



The Varstätt Principles

The principles that separate work that
ships from work that drifts

Introduction



Models don't ship.

Most software projects fail the same way. Unclear scope turns six weeks into six months. A \$15K budget becomes \$50K. Developers build the wrong thing beautifully, on time, exactly as specified.

Most of that time goes into models of the work: calls, meetings, proposals, plans, boards, mockups. Models don't ship. The only thing that matters is the thing itself: working software in production.

After building software since 2011, five areas separate work that ships from work that drifts: Discovery, Delivery, Diligence, Partnership, and Philosophy.

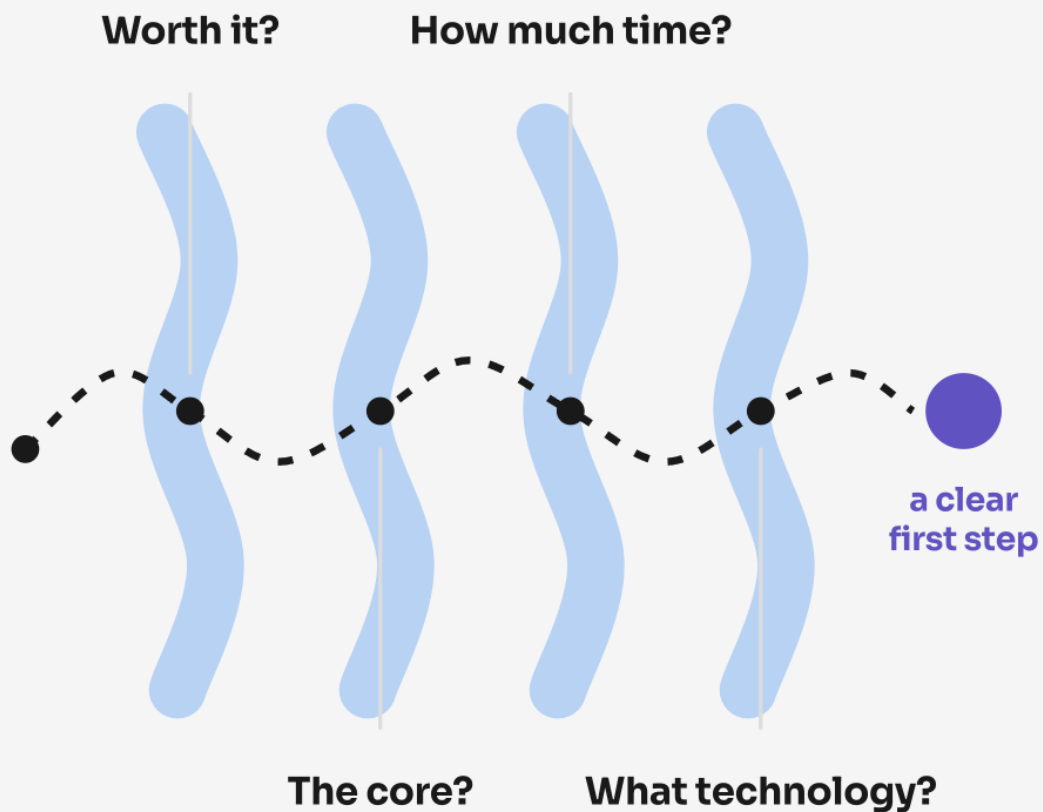
Contents

01 Discovery	1
Worth Building	2
Find the Core	3
Appetite, Not Estimates	4
Scope Shaping	5
Context Over Purity	6
Saying No	7
02 Delivery	8
Continuous Flow	9
WIP One	10
Production Is Done	11
Feature Flags	12
Scout Rule	13
Async Updates	14
Quality Gates	15
Default Stack	16
03 Diligence	17
No Split	18
Monitor Day One	19
Observe and Improve	20
Pause, Not End	21
Incident Response	22
Documentation	23
04 Partnership	24
Onboarding	25
Pricing Conversation	26
Transparency	27
Weekly Accountability	28
Priorities, Not Scope	29
Async First	30
Exit Freedom	31
05 Philosophy	32
Business Cost	33
Hours Are Wrong	34
Consolidation	35
Client Decides	36
Solo Developer Model	37
Client Owns Everything	38

01

Discovery

Figuring out what's worth building before
anyone writes code



Four questions answered honestly before anyone writes a line of code. Discovery produces a clear first step, not a complete roadmap.

Most founders hire developers before they know what they're building. They describe a vision, the developers nod, and everyone starts. Six weeks later, the thing that got built isn't the thing that was needed. Discovery exists to prevent that.

It's not a big process. It's four questions answered honestly before anyone writes a line of code — plus the discipline to shape scope and say no.

Is This Worth Building?

The client deserves a straight answer here, not encouragement. If there's no validated demand, if the problem doesn't exist the way the client thinks it does, or if the market is misunderstood — the developer says so. The client owns the business risk. The developer's job is to give an accurate picture so the client can decide with open eyes.

What's the Core Feature?

For a new product, it's the one capability that creates unique value — the reason someone would choose this over doing nothing. For an ongoing product, it's the next thing that unlocks the most value. Everything else waits. The core can't be found by listing features. It's found by asking what breaks if it's removed.

How Much Time Is This Worth?

Appetite replaces estimates here. The question isn't "how long will this take?" — it's "what can the developer deliver in six weeks that's actually useful?" Time-box first, then fit the scope inside it. This produces a real constraint instead of a prediction that's wrong by the time anyone reads it.

What Technology Fits?

The right technology is the one that fits the client's situation. Existing infrastructure, credits on a platform, a team already running something — that matters. Fewer services means less maintenance. Developer preferences don't enter into it.

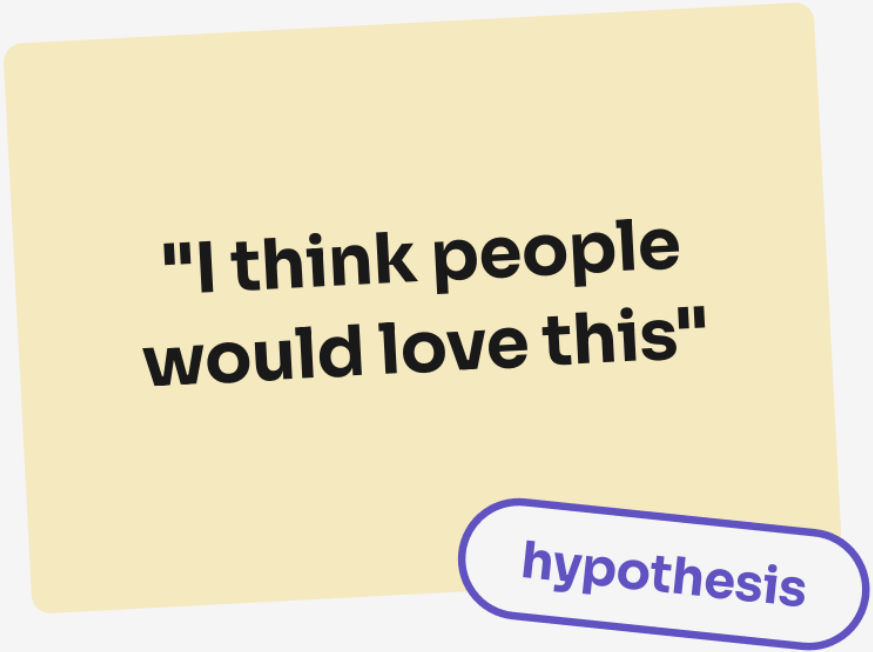
What's Ahead

Discovery produces a clear first step, not a complete roadmap. Roadmaps are fiction written in advance. What the client gets instead is an appetite boundary, a defined core, and technology choices grounded in business reality.

Six principles walk through Discovery step by step — starting with whether the idea is worth pursuing at all, narrowing down to the core feature, setting time boundaries, shaping scope to fit the appetite, making pragmatic technology choices, and having the discipline to say no when it's not the right fit.

Worth Building

Most things aren't worth building — figure that out first



**"I think people
would love this"**

hypothesis

"I think people would love this" is not market research. It's a hypothesis. Treat it like one.

The hardest truth in this work: most things aren't worth building. Not because they're technically impossible. Because the business case

doesn't hold, the market doesn't exist, or the problem isn't real. The developer's job is to figure that out before anyone writes a line of code.

Advise, Don't Decide

The developer isn't the business owner. The risk isn't theirs. But they've seen the patterns — solutions hunting for problems, products nobody asked for, "essential" features that sat unused. Use that. Ask: "Have you validated demand?" "How are people solving this today?" "What happens if you don't build this at all?" Have an honest conversation. Then [the client decides](#).

Red Flags Worth Naming

Unvalidated demand is the most common. "I think people would love this" is not market research. It's a hypothesis. Treat it like one.

Non-existent problems show up as cool technology looking for an application, or inefficient processes that aren't painful enough for anyone to actually change.

Market misunderstanding means building for the client's edge case instead of the customer's common case.

Already-solved problems deserve scrutiny. Competitors exist for reasons. "We'll do it better" needs evidence, not confidence.

None of these are automatic disqualifiers. All of them are worth discussing out loud.

The Honest Conversation

Say it plainly: "I don't think this has product-market fit yet." Or: "What if we tested demand with a landing page before building anything?" Some clients push back. Some know their market better than the developer does. That's fine — document the concern, build what the client wants, and learn together.

The clients worth keeping appreciate the honesty. They've been burned before by teams who built exactly what was asked and delivered exactly the wrong thing.

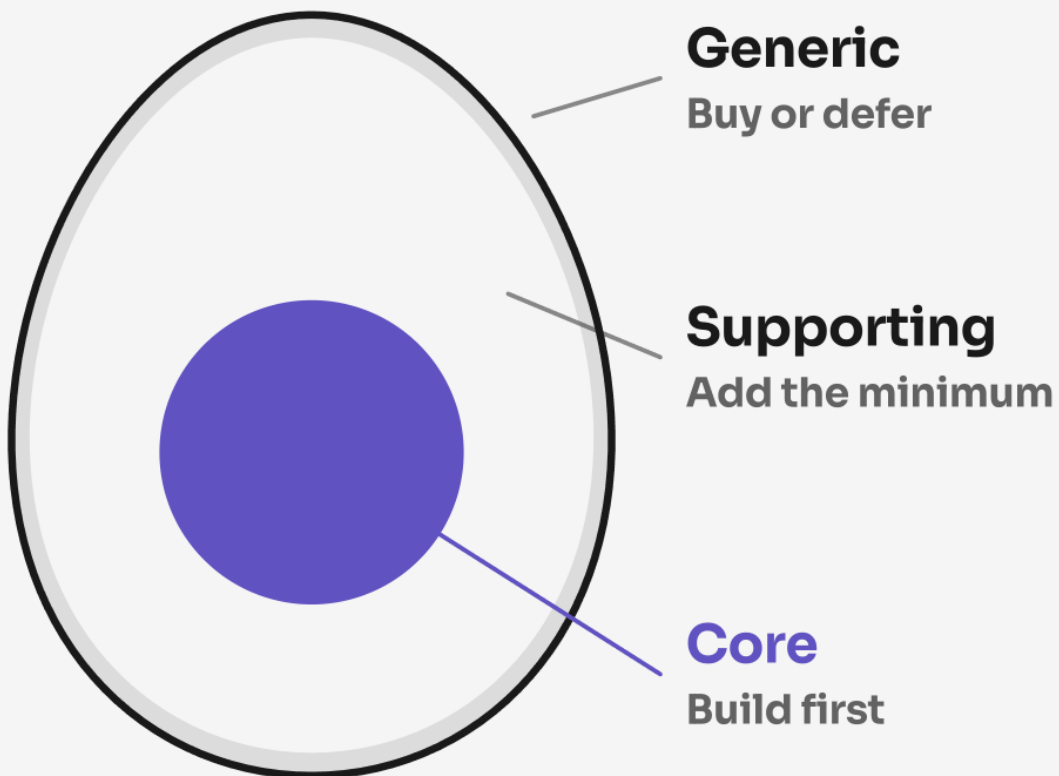
What This Prevents

The worst outcome in this work isn't a buggy launch or a missed deadline. It's perfect execution of the wrong idea. Software nobody uses. Nobody can prevent every failure. But the obvious ones are preventable.

Find the Core

The essential capability gets built first — everything else waits

Build the core first



Build the core first. Add the minimum supporting features to make it usable. Buy or defer everything generic.

Everything can't be equally important. The hardest part of any project isn't building — it's figuring out what actually matters.

The Feature List Trap

When scoping something new, the client provides a list: authentication, dashboard, notifications, export, integrations. That's not a vision. That's a backlog with no spine.

The [core domain](#) is the single capability that creates competitive advantage — the reason the thing exists. "This app helps recruiters find qualified candidates by analyzing resumes against job requirements using AI." That's a core. Everything else either supports it or doesn't belong yet.

The dashboard that presents matching results is a [supporting domain](#). It enables the core to be useful, but it's not the value itself — swap it for a different UI and the product still works. Authentication? Notifications? [Generic domains](#). Commodity. Buy them or defer them entirely.

Build the core first. Add the minimum supporting features to make it usable. Buy or defer everything generic.

Existing Software Is Messier

Ongoing projects don't have the luxury of a clean slate. There's debt, feature requests, bugs, and performance issues all competing for attention at once.

The client still can't do everything. The question is the same: what's the immediate priority that unlocks the most business value right now? Sometimes it's a new feature. Sometimes it's a critical bug costing customers. Sometimes it's a performance fix that unblocks everything else.

But it's [one thing](#). Pick it, execute it, move to the next.

Where Projects Waste Effort

Most teams spend the majority of their time building supporting and generic domains from scratch — user management, payment flows, file storage, email delivery. Things that have been solved a hundred times.

That's not competitive advantage. That's operational necessity. The market doesn't reward anyone for writing their own email queue.

Every hour spent on generic infrastructure is an hour not spent on the thing that actually differentiates the product.

The Forcing Question

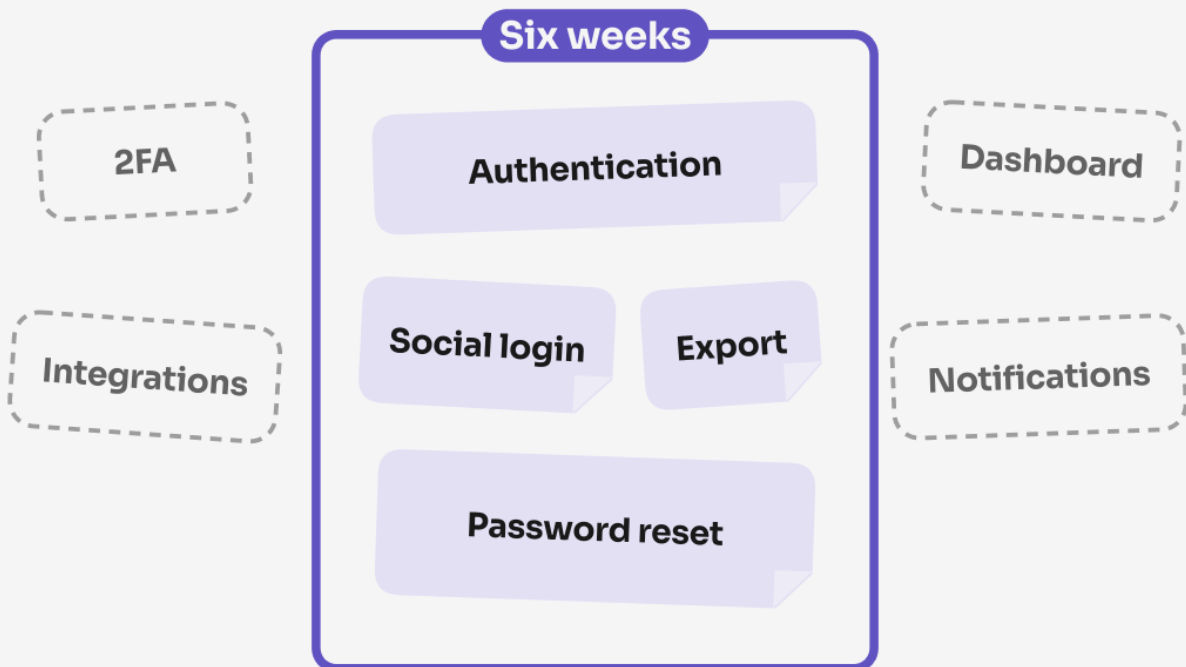
Ask it directly: if only one thing could be built, what creates the most value?

That's the core. Start there.

Appetite, Not Estimates

Time boxes replace estimates — fix the budget, flex the scope

Fixed budget, flexible scope



Fix the budget, flex the scope. If everything can't fit in six weeks, good: be honest about what's essential versus what's nice-to-have.

Estimation is waste. Estimates are based on known unknowns, but real problems are unknown unknowns. By the time the estimate is finished, the thing could've already been built.

Why Estimates Fail

Ask "how long to build authentication?" and the answer is a number. What that number won't include: social login breaking behind a corporate proxy, password reset emails getting spam-filtered, 2FA edge cases for international phone formats. None of that was in the estimate. All of it affects the timeline.

