerges mid-implementation, the agent must replan *while* holding all current implementation state in context. This split focus produces suboptimal plans.

Context exhaustion: Long-running tasks fill the context window with artifacts of the journey — code snippets tried and abandoned, debugging output, file contents examined but not modified. Eventually, the original requirements scroll out of the working memory.

What we needed wasn't smarter general agents. We needed specialized agents with clear boundaries.

. . .

Enter the Plan Subagent: Separation of Concerns at the Agent Level

Claude Code v2.0.28 introduces something elegant: a dedicated Plan subagent that exists solely to think through problems before any code gets written.

The Architecture Shift

When you task Claude Code with something like *"refactor our authentication to support OAuth 2.0 with PKCE flow,"* the workflow now bifurcates:

The Plan subagent:

- Receives only the requirements and current codebase context
- Analyzes architecture, identifies files to modify, spots potential conflicts
- Produces a structured implementation plan with clear phases

- **Then exits, freeing its entire context**

The main agent:

- Receives a clean plan with no exploratory baggage
- Executes with full context dedicated to implementation
- Can call the Plan subagent again if requirements change, passing only the delta

This isn't multitasking — it's task isolation. Each agent maintains a focused context for its specific domain.

Why This Matters: A Real Example

I recently refactored a Node.js API from JWT to session-based auth. Without subagents, the autonomous agent spent considerable time exploring the codebase, identifying dependencies, considering migration strategies — all while trying to simultaneously implement changes. The context became bloated with both *"what should we do?"* and *"what are we doing?"* competing for attention.

With the Plan subagent, the workflow changed fundamentally. The Plan subagent spent its entire context budget analyzing the architecture. It identified that our Redis configuration would need updates, that the user model had middleware dependencies on JWT verification, that logout semantics would need rethinking. It produced a clear four-phase migration plan, then exited.

The main agent executed that plan with zero planning overhead. When we hit an unexpected third-party library incompatibility mid-migration, we resumed the Plan subagent with just that constraint. It adjusted the plan — swapping the problematic library for an alternative — without reanalyzing the entire architecture.

The result wasn't faster completion time. It was **more coherent implementation**. The code showed clear architectural intent rather than the usual "evolved as I went" pattern you get from context-confused agents.

. . .

Resumable Subagents: Maintaining State Across Invocations

Here's where v2.0.28 gets particularly interesting from a systems design perspective.

Previous versions treated subagents as ephemeral — invoke, execute, discard. If you needed refinement, you started from scratch. This works for simple tasks but breaks down when planning complex systems.

Imagine planning a microservices migration. The Plan subagent proposes splitting your monolith into service boundaries. You review and realize one proposed service boundary cuts across a transaction boundary that requires distributed transactions. You need to adjust the plan.

Before v2.0.28, that meant:

1. Restart the Plan subagent
2. Re-provide all original context
3. Explain the constraint
4. Hope the replanning doesn't lose other good decisions

Now, Claude can **resume subagents**, maintaining conversational state:

```
You: "Plan the microservices split for our e-commerce backend"
Plan Subagent: [Proposes 5 service boundaries including split payment processin
You: "The payment flow requires atomic transactions. Can't split it."
Plan Subagent (resumed): [Keeps other 4 boundaries, adjusts payment service to
```

The resumed subagent remembers its reasoning, the architectural constraints it already considered, and why it made specific decisions. It adjusts the specific element that changed without throwing away the entire plan.

> *This transforms iterative planning from expensive replanning to efficient refinement.*

. . .

Dynamic Model Selection: Matching Capability to Complexity

V2.0.28 introduces another architectural pattern that changes how we think about agent coordination: **dynamic model selection.**

Not all problems require the same cognitive capacity. Writing complex architectural plans benefits from maximum model capability. Generating repetitive test cases doesn't. Previously, you chose a model for the entire session. Now, subagents dynamically select models based on task complexity.

