

What I've Stepped Into and Commanded Back to Sanity

1. The 30-Minute Tab vs Spaces War

Context:

Two senior devs argued for half an hour about whether the team should use tabs or spaces. One pulled out PSR-12 to justify spaces. The other insisted on tabs for accessibility. Meanwhile, critical bugs were left unresolved.

Problem:

Pure ego war. No decision-making authority. No shared rulebook. Wasted time, delayed delivery.

Action:

I enforced a one-line rule and set up a formatter. End of debate.

Outcome:

- * Zero formatting debates since
- * Clean PRs
- * Faster onboarding

Lesson:

If your devs fight over linters, you don't have a delivery problem — you lack leadership.

2. The “I’m Not a Writer” Excuse

Context:

A founder asked his senior dev to write a simple rulebook. The reply: “I’m not a writer.” He pointed to 400+ pages of PSRs and called it done.

Problem:

No actionable standard. Juniors guessed. Codebase became an inconsistent mess.

Action: I stepped in to create a **2-page rulebook**, printed it, shared it in Slack, and enforced it in code reviews.

Outcome:

- * Code consistency across team
- * Reduced PR friction
- * Faster handovers

Lesson: If your standards don’t fit on a few pages, you don’t have rules — you have chaos with footnotes.

3. The Backend Kingdom

Context:

One senior backend dev built a massive, undocumented system only he understood. Nobody else touched it.

Problem:

- * No one could fix or extend his modules
- * Features took longer
- * High risk if he left

Action:

I took the lead to break apart the monolith, documented key patterns, refactored for team ownership, and enforced code readability.

Outcome:

- * New devs onboarded in weeks, not months
- * No more siloed risk
- * Backend became collaborative

Lesson:

If one dev “owns” the whole backend, your system is a single resignation away from collapse.

4. The Feature Rebuild to Avoid Responsibility

Context:

A “senior” refused to use a teammate’s module because “I didn’t write it, I won’t take responsibility.”

Problem:

He re-implemented existing features from scratch, creating:

- * Duplicate logic
- * More bugs
- * Confusion in the codebase

Action:

I enforced architectural boundaries, clarified ownership expectations, and removed the dev after repeated ego blocks.

Outcome:

- * Feature reuse normalized
- * Codebase simplified
- * Morale improved across the team

Lesson:

If a dev refuses to build on teammate code, you don’t have a coding issue — you have an ego problem.

5. The Quiet Resignation of Top Talent

Context:

New hires joined a dev team and quickly saw the red flags:

- * No tests
- * Weekly production fires
- * No leadership
- * No clear rules

Problem:

The AAA players — those with high standards and experience — started asking, “What shithole have I landed in?”

They waited a few weeks to see if things improved.

They didn’t.

So they quietly started job-hunting.

Observation:

- * The best talent left fast.
- * The C-players and juniors clung to the job for security.
- * What remained was a mediocre team stuck in endless complaining and stagnation.

Action:

I reset technical leadership, imposed structure, introduced sanity-level processes, and cleaned up the worst tech debt to restore trust.

Outcome:

- * New hires stopped ghosting
- * Retention improved

* The delivery pace returned to sustainable

Lesson:

Top talent doesn't quit loudly — they disengage silently.

If your best people leave within 90 days, your system isn't broken — it's repellent.

6. The Junior-Only Money Trap

Context:

One company built their team entirely out of junior developers to “save cash.” No leads, no seniors, no tech strategy.

Problem:

Juniors needed constant guidance. They weren’t confident making decisions. Every step required validation. Every delivery dragged.

What happened:

- * Features took forever
- * Quality dropped
- * Fires became normal
- * And senior mentors were nowhere to be found

Action:

I stepped in to define structure, set decision boundaries, and eliminate the false economy of “cheap labor.”

Outcome:

- * Delivery predictability returned
- * Juniors had real guardrails
- * Velocity went up despite a smaller team

Lesson:

A team of juniors without guidance isn’t cheap — it’s a liability. It burns cash silently and delivers nothing on time.

7. Chronic Underestimation and Delivery Chaos

Context:

The founder of one company repeatedly underestimated project timelines — for years. Every delivery was late. Devs gave optimistic estimates under pressure, knowing they'd be blamed when things slipped.

Problem:

- * No buffer
- * No risk accounted for
- * Trust eroded with clients
- * Team morale cratered
- * Weekly standups became public shaming rituals

What happened:

Devs said: "2 weeks." I said: "Add 2 weeks for unknowns. Add 2 more for peace of mind."

Still, the owner pushed to tell clients "2 weeks." Every time — it blew up.