This isn't a planning failure. It's the nature of software. The [#NoEstimates](#) movement exists for a reason — estimation is difficult, often impossible, and almost always misused as a commitment.

Scope to a Time Box

Instead of asking "how long will this take?", ask "how much time is this worth?" That's appetite-based scoping. The client sets the time box first, then the developer figures out [what fits inside it](#).

Six weeks for this problem. What's the most valuable version that can ship in that window? If everything can't fit in six weeks, good. That means being honest about what's essential versus what's nice-to-have.

How It Works in Practice

Start with one question: "Six weeks — what's the single capability that creates real value?" Build that. Get it in front of users. See what actually happens.

Then, if needed: "Another six weeks. What should be added based on what was learned?" Iterative scoping beats upfront estimation every time, because it's grounded in reality rather than speculation.

When Clients Want a Total

Some clients will ask for a fixed-price total. That's a fair question with an honest answer: "Phase 1 is six weeks at \$X/week. After that, both sides will know whether Phase 2 is needed and what it should include. That's more accurate than a year-long estimate that'll be wrong."

Some clients need the false certainty of a fixed price. This approach isn't for them. For clients who value flexibility and honesty over fabricated precision, appetite-based work is the better deal.

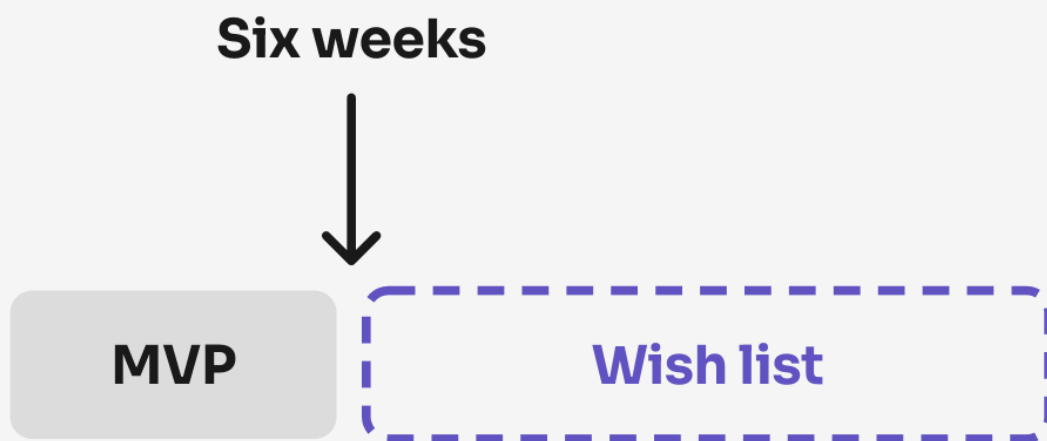
What the Developer Is Actually Selling

The developer isn't selling the ability to predict the future. That's a lie, and clients who've been burned know it.

The developer is selling incremental value, real adjustments based on what both sides learn, and honest communication throughout. That's the offer.

Scope Shaping

Fixed time, flexible features



Six weeks set the line. What fits is the MVP; the rest waits on the wish list.

The principle is simple: [fix time and budget, flex scope](#). The client has an [appetite](#). The [core is known](#). Now the developer needs to shape what

actually fits inside the time box. This is the craft between strategy and execution.

Flex the Feature, Not the Timeline

Most features can be built a thousand different ways. The full version with custom configuration, responsive design, and flexible workflows. Or the shaped version that gets users doing the new thing within the appetite.

Instead of user-configurable settings, use presets. Instead of responsive design across every breakpoint, ship desktop only. Instead of custom workflows, use templates that cover most of the ideal flow but take a fraction of the time to build.

Only with Client Approval

This only works with client approval. No flexing behind their back. Ever.

But when the other principles are followed — [transparency](#), real commitment, trustworthiness — clients rarely push back. They see the real scenario. They understand work isn't predictable. When the developer is honest about what fits and what doesn't, and proposes a solution that delivers what must be there, clients appreciate the approach.

The client approves the flex. The developer ships on time. Then the rest gets scheduled.

Less Is Often Enough

And sometimes? The flexed version turns out to be enough. The full thing never gets built because reality proved the simpler version was all anyone needed. Less code, same outcome.

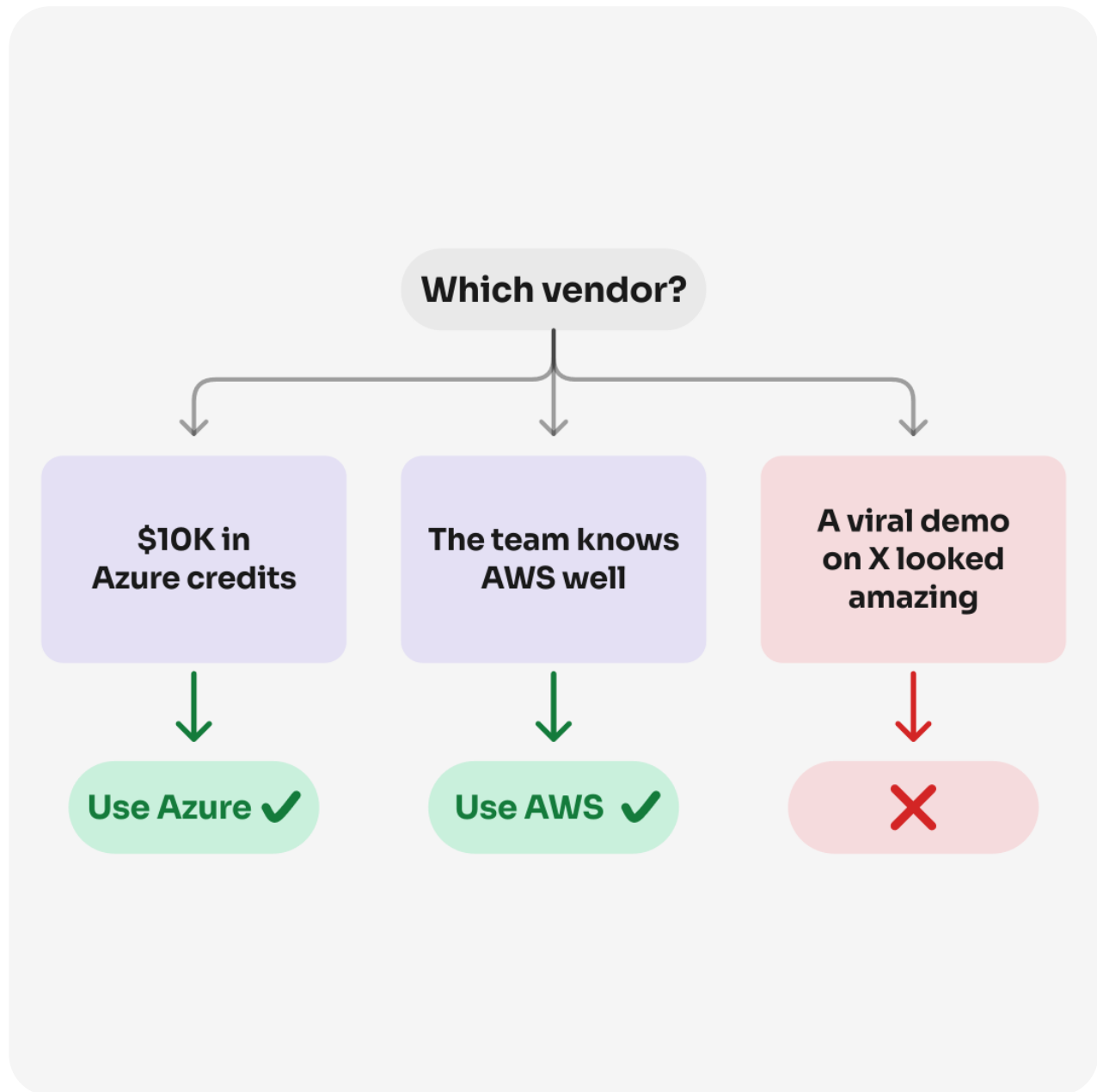
Real Deadlines vs Fake Ones

Most deadlines are fake. A real deadline is when the work has no value after the date. An event management app after the event is worthless. That's a real deadline.

"We need this by Q3" is usually just a wish. Know the difference. Shape scope against real constraints, not arbitrary dates.

Context Over Purity

Pragmatic decisions beat perfect theory



The best technology stack is the one that fits the client's actual constraints.

The best technology stack is the one that fits the client's actual constraints. Not the one that's theoretically pure. Not the one that's fashionable on Hacker News this month.

Business Constraints Win

The client has \$10K in Azure credits from a startup program? Use Azure. The client's team knows AWS well? Use AWS. The existing infrastructure runs on Google Cloud? Stay there.

None of these are wrong choices. They're right business decisions. Yes, the developer has preferences. Yes, some stacks are objectively better for specific cases. But context beats preference every time.

Fewer Moving Parts

When making technology choices, bias toward fewer services and fewer integrations. Every additional tool is another thing to learn, maintain, and debug. During discovery, this means asking: "Can the platform already chosen handle this, or is something else truly needed?"

Most of the time, the chosen platform handles it. The [consolidation](#) argument runs deeper than a discovery decision — it's a philosophical position explored fully in the Philosophy section.

The Default Stack Isn't an Accident

A good [default stack](#) is predictable, well-documented, and handles common cases without drama. The developer adapts when business needs require it, and defaults to proven simplicity when they don't.

In Varstatt, that's React, Node.js, and Firebase — covering the vast majority of what clients actually need. Same language on the front end and back end. A unified mental model. A massive ecosystem. Not because

JavaScript is the best language ever invented, but because this stack is proven at scale by thousands of teams.

How to Have the Conversation

Lead with reasoning, not dogma. "The recommendation is X because of these constraints. If the client has strong preferences for Y, the developer can work with that — here are the tradeoffs."

No lectures. No dogma. Context-appropriate choices with clear reasoning.

Saying No

Not every project deserves the code

No is easier at the start



Say no early. Take work out of desperation and they'll cut corners on principles. And it never finishes well.

Every developer needs a line. Not a vague sense of discomfort — an actual filter. Projects that cross it get declined. No negotiation, no "just this once."

Define the Filter

The filter is personal. It reflects what the developer believes about the work they put into the world. Some won't build surveillance tools. Some won't work with industries they find exploitative. Some draw the line at dark patterns designed to manipulate users.

The specifics don't matter as much as having them. Without a clear filter, every questionable project becomes a debate — and when money's tight, the debate always ends the same way.

In Varstatt, the filter is straightforward: the work must contribute to humanity, or at least not hurt it. Projects that play on people's weaknesses — gambling, exploitation, manipulation — get declined. Period.

When It Actually Gets Hard

This sounds obvious until savings are running low and a well-paying project lands in the inbox. An escort directory with a polished PRD. A fantasy sports gambling platform with a tight deadline and real budget. The brief looks professional. The money looks good. The need feels real.

Say no anyway.

The moment the developer takes work out of desperation, they've already lost. They'll resent the project. They'll cut corners on principles. And it never finishes well. Never.

How to Decline

The how matters less than the doing. Keep it short. Keep it professional. Don't preach. Don't explain the moral framework. Just decline.

"Thanks for sharing the detailed requirements. After reviewing everything, this project isn't the right fit. If there are other ventures in the future, feel free to reach out."

No sermon. No bridge burned. Door open for different work.

The clients the developer says no to aren't their clients. The projects that conflict with how the work gets done will drain worse than an empty bank account. These principles aren't a luxury affordable only when business is good. They're the foundation that makes good business possible.

Say no early. Say no clearly. Say no when it hurts.

02

Delivery

Continuous flow replaces sprint ceremonies and batch releases



Working in production is done.

No plan survives contact with development. The question isn't how to prevent change — it's how to absorb it without losing momentum.

Continuous Flow Over Sprints

Weekly billing creates the payment rhythm. Not sprints. This is [Kanban](#): work flows continuously from a priority queue. It comes in, gets ranked, gets executed, gets deployed. No velocity tracking. No burn-down charts. No planning poker.

The only question that matters: what's most important right now?

For meaningful phases — major features, big rewrites — think in [6-week cycles](#). Long enough to ship something end-to-end valuable. Short enough to stay honest. But these aren't fixed time-boxes with commitments baked in at the start.

WIP Limit of 1

One task at a time. Finish it. Deploy it. Then move on.

Starting something new? Stop the current task first. This forces honest prioritization and kills context switching. It prevents the trap of being "90% done on five things" while nothing is actually working. The client sees consistent forward motion. The developer stays focused.

Production Is Done

Code review isn't done. QA passed isn't done. Merged isn't done. Working in production is done.

This changes how the developer thinks about deployment. If deploying is hard, it gets avoided. If it's easy, it happens constantly. Feature flags handle incomplete work — deploy behind the flag, keep building, flip it when it's ready. The main branch stays production-ready. Always.

Scout Rule

Leave code better than it was found. That's the entire refactoring strategy.

No scheduled debt sprints. No percentage allocation. Just improve what gets touched. With a solo developer and consistent patterns, technical debt stays manageable without ever making it a project.

Communication While Delivering

Regular updates go out after each work session — not at end of day, but when the work is actually done. Meetings happen only for decisions that genuinely need real-time discussion. Everything else is written.

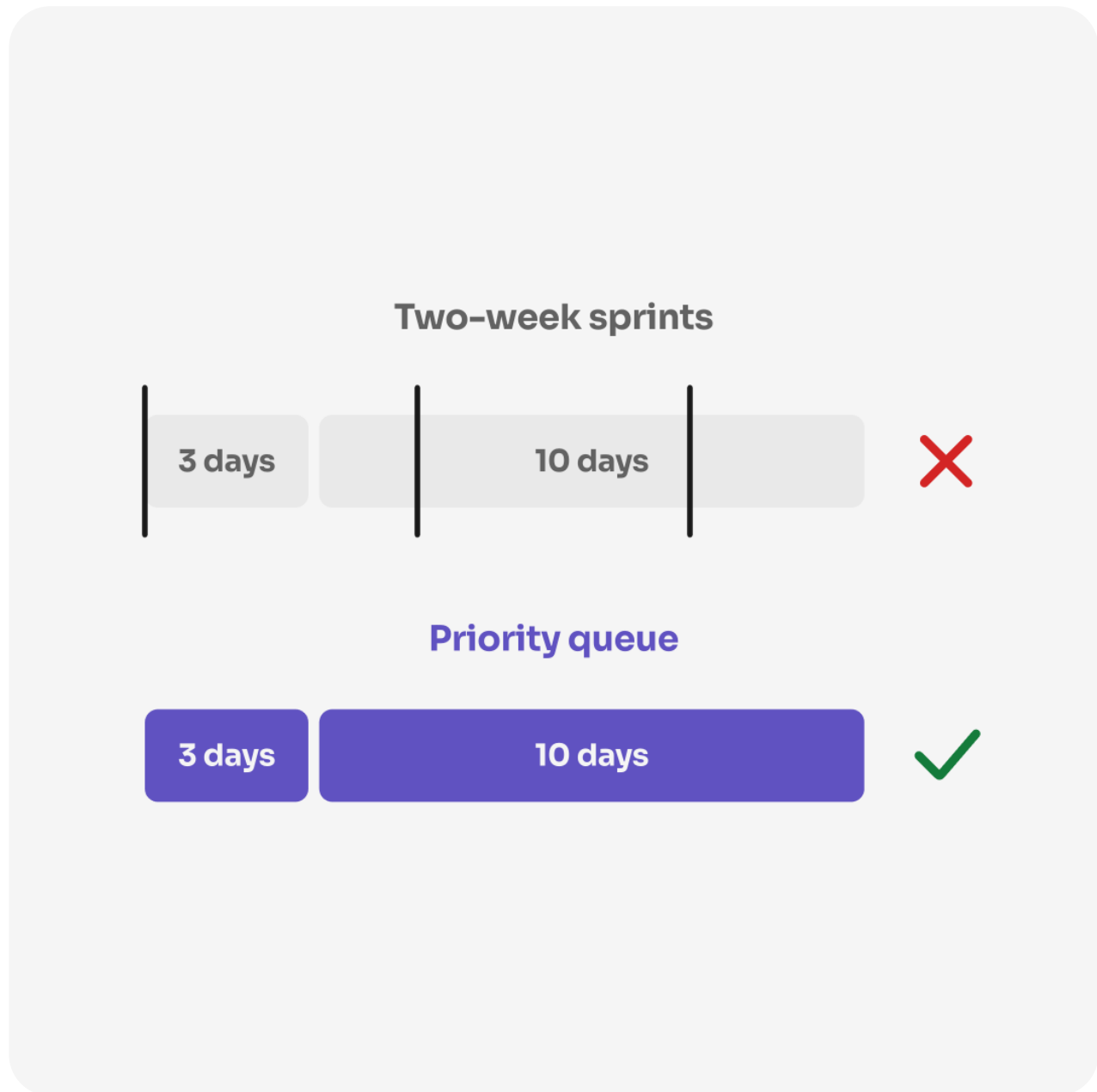
This keeps calendars empty and focus time protected. Communication overhead stays near zero.

What's Ahead

Eight principles govern how delivery works in practice — from the overall flow of work, through the constraints that keep it honest, to how quality is maintained, how progress gets communicated, and the tools used to ship.