The model hierarchy:

- **Opus 4.1**: Deep reasoning for complex architecture and planning
- **Sonnet 4.5**: Balanced capability for standard implementation work
- **Haiku 4.5**: Fast execution for repetitive, well-defined tasks

In practice, this means:

Complex planning → Opus analyzes architectural implications, considers failure modes, proposes robust solutions

Feature implementation → Sonnet handles standard development work — the sweet spot of capability and speed

Test generation, simple refactoring → Haiku executes quickly on well-defined patterns

The routing happens transparently. You don't configure which model to use — the system matches capability to complexity. The Plan subagent might use Opus to design a complex migration strategy, then hand off to Sonnet for implementation, which delegates to Haiku for generating test boilerplate.

This isn't about cost optimization (though that's a benefit). It's about **appropriate cognitive load**. Using Opus to generate test fixtures wastes its reasoning capacity on pattern matching. Using Haiku to plan complex refactors underserves the problem's complexity.

. . .

Production Controls: The — max-budget-usd Flag

Let's talk about the elephant in the room: autonomous agents in production environments.

Exploratory workflows are inherently unbounded. An agent might try five approaches before finding the right solution. That's exactly what you want when exploring complex problems. But when that agent runs unsupervised — generating documentation, running code reviews, performing refactors — unbounded exploration becomes a problem.

V2.0.28 introduces `--max-budget-usd` —a hard limit on operation costs:

```
claude --max-budget-usd 50.00 "refactor authentication system"
```

When API usage for that operation hits the limit, execution stops. No gradual slowdown, no warnings — it halts.

Why this matters for production systems:

Scheduled automation: Run overnight code audits or documentation updates with predictable resource consumption.

Experimental workflows: Let agents explore multiple implementation strategies without runaway execution.

Multi-agent coordination: Set budgets for each subagent in complex workflows, ensuring no single component consumes excessive resources.

The implementation details matter:

- Budget is per-operation, not per-session
- Counts all model calls including subagents
- Accounts for different model pricing *(Opus vs. Haiku)*
- Provides detailed breakdown on termination

This transforms autonomous agents from *"requires constant supervision"* to *"safe for unattended execution with defined boundaries."*

• • •

The Developer Experience Details That Compound

Major features get headlines, but v2.0.28 includes refinements that show production maturity.

Git-Based Plugin Branching

Plugins and marketplaces now support branch/tag syntax:

```
/plugin install username/plugin#dev
/plugin install username/plugin#v2.1.0
/plugin install username/plugin#fix/context-issue
```

This enables proper plugin development workflows. Point staging environments at development branches, production at stable tags, and test environments at feature branches. Previously, this required maintaining multiple plugin repositories or complex local path configurations.

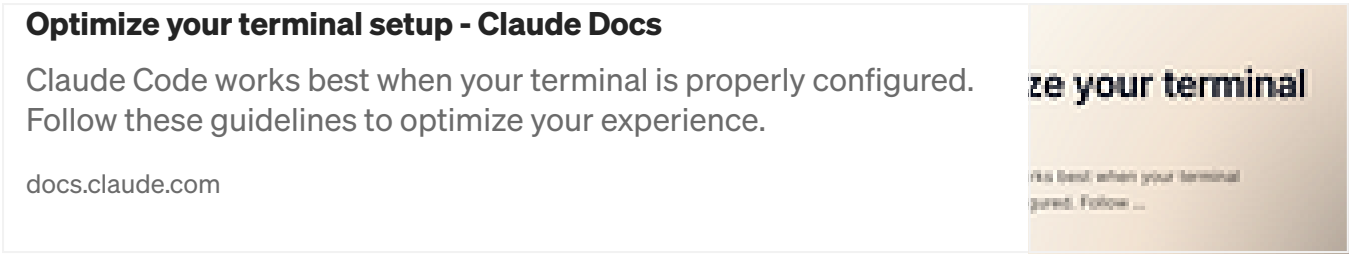
For teams building internal Claude Code plugins, this is the difference between *"maybe we can make this work"* and *"we can properly version and test this."*

The VS Code Shift+Enter Fix

Small bugs compound. The `/terminal-setup` command was inserting backslashes when using Shift+Enter for multi-line input in VS Code's integrated terminal.

This broke command history, created syntax errors in pasted scripts, and forced manual cleanup. Not dramatic, but multiplied across dozens of daily interactions, it added friction to the development loop.

Fixed in v2.0.28. Small quality improvements in developer tools matter.



