~ $ don't panic, automate.

A Vibe Coder's Guide to Agentic Engineering

Adam Charnock • Dominik Grusemann • Traun Leyden

bettervibe.org

v1.0 • 2026

A Vibe Coder's Guide to Agentic Engineering

“Agentic engineering is about preserving the quality bar of professional software.”

— Andrej Karpathy

// CONTENTS

// INTRODUCTION

A year ago, “AI coding” mostly meant autocomplete. Today, an agent can read a ticket, plan the work, write the code, run the tests, and open the pull request — in the time it used to take to set up a branch. The leverage is real. So is the mess underneath it.

Most engineers who have shipped serious AI-assisted work have felt the same two things. First: a stretch of speed that feels like cheating. Then the comedown — code that works but you don’t quite understand, a codebase drifting somewhere you didn’t choose, a review queue full of pull requests no human wants to read. We call it the Vibe Coding Hangover, and avoiding it is the reason this guide exists.

Agentic engineering is the discipline of directing AI agents to ship production code without losing the quality bar, the architectural thread, or your own understanding of the system. It is not better prompting. It is the system you build around the agent so the agent can do its best work and you can stay in charge of the result.

WHAT THIS IS — AND ISN’T

A field manual, not a textbook. Ten practices, organized around the natural arc of a piece of work:

- **Part I — Plan.** Spec the work, choose the architecture, set the rigor before you prompt.
- **Part II — Implement.** Run the agent inside fast feedback loops, with the tools and context it needs to be useful.

- **Part III — Review & Sustain.** Catch what the agent missed, manage the debt it creates, and keep the codebase yours over time.

Each chapter is short on purpose. We have written the practices in the form we wish someone had handed us a year ago: the rule, the reason behind it, and just enough texture to apply it tomorrow morning.

WHO THIS IS FOR

Working software engineers who already ship code with an agent in the loop and have started to notice the seams. You have used Claude Code, Cursor, or Codex in earnest. You have felt the speed. You have also seen a pull request you couldn't fully defend, a refactor that quietly went sideways, a teammate ask "who actually wrote this?" — and wondered whether the velocity is worth the drift.

If you are still deciding whether to adopt AI coding tools at all, this is too far downstream. If you are looking for prompt templates or a list of MCP servers, you will be disappointed. The practices here assume agents are already part of your workflow; the question is how to keep them honest.

WHERE THIS COMES FROM

The three of us have shipped production code with AI agents daily for the better part of two years — across bare-metal Kubernetes infrastructure, B2B SaaS, autonomous driving systems, and consumer apps. None of us came to this from research. We came to it from things breaking in production and trying to figure out why.

What you are reading is the consensus view that emerged from those scars, sharpened against the public writing of practitioners we trust, and tested in the workshop we run on this material in Munich. None of it is settled. The tools change every month, the models change every quarter, and some of what is in here will look quaint within a year. We have tried to write the parts we expect to age the slowest — the ones about how humans and agents work together, not the ones about which CLI flag to pass.

HOW TO READ IT

Linearly, if it is your first pass. The three parts build on each other, and several practices in Part III only make sense once you have internalized Part I.

As a reference, after that. Each chapter stands alone. Skim the contents, find the practice that matches the failure mode you are currently looking at, and start there.

If you only have ten minutes: read Chapter 1 (`Plan Before You Prompt`) and Chapter 9 (`Manage Technical Debt Actively`). The first is where most of the leverage lives. The second is what happens when you skip the first.

ONE LAST THING

Agentic engineering is not about getting the agent to do more. It is about preserving the things that made you good at this work in the first place: judgment, taste, ownership, the willingness to read what was written and decide whether it should stay. The agent is a force multiplier. You are still the engineer.

Turn the page.

// PART I

Plan.

Before you prompt: spec the work, choose the architecture, set the rigor.

// 1. PLAN BEFORE YOU PROMPT

Write a `plan.md` before touching code. Without a spec, AI generates plausible but directionless output. The plan is your anchor — revisit it after every wave of work to catch drift.

A brief written spec is worth ten corrective prompts.