Continuous Flow

Kanban over sprints — a priority queue, not fixed time boxes



Work doesn't fit neatly into week-long boxes. A priority queue, not fixed sprints.

Kanban, not Scrum. A priority queue, not fixed sprints. Continuous delivery, not batch releases.

Work doesn't fit neatly into week-long boxes. Some tasks take three days. Some take ten. Forcing them into artificial time boundaries adds ceremony without adding value.

No Ceremonies

There's no velocity tracking, no planning poker, no burndown charts. Just a priority queue. Pull the next task, execute it, deploy it, repeat.

The Priority Queue

The client maintains a [ranked list](#). The top item is the highest priority. The developer works top-down: finish what's in front, then pull the next thing.

Priorities shift? The client reorders the list. No sprint replanning, no ceremony. The developer is always working on what matters most right now.

This requires discipline on both sides. The client has to actually prioritize — not mark everything urgent. When it works, it cuts through ambiguity fast. No scope debates, no negotiation. Just a clear stack and continuous execution.

Six-Week Cycles for Bigger Bets

For significant features, think in [six-week cycles](#) — long enough to deliver something end-to-end valuable. A core capability implemented, integrated, tested in production, and iterated on based on real feedback.

A cycle isn't a deadline. It's a planning horizon. "In six weeks, we expect X to be working." Comes in at five weeks? Great. Slips to seven? Fine. The cycle serves orientation, not ceremony.

Why This Works

Software development isn't predictable enough for two-week sprints. Requirements clarify during development. Priorities shift based on what users actually do. Technical surprises surface at the worst times.

Continuous flow embraces this reality. The developer is always working on the most important thing, even as that definition changes. There's no overhead from sprint planning, no theater around velocity, no rituals that exist to feel productive rather than be productive.

Build it. Deploy it. Get feedback. Pull the next priority.

WIP One

One thing in progress at a time — finish it, deploy it, move on

Four tasks at a time



One task at a time



Four tasks at 80% ship nothing. One at a time, the same hours ship all four.

One task in progress at a time. That's the default. It prevents most of the chaos.

How It Works

If the current task isn't done, don't start another one. If priorities shift, stop the current work first. If the client needs something urgent, pause current work and start the urgent task.

Can the developer work on two things at once? Sure — at their own risk. Without solid CI/CD and clean branching, two in-progress tasks turn into merge conflicts, half-finished features, and context that doesn't fit back together. Most of the time, it's not worth it.

Blocked Is Not In Progress

When a task can't move forward — waiting on client feedback, a third-party dependency, an external approval — it's blocked. Mark it as blocked, document what it's waiting on, and move to the next task. That's normal workflow, not an exception.

The key distinction: blocked means parked, not abandoned. The developer checks regularly whether the blocker has cleared. When it does, the blocked task takes priority over whatever else is queued — unless something is already in progress, in which case finish that first.

This keeps the board honest. "In progress" means someone is actively working on it right now. "Blocked" means it's waiting on something specific. No ambiguity.

Why Developers Resist It

"These tasks are related, I can work on both." Related isn't the same as identical. Finish one, then start the other.

"I'm more productive when I switch tasks after hitting a wall." That's context switching dressed up as productivity. Maintaining two mental

models simultaneously is slower, not faster. The developer isn't beating the system — they're fooling themselves.

Why Clients Love It

Every [status update](#) becomes clean: "Working on X, it's Y% done, Z is blocked on client feedback." That's useful.

Compare that to: "A is 60% done, B is 30%, C is in review, D is waiting on E." Nobody knows what's happening, including the developer. Progress is invisible. Things fall through cracks.

One thing in progress means one honest answer when someone asks where things stand.

What It Prevents

The "90% done on five things" trap. Nothing shipped, nothing delivering value, just complexity piling up. The client can't use 90% done. Nobody can deploy 90% done.

It also closes the back door on scope creep. Adding something new means first stopping what's current. That forces an honest conversation about [priorities](#) instead of quietly absorbing extra work.

Production Is Done

Work is only done when it's live in production

Production is done

Code review



QA passed



Merged



Working in production



Done means one thing: released to real users in production, with monitoring in place.

"Done" is one of the most abused words in software. Code review passed? Not done. QA signed off? Not done. Merged to main? Still not done. Done

means one thing: released to real users in production, with monitoring in place.

Everything else is in progress.

Deploy Is Not Release

These two words get used interchangeably. They shouldn't.

Deploy means the code is running in production. It's on the servers, it's executing, but users may not see it yet. A deploy behind a disabled [feature flag](#) is still a deploy — the code is live, but the feature isn't.

Release means users interact with it. A feature flag flipped to 1% of traffic is a release. A new page accessible to everyone is a release. An A/B test running against real users is a release.

The distinction matters because it separates the risk of shipping code from the risk of exposing functionality. Deploys should be boring and frequent. Releases are business decisions — they happen when the feature is ready, not when the code happens to land.

What Doesn't Count

Passing code review means the code looks right to someone. It doesn't mean users are getting value from it. QA approval means it works in a controlled environment. Production is not a controlled environment. Staging is close, but staging isn't production — real traffic, real data, and real edge cases live in production only.

Work that isn't deployed doesn't count. For anyone.

Why the Definition Matters

When release equals done, ambiguity disappears. No more "it's done on my machine." No more "it's basically done, I just need to deploy it." The definition forces the developer to make deployment simple. If deploying is painful, it gets avoided. If it's routine, it happens constantly — and that's exactly the behavior both sides want.

It also kills the "one more thing" trap. The moment done means released, there's no room for "let me just add this small thing before it goes out." It's live. Improvements go in the next release.

Feature Flags Bridge the Gap

Deploy early, release when ready. Push code to production with the flag disabled. Keep developing. Test against the real production environment. Flip the flag when it's ready. The main branch stays production-ready at all times — no long-lived feature branches, no integration hell at the end of a sprint.

Incomplete work gets deployed safely. Complete work gets released deliberately. The two don't have to wait on each other.

Deployment Is a Process, Not an Event

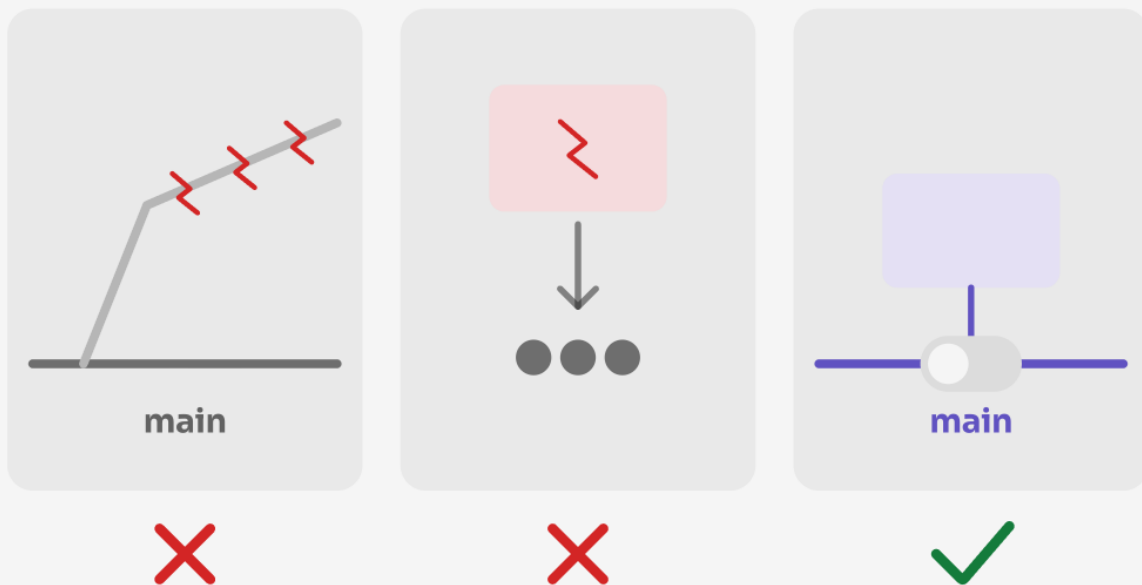
Deploy multiple times a day if the work supports it. Once a week if that's the rhythm. The frequency matters less than the friction. Automated CI/CD, a full test suite, one-command deployments — when that infrastructure is in place, deploying stops being a discipline and starts being the natural way work happens.

Production becomes the default state, not a destination visited occasionally.

Feature Flags

Flags decouple deployment from release — ship safely, release when ready

Ship in chunks, switch on when ready



Both options are bad. Feature flags are the way out.

Main branch should always be [production-ready](#). Feature flags make that possible by decoupling deployment from release.

The Branch Problem

A feature takes three days to build. It can't be deployed half-finished, so it sits in a branch. Meanwhile, main keeps moving. Merge conflicts pile up. By the time it's ready, integration is a slog. The alternative — deploying broken functionality — is worse. Both options are bad. Feature flags are the way out.

How They Work

The developer deploys the code behind a flag that's disabled in production. The feature exists in the codebase but users can't see it. Development continues, testing happens against real production infrastructure, then the flag gets flipped when it's ready. The feature goes live instantly, no deployment needed.

Why It Matters

No long-lived branches means no integration hell. The developer works on main, everything stays compatible. Testing happens in the actual production environment — staging never fully matches production. The database is different, third-party APIs behave differently, network conditions vary. Testing behind a flag catches real issues before real users hit them.

Implementation

Start simple. A boolean check is enough:

```
if (featureFlags.newCheckout) {  
  return <NewCheckout />;  
}
```

From there it can get more precise. Environment-based flags are enabled in staging and off in production. User-based flags target admins or

internal testers. Percentage-based flags roll out to 10% of users before going broad. Dedicated feature flag services handle the complexity if needed. Don't over-engineer it before there's a reason to.

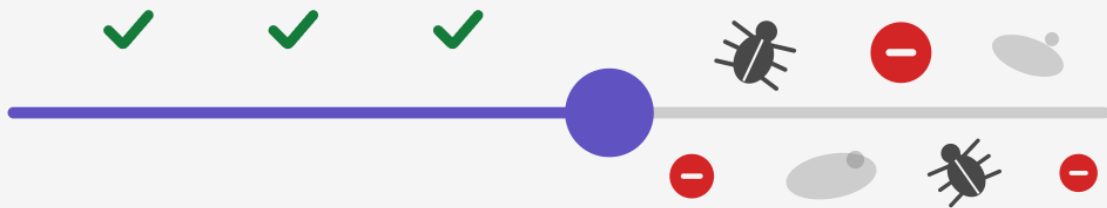
What This Prevents

The "big bang" deployment. Two weeks of work shipped all at once. Something breaks. Nobody knows which change caused it. Rolling back means losing everything. With feature flags, each change is isolated. Problems are obvious. Rollback is surgical — flip a flag, not revert a month of commits.

Scout Rule

The codebase gets better with every change — no separate refactoring projects

**Leave the product better
than it was found**



Leave the code better than it was found. That's the entire technical debt strategy.

Leave the code better than it was found. That's the entire technical debt strategy.

While There Anyway

The developer is working on feature X in file Y. File Y has cruft — inconsistent naming, muddy logic, missing tests, unclear structure. Clean it up. Rename the variables. Extract the method. Add the test. No separate refactoring project needed. Just improve what gets touched, as it gets touched.

This is how debt gets paid down without anyone noticing. Continuous improvement baked into normal work, not a special-case interruption that needs to be scheduled, justified, and prioritized against features.

Technical Debt Is the Developer's Problem

The client shouldn't have to care about technical debt. They shouldn't hear about it in standups, see it on the board, or be asked to prioritize it against features. Technical debt is the developer's environment — like a carpenter's workshop. Keep it clean so the work stays efficient.

The same goes for testing. Unit tests, integration tests, whatever the project needs — the developer writes them to sleep well at night, not because the client asked. The client cares about working software. How the developer ensures it works is their responsibility.

Why Context Matters

The best time to improve code is when the developer is actively working in it. The understanding is already there. The code has just been read line by line. That context is valuable — use it. An hour later, the developer has moved on. A month later, the details are forgotten entirely.

No permission needed. No explaining [technical debt](#) to non-technical stakeholders. Better code is just a side effect of shipping the feature.

When Debt Blocks a Feature

Sometimes the debt is in the way. Can't build the new feature without fixing the data model first? Good — fixing it is the first step of the task. It's not a separate ticket. It's not a negotiation with the client. It's part of the work.

Major architecture changes, performance work, security fixes, foundational upgrades — these can't always be done in passing. But the trigger is always the same: the business need is real and the cost of not doing it is blocking progress.

The Solo Advantage

Most technical debt compounds because multiple developers create inconsistent patterns. Different styles, conflicting approaches, disagreements never resolved. Working [solo](#), the developer has consistent patterns throughout. The scout rule keeps them clean.

When a second developer eventually joins, they're joining a codebase with clear conventions and established quality. That's far easier to onboard into than a mess accumulated over years of well-intentioned shortcuts.

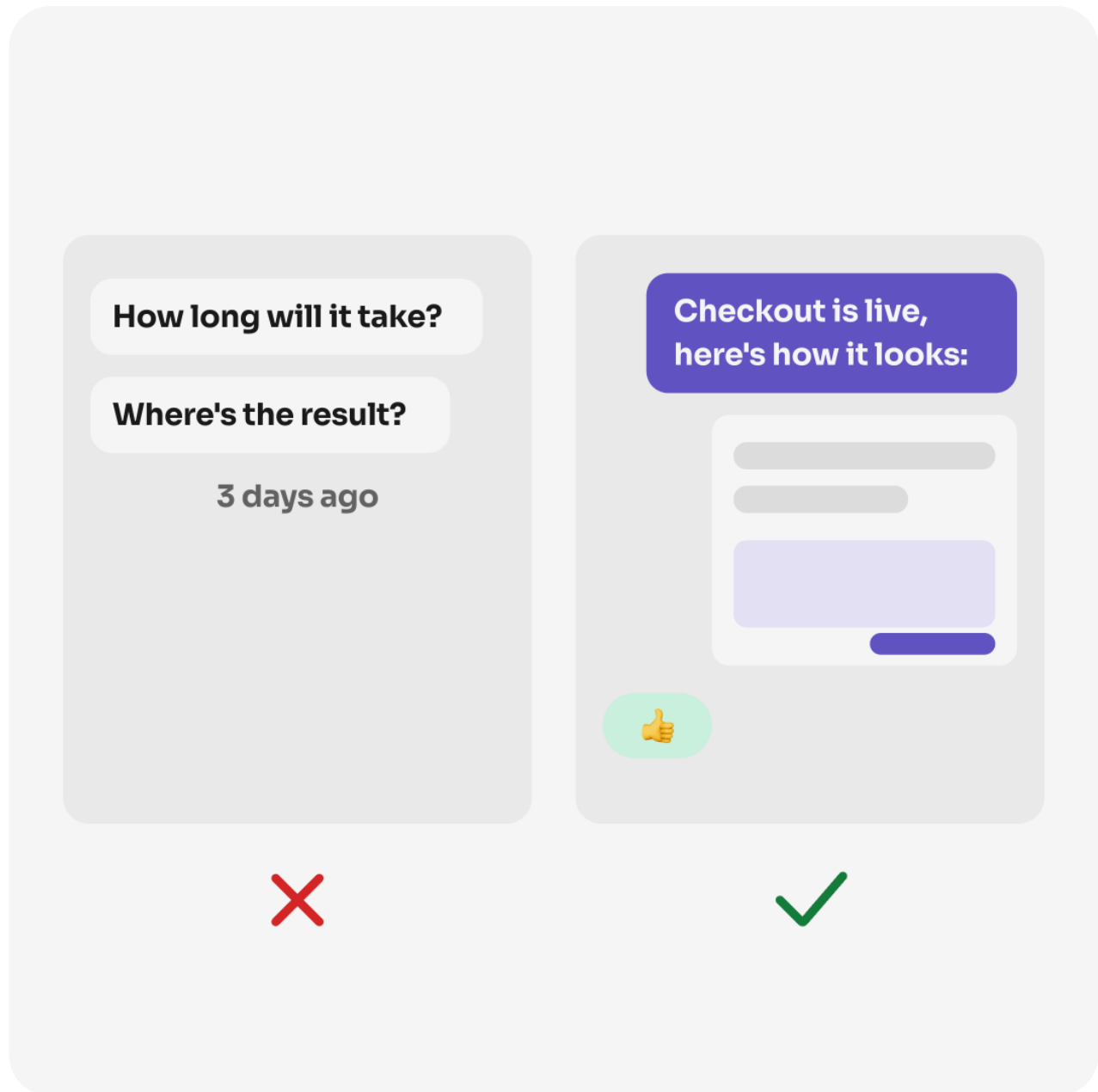
In Practice

Before: muddy variable names, tangled control flow, duplicated logic.

After: clear names, obvious flow, extracted shared logic. It takes five extra minutes. It saves hours of confusion later. That trade is always worth making.

Async Updates

Progress is communicated asynchronously, not in meetings



The async update replaces the status meeting entirely.

Writing is the delivery communication layer. Every work session produces a written artifact — not a meeting, not a call, not a Slack ping asking "how's it going?"