.gitignore and Custom Tool Discovery

Previously, Claude Code respected `.gitignore` when discovering custom subagents and commands. This created a conflict: developers wanted to keep personal configurations local (gitignored) but still have them discovered by Claude Code.

V2.0.28 decouples git concerns from tool discovery. `.gitignore` status no longer affects whether Claude Code finds custom configurations.

Practical impact: You can now maintain:

- Team-wide subagents in `.claude/agents/` (committed)
- Personal experimental agents in `.claude/local-agents/` (gitignored)
- Both discovered and available

This respects the separation between "what belongs in version control" and "what tools should know about locally."

. . .

The Multi-Agent Architecture Pattern

V2.0.28's most significant contribution might not be the features themselves — it's the architectural pattern they establish.

The Plan subagent demonstrates **agent specialization**. Not general agents that do everything, but focused agents with clear domains and isolated contexts.

This pattern extends naturally:

Security Auditor Subagent: Analyze code changes for vulnerabilities, maintain context about security patterns, exit after audit

Performance Profiler Subagent: Examine algorithmic complexity, identify bottlenecks, suggest optimizations in isolation

Documentation Generator Subagent: Maintain understanding of API contracts, generate docs without polluting implementation context

Test Coverage Analyst Subagent: Track untested code paths, generate missing test cases, exit cleanly

Each operates independently with full context dedicated to its domain. Each can resume if requirements change. Each can dynamically select appropriate model capacity for its tasks.

The main agent orchestrates — receiving outputs, delegating work, coordinating across specialized domains. Like a tech lead coordinating a development team, not a solo developer context-switching across every role.

This architecture isn't fully realized yet. V2.0.28 gives us the Plan subagent and the infrastructure to build more. But the pattern is clear, and the possibilities are significant.

. . .

Common Pitfalls in Production Adoption

After working with v2.0.28 across several projects, three patterns of misuse emerged:

Pitfall 1: Treating Plans as Gospel

The mistake: Implementing Plan subagent output without review, assuming AI-generated plans are inherently optimal.

The reality: Plans are probabilistic. The subagent makes assumptions about your codebase, dependencies, constraints. Sometimes those assumptions are incorrect.

I caught a Plan subagent proposing a migration strategy that relied on a deprecated library — not because it was wrong about the architecture, but because it didn't know our internal deprecation policy. The plan was architecturally sound but practically unimplementable.

The solution: Treat plans like pull requests from a senior engineer. Review critically, challenge assumptions, verify against constraints the agent couldn't know. Plans are starting points for refinement, not final specifications.

Pitfall 2: Budget Limits Without Calibration

The mistake: Setting `--max-budget-usd` limits arbitrarily low "to be safe."

The reality: Too-low budgets cause mid-task termination, leaving partial work that's harder to resume than starting fresh.

The solution: Run tasks without limits initially to establish baselines. Track actual API costs for different task categories. Then set limits at reasonable multiples (perhaps 1.5x typical cost) to prevent runaway execution while allowing legitimate exploration.

Pitfall 3: Ignoring Model Selection Signals

The mistake: Treating dynamic model selection as a black box, never examining which tasks run on which models.

The reality: Default selection is good but not always optimal for your specific codebase patterns.

The solution: Review logs periodically. If Haiku consistently struggles with tasks you consider routine, perhaps your codebase has complexity the system underestimates — adjust subagent model preferences. If Opus gets used for simple test generation, configure those subagents to prefer Haiku.

The `model:` field in subagent definitions allows overrides when your domain knowledge exceeds the system's heuristics.

. . .

What This Means for Autonomous Development

V2.0.28 represents a shift from *"AI coding assistant that sometimes surprises you"* to *"production system with predictable behavior and defined boundaries."*

Agent specialization addresses the context pollution problem. No more mixing planning and execution in a single overloaded context.

Resumable state enables iterative refinement without expensive replanning. Plans evolve rather than restart.

Dynamic model selection matches cognitive capacity to problem complexity. Not all problems need maximum reasoning capacity.

Production controls make unattended execution viable. Bounded exploration with defined limits.

Developer experience refinements show maturity. Small fixes compound into smoother workflows.

This isn't the final evolution of agentic development. The multi-agent architecture pattern v2.0.28 establishes will likely expand. More specialized subagents for specific domains. Better coordination between agents. Learned optimization of task delegation.