BEST PRACTICES:

- Right-size the artifact to the task. Match the weight of the plan to the risk of getting the task wrong. The annoyance test: if you'd be annoyed to have the agent interpret requirements differently than you meant, write a spec. If you could fix it in one follow-up prompt, skip it. CRUD forms get a sentence; billing state machines get a document.
- Prefer one editable plan file over hidden artifacts. A markdown file you can open in your editor and revise beats a buried plan file you can only see through the agent's UI. You want to be able to edit, version-control, and grep your plans.
- Plan with the most capable model you have; implement with the cheapest one that can handle the task. Architecture and design decisions are where reasoning quality compounds — pay for it. Implementation, when the plan is well-specified, is often mechanical enough for a smaller model. In Claude Code, `/model opusplan` automates this split.
- Bound each phase with its own clean context. Before planning a new feature, clear context if you've been working on something unrelated — irrelevant history is noise. During planning, push exploratory research into

subagents so it doesn't pollute the planner's window. After planning, hand the spec to a fresh session for implementation. Each handoff is a context reset; the spec file is what survives.

- Plan conversationally. The most reliable planning workflows are dialogues, not one-shot generations. The agent asks one clarifying question at a time, proposes 2–3 alternatives, presents the design in chunks for approval. Monolithic plans dumped in a single response are too dense to review and miss the assumptions a back-and-forth would have surfaced.
- Own the architecture; let the agent plan within it. The agent is good at expanding a clear intent into structured tasks. It's bad at deciding which parts of the system should exist. Make the architectural calls yourself, then ask the agent to plan within them — and explicitly ask it to flag over-engineering before implementation.
- Use your voice. Talk to the agent instead of typing. When you type, you self-edit and slim down context. When you speak, you include the reasoning you'd normally skip. In our experience, telling the agent exactly what you wouldn't bother writing down can be extremely useful.
- Don't anchor the agent to your answer. Describe the problem, not your hunch about the solution. Models are eager to ratify what you've already proposed, so leading with "I was thinking we should…" usually gets you a confident expansion of your own bias. Let the agent reason from the problem first, then compare. When its take and yours diverge, that's where the real planning happens.

SOURCES:

- https://lucumr.pocoo.org/2025/12/17/what-is-plan-mode/
- https://martinfowler.com/articles/exploring-gen-ai/sdd-3-tools.html
- https://blog.scottlogic.com/2025/11/26/putting-spec-kit-through-its-paces-radical-idea-or-reinvented-waterfall.html

// 2. AGENTS.MD/CLAUDE.MD

The LLM starts every conversation with amnesia. Re-explaining your codebase, conventions, and build commands in every prompt is wasted effort — and it never quite sticks. Persist that knowledge in the repo instead.

BEST PRACTICES:

- Short — a good context file is short and load-bearing. Include only what the agent can't infer from the code.
- Non-obvious — conventions, architecture choices, tooling quirks. "We use bun, not node."
- Tech stack — Every `AGENTS.md` needs a `## Tech Stack` section.
- No `/init` — Do not use `/init`. Auto-generated context files measurably reduce success rates.
- Progressive disclosure — keep the always-loaded file small; reveal task- and path-specific context only when relevant.
- Do not describe anything that the `formatter` or `linter` can do.
- Use procedural workflows (numbered lists) for recurring tasks. Put them in slash commands or skills, not in `AGENTS.md` itself.
- Use decision tables when 2–3 valid approaches exist.
- Pair every "don't" with a "do" — a prohibition without an alternative makes the agent over-explore.

SOURCES:

- https://www.augmentcode.com/blog/how-to-write-good-agents-dot-md-files

- https://www.humanlayer.dev/blog/writing-a-good-claude-md

// 3. ARCHITECTURE OVER SYNTAX

You're going to spend a lot of time writing code with agents. But the code itself is rarely where the real leverage is. Architecture is.

A well-written function improves one small corner of your project. A well-designed system improves `everything` built on top of it. And a bad architectural choice doesn't just create one problem — it creates a category of problems that follow you for months. The consequences compound in both directions, fast.

This is where your attention belongs. Not on whether a loop is elegant, but on how pieces fit together, how data flows, how boundaries are drawn. That's what determines whether your project succeeds or slowly collapses under its own weight.

YOU AND THE AGENT ARE A TEAM

Stop thinking of your coding agent as a tool. Think of it as a teammate — one that's complementary to you, not competing with you.

The agent is fast, consistent, and doesn't get tired at 2am. It can hold a hundred edge cases in its head without losing track. For mechanical, repetitive, detail-heavy implementation, it will outperform you every time. It'll write the fifteenth input check with the same care as the first.