Write After Every Session

After each work session, the developer posts an update. What got done. What's next. What decisions were made and why. A screenshot, a link to the deployed feature, a short paragraph of context. Simple.

The client sees progress continuously. No waiting for a weekly status meeting. No "any updates?" emails because the update is already there. The async update replaces the status meeting entirely.

What Goes in an Update

Keep it concrete. "Implemented the checkout flow. Stripe integration is working in staging. Still need to handle the webhook for failed payments — doing that next." Link to the deployed change if possible. Attach a screenshot if it's visual. Flag blockers explicitly: "Blocked on: need the production API key from the client."

Avoid vague updates. "Worked on the backend" tells nobody anything. Name the thing. Show the thing.

Decisions Get Captured in Real-Time

When the developer documents a decision right after making it, the reasoning is still fresh. "Went with approach X because Y" — that context lives in the thread forever. Six months later, someone asks why — the answer is already there. No reconstructing half-remembered conversations.

This also protects both sides. When the client asks "why did we do it this way?", the answer is already in the thread. No defensiveness needed. Just a link.

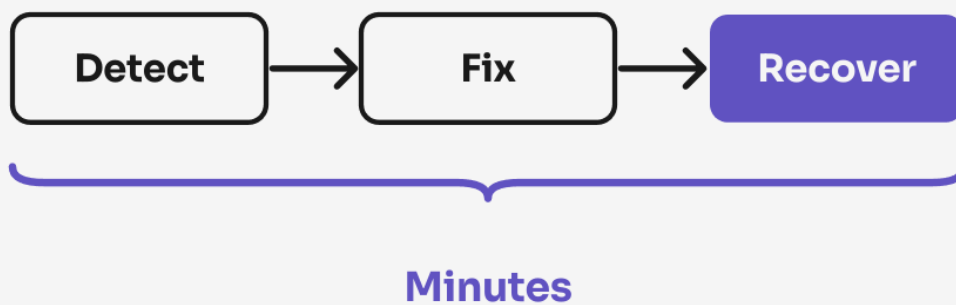
The Compound Effect

Updates accumulate into a project history. After three months anyone can scroll back and see every decision, every feature shipped, every blocker resolved. That's more useful than any retrospective. It's real-time [documentation](#) that writes itself as a side effect of working.

Quality Gates

Safe to fail, fast to fix

Safe to fail, fast to fix



The goal isn't zero bugs. It's zero surprises that can't be recovered from in minutes.

Software development has two schools of thought about quality.

Fail-safe tries to prevent every issue. More tests, more checks, more approval gates. In some domains this makes sense — banking, medical devices, regulated industries. When regulations force the hand, comply.

Safe-to-fail acknowledges reality. Bugs will happen. No amount of testing catches every issue before it reaches production.

Pick [safe-to-fail](#).

Instead of preventing every bug, focus on how quickly the developer can detect, fix, and recover. This is where the other principles earn their keep. [Monitor from day one](#). Deploy small chunks often. Use [feature flags](#) to roll back instantly. The safety net isn't a test suite — it's the ability to respond.

About tests specifically: it depends on the project. Sometimes there's a dedicated tester. Sometimes the developer writes unit and integration tests. Sometimes it's deploy-and-pray — check it in production.

In Varstatt, here's a strong opinion: using TypeScript correctly can replace the majority of unit tests otherwise written for data validation and type safety. When data structures are enforced at compile time, entire categories of bugs disappear before anything runs.

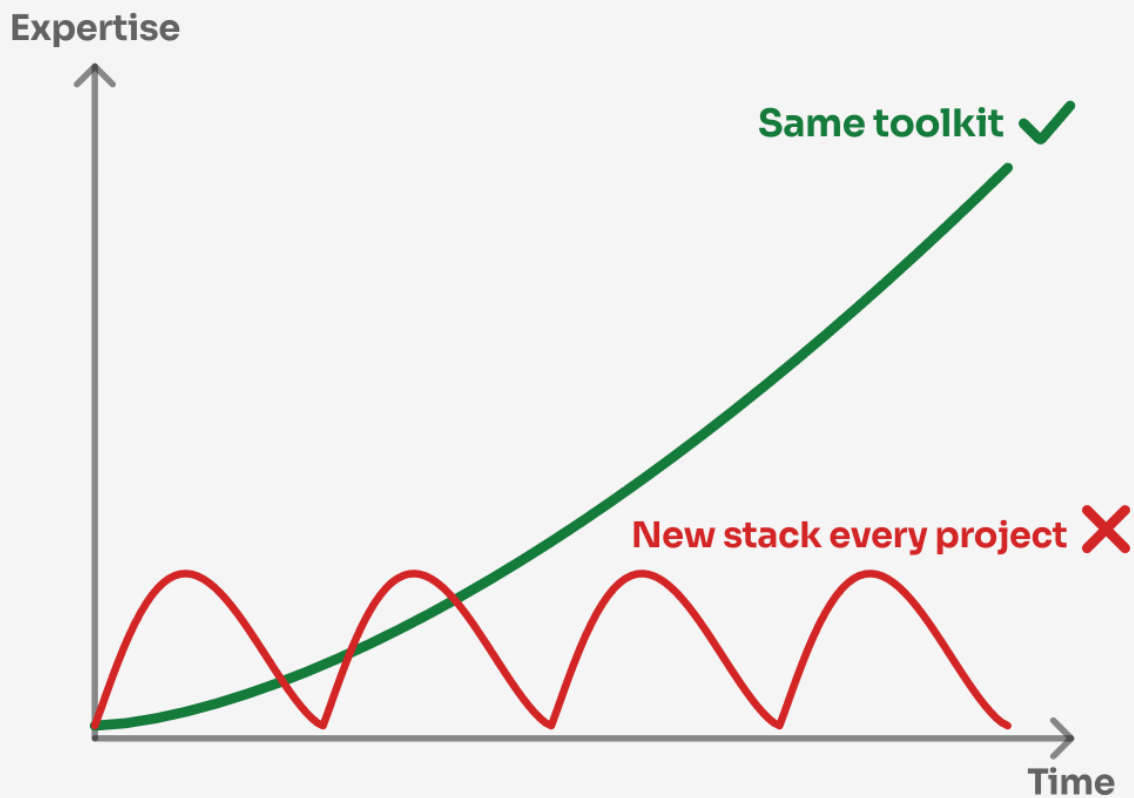
Sometimes the developer deliberately skips TypeScript to allow flexibility in data. In those cases, don't assume data will be there as expected — handle the absence gracefully instead of crashing.

"Test adequately for the risk level" means exactly what it says. A payment flow gets more scrutiny than a settings page. A user-facing feature gets more attention than an admin tool. Match effort to the stakes.

The goal isn't zero bugs. It's zero surprises that can't be recovered from in minutes.

Default Stack

Deep expertise in familiar tools beats starting fresh each time



Consolidation isn't stubbornness. It's expertise compounding over time.

Would a client trust a mechanic who asks what tools to use to fix the car? Of course not. The client expects the mechanic to have their own tools, know them deeply, and get the job done.

But if the client comes with suggestions — a specific part, a preferred brand — a good mechanic adapts. They expand their default stack with the client's input.

A default stack is what the developer uses when building from absolute scratch, when the client has no existing infrastructure or opinion. Without one, tool decisions happen on every project instead of building expertise.

Within the default stack the developer accumulates everything: know-how, assets, patterns, scripts, shortcuts. They know what works. They know the pitfalls. They ship faster because they've solved these problems before.

One Language, End to End

The strongest default stacks use one language across the entire stack — frontend, backend, infrastructure. Same mental model, same tooling, same debugging. The specific platforms and frameworks follow the [consolidation](#) principle: fewer moving parts, one coherent toolkit.

In Varstatt, that language is JavaScript and TypeScript everywhere. This stack builds 80% of modern web applications for almost any niche, any usage. Full-stack, end-to-end product development from a single coherent toolkit.

How Far Is Too Far

No default stack covers everything. Clients bring existing infrastructure, preferred tools, specific requirements. The question isn't whether to adapt — it's how much adaptation the developer can absorb before the expertise advantage disappears.

Think of it as vector distance — how far is the client's needed stack from the developer's default? A different cloud provider with the same language and patterns? Close enough. Same principles, slightly different tools. A completely foreign language with an unfamiliar backend? That's far. Probably not the right mechanic.

Of course the developer adapts. But adaptation has a cost. The further from the default, the less benefit from years of accumulated expertise. At some point the distance is too great and the developer is learning on the client's dime. That's not honest.

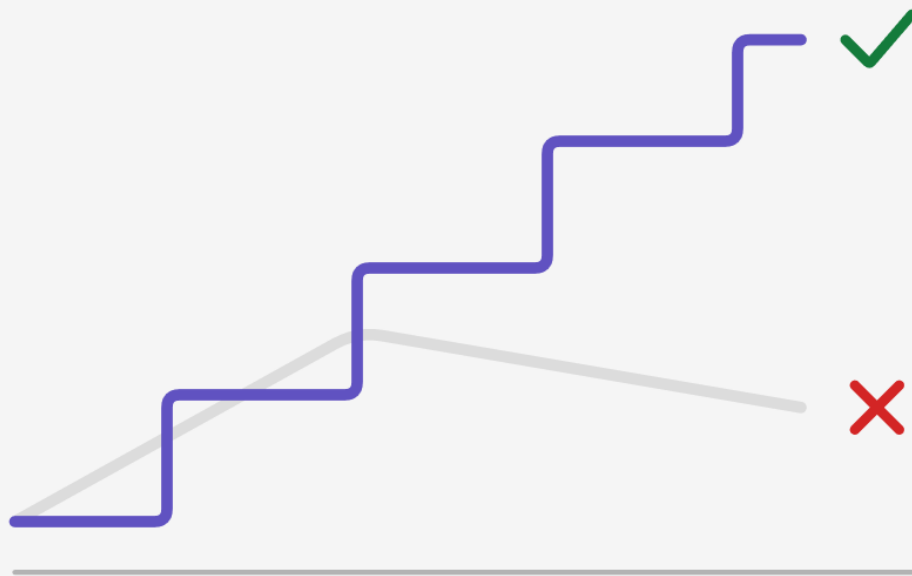
Consolidation isn't stubbornness. It's expertise compounding over time.

03

Diligence

Software evolves or dies — quality and
discipline keep it alive

Software that doesn't evolve dies



Improve based on those patterns — not a roadmap
someone wrote before the product existed.

The biggest lie in software is "we're done building, now we maintain it." There's no such thing. Software evolves or it dies.

The developer ships a feature, watches how people actually use it, fixes what breaks, improves based on what they learn, and adds what the reality demands. That's not a build phase followed by a maintenance

phase. That's just development — continuous, grounded in what's actually happening instead of what anyone assumed six months ago.

The difference between "active development" and "maintenance" is velocity, not nature. Some weeks the developer ships something significant. Other weeks it's an edge case three users hit on a Tuesday. It's all the same work.

Monitor From Day One

Monitoring isn't something to bolt on after launch. Analytics, error tracking, and alerts belong in the initial build. Waiting until after the first deploy to ask "is this working?" means losing the window to catch the first wave of real-world problems.

The developer needs to know: Is it working? Are people actually using it? Where do they get stuck? What falls apart under real load? Those questions can't be answered without instrumentation that was there from the start.

Watch What It Actually Does

Observability means watching how the software behaves in the wild — not how anyone thought it would, but how it does. Users do unexpected things. Edge cases surface. Performance bottlenecks appear under load nobody simulated.

Analyze what's happening continuously. Patterns emerge fast when the developer is paying attention. Improve based on those patterns — not a roadmap someone wrote before the product existed.

Pause Instead of Stop

When work naturally slows — waiting on a client decision, only low-priority improvements on deck, a seasonal quiet period — the client pauses the engagement. The client resumes when real work returns.

There's no forced "project end" decision. The relationship continues. The codebase stays familiar. Context doesn't evaporate. The engagement scales to match actual need. That kind of flexibility only works with weekly accountability. Monthly contracts can't bend this way.

What's Ahead

Six principles define what diligence looks like in practice — starting with the fundamental rejection of the build-versus-maintain split, then covering how to instrument, observe, sustain the work over time, respond to incidents, and document decisions.

No Split

Development and maintenance aren't separate phases

Maintenance is still development



No handoffs, no lost context. Just continuous work on running software.

The industry pretends there's a clean line between "building new features" and "maintaining existing software." There isn't.

What Actually Happens

The developer builds a feature. Deploys it. Users start using it. They hit edge cases nobody anticipated. The developer fixes those. The client asks for small improvements. Those get added. Performance degrades at scale. The developer optimizes. Meanwhile the next feature is being built, which connects to the first one, which requires changing how the first one works, which surfaces assumptions buried in the original implementation.

Is that "building" or "maintaining"? Both. Simultaneously.

The False Distinction

The classic split looks clean on paper: a development team builds new things, a maintenance team fixes what's broken. Different teams, different skills, clean handoffs.

What actually happens is information loss. The development team never sees how their features perform in production. The maintenance team doesn't understand the original design intent. Quality suffers. Velocity suffers. Context gets destroyed at every handoff.

No Handoffs, No Lost Context

When [one person](#) builds the feature, deploys it, watches it in production, fixes the issues, optimizes the slow parts, and integrates the next thing — there's nothing to hand off. No phase transitions. No "that's not my problem, I'm on the build team." Just continuous work on running software, informed by how it's actually being used.

That's the whole model. It's simpler and it's more honest.

Maintenance Is Development

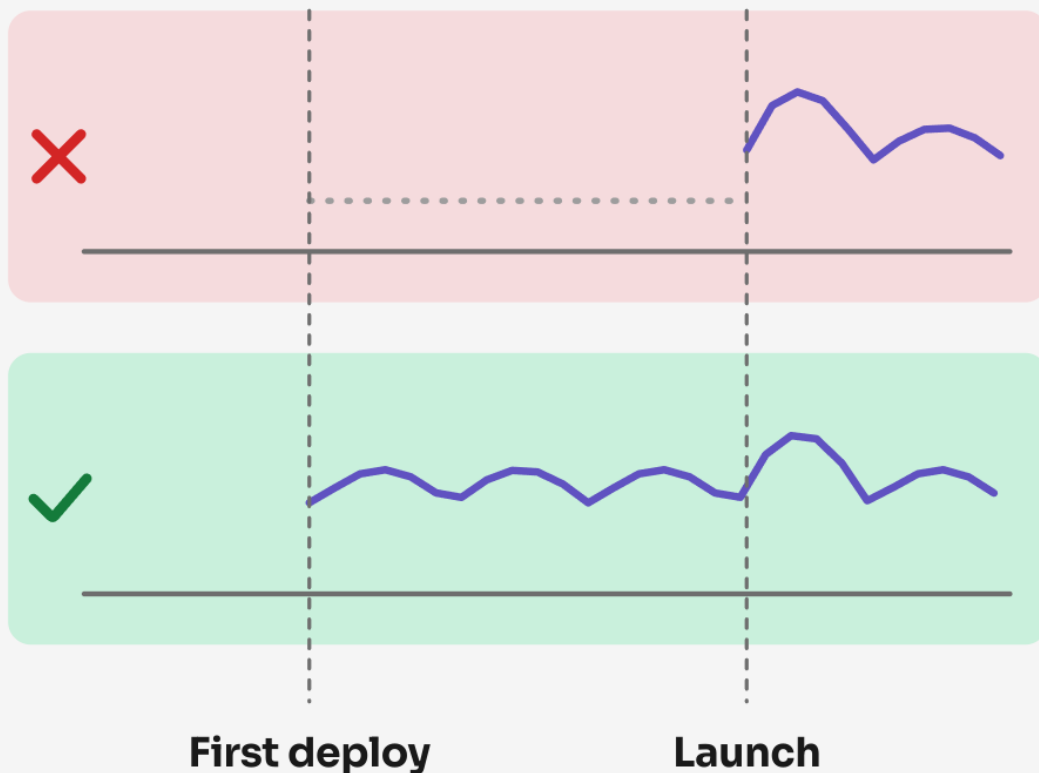
Fixing a bug is development. Understanding why users hit an error, tracing the root cause, writing a fix, deploying it safely. That's software development.

Optimizing performance is development. Profiling the application, finding the bottleneck, refactoring for efficiency, measuring the result. That's software development.

Call it whatever. The work is the same. The skills are the same. The only thing that changes is the label people put on the ticket.

Monitor Day One

Observability starts from the very first deploy



Observability isn't a polish task. It's infrastructure.

Nobody can fix what they can't see. And without watching from the first deploy, the project starts blind at the worst possible moment.

Observability isn't a polish task. It's infrastructure. Treat it that way.

Don't Wait for Launch

"We'll add monitoring after launch" is how teams spend launch week guessing. That week has the highest traffic, the most edge cases, and the most bugs nobody has seen before. It's exactly when visibility is needed — and exactly when it won't exist if it was deferred.

Set up monitoring during development. Not after. Not when things break. Before the first deploy.

What to Put in Place

Four categories matter from day one:

Error tracking — capture exceptions the moment they happen, not when a user eventually files a complaint.

Uptime monitoring — monitor a simple health check endpoint. The developer should know the service is down before the users do.