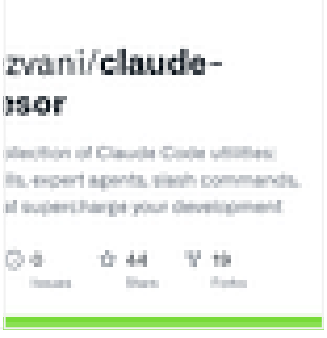
But for teams adopting autonomous coding tools today, v2.0.28 crosses a meaningful threshold. The foundation is solid. The architecture is extensible. The production readiness is real.

A Collection of Subagents, Skills, and Slash commands for your daily use in this Github Open Source Repository:

GitHub - alirezarezvani/claude-code-tresor: A world-class collection of Claude Code utilities...

A world-class collection of Claude Code utilities: autonomous skills, expert agents, slash commands, and prompts that...

github.com



• • •

Getting Started: Technical Prerequisites

If you're evaluating v2.0.28 for production use, here's what matters:

Installation:

```
npm install -g @anthropic-ai/claude-code@2.0.28
```

Key documentation:

- Claude Code Subagents Guide
- Claude Agent SDK
- Best Practices for Agentic Workflows

Testing approach:

Start with the Plan subagent on complex refactoring tasks. Compare plan quality and implementation coherence against previous approaches.

Experiment with custom subagents for repetitive patterns in your codebase. Test generation, documentation updates, style enforcement — tasks with clear inputs and outputs.

Set budget controls on non-critical automation. Observe how limits affect completion rates and resource consumption.

Monitor dynamic model selection. Verify that routing aligns with your understanding of task complexity.

Integration considerations:

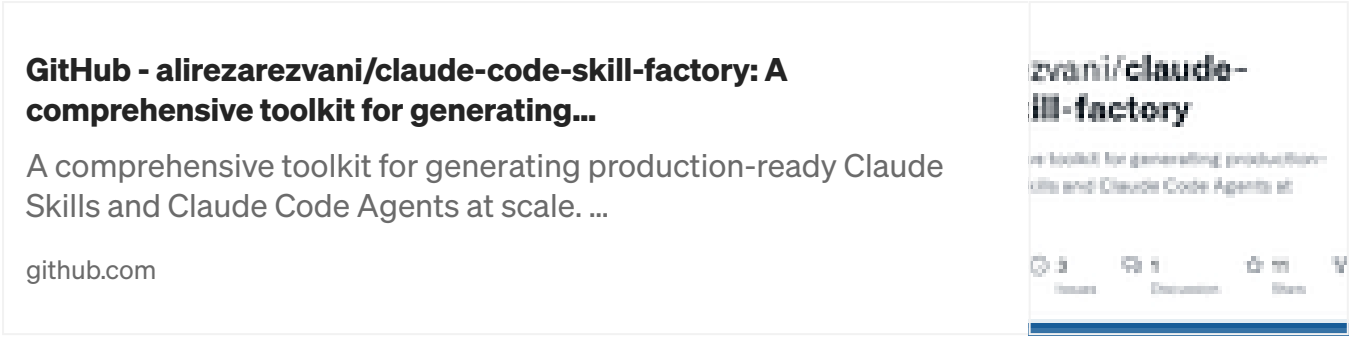
For CI/CD integration, budget controls become essential. Set appropriate limits for automated tasks to prevent runaway execution in unmonitored pipelines.

For team adoption, share successful subagent definitions in version control. Custom subagents work best as shared resources, not individual configurations.

For production deployment, monitor API usage patterns initially. Establish baselines before setting hard limits.

Your Feedback is Wanted — Call:

Try out my new Open Source Project for building Claude AI and Claude Code Agents Skills with the Agent Skills Factory at scale.



. . .

The Engineering Perspective

Claude Code v2.0.28 moves autonomous coding from experimental capability to production-viable architecture. The Plan subagent, resumable context, dynamic model routing, and execution controls form a coherent system for complex development workflows.

The multi-agent pattern it establishes has implications beyond current features. As specialized subagents proliferate — security analysis, performance profiling, documentation generation — the coordination layer becomes increasingly important. How do agents share context? How do we prevent redundant work? How do we maintain coherence across specialized domains?