But it can't reason about the big picture. It doesn't know `why` you're building what you're building. It can't weigh your business constraints against your technical ideals, or feel that something is off before it can articulate what. That's your job.

So the division is clear: the agent brings speed, consistency, and tireless attention to detail. You bring reasoning, context, and judgment on questions that don't have clear right answers. Your job is to set the direction, communicate your design clearly, and review what comes back. The agent's job is faithful execution.

When this loop works — you thinking clearly about structure, the agent executing faithfully on implementation — you move faster without sacrificing quality. You explore more options without burning out. That's the goal. Not to replace your thinking with the agent's output, but to free your thinking from the work that was always grinding it down.

THE BOUNDARY BETWEEN ARCHITECTURE AND CODE IS BLURRY

In interviews, coding and system design are separate exercises. In actual work, they bleed into each other constantly. You're writing a function and realize the data model is wrong. You're sketching a system and realize you need to prototype something to check your assumptions.

You can't just "do architecture" in isolation and hand it off. Don't let the tail wag the dog: you should be designing the architecture with high intentionality, and the coding agent should be aligning its code to your architectural decisions. The decisions at each abstraction level influence the others, and if you're not paying attention to that interplay, things drift out of alignment fast.

It's no coincidence that this is also what separates senior engineers from junior ones. Early in your career,

strong coding skills can carry you. The further you go, the more you're judged on system-level thinking. The coding agents will act as a force multiplier on this gap.

FOCUS WHERE IT MATTERS

Your attention is the scarcest resource in the entire workflow. Not compute, not tokens — your attention.

When you spend an hour thinking carefully about your system's boundaries, you save dozens of hours of rework later. When you spend that hour tweaking CSS or rewriting a utility function that already works, you've burned your most valuable resource on something that barely moves the needle.

Get honest about where your time is going. If most of your day is in the weeds and barely any of it is on structure, you've got the ratio backwards.

USING AGENTS FOR DESIGN RESEARCH

Agents aren't just for writing code — they're useful for exploring the design space. But think of the agent as a research assistant, not an architect. It can survey options and surface approaches you hadn't considered, but its knowledge might be outdated and it doesn't have your full context.

Use it iteratively. Start broad: "What are the common approaches for X?" Then narrow: "Given these constraints, compare A and B." Then validate outside the agent — read the docs, check the GitHub issues, talk to someone who's used it in production.

Be explicit about your constraints when you prompt. Scale requirements, team size, deployment environment, timeline. Vague prompts get vague answers.

LEVERAGE ARCHITECTURE DESIGN BEST PRACTICES: WRITE THINGS DOWN

Design documents aren't bureaucracy. They're thinking tools.

When you write down your architectural reasoning, you're forced to make assumptions explicit. You discover gaps you didn't know were there. You notice when two decisions contradict each other. The act of writing `is` the thinking.

Even solo, a short doc capturing your key decisions and rationale will save you enormous confusion three months from now. On a team, it's even more critical — design docs prevent the kind of slow divergence that kills projects. It doesn't have to be formal. A markdown file with a few sections is fine. The point is clarity, not polish.

PROTOTYPING IS A SUPERPOWER NOW

Prototyping used to be expensive. Building a throwaway implementation to test an assumption might take days, so people skipped it. They'd guess at the architecture and hope.

Now you can spin up a prototype in hours. Want to know if that message queue handles your throughput? Prototype and benchmark it. Not sure whether a relational or document database fits better? Prototype both and compare. The cost of validating assumptions has dropped dramatically.

These throwaway prototypes aren't wasted work. They're some of the highest-leverage work you can do, because they

turn architectural guesses into decisions backed by real data.

// 4. HAVE EMPATHY FOR THE AGENT

Empathy for the agent's situation gets the most out of it, somewhat like being a manager. The LLM starts every conversation with amnesia. It doesn't know your codebase, your conventions, or the discussion that led to this task. We forget how much of our own context we carry by default. The agent has none of it.

Brief it the way you'd brief a competent direct report on their first day:

- The goal, in your words.
- The constraints. What cannot change, what must not break.
- What you already tried, and why it didn't work.
- The audience for the output. Throwaway prototype and production code call for different work.
- The urgency-versus-thoroughness tradeoff. An agent told nothing will pick a default, and it may not be the one you want.