Analytics — track the actions that matter: signups, activations, key feature usage. Not everything. The things worth acting on.

Performance tracking — measure response times, database query latency, and API endpoint slowness. A baseline is needed before degradation can be spotted.

In Varstatt, that typically means Sentry for error tracking, Google Cloud Logging for infrastructure visibility, and Simple Analytics for usage — but the specific tools matter less than having them from the first deploy.

The Feedback Loop

With monitoring: a spike in errors appears, gets traced to a bad deploy, rolled back or fixed, numbers verified back to normal. Done in an hour.

Without monitoring: a user reports something feels wrong. The developer can't reproduce it. Asks for details. The user doesn't remember. The problem persists for days.

Monitoring compresses the gap between "something broke" and "problem solved."

What to Actually Watch

Focus on what's actionable. Too many dashboards become noise everyone learns to ignore.

Watch where users get stuck. Watch what breaks and how often.

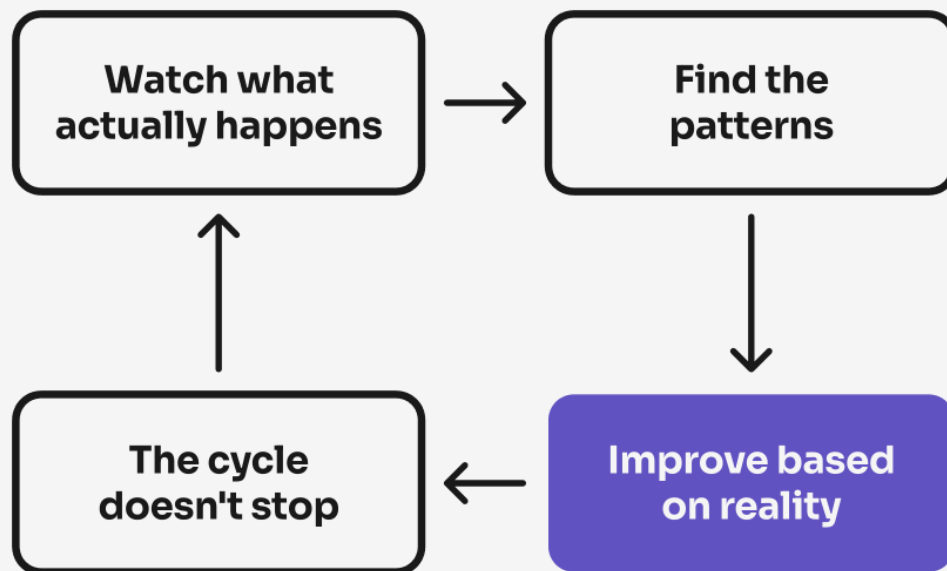
Watch what's slow. Watch the business numbers — signups, conversions, retention — whatever the product actually lives or dies by.

The Cost Argument

Basic monitoring is often free. Paid tiers run \$20–50 a month. One critical bug caught before it reaches users pays for years of that. The ROI isn't complicated.

Observe and Improve

Real data drives continuous improvement



Reality-based improvements beat assumption-based development every time.

Software gets built on assumptions. It gets improved on reality. Those are two very different inputs, and only one of them is honest.

Watch What Actually Happens

Users do unexpected things. Always. The happy path the developer designed? Maybe half of people follow it. The other half find creative ways to stress the assumptions — submitting forms with data nobody anticipated, combining features in ways nobody considered, hitting edge cases nobody knew existed.

Don't watch what was expected to happen. Watch what actually happens. There's a difference, and it matters.

Find the Patterns

One user hitting an error is noise. Ten users hitting the same error is a pattern — and a pattern means something is wrong with the software or the UX, not the user.

Usage data tells the same story. Feature X gets hammered daily. Feature Y sits untouched. That's a pattern too. Maybe X needs optimization. Maybe Y needs to be cut. A page that loads slow in one specific scenario but fast everywhere else? That's not random — it's a particular query, a particular data condition, something real and specific the developer can fix.

Data reveals what assumptions miss.

Improve Based on Reality

The client says users need feature Z. The developer checks the analytics. Nobody uses feature Y, which does something similar. Before building Z, the question is: has anyone [validated this need](#)? That's a better question than "how long will it take?"

Performance feels sluggish. The developer checks monitoring. 95% of requests are fast. 5% are slow — all hitting one specific database query. Optimize that query. Problem solved. No guessing.

Bug reports seem scattered. The developer digs into the error logs. Every single error comes from mobile Safari. Not random. Browser-specific. Fixable.

Reality-based improvements beat assumption-based development every time.

The Cycle Doesn't Stop

Build. Deploy. Observe. Analyze. Improve. Deploy again. That's the loop. Not "build, deploy, done."

Software that doesn't evolve dies. Markets shift. User behavior changes. Competition moves. Without observing and improving, it's just hoping — and hope isn't a strategy.

What It Actually Takes

Humility. Assumptions will be wrong. That's not a failure, it's the nature of building things. Adapt.

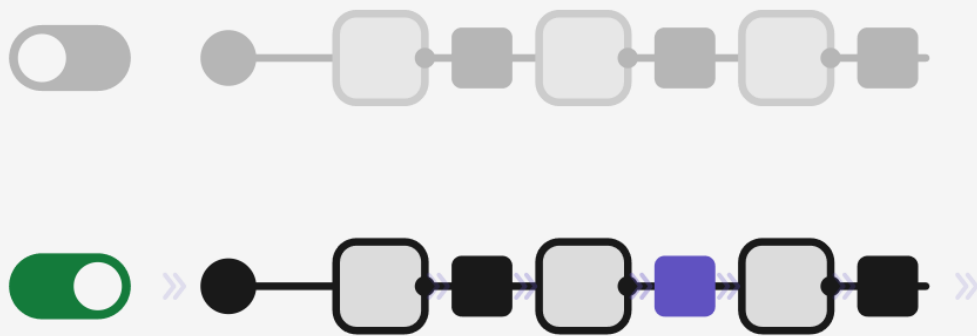
Curiosity. When users behave unexpectedly, ask why. What does this pattern mean? What are they actually trying to do?

Action. Observing and analyzing accomplish nothing on their own. The loop only closes when the improvement ships.

Pause, Not End

Engagements pause gracefully rather than abruptly end

Pick up where things left off



When work resumes, pickup is immediate.

Projects don't move in a straight line. Work surges, then slows. Waiting happens. Real engagements look nothing like a steady retainer — and the relationship model shouldn't pretend otherwise.

Why Pausing Matters

The client is waiting on a decision. Direction needs to be validated, stakeholders need to align, and that takes weeks. There's nothing urgent — no critical bugs, no pressing features, just a backlog of nice-to-haves that can wait.

Sometimes the business slows seasonally. Development should follow. Sometimes the client has spent the allocated budget and needs to generate revenue before continuing. All of these are legitimate. None of them should force an awkward ending.

Without a pause mechanism, three ugly outcomes emerge:

Forced continuity — no urgent work but the client keeps paying because the contract demands it. Resentment builds quietly.

Hard endings — a temporary break is needed but the only option is a full cancellation. Restarting means finding someone new and explaining everything from scratch.

Scope-stretching — not enough real work, so busywork gets invented to fill the hours. Everyone knows it's fake.

How to Structure It

The engagement model should make pausing a first-class option, not a special case. That means:

- **No penalty for stopping.** The client shouldn't feel locked in when there's no work to do.
- **Context survives the gap.** Code, documentation, task boards — everything stays where it is. When work resumes, pickup is immediate.

- **Billing matches activity.** Short billing cycles make this natural. The shorter the cycle, the easier it is to start and stop without awkward conversations.

The specific mechanism depends on the model — subscription, retainer, project-based. What matters is that pausing is built in, not negotiated each time.

The Varstatt Model

In Varstatt, [weekly billing](#) makes this straightforward. The client cancels the subscription. Billing stops immediately. When the client is ready, they resubscribe and pick up where things left off. No re-onboarding. No lost context. No penalty.

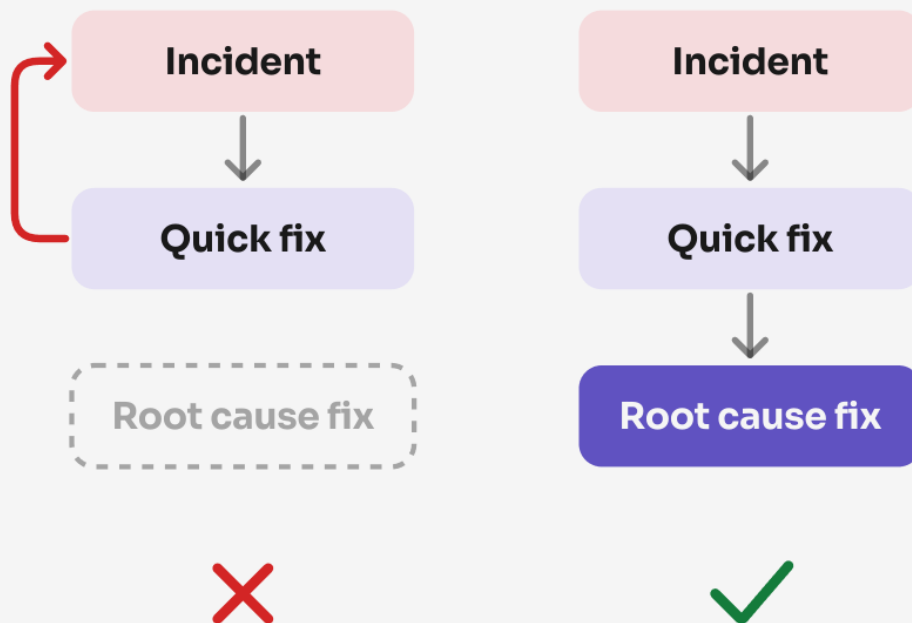
There's no awkward "we need to end the engagement" conversation. Billing pauses. The relationship continues. Context stays intact. Trust holds.

Billing Should Reflect Reality

Sometimes the developer is shipping major features fast. Sometimes it's iterating on small things. Sometimes both sides are waiting. The engagement shouldn't pretend otherwise — and neither should what the client is charged.

Incident Response

Stop the bleeding first, then find the root cause



The quick fix buys time — the root cause fix is the actual work.

Something breaks in production. The response has two phases, and mixing them up is the most common mistake.

Stop the Bleeding

Phase one is recovery. Figure out what's happening — check logs, reproduce the issue, determine the blast radius. Infrastructure or code? How many users affected? These questions narrow the search fast.

Then find the smallest possible fix to restore the system. Don't try to understand the full picture yet. Ship the patch. Breathing room restored.

Find the Root Cause

Phase two is surgery. Schedule the deep investigation. Find why it actually broke, not just what broke. Fix it properly. Ensure the same failure can't happen again.

The temptation is to skip phase two once the system is back up. That's how the same incident happens three times. The quick fix buys time — the root cause fix is the actual work.

Prevention Over Response

The best incident response is the one that doesn't happen. [Monitor from day one](#) so the developer knows before the client does. Deploy small to isolate changes. [Feature flags](#) to kill a bad release in seconds. [Safe-to-fail](#) means the system was designed for this moment.

When prevention works, incidents become rare. When they do happen, the same monitoring and tooling that prevents most incidents also makes the ones that slip through easier to diagnose.

In Practice at Varstatt

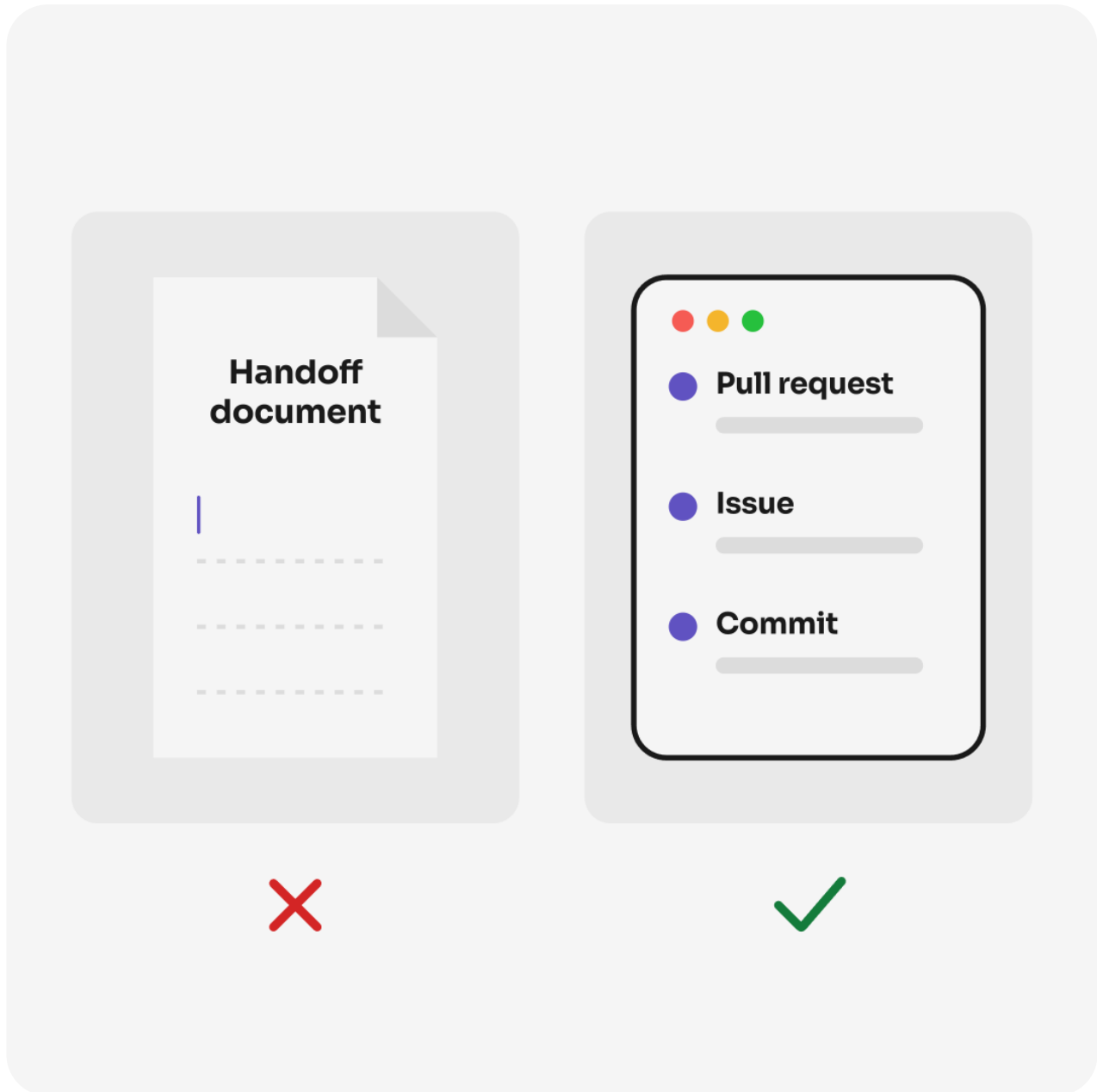
Something breaks — the developer drops everything and gets to a computer. No escalation matrix. No on-call rotation. No waiting for someone else to triage. The developer responds directly, immediately or as soon as physically possible.

Communication is simple. The client provides whatever data they have. The developer goes autonomous the moment something's wrong — no waiting for the client to diagnose the problem.

Postmortems are not a ceremony. They're the same [async updates](#) sent for normal work: what broke, why, what was done, what will prevent it. Written in the regular channel, in the regular format. No incident report templates. No blameless retro rituals. Just honest updates about what happened and what changed.

Documentation

Transparent process creates documentation as a side effect



Nobody sits down to write a handoff document — the trail already exists.

The best documentation is a side effect, not a deliverable.

Process as Documentation

If a new developer needed to take over tomorrow, what would they find? The answer depends on how transparent the process is. Pull requests that explain what changed and why. Issues that capture decisions. Commit history that tells the story of the project.

When the process is transparent, documentation happens automatically. Nobody sits down to write a handoff document — the trail already exists.

The question isn't "what should be documented?" It's "is the process transparent enough that documentation happens naturally?"

When to Write Deliberately

Sometimes the process doesn't capture it. A complex architectural decision that spans multiple PRs. A non-obvious business rule that isn't evident from the code. A deployment quirk that would trip up the next person. These are worth a few paragraphs in the repo.

But don't maintain wikis nobody reads. Don't write ADRs for decisions that are obvious from the PR. Don't create documentation for documentation's sake. If the process already captured it, writing it again is waste.

AI Changed the Economics

The AI era collapsed the cost of documentation. Decisions happen in minutes — AI documents them in seconds. Pull request descriptions, decision records, summaries of conversations — all generated from work that already happened.

This doesn't mean generating documentation for everything. It means the excuse of "documentation takes too long" no longer holds. When the cost

drops to near zero, the only question left is whether the process feeds the right context to the tools.

At Varstatt

In Varstatt, documentation is a byproduct of [transparency](#). The client sees the boards, the code, the PRs, the messages. Every [async update](#) documents progress. Every PR explains the change. The client already has the documentation — they just don't call it that.

AI generates markdown decision records from actual conversations, written directly into the repository. No Confluence pages. No separate wikis. The repo is the single source of truth for both code and decisions.

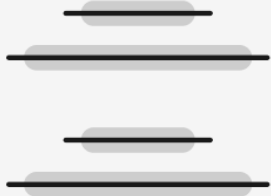
04

Partnership

Trust, transparency, and accountability
shape the client relationship

Hard conversations? Fix the process

Hard conversation



- ✓ Full visibility
- ✓ Weekly accountability
- ✓ Priorities over scope
- ✓ Client-controlled exit



Hard conversations are a symptom, not a condition. Fix the process and the symptom disappears.

Methodology and relationship can't be separated. How the work happens shapes how the engagement works, and vice versa. Get one wrong and the other falls apart.

Full Visibility

The client sees everything. The task board, the documentation, the repository, the work-in-progress commits. Not a curated summary — the whole thing. After every session there's an update: what got done, what's next, what questions are open.

This demands trust. Trust that the client won't second-guess every micro-decision. But it also creates trust. Nothing is hidden. There are no surprises. Problems surface early, not at delivery.

The best clients disappear for weeks while the developer works, then return with feedback and fresh ideas. That pattern only works when visibility is complete.

Weekly Accountability

Short billing cycles create a tight feedback loop. Good work and the relationship continues. Bad work and the client leaves. There's nowhere to hide.

No long-term contracts to absorb mediocre output. Both sides carry mutual risk — the client risks a short-term payment, the developer risks immediate cancellation. That tension keeps everyone honest.

Priorities Over Scope

Fixed scope creates scope creep. That's the paradox. Remove the scope and there's nothing to creep. Instead, there's a priority queue. The client wants a new feature? Great — where does it sit in the queue?

The [WIP limit](#) enforces this. Adding something means stopping something else. That forces real decisions about what actually matters right now.

Async by Default

Written updates are the default. Task comments capture decisions. Slack handles quick questions. The exception is strategic inflection points — architecture changes, major pivots — where a conversation is worth the cost.

Eliminating unnecessary meetings is the single biggest productivity gain available. Most discussions don't need to be synchronous.

Client-Controlled Exit

Easy cancellation isn't a bug — it's a feature. No service provider will ever tell a client to stop paying. The incentive always points toward continuation. Here it's flipped: the client controls the exit. The client decides when the value justifies the cost. That's an honest tension, and it's acknowledged from day one.

Hard Conversations? Fix the Process

Most advice about hard conversations assumes an adversarial relationship. The client wants X, the developer thinks X is wrong, now someone needs a script.

Here's a different take: if a playbook for hard conversations is needed, the process is broken.

Software development is a business cost and the client decides. The developer can consult. The developer can recommend. The developer can flag risks and present alternatives. But the final decision is the client's. They pay, they choose. This isn't passive — it's honest.

When transparency is real, the client sees everything. When weekly accountability is real, there are four exit points every month. When

priorities replace scope, the conversation is always "what's most important next?" not "where's everything I was promised?"

In this environment, blow-ups don't happen. Disagreements get resolved in async messages and discovery calls, not in dramatic confrontations.

The clients who generate hard conversations are the ones the other principles should have filtered out — [saying no](#) at the start, [week zero](#) as a mutual audition, easy exit when fit isn't right.

Hard conversations are a symptom, not a condition. Fix the process and the symptom disappears.

What's Ahead

Seven principles shape how the partnership works — from onboarding and pricing that set the right expectations, through full transparency as the foundation, the accountability mechanism and how work gets managed, to the communication model and the freedom to walk away.

Onboarding

How the first week sets the tone for everything



Real work together is the only reliable signal.

The engagement doesn't start when the first commit lands. It starts before that — the mutual audition where both sides figure out if this actually works.

Do Real Work Before Committing

The principle is simple: work together before anyone commits long-term. The developer researches the problem independently. Responds with their understanding, questions, and a rough direction. Has a strategic conversation — not a sales pitch. Then sets everything up together, so the first paid week starts with real work.

Only after code is committed and building starts do real answers emerge. Everything before that is theory.

The Mutual Audition

Despite being a trial period, the developer communicates like it's real work. Regular [async updates](#). Full [transparency](#). The developer is testing the client as much as the client is testing the developer.

Can both sides work together? Does communication flow? Are expectations realistic? Is the project what the client described?

These questions only get honest answers through actual work — not proposals, not sales calls, not references. Real work together is the only reliable signal.

Why This Protects Both Sides

The client doesn't commit money before seeing how the developer works. The developer doesn't commit effort before knowing the client is a real fit. A week of working together tells more than any proposal ever could.

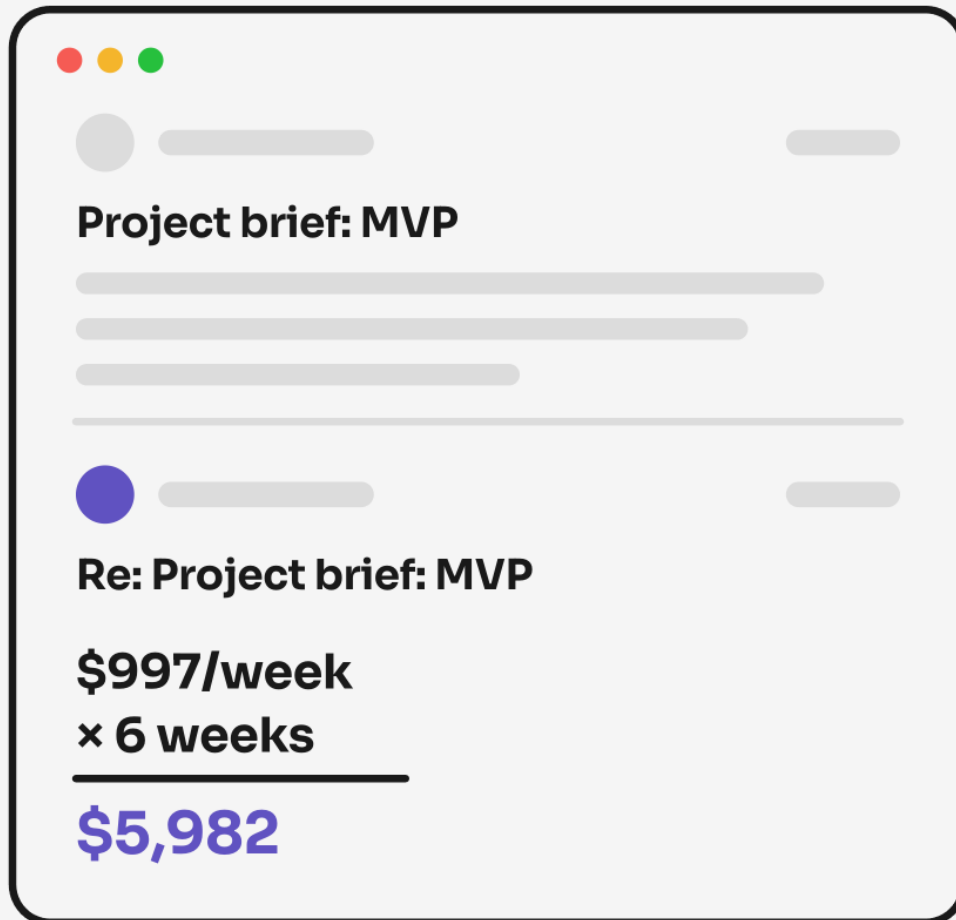
The Varstatt Flow

The developer receives a project brief. Before responding, Varstatt does independent research — understands the problem, drafts potential solutions, figures out why this matters and what success looks like. Then responds with an understanding of the problem, a rough timeline, and follow-up questions.

A discovery call gets scheduled. If the client subscribes, week 0 begins: a free onboarding week. During it Varstatt analyzes the flow, bootstraps the development environment and prepares access. It builds nothing yet, so week 1 starts with real work. Only if all is good does the client continue the subscription.

Pricing Conversation

The price is on the website — here's how the conversation goes



Fixed retainer, estimated timeline, total investment is multiplication. Nothing more.

Pricing should be public. If a prospect reaches out, they've already seen it. Self-selection starts before anyone talks. No hidden fees. No "it depends" pricing theater.

Keep It Simple

The pricing conversation itself should be simple: fixed retainer, estimated timeline, total investment is multiplication. Nothing more. This fee structure allows the developer to do good work, keep fewer clients, and give each one continuous results.

The Hours Red Flag

If a prospect needs corporate framing, the developer can explain it as a part-time engagement. But this is usually a red flag. When the client needs to hear hours, the project will fail.

Why? Because [counting hours](#) means the client is buying time, not outcomes. They'll optimize for hours spent instead of results delivered. They'll question whether a 30-minute task was really 30 minutes. They'll want timesheets. And the developer will spend more energy justifying time than doing actual work.

Let the Model Self-Select

The best clients never question the price. They see the value, they see the model, they commit. The ones who push back on hours or demand to understand "what amount of work they get" — those are tire-kickers or corporate procurement people optimizing for the wrong metric.

The developer can try to explain with examples of what typically gets delivered in a week. Sometimes that helps.

But if the conversation keeps circling back to hours and outputs and utilization rates, listen to that signal. It's telling the developer this client doesn't fit the model. And forcing a fit will make both sides miserable.

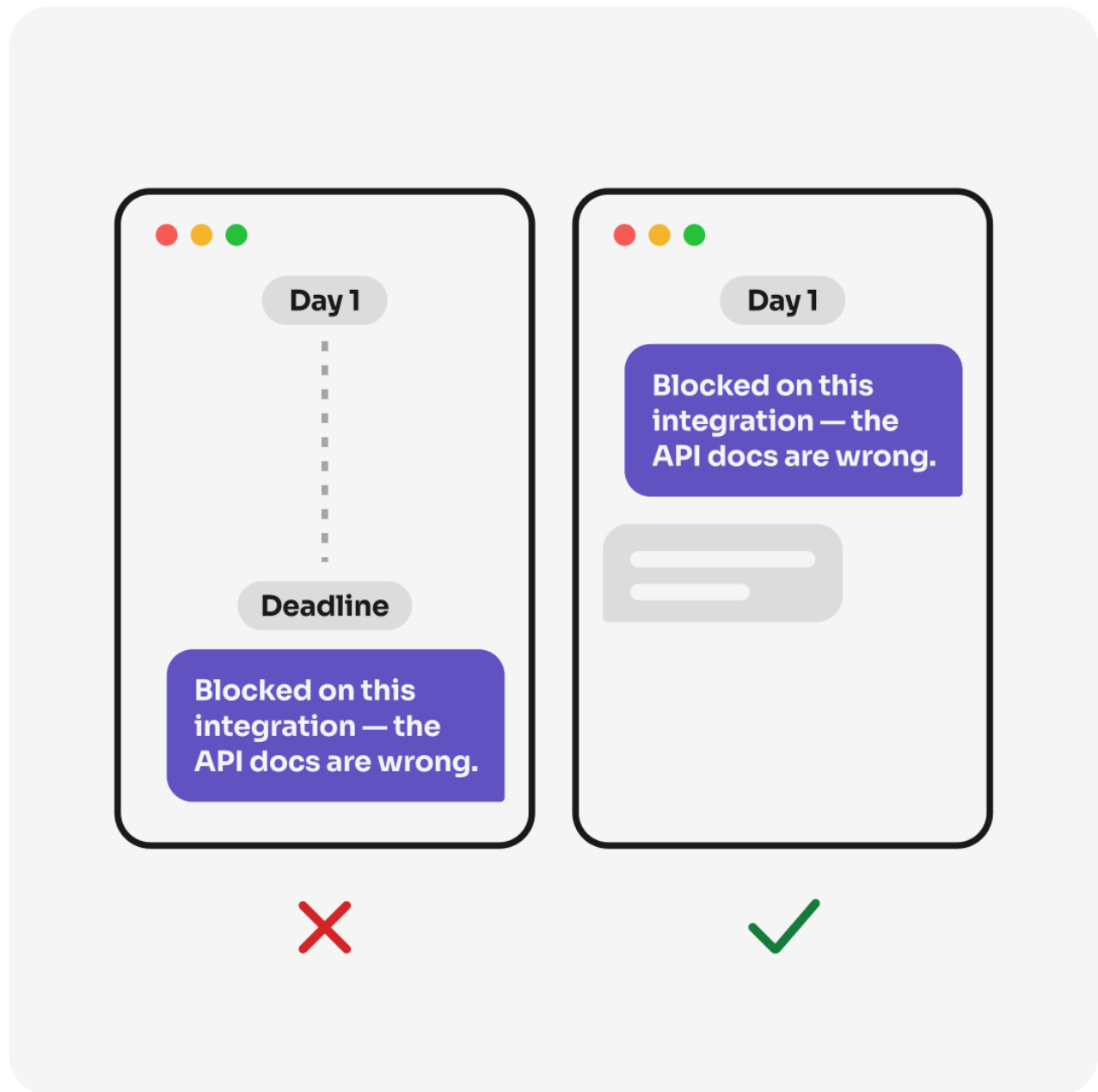
Price publicly. Explain simply. Let the model self-select.

In Practice at Varstatt

In Varstatt, the initial timeline is included in the response to the brief: the fee is X per week, the estimated project length is Y weeks, total investment is X times Y. On the discovery call, Varstatt repeats it. That's the entire pricing conversation.

Transparency

Full visibility into progress, problems, and decisions



When something's stuck, the client knows immediately.

The client sees everything. Not as an ideal — as a practice. Full access to the task board, the repository, the documentation, and the ongoing work-in-progress. No staged reveals. No "we'll show you when it's ready."

Full Access, Always

The task board is shared with the client from day one. Current priorities, what's in progress, what's done, and what's coming next — all visible. The code repository belongs to the client — owner access, not view-only. Every commit, every pull request, every comment thread is readable. [Documentation](#) lives in shared spaces: design decisions, architecture notes, troubleshooting guides. All of it. All the time.

Regular [updates](#) land in task comments or a shared channel. What got done, what's next, what's blocked. The client doesn't have to ask.

Development Is Messy — The Client Will See That

The client will see commits that say "WIP — doesn't work yet." They'll see "trying a different approach" followed by a revert. That's not a problem. That's honest work.

Hiding the mess creates false polish. It feels professional but it's dishonest. The work is messy. Pretending otherwise builds a relationship on performance instead of reality. The client isn't paying for clean commit history. The client is paying for working software.

Problems Surface Early

When something's stuck, the client knows immediately. "Blocked on this integration — the API docs are wrong." The client can help, escalate, or adjust expectations. None of those options exist when the problem stays hidden until a deadline.

Decisions are documented in real-time. Six months later, when someone asks why the team went with approach X, the answer is already written down. Context doesn't disappear when people change.

The Status Anxiety Loop

When clients don't know what's happening, they ask. The developer stops to write updates. Less work gets done. The loop repeats.

Transparency breaks the loop. Status is always visible. No one has to ask. The best working relationships are ones where the client can disappear for two weeks, come back, check the board, see clear progress, and trust that things are moving. That only works when there's nothing hidden.

Surprises come from assumptions and opacity. Both die in the light.

Weekly Accountability

Weekly billing creates a feedback loop with real consequences

Do you deliver well?

Invoices	
Week 5	Upcoming
Week 4	Paid
Week 3	Paid
Week 2	Paid
Week 1	Paid

Did the developer deliver value this week? The client continues.

Short billing cycles create accountability. The shorter the cycle, the tighter the feedback loop — and the more honest the relationship becomes.

Feedback That Actually Bites

Did the developer deliver value this week? The client continues. Did the developer drop the ball? The client stops. There's no contract to hide behind, no sunk cost keeping the client locked in. Weekly accountability creates a feedback loop with teeth. Both sides feel it immediately, not twelve months from now when renewal rolls around.

Why Some Developers Fear This

The common objection: "What if a client has unrealistic expectations and cancels unfairly?" It's possible. It's rare. But it's real, and the developer takes that risk consciously. Mitigation comes through honest communication, clear expectation-setting from day one, and doing work worth being proud of. If the fear is that the work won't hold up to weekly scrutiny, that's not a billing problem. That's a deeper problem.

Why Confident Developers Prefer It

No need to win clients with polished proposals or sell them on estimates. Just show what got built. No political maneuvering, no performance theater for a manager, no waiting on a quarterly review. Deliver value or don't. The feedback is immediate and honest. That's freedom — the kind that comes from trusting the output.

Mutual Skin in the Game

The client risks paying for a week that doesn't deliver. The developer risks losing the engagement for a week that falls short. Both parties stay alert. Both parties optimize for actual results, not optics. No contract engineered by lawyers produces that kind of alignment.

Relationships That Last Because They Work

A long-term relationship with an easy exit sounds contradictory. It isn't. When the client stays, it's because the work keeps earning it — not because breaking a contract would be painful. When it ends, it ends cleanly. No negotiation, no awkward wind-down, just a [pause](#) or a cancel and a professional goodbye. That's a healthier foundation than obligation.

Priorities, Not Scope

Priorities replace fixed scope — alignment over negotiation

There's only what's next



There's no "in scope" versus "out of scope" — there's only what's next.

