[](./15 unbreakable laws of software engineering that keep breaking us _ by Devlink Tips _ May, 2025 _ Medium_files/1_DBoF-S1OR5UIH7peTQFfRw.png)

Introduction: the gravity of coding laws and how we still fall

Ask any senior engineer about the “laws” of software development, and you’ll get a smirk, a sigh, or a war story involving YAML and a 3 a.m. deploy. These aren’t just abstract ideas taught in dusty CS textbooks they’re the brutal truths that slap you mid-sprint when your estimate was “just a few hours.”

Why do we keep repeating the same mistakes? Because software engineering isn’t just writing code it’s juggling complexity, deadlines, people, and the absurdity of Jira tickets labeled “minor bug,” which then break production.

These laws exist because we’re not just coding for machines we’re coding **with** humans, **for** humans, and **against** entropy.

In this article, we’re going to walk through 15 software engineering laws not 13, because we like pain that define (and sometimes destroy) your dev life. But we’re not doing it the textbook way. This is the **“developer who’s had 3 coffees and just broke staging” version** casual, honest, and sometimes funny, because if you don’t laugh, you’ll cry.

By the end, you won’t just know the laws. You’ll remember the last time each one bit you in production.

Ready?

Next: **moore’s law of expectations** where faster tech means crankier users. Want me to continue?

Section 1: moore’s law of expectations a.k.a. why nobody waits anymore

Remember when websites took **10 seconds** to load and we were like, “Wow, the internet is magic!” Yeah, those days are gone. Moore’s Law said computing power doubles every ~2 years but nobody told users that their *patience* doesn’t.

Now if your app takes more than 1.5 seconds to load, users are already rage-closing tabs and tweeting, “This site is trash.”

Welcome to the **Moore’s Law of Expectations**: as tech gets faster, user expectations get *stupidly* higher.

Real dev pain:

- You spent 6 months optimizing a backend pipeline. Users? “Still slow.”
- You added a loading spinner with a cute animation. Users? “Why is this even loading?”
- You ship a killer ML model that takes 2 seconds to process results. Users? “I thought this was real-time?”

Fun fact:

A Google study found that **53%** of mobile users bounce if a page takes longer than 3 seconds to load. That’s it. No love for your brilliant CSS grid.

Pro tip:

- Build fast, but **design for impatience**. Skeleton screens > spinners.
- Preload intelligently. Anticipate rage.
- And test on a potato phone. Seriously.

Because no matter how fast your backend is someone, somewhere, is using Safari on hotel Wi-Fi.

Section 2: zawinski’s law of software complexity every app becomes email

Jamie Zawinski once said:

> *“Every program attempts to expand until it can read mail. Those programs which cannot so expand are replaced by ones which can.”*

It was meant as a joke.
It is now… **prophesy.**

You start building a focused, minimal app:

> *“It’s just a calendar!”*
> *Two sprints later:*
> *“We should add notifications, comments, user messaging ”*
> *Before you know it, your productivity tool has a markdown editor, Slack clone, Gmail sync, and a broken inbox no one uses.*

Real dev example:

- Dev 1: “Let’s build a to-do list app.”
- Dev 2: “Can users assign tasks?”
- Dev 3: “Let’s add @mentions and DMs.”
  Now you’re writing regex to parse emojis in chat threads. Congrats — you’re Google Wave.

The rabbit hole:

Zawinski’s Law doesn’t mean devs love email — it means **scope creep is inevitable**. Product managers, users, and your own brain will keep shouting, *“Just one more feature.”*

But every feature adds complexity. And complexity kills simplicity. And then you’re debugging your own Slack clone inside a Pomodoro app.

Dev survival tip:

- Say “no” like it’s your job (because it is).
- Separate “cool idea” from “critical feature.”
- Fight feature bloat like it’s legacy code trying to enter prod.

Because once your app does email, someone will ask,
“Can it do video calls too?”

Section 3: hofstadter’s law (it always takes longer even

when you know that)

> ***"It always takes longer than you expect, even when you take into account Hofstadter's Law."**Douglas Hofstadter, also a time-traveling senior dev probably*

There's a universal truth in software engineering: **nothing ever ships on time.** Not because you're lazy. Not because the tech is broken (okay, maybe a little). But because building software is like fixing a car while it's still on the highway in a thunderstorm and someone just asked for Bluetooth support.

The law in action:

- You estimate a feature will take **3 days.**
  It takes **8**, and that's if you're lucky.
- You plan a "simple refactor."
  Two weeks later, you're crying over circular dependencies and wondering why your linter hates you.

Why it hurts:

- Software projects are *nonlinear* one tiny issue (like that async bug) can derail the whole timeline.
- Unexpected blockers are expected.
  ("Oh, we forgot auth." "Wait, is that API still supported?")

Real data:

A Standish Group CHAOS report found that **only 29% of software projects** are completed on time and within budget. That's not a stat that's a cry for help.

Dev tips to beat (or at least manage) the curse:

- Always add **buffer** time. Then add more.
- Break features into micro-deliverables.
- Use the sacred rule: *Double the estimate. Then double it again if a manager is watching.*

Because even when you *expect* things to go wrong, they somehow find a new way to surprise you.

Section 4: gall’s law complex systems are born simple or die trying

> ***“A complex system that works is invariably found to have evolved from a simple system that worked.”**John Gall, probably watching someone over-engineer a to-do app*

Gall’s Law isn’t just about systems it’s about the egos that try to build them.

We’ve all seen it: a junior dev wants to impress, an architect wants to make their mark, and suddenly your weekend side project has CQRS, GraphQL federation, an event-driven microservice mesh, and… three users.

What Gall really means:

Don’t try to ship a spaceship on day one.
Build a skateboard. Then a bike. Then maybe, just maybe, the hoverboard.

If your MVP has a service registry, gRPC, and a custom logger before it even renders a button… you’re not building software. You’re building suffering.

Real example:

- The simple system: monolith with REST endpoints.
- The complex “rewrite”: 18 microservices, CI/CD that fails silently, and a `k8s` config only one person understands.

Guess which one’s still up?

Dev wisdom:

- Build the **simplest working thing**. Evolve it later.
- Complexity is easier to add than remove.
- Design for today’s problems, not imaginary scale that might never come.

Because if you try to launch with version 5.0 of your system before version 1.0 ever runs…
You’re not following Gall’s Law **you’re fighting physics.**

section 5: conway’s law your org chart is your architecture

> ***“****Any organization that designs a system will produce a design whose structure is a copy of the organization’s communication structure.”**Melvin Conway, aka the guy who saw your messy codebase coming*

Conway’s Law is the reason your frontend talks to four different backends maintained by three different teams in four different time zones… badly.

It’s not because the system was badly designed.
It’s because your **team** was.

Real dev situation:

- Team A builds a service in Java.
- Team B builds theirs in Go.
- Team C only communicates in Slack emojis.

You end up with a “modular architecture” that’s really just a **dev org family tree** with all the passive aggression baked in.

Conway’s Law in action

- Microservices built by siloed teams? They don’t talk.
- Monolith maintained by one senior dev? It’s clean… until he quits.
- Product splits frontend/backend teams? Prepare for mismatched APIs and Figma vs reality arguments.

Pro tip:

- Align teams with the **user experience**, not just the tech stack.
- Make communication **part of the system design**.
- The more walls between devs, the more duct tape in your system.

Because software doesn’t exist in a vacuum.
It exists in meetings, Slack channels, and confused Jira tickets.

Section 6: brook’s law (adding more devs makes it worse, not better)

> ***“Adding manpower to a late software project makes it later.”**Fred Brooks, whispering this into the void every sprint planning*

Imagine your project’s on fire. The deadline’s in two weeks. Panic sets in.
What does management do?
Add three new devs to “help.”

Now you’re onboarding, explaining the weird parts of the codebase, fixing merge conflicts, and answering the classic new-hire question:
“Hey, what does `**initLegacyWorkerFlag(true)**` **do?”**
(No one knows.)

Why this law still hits:

- New devs slow you down before they speed you up
- Communication overhead increases faster than productivity.
- Knowledge sharing ≠ instant productivity. It takes time. A lot of it.

Real world chaos:

- A two-dev feature becomes a six-dev Frankenstein.
- Every meeting becomes “Who’s doing what again?”
- Your Git history turns into a graveyard of half-baked ideas and broken hotfixes.

Study says:

A 2020 Harvard Business Review (https://hbr.org/2020/09/why-do-we-keep-adding-people-to-late-software-projects) revisit of Brooks’s Law showed teams that scaled too fast saw increased bugs, more missed deadlines, and lower morale.

Pro tip:

- Instead of throwing bodies at the problem, reduce scope.
- Delay the feature, not your sanity.
- Let small, focused teams own vertical slices not entire pizzas.

Because code doesn’t magically write itself faster with more fingers on the keyboard.
It just breaks in more places.

Section 7: linus’s law many eyes, shallow bugs or utter chaos

_“Given enough eyeballs, all bugs are shallow.”
_Eric S. Raymond, referencing Linus Torvalds, probably not your overworked sprint team

On paper, Linus’s Law sounds beautiful:
More people looking at code = bugs get caught faster.
In reality? It’s more like:
More people = more comments like “Can we rename this to something more semantic?”

What it really means:

Linus’s Law *assumes* everyone reviewing your code is:

- Competent
- Focused
- Actually reading it
  Spoiler: that’s rare.

The PR from hell:

- 4 devs, 12 review comments, and no merge after 3 days
- Half the comments are nits, one is a philosophical debate on naming conventions, and one dev just left a “□” emoji and dipped
- Merge conflict party every time you rebase

It only works when:

- Reviewers are actually engaged
- Code is clean, isolated, and understandable
- You don’t need a README for a single method

Dev tips to make it work:

- Keep PRs small and focused (like feature flags for reviewers)
- Use tools like Reviewpad (https://reviewpad.com/) or Danger.js for automated pre-checks
- Set rules: “No more than X files per PR”
  Because no one wants to review 74 files of refactors during lunch.

□ Bonus reality:

Sometimes it’s not about eyeballs it’s about **who’s got the braincells left at 5:30pm.**

Section 8: postel’s law be kind with input, ruthless with output

> ***“Be conservative in what you do, be liberal in what you accept from others.”**Jon Postel, Internet pioneer and first person to predict JSON spaghetti*

Postel’s Law is a nice way of saying:

“Your API should tolerate weird inputs but never produce them.”

Unfortunately, most systems interpret this as:

“Let’s accept anything and hope it works.”

Cue the horror of parsing user-submitted CSV files with emoji headers.

Real-life chaos:

- Frontend sends `isEnabled: "yes"`
- Backend accepts `"yes"`, `"true"`, `"1"`, `"yup"`, and `"heck yeah"`
- Then your downstream service chokes because it expected a boolean

Seen in the wild:

- APIs accepting timestamps as strings, numbers, or `"ASAP"`
- Form fields that let you enter “February 31st”
- “Flexible” systems that create more bugs than they catch

When it works:

- You build backward-compatible APIs
- You gracefully handle version mismatches
- You avoid breaking the client just because `"optionalField"` was missing

Dev tip:

- Accept junk politely, sanitize it, and respond like a strict schoolteacher
- Use libraries like Joi (https://joi.dev/) or Zod (https://zod.dev/) to validate the chaos
- Validate on both frontend **and** backend. Trust issues are healthy here.

Because one man’s flexible API is another man’s tech debt support ticket.

Section 9: the pareto principle (80% of bugs, 20% of the code)

> _*“80% of consequences come from 20% of the causes.”*
>
> _*Vilfredo Pareto, who definitely debugged JavaScript in his past life*

The Pareto Principle shows up everywhere including your codebase.

That **20%** of gnarly, old, spaghetti-like code?

Yeah, **that’s** where 80% of your bugs live.

And somehow… it’s always in the same damn `utils.js`.

Real-life horror:

- You spend hours optimizing a new feature.
  The bug? It's in a shared helper last touched in 2018 by a dev who now runs a goat farm in Iceland.
- Or maybe it's in `if (value == 'true')` because someone didn't believe in strict equality.

The usual suspects:

- Utility functions with 20+ if-else branches
- Shared libraries no one wants to refactor
- 2,000-line service files named `mainService.ts`

Dev wisdom:

- Identify the "hot spots" in your