Add screenshots of the Slack thread or ticket that prompted the request. At Lithus we also point it at our `llms-full.txt` in some cases (https://lithus.eu/llms-full.txt).

Be honest about your own uncertainty. "I think we should do X, but I don't know as I haven't worked with this stack before" gives the agent permission to disagree. Without that line it commits to your framing, and you lose a second opinion you were never going to read otherwise.

Tell it to ask questions. It will sometimes do this on its own, but not always, and the question it asks usually reminds you of context you forgot to include.

Empathy you'd otherwise repeat every session belongs in `CLAUDE.md`, or your agent's equivalent. Codebase conventions, who you are, what you care about. Write it once, stop retyping it.

// PART II

Implement.

Working with the agent: small tasks, the right tools, effort tending to zero.

// 5. CHOOSE TOOLS FOR THE AI, NOT FOR YOU

Mandatory checks are another form of important feedback, such as pre-commit hooks.

At Lithus we use lefthook pre-commit hooks for: secret scanning (gitleaks), formatting (rustfmt, gofmt, black, prettier, nixfmt), linting (clippy, go-lint, statix, deadnix), blocking commits of unencrypted Terraform state, and requiring documentation updates.

These pre-commit hooks force the agent through a mandatory feedback loop. The commit doesn't land until the checks pass; every rejection becomes a correction the agent has to make before moving on.

It should be easy for the agent to reproduce issues. Ideally, one command brings the project to a working state. Mise or devenv are good options here.

Programming language choice is very relevant here too. Pick languages that give fast feedback at compile-time. Static type errors are easier for an agent to address than runtime ones it has to reproduce first.

The underlying principle is to build feedback loops. If you find yourself copy-pasting into the prompt window, that is likely a candidate for a feedback loop.

// 6. EFFORT IS TENDING TOWARD ZERO

The effort of actually wielding your chosen tools has historically fallen to you, the engineer. Some tools you enjoyed wielding, but some you did not. Perhaps some of these tools were conceptually harder to grasp, or required more effort than you were willing to give.

Now you have a new option. You can now reap the benefits of tools you would previously avoid. The tradeoff is that you must be willing to learn these tools top-down, not bottom-up. Understand the principles and the best practices, not the syntax.

Some examples of tools that may fall into this category for you:

- **Rust** — rapid feedback, and even though the type system is a head-scratcher, agents deal with it fine.
- **Nix** — a paradigm shift, but it catches system configuration issues at evaluation time, before runtime.
- **Kubernetes** — a well-known system that agents can easily deploy to, and one they are excellent at diagnosing issues on.
- **Schema-first APIs** — OpenAPI, Protobuf, GraphQL. Tedious to set up. A free reviewer forever after.
- **Infrastructure as code** — Terraform or Pulumi over clicking through cloud consoles. Declarative state is painful to learn, and easier for agents to reason about than UI clickpaths nobody can audit.
- **Strict linter and type-checker configs** — clippy at full volume, mypy strict, the ESLint rules humans turn off. The agent absorbs the friction; the codebase keeps the safety.

The principle has some limits. Tools with sparse training coverage will require the agent to have some documentation on-hand, which will increase token use.

// PART III

Review & Sustain.

Closing the loop: feedback, specialist reviews, keeping technical debt honest.

// 7. BUILD FAST FEEDBACK LOOPS

If the AI can't test its own work quickly, it can't self-correct. This means:

- Easy-to-run test commands — not a 12-step manual process
- Pure functions that can be tested in isolation
- Architecture that separates business logic from I/O
- Access to development and production logs

The faster the loop closes, the sooner mistakes get caught before they compound.

// 8. CODE REVIEWS

When agents write most of the code, a single human reviewer becomes the bottleneck. The shift that matters is moving from one human reading every diff to a panel of specialist agents reviewing in parallel — with the human as the final checkpoint, not the first one.

USE A PANEL, NOT A SINGLE REVIEWER

A generalist reviewer produces generic advice. A panel surfaces issues that no single reviewer could hold in mind simultaneously.

The practical prompt is short:

> 💡 **PROMPT**
>
> Spawn a red team of agents to review this work[, ensure you include an X expert]

Claude picks reasonable specialists on its own, but it helps to name them explicitly:

- best-practices expert
- DRY expert
- `<language>` expert
- security expert
- maintainability expert
- SOC 2 / ISO 27001 compliance expert

For larger diffs, run two rounds. The first round catches the critical issues; the second round picks up important findings the first one missed. It is token-heavy, but cheap relative to the cost of a missed regression.

TELL THE AGENT WHAT NOT TO FLAG

The most counterintuitive finding from production deployments: a security reviewer's prompt is mostly a list of things to `ignore` — theoretical risks, defense-in-depth nits, unchanged code, "consider library X" suggestions. Without an explicit ignore list, the panel produces a firehose of speculative warnings that developers learn to tune out within a month.

This generalizes beyond review. Generate an explicit `REVIEW.md` for this task and reference it on review prompts.

TIER THE REVIEW TO MATCH THE DIFF

Frontier-model tokens are wasted on a typo fix. A workable classification:

- **Trivial** — ≤10 lines, ≤2 files. Coordinator can downgrade from Opus to Sonnet.
- **Lite** — ≤100 lines. Spawn a single agent to review this work.
- **Full** — >100 lines, or any change touching auth, crypto, or other security paths. Spawn a team of expert agents to review this work.

Published cost data puts trivial reviews near $0.20 and full reviews near $1.68. Tiering is the most direct answer to the "Claude Code is too expensive" objection.

REVIEW BEFORE THE PR, NOT AFTER

Run the panel on the developer's laptop before the PR opens — same agents, same prompts, same risk tiers as CI. The human reviewer becomes the final checkpoint, not the

first one. Every finding the panel produces also becomes a candidate prompt rule for the next iteration, which is where compound engineering starts to pay off.

// 9. MANAGE TECHNICAL DEBT ACTIVELY

You already know what technical debt feels like, even if you've never used the term. It's that moment where a change that should take twenty minutes takes two days. It's when you fix a bug in one place and three more pop up somewhere else. It's the slow, creeping realization that your codebase is fighting you.

At its core, technical debt is complexity that's slowing you down. Duplicated logic scattered across files. Tightly coupled components where touching one thing breaks five others. That Rube Goldberg machine you built sprint by sprint, where nobody can trace the full path from input to output anymore.

And in the age of agents, this problem gets worse faster than it ever has before.

NOT ALL DEBT IS CREATED EQUAL

Sometimes technical debt is the right call. If you're building a prototype, pile it on. You're trying to validate an idea, not build a cathedral. Debt in throwaway work isn't debt — it's efficiency.

If you're working on a long-lived system — something people depend on for years — your tolerance should be close to zero. Complexity compounds. What costs you an hour today costs you a day next quarter and a week next year.

Most of us live somewhere in between. You're building a SaaS product, the product is evolving, and some parts won't survive the next pivot. Being precious about those parts is a waste. But the core? That needs to be solid.

The question you should always be asking is: how long does this code need to live?

AGENTS WILL BURY YOU IN DEBT IF YOU LET THEM

Agents don't care about your codebase. They care about completing the task you gave them. That's it.

If you say "fix the performance issue," an agent will fix it — probably by bolting on a caching layer or wrapping things in a new abstraction. Did it work? Yes. Is the underlying design any better? Almost certainly not. You just got another layer on the pile.

The agent optimizes for the reward signal (task completed, tests pass) and ignores everything else. It doesn't check whether the solution is consistent with the architecture. It doesn't ask whether there's already a utility function that does the same thing. Left unchecked, your complexity grows with every task — and the agent isn't being malicious. The problem is that "what you asked" and "what you actually need" aren't always the same thing.

HOW TO KEEP IT UNDER CONTROL

Know your stack. You can't guide an agent toward good solutions if you don't understand what good looks like. You don't need to write every line yourself — but you need to understand every "semantic chunk" and know whether it belongs.

Ensure best practices. If your codebase has patterns for database connections, tell the agent. If there's a utility module that should be reused, point to it. Build these constraints into your reusable prompts, rules files, and

agent configs. Make the guardrails part of the workflow, not something you remember on a good day.

Catch problems early. Review the code, run the tests, watch runtime behavior. Feed problems back into the agent while context is fresh. This matters more with agent-generated code because the volume is so much higher.

Prevent what you can, detect what you can't prevent, fix what you detect before it metastasizes.

THE VIBE CODING HANGOVER

You've been shipping fast. The agent is cranking. Tickets are closing. Then one morning you open the project and… what is this? Three different auth patterns. Circular imports. A file structure that makes no sense. You're staring at your own codebase and you don't recognize it.

That's the Vibe Coding Hangover.

It hits hardest under pressure to deliver something new, when you realize the foundation can barely support what's already there. Your options all suck: patch it (more debt), refactor it (explain the velocity drop to stakeholders), or rewrite it (feels like admitting failure). All of them could have been avoided by paying attention along the way.

THE SNEAKY STUFF: DEAD CODE AND LOST UNDERSTANDING

Not all tech debt screams at you. Dead code is a perfect example — agents rewrite functions and leave the old versions behind. Nothing breaks, but your codebase fills with ghosts. Every search returns noise. You burn tokens sending useless context to the agent on the next task. Get

tooling to catch it: compiler warnings in Rust, linters in TypeScript. This matters more now because agents generate dead code at scale.

But the really dangerous one is not understanding your own system. This is latent debt — it doesn't show up until something breaks and you have no mental model of how the pieces fit together. If you fall too far behind, the agent starts looping on fixes that don't work because the problem is architectural, and you can't help because you don't understand the architecture either. At that point, your only option is often to start over.

Don't let this happen. Read the code the agent writes. Sketch how the system works, even if it's boxes and arrows on a napkin. The goal isn't perfect documentation — it's a mental model close enough to reality that you can intervene when things go sideways.

STRATEGIES THAT ACTUALLY HELP

Snippet databases. Keep a collection of battle-tested code patterns you've already debugged and validated. Point the agent at them. Instead of reinventing solutions (and reinventing new bugs), it reuses something proven. You move faster and take on less debt. One of the rare genuine free lunches.

Lean toward stronger typing. Traditionally, strong typing meant more boilerplate and cognitive overhead. But agents eat that overhead for breakfast. They'll happily generate all the type annotations and interfaces you want, and you get explicit data structures, better tooling, and fewer mystery bugs. TypeScript over JavaScript. Python type

hints everywhere. If the ecosystem supports it, Rust or Go.

One caveat: don't fight your ecosystem. If you're doing ML work, you're using Python — that's where the libraries live. The point is to choose the strongest typing your ecosystem supports, not to abandon your ecosystem in pursuit of type safety.

THE BOTTOM LINE

Technical debt in the agent era moves faster and hides better than it used to. But if you understand your tools, set clear constraints, and build feedback loops into your workflow, you can move fast without waking up to a codebase you can't recognize.

The agents are only as good as the guardrails you give them. Build the guardrails.

// INSTRUCTORS

Not academics — practitioners. All three ship production code with AI daily.

Adam Charnock

linkedin.com/in/adamcharnock

Founder of Lithus, where he manages bare-metal Kubernetes infrastructure for SMEs. Previously contracted at Twitter and Blocknative. Adam has run production systems at scale for 18 years — he teaches how to calibrate control and let agents handle the right things.

Dominik Grusemann

linkedin.com/in/dominikgrusemann

Co-Founder of Marbles AI and former CTO of Chatchamp (acquired 2023). Previously led BMW's autonomous driving department toward agile. Dominik focuses on CLAUDE.md best practices and compound engineering.

Traun Leyden

linkedin.com/in/tleyden

Co-Founder of Fluensy.app. Ex-Databricks / Couchbase senior engineer with 20+ years shipping backend systems in Go, Python, and Rust. Traun brings deep experience in building reliable, well-tested software — and knows exactly where AI-generated code breaks down in production.

~/bettervibe $ cat about.md

AI can get you to a demo fast. Engineering judgment is what keeps it running in production. bettervibe teaches the craft in between.

A recurring hands-on workshop in Munich for developers who already ship with AI. Bring your laptop and a real project. We work it together: planning, agent orchestration, review loops, and the engineering judgment AI still requires. You leave with code in your repo and a methodology you can use the next morning.

// INSTRUCTORS

Adam Charnock — Founder, Lithus. Eighteen years of production infrastructure (Twitter, Blocknative).

Dominik Grusemann — Cofounder, Marbles AI. Former CTO of Chatchamp (acquired 2023). Previously led BMW's autonomous driving department toward agile.

Traun Leyden — Cofounder, Fluensy.app - Speak As Brilliantly As You Think. Twenty years shipping backend systems (Databricks, Couchbase).

Next dates and registration:
bettervibe.org