The moment a fixed scope is defined, a negotiation is created. Every conversation becomes about what's in and what's out. The client starts

counting. "That's related, it should be included." "That's basically the same work." Arguments follow. Relationships strain. Projects derail. Scope creep isn't caused by clients asking for too much — it's caused by scope existing in the first place.

The Priority Queue

Replace scope with a priority queue. The client maintains a ranked list: most important at the top, least important at the bottom. The developer works top-down, always building the highest-priority thing.

The client wants to add something? Add it. Just decide where it ranks. The client wants to change direction? Reorder the list. There's no "in scope" versus "out of scope" — there's only what's next.

One Thing at a Time

A [WIP limit](#) enforces real prioritization. Adding a new task means stopping the current one. When something urgent comes up, the question is simple: does the current work stop, or does this go into the queue? The client chooses. But the choice is explicit. Both can't happen at once.

This kills the "everything is urgent" problem. When everything is urgent, nothing is. The queue forces actual business thinking — what matters more right now, this or that?

What the Client Learns

The client can have anything. The client can't have everything — not all at once. The queue makes that visceral. "Yes, the developer can build that. What stops in order to start this?" It's a hard question. It's also the right question. Better answers lead to better products.

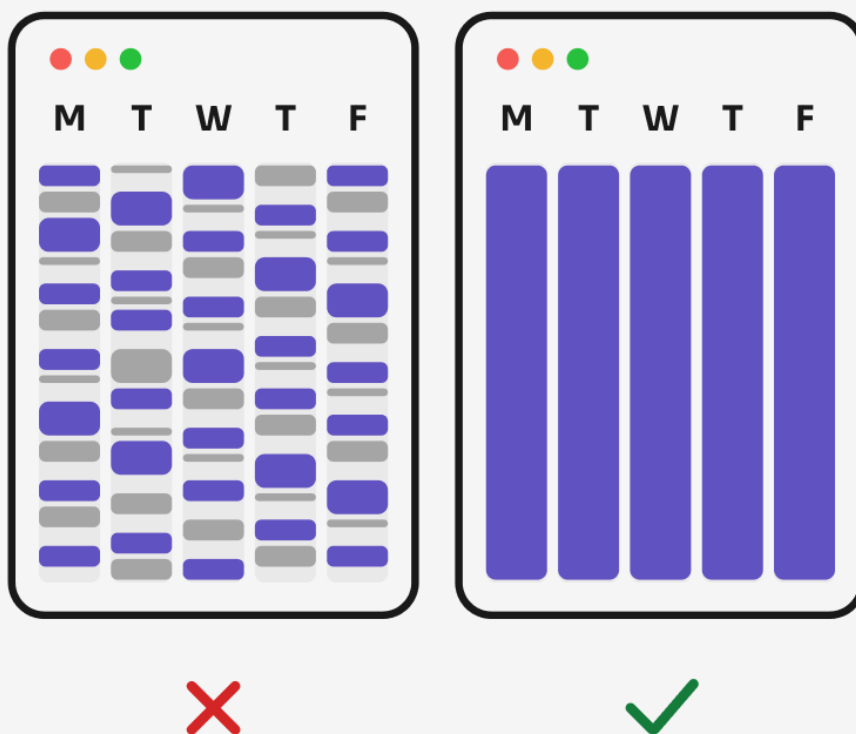
What Both Sides Get

Both sides always know what to work on next. No ambiguity, no scope debates, no defending line items. Just the top of the queue and the work itself. Simple. Effective.

Async First

Async communication protects focus time and respects everyone's calendar

Meetings kill focus



Meetings are a last resort, not a default.

Meetings are a last resort, not a default. This isn't about introversion or avoiding people — it's about protecting the conditions that make good work possible.

Why Async Shapes the Partnership

Async-first means defaulting to writing, and treating synchronous time as a scarce resource. This sets the tone for the entire relationship: trust over surveillance, results over presence, written records over half-remembered conversations.

It also means no one's calendar dictates the pace of work. The developer works when they're sharpest. The client reviews when it's convenient. Nobody waits on the other to "be available."

When to Actually Meet

Some things are genuinely better live. Strategic decisions where multiple paths are on the table. Complex requirements that are easier to walk through than describe. Major pivots that affect the whole direction of the work. Occasional check-ins to make sure the relationship is solid.

These are exceptions. Treat them that way. Tactical work doesn't need a meeting. "Built the feature, here's how it works, let me know if it needs adjustments" is a message, not a calendar invite.

What Disappears

Daily standups go away — everything they surface is already in the task board. Sprint planning meetings go away — [no sprints](#), no planning ceremonies. Status meetings go away — status is always visible. The "quick sync" that somehow eats 45 minutes and produces nothing goes away.

All the process theater that makes developers hate agile disappears.

The Math

The average developer loses [11 hours a week to meetings](#) and gets maybe 20 hours of real focus time. Flip the default and that becomes 1-2 meeting hours and 38+ hours of uninterrupted work. That's not a marginal gain. That's doubling productive capacity without hiring anyone.

Calendar creep is a productivity killer. "Quick syncs" that run 30 minutes. Half the day in meetings, the other half recovering from them. Context switching compounds the damage — the developer is deep in a problem, a meeting interrupts, and it takes [20 minutes to get back](#). Do that five times a day and nothing gets built.

Handling the "I Need You Available" Client

The developer is available. An async message gets a response within a few hours. For real emergencies, there's a direct channel. But "emergency" means production down, data loss, or a security breach — not a new feature idea that just came to mind. The right clients understand this. The ones who can't adapt are telling the developer something useful about whether the relationship will work.

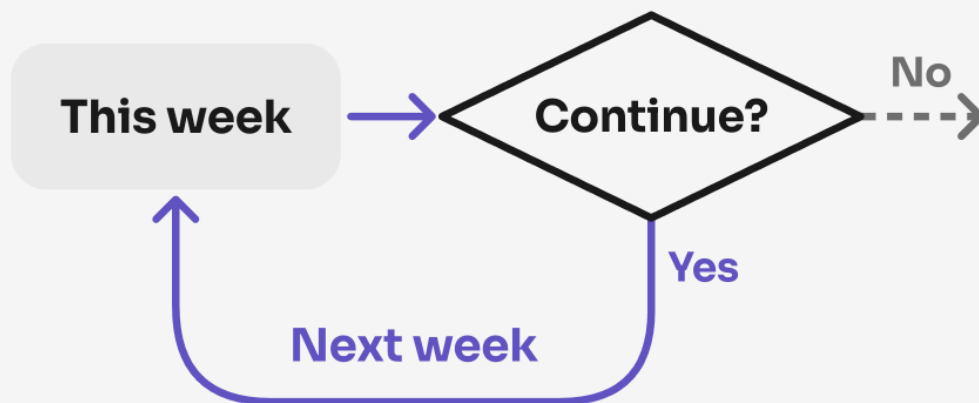
In Varstatt, that means Slack for day-to-day communication and a shared phone number for genuine emergencies.

That's a filter. A good one.

Exit Freedom

Clients can leave at any time — no lock-in, no hard feelings

Leave anytime



The client is never far from a clean stop.

The best accountability mechanism is easy cancellation. Not contracts. Not milestones. Not quarterly reviews. Just: the client can leave whenever they want.

Why Easy Exit Matters

No penalty clauses. No notice period. No awkward conversation about "winding down." The code, documentation, and task board belong to the client — [full ownership from day one](#). Nothing is held hostage. There's nothing to "hand off" because there's nothing the developer is holding.

Short billing cycles make this mechanically simple. The shorter the cycle, the more natural exit points. The client is never far from a clean stop.

The Confidence Filter

A developer who isn't confident in their work finds easy cancellation frightening. The client could leave anytime. No contract to hide behind, no sunk cost keeping the client locked in. But that fear is telling — the developer isn't sure they're delivering value. The fix isn't a better contract. It's better work.

For confident developers, it's liberating. Ship working software. Solve real problems. The client stays. That's the whole loop. No relationship management theater. No convincing stakeholders the work matters. Just a direct signal: if work quality drops, the client cancels. Adjust or move on. Clean feedback.

What It Prevents

Resentment buildup. The client feels trapped, the developer feels obligated to underperform, the relationship sours. Nobody wins. Easy exit short-circuits all of that. If it's not working, end it cleanly. No bad feelings, no forced continuation of something broken.

Graduates, Not Dropouts

Some clients will cancel because they've outgrown the model. They need a bigger team, more capacity, different specialization. Call them graduates. They didn't leave because the developer failed. They left because they succeeded.

Leave the door open. The best former clients come back for new projects, refer others, or return when they need focused execution again. Restarting should be as simple as picking up where things left off. No new proposals. No re-onboarding. The repo is where it was left.

Clean exits require zero effort when ownership was never ambiguous.

At Varstatt

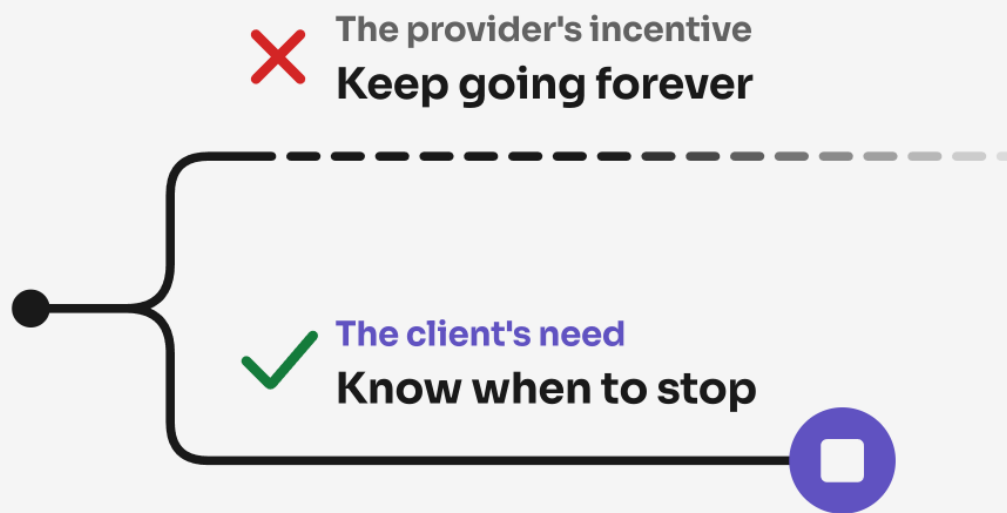
In Varstatt, [weekly billing](#) creates four natural exit points every month. Offboarding is a Stripe cancellation. After the subscription cancels, the developer stops producing work. No refunds for paid weeks — but the developer can finish things up or prepare additional materials until the current period expires if the client asks.

05

Philosophy

Fundamental assumptions about what
software development is — and isn't

Who the work is for



The service provider's incentive is to keep going forever.
The client's need is to know when to stop.

These aren't practical guidelines dressed up as principles. They're fundamental assumptions about what software development is — and what it isn't.

Software Is a Business Cost

Developers like to think they're craftspeople. Artists. Elite engineers building cathedrals in code. They're not. Software development is an operational cost — like accounting, customer service, or office supplies. Essential? Yes. Special? No.

Clients don't want beautiful code. They want business problems solved at a reasonable price. That mindset shift changes everything: service-oriented instead of ego-driven, predictable instead of creative-on-a-whim, cost-effective instead of technically-interesting-first.

One Platform Beats Many

Every integration point is a failure point. Every API key is a maintenance burden. Every third-party service is another thing to monitor, pay for, and eventually migrate away from. Developers love microservices because they're interesting. Clients suffer the operational complexity.

Platform consolidation is the answer. One platform covering authentication, database, hosting, serverless functions, file storage, and monitoring — one bill, one mental model. Less sophisticated than best-of-breed for each category? Sure. Good enough for most use cases? Absolutely. Fewer services means less integration complexity, less maintenance, and less that can go wrong.

In Varstatt, that platform is Firebase.

Hours Are the Wrong Metric

Time tracking is productivity theater. Hours logged mean nothing on their own. Did the developer solve the problem? Did they ship working software? Did they deliver value? Those are the real questions.

Hourly pricing creates a perverse incentive: work slower, earn more. Work faster, earn less. A fixed weekly rate aligns incentives — deliver value, keep the client happy, and the relationship continues.

The Client Decides When Done

The service provider's incentive is to keep going forever. The client's need is to know when to stop. That tension is real, and pretending otherwise is dishonest.

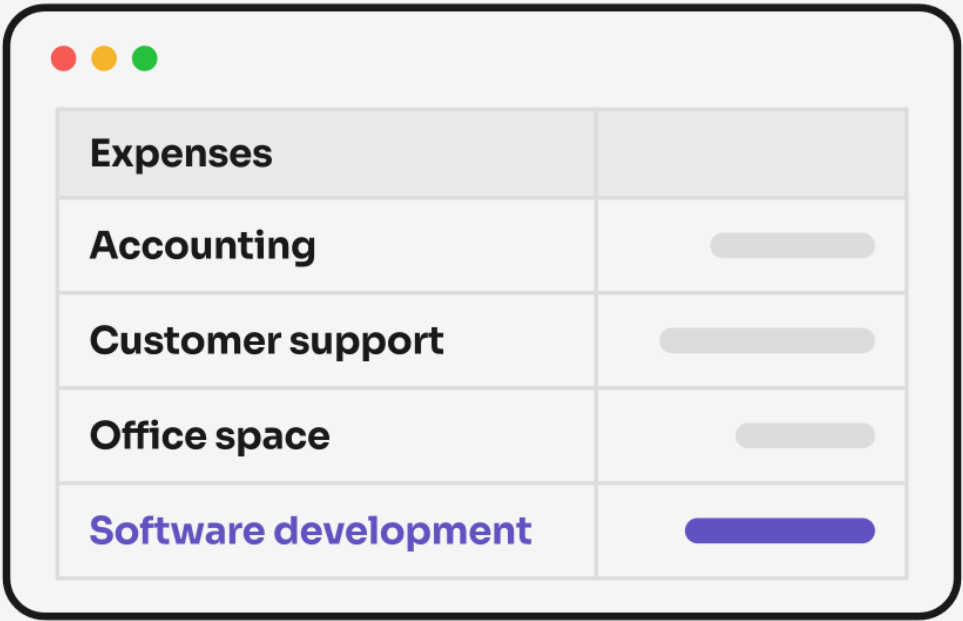
The client controls the exit. The developer will never tell a client to stop paying — but the client can decide when the value no longer justifies the cost. That's a business decision, not a technical one. Software is never truly "done," but business objectives get met and ROI thresholds get reached. The client decides when that moment is.

What's Ahead

Six principles spell out these beliefs in detail — starting with what software development actually is, then addressing how it should be priced, how systems should be built, who ultimately holds the reins, why the solo model works, and who owns the output.

Business Cost

Software development is an operational expense, not a craft



Business cost — the same category as accounting, customer support, and office space.

Not a craft. Not an art form. Not a calling. Business cost — the same category as accounting, customer support, and office space. Essential, yes. Precious, no.

The tech industry spent decades convincing developers they're an elite class. Conferences celebrate "10x engineers." Hiring culture treats developers like scarce, pampered talent. That framing creates entitlement. Entitlement creates bad service.

The Developer Is Here to Solve Problems

Clients don't want beautiful code. They don't care about architecture decisions. They want their business problems solved at a reasonable cost, delivered predictably, without drama.

That's not a cynical view. That's an honest one. Service orientation starts when the developer stops centering their own preferences and starts centering the client's outcomes.

Craft Serves the Business

Yes, software development involves craft. So does accounting. But accountants don't demand creative freedom to explore interesting tax theories — they deliver reliable, predictable, valuable work. Development is the same.

Craft in service of business goals is good. Craft for craft's sake is self-indulgence billed to the client.

Stop Making Technical Decisions Based on What's Interesting

Don't debate microservices versus monoliths based on what's technically exciting. Choose what's [operationally simplest](#) for the client's actual situation. Don't insist on test-driven development because it's "proper engineering" — test adequately for the risk level, nothing more.

Don't gold-plate solutions. Solve the actual problem, not a perfect theoretical version of it.

Predictability Is the Product

Clients need to plan. That means predictable delivery, predictable communication, predictable cost. When the developer treats development as an operational function rather than a creative endeavor, surprises stop happening.

Predictability is what earns long-term relationships. It's more valuable than any technical sophistication added to the codebase.

Why This Matters

Clients have dealt with developers who treat every decision like an art critique. Technical debates without business context. Perfection becoming the enemy of done.

A developer who optimizes for business outcomes — who delivers reliably and keeps costs honest — is refreshing. That's who gets called back.

Hours Are Wrong

Hourly billing misaligns incentives — fixed rates fix it

Results delivered over hours logged



The client doesn't care how long it took. The client cares whether it works.

Hourly billing is productivity theater. It measures the one thing that doesn't matter — time — and ignores the one thing that does: whether the work delivered value.

Ten hours of bad code is worse than two hours of good code. The hours don't tell anyone which one they got.

What Hours Actually Measure

Hours measure presence. Nothing else. They don't measure quality, judgment, or whether the problem got solved. They don't tell the client if the developer was sharp or stuck in circles.

A senior engineer might solve in an hour what a junior takes ten hours to fumble through. Under hourly billing, the senior earns less for delivering more. That's not a pricing model — it's a punishment for expertise.

The client doesn't care how long it took. The client cares whether it works.

The Incentive Problem

Hourly pricing creates a perverse incentive: work slower, earn more. Work faster, earn less. Every efficiency gain comes out of the developer's pocket.

It also requires surveillance to function. Time tracking software, activity logs, screenshot monitoring. The client isn't hiring a professional — they're babysitting one. That's not a working relationship. It's a power struggle dressed up as accountability.

How Fixed Rates Fix This

A fixed rate regardless of hours spent flips the incentive: deliver value efficiently, and the work is done. Solve the problem in ten hours? Great. Takes forty? Also fine. No logging required.

There's no time tracking to game. No reason to pad hours or drag work out. The only question is: did the developer ship what was promised?

Transparency Without Surveillance

No time tracking doesn't mean no visibility. The client sees exactly what's being built — [task updates, commits, deployed features](#). The client just doesn't see how many hours each piece took.

That's fine. Hours don't matter. Results do.

What Makes This Work

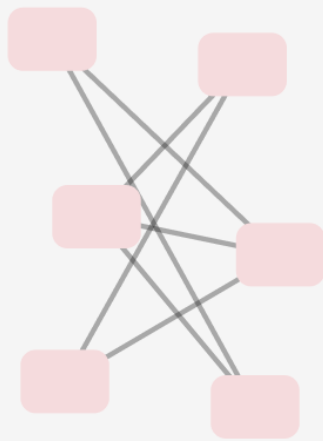
It requires trust in both directions. The client trusts the developer isn't slacking. The developer trusts the client won't treat a fixed rate as a license for infinite scope.

[Weekly delivery](#) creates that trust naturally. The developer delivers value, the relationship continues. The developer doesn't deliver, the client cancels. No surveillance needed when accountability is built into the model.

Consolidation

Fewer tools, vendors, and moving parts make better systems

One platform beats many



Every service added is another potential failure point.

Every service added is another potential failure point. Most developers know this and add the services anyway, because best-of-breed feels responsible. It isn't.

The Sprawl Problem

Picture a standard web app stack: Auth0 for authentication, Stripe for payments, S3 for storage, PostgreSQL on Heroku for the database, SendGrid for email, Algolia for search, Twilio for SMS. That's seven services. Seven bills. Seven sets of API keys to rotate. Seven dashboards to check when something breaks at 2am.

None of that is exotic. That's just a basic CRUD app with common features. And the developer is already managing integration complexity like it's a full-time job.

The Consolidated Alternative

One platform can cover authentication, database, file storage, hosting, serverless functions, and analytics — all wired together by default, all on one bill, all documented in one place. Less powerful than best-of-breed in every individual category? Sometimes. Good enough for the vast majority of products? Absolutely.

Nobody wins by using the most sophisticated tools. The win comes from shipping and staying sane.

The Hidden Cost of Every Integration

Each external service comes with a maintenance burden that compounds over time: initial setup, ongoing monitoring, version updates, security patches, API changes nobody asked for, billing anomalies, and documentation that's always slightly out of date. Multiply that by the number of services and the developer has created a second job that produces nothing users care about.

When to Deviate

Specialized requirements justify specialized tools. High-volume transactional email, complex search at scale, strict data residency requirements — these are legitimate reasons to reach for a dedicated service. But a clear business case is needed, not a vague feeling that a dedicated tool is "more professional."

Default to consolidation. Deviate when the operational cost is worth it, and only then.

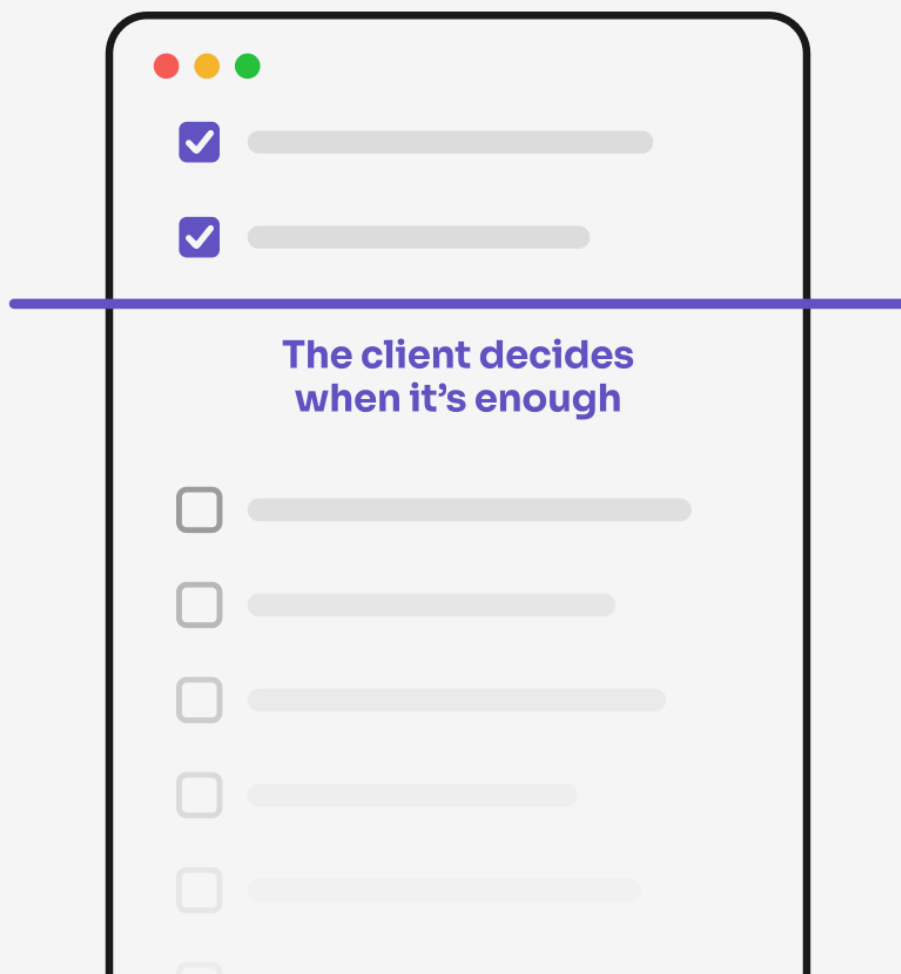
The Migration Myth

"We'll consolidate now and migrate to dedicated services later." That migration almost never happens. The cost is too high, the timing is never right, and the existing system is just functional enough to keep deferring it. Choose platforms the project can live with long-term. The early decisions stick.

Client Decides

The client controls when the engagement ends

The roadmap is infinite



The client decides when the engagement ends.

Software is never truly done. There's always another feature to build, another edge case to handle, another thing to improve. But engagements end. The question is: who calls it?

The Incentive Problem

Every service provider wants projects to keep going. That's not cynicism — it's basic economics. More months of work means more revenue. The conflict of interest is obvious, and pretending it doesn't exist is dishonest.

The developer has an incentive to continue. The client has an incentive to optimize spending. That tension is real. Acknowledging it upfront is the only respectful thing to do.

The Client Is in Control

The client decides when the engagement ends. Not contract terms. Not project milestones the developer defines. The client.

The developer keeps going as long as the client finds value in it. The client stops when the cost no longer justifies the return. That's how it should work — it's the client's money, the client's business, the client's call.

Information, Not Decisions

The developer's job is to keep the client informed, not to make the call. "Here's where things stand. Here's what could be added next. Here's what that would cost." The client takes that information and decides whether it's worth it.

Business objectives get met. ROI thresholds get reached. Budgets shift. Priorities change. Those are business decisions, and the client is the one running the business.

What This Prevents

Without client control, the endless project problem emerges. The developer keeps finding "necessary" work. The client keeps paying. Nobody acknowledges when enough is enough.

It also prevents the opposite: the developer deciding the project is done and walking away while the client still has unmet needs. When the client controls the ending, the client controls whether their needs are actually met before it wraps up.

Why This Is a Philosophical Position

Most service models are designed to keep the client locked in. Long-term contracts, high switching costs, proprietary systems [the client can't take with them](#). All of that serves the provider, not the client.

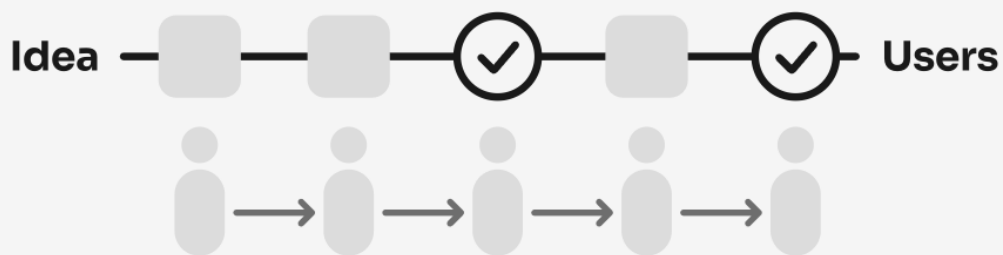
Client control of the exit isn't a pricing detail — it's a statement about who the work is for. If the answer is "the client," then the client decides when it's done. Everything else is just words.

Solo Developer Model

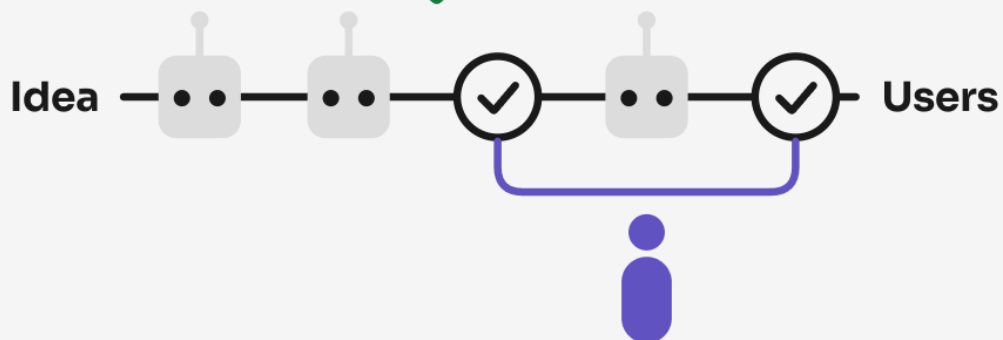
One developer, full stack, AI-augmented

One developer, full stack, AI-augmented

✗ Before



✓ Now



One developer simplifies everything.

The AI era changed the math. A solo developer with modern AI tools can rival the output of a small team from five years ago. For most startup projects, this is fully enough and dramatically more cost-effective.

Why One Developer Works

The advantages are concrete. No context switching between team members. No handoff documents. No standup meetings to synchronize. No merge conflicts from parallel work streams. No "I thought you were handling that" conversations. One person holds the full context of the entire system in their head.

A modern product engineer can deliver and iterate across the full software development lifecycle. Frontend, backend, infrastructure, deployment, [monitoring](#). For most projects, the pace is exactly right for the business to absorb it. The developer ships, the client integrates, the developer ships again.

Other principles follow naturally: [WIP limit of one](#), [scout rule](#), no handoffs, no coordination overhead. One developer simplifies everything.

When It Breaks

When the startup scales. When one developer can't cover the surface area anymore. When the client needs three features shipped simultaneously. This is the natural limit.

And that's fine. When it breaks, the developer adapts. Assemble a small team and manage it. Bring in a designer. Join the client's existing team with a specific area of responsibility. The arrangements get custom, harder to templatize — but the developer handles it.

Some clients need the developer to be the entire tech team. Others need one specialist in a bigger org. The solo model is the default, not the dogma.

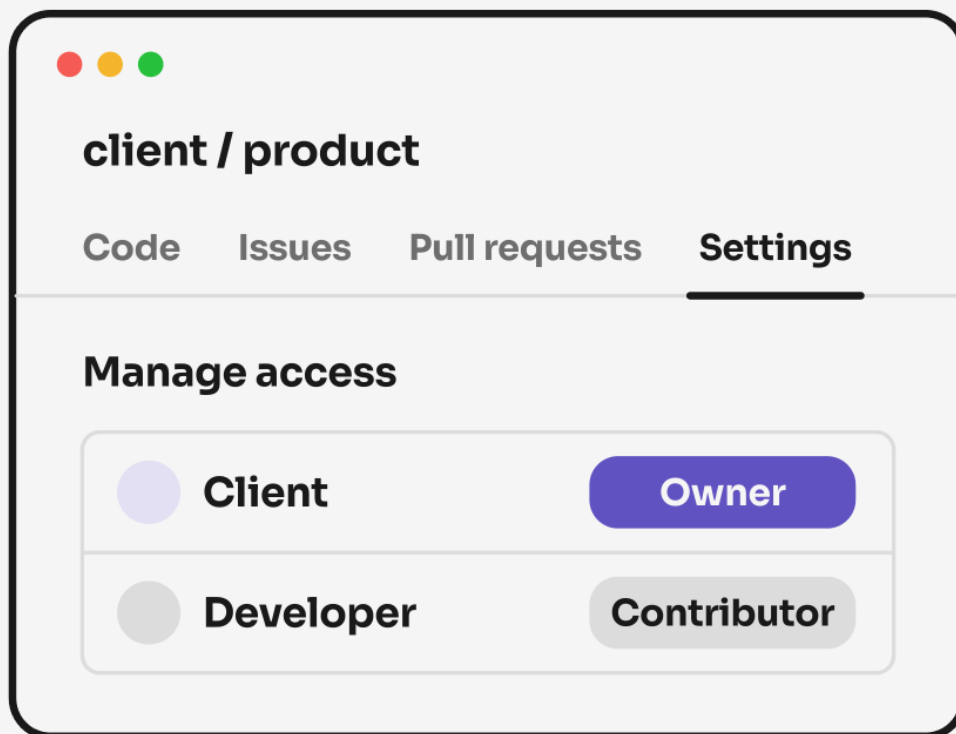
The Sweet Spot

For most engagements — early-stage products, MVPs, ongoing iteration on established apps — one developer with AI tooling, deep stack expertise, and a proven process delivers more than a team of three with coordination overhead.

Solo isn't a limitation. It's a competitive advantage — until it isn't. Know when that moment comes.

Client Owns Everything

Zero lock-in by design



The developer is a contributor. Never an owner.

The client owns the repository. The client owns all services. The client owns the billing for every tool and platform. The developer is a contributor. Never an owner.

From Day One

This isn't a policy bolted on at the end of an engagement. It's how things are set up from the start. Every repo is created under the client's account. Every cloud service is billed to the client's card. Every API key belongs to the client.

If the client cancels tomorrow, they lose nothing. No code held hostage. No credentials the developer controls. No proprietary framework the client can't run without the developer. The client takes their repo, hands it to another developer, and keeps going.

Why It Makes Exit Real

[Exit freedom](#) is meaningless if the developer controls the infrastructure. "The client can leave any time" rings hollow when the code depends on the developer's proprietary tooling or the deployment pipeline requires the developer's involvement.

No proprietary software. No custom frameworks that only the developer understands. [Standard tools, standard patterns](#), standard deployment. If something needs to be proprietary, the developer is a contributor to it — not the owner of the asset.

The Business Logic

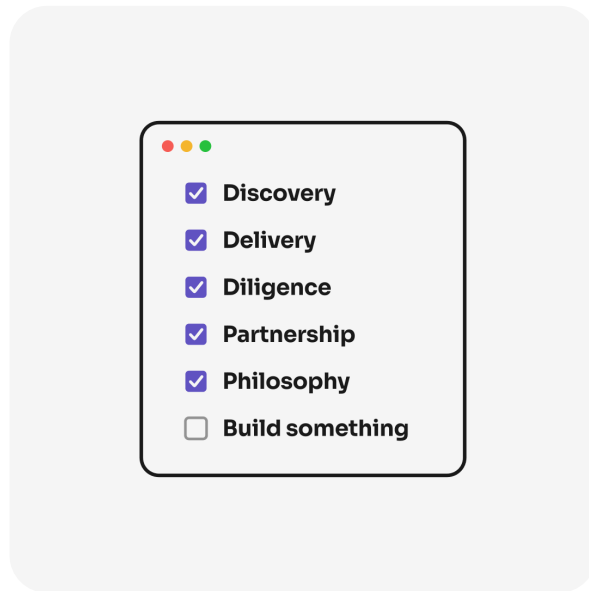
This connects directly to [software as business cost](#). If development is a business expense like office supplies, then the outputs belong to the business. Nobody lets the office supply vendor own the filing cabinets.

When the client owns everything, the retainer is purely for the developer's time and expertise. Cancel and everything stays. Nothing disappears. Nothing breaks.

Portable by Default

Some clients will leave and come back. Some will graduate and outgrow the model. Some will hand the codebase to a bigger team. All of these are fine because the developer built zero dependency on themselves.

The best service providers make themselves easy to leave. Paradoxically, that's exactly why clients stay.



Let's build something together.

These principles aren't rules to follow blindly. They're a foundation to adapt as the work demands. The best way to test any of this is to build something together.

jurij@varstatt.com

[@juritokarski](#)

[linkedin.com/in/juritokarski](https://www.linkedin.com/in/juritokarski)

varstatt.com