Deeper impact:

Every failed delivery chipped away at the team's pride.

People wanted to win — but leadership made it structurally impossible.

No matter how hard they worked, they walked into standups knowing they'd be blamed.

The team didn't just lose trust in the process — they lost belief in the *possibility* of victory.

Action:

I re-trained the estimation culture.

Shifted internal timelines vs. external promises.

Built buffers into planning and reframed the narrative: “We deliver early when possible — never late.”

Outcome:

- * Clients got honest, predictable timelines
- * Team had breathing room to actually build
- * Morale improved — because people could finally win again
- * Delivery confidence rebuilt from the ground up

Lesson:

Optimism isn't a strategy. Hope kills delivery.

If your team never wins, they won't stay.

Estimate for *peace of mind* — not to look good on a kickoff call.

8. The Pressure Cooker From Client Deadlines

Context:

A founder agreed to build a custom app for a major client — and accepted their timeline: 45 days. Alongside 20 custom features.

Problem:

- * Timeline was unrealistic from day one
- * Client submitted 5 pages of change requests mid-build
- * No scope control
- * Devs worked weekends, burned out
- * No recovery time between sprints
- * Launch was buggy, rushed, and flagged by app stores

Compounding failure:

- * Additional “must-have” features were accepted days before release
- * The app hit the stores with bugs
- * 50k users flooded in, features broke
- * Negative reviews couldn’t be undone
- * Codebase was littered with rushed patches

Team impact:

- * They hoped launch would mean relief
- * Instead, a post-launch firestorm started
- * Bugs piled up, pressure restarted, trust collapsed

Action:

I stepped in and advised immediately to remove anything not critical to V1. I forced the conversation: move features to V2.

Some massive items were cut just in time.

Outcome:

- * Launch delay was reduced
- * The fallout was ugly — but could've been catastrophic
- * Team saw someone finally push back against the chaos

Lesson:

Accepting a client's fantasy timeline doesn't win trust — it guarantees disappointment.
Leaders must protect the team's ability to deliver — not sell them into the ground.

9. The Irreplaceable Developer Trap

Context:

A developer realized the company had no backup plan — and used that leverage. He openly rejected work requests, cherry-picked tickets, and violated team rules in Slack. He even argued with the founder, knowing no one else could take over his module.

Problem:

- * Devs built fragile silos of personal code
- * Refused to collaborate
- * Made systems intentionally opaque
- * Leadership had no leverage

Action:

I exposed the single points of failure, enforced documentation, and made rulebook compliance mandatory. I advised the founder to terminate repeat violators *publicly* — so the team saw that loyalty to the mission mattered more than one person's comfort.

Ongoing mitigation:

- * Enforced pairing and cross-review of critical modules
- * Announced continuous hiring to build optionality
- * Made clear that “no one is irreplaceable” is policy — not theory

Outcome:

- * Toxic leverage behavior disappeared
- * Team saw leadership had a spine
- * The codebase became maintainable — by design

Lesson:

If someone believes they're irreplaceable, they're dangerous.

Systems must be built so that no one person becomes a hostage taker.

10. The Pushback Against Change

Context:

I introduced a Git strategy shift — from classic GitFlow to a tailored GitHubFlow-style process. The goal was simple: stop batching features and let teams ship smaller releases, faster, safer.

Problem:

The devs pushed back hard:

- * "I've never seen anything like this. It's bullshit."
- * "You'll get merge conflicts."
- * "No other company I've worked at does it like this."

They defended a broken system — even though it caused **production failures weekly**. Releases felt like bomb defusal operations. Still, fear of change kept them clinging to the known.

Even more dangerous: senior devs made ****purely emotional assessments**** with zero data. If the founder had listened to that noise, the team would still be stuck in release hell.

Action:

I didn't argue. I asked: **"Do you have a better idea that fixes the current problem?"**
Silence.

Then I said: **"Let's test this on ONE product for two weeks. If it's painful, we adjust. If it's a disaster, we drop it. Agreed?"** They agreed.

Outcome:

- * After 2 weeks, it worked perfectly
- * The same devs who resisted said: **“It’s totally fine.”**
- * The system is still in use today

Lesson:

Never let emotion masquerade as data — especially from the loudest voices in the room.

Change always triggers pushback — especially from devs who aren't strategic.

You don't win with brute force. You win with framing, options, and structured pilots.

Want me in your corner?

Download my Sales Letter (PDF) to see how I step in, what I offer, and what it costs:

👉 docsend.com/view/4xdt3jbid22j7w83

VIKTOR JAMRICH
CONSULTING