These are interesting systems design problems. V2.0.28 provides the foundation to explore them.

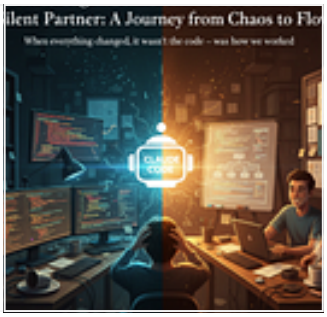
For teams currently using AI coding tools, this release merits evaluation. The architectural changes are substantial. For teams waiting for production readiness, the necessary controls are now present.

The future of autonomous development is multi-agent, specialized, and bounded. V2.0.28 shows us what that looks like in practice.

The AI Agent That Became Our Team's Silent Partner: A Journey from Chaos to Flow

When everything changed, it wasn't the code – it was how we worked

alirezarezvani.medium.com



. . .

✨ *Thanks for reading! If you'd like more practical insights on AI and tech, hit* ***subscribe*** *to stay updated.*

I'd also love to hear your thoughts — drop a comment with your ideas, questions, or even the kind of topics you'd enjoy seeing here next. Your input really helps shape the direction of this channel.

. . .

About the Author

Me, Alireza Rezvani work as a CTO @ an HealthTech startup in Berlin and architect AI development systems for my engineering and product teams. I write about turning individual expertise into collective infrastructure through practical automation.

Looking forward to connecting and seeing your contributions — check out my open source projects on GitHub!

Connect: Website | LinkedIn
Read more: Medium Reza Rezvani
Explore: GitHub

Claude Code | Context Engineering | Generative Ai Tools | Software Development | Software Engineering

Following

Written by Reza Rezvani

938 followers · 71 following

As CTO of a Berlin AI MedTech startup, I tackle daily challenges in healthcare tech. With 2 decades in tech, I drive innovations in human motion analysis.

No responses yet

Bgerby

What are your thoughts?

More from Reza Rezvani

In nginity by Reza Rezvani

I Let Claude Sonnet 4.5

IMAGINE this: It's 6 a.m., the kind of quiet dawn where the world's still wrapped in that soft, hazy light filtering through your blinds...

Sep 29

Reza Rezvani

"7 Steps" How to Stop Claude Code from Building the Wrong Thing (Part 1): The Foundation of...

Learn how to stop Claude Code from rewriting your architecture with vague prompts. This guide introduces Spec-Driven Development...

Sep 17

In nginity by Reza Rezvani

Claude AI and Claude Code Skills: Teaching AI to Think Like Your Best Engineer

Oct 19

Reza Rezvani

I Discovered Claude Code's Secret: You Don't Have to Build Alone

I've been coding long enough to know that the late-night debugging sessions aren't glamorous. They're just necessary.

Sep 19

See all from Reza Rezvani

Recommended from Medium

In Realworld AI Use Cases by Chris Dunlop

The complete guide to Claude Code's newest feature "skills"

Claude Code released a new feature called Skills and spent hours testing them so you don't have to. Here's why they are helpful

Oct 21

Reza Rezvani

“7 Steps” How to Stop Claude Code from Building the Wrong Thing (Part 1): The Foundation of...

Learn how to stop Claude Code from rewriting your architecture with vague prompts. This guide introduces Spec-Driven Development...

Sep 17

In AI Software Engineer by Joe Njenga

This Viral DeepSeek OCR Model Is Changing How LLMs Work

This DeepSeek OCR model hit an overnight success not seen in any other release—4k+ GitHub stars in less than 24 hours and more than 100k...

6d ago

In Dare To Be Better by Max Petrusenko

Claude Skills: The $3 Automation Secret That's Making Enterprise Teams Look Like Wizards

How a simple folder is replacing $50K consultants and saving companies literal days of work

Oct 17

Manojkumar Vadivel

The .claude Folder: A 10-Minute Setup That Makes AI Code Smarter

If you're new to Claude Code, it's a powerful AI coding agent that helps you write, refactor, and understand code faster. This article...

Sep 15

Agen.cy

20+ Genius Ways Power Users Are Using Claude Code Right Now

Here are 10+ ways power users are using Claude Code🧵

5d ago

See more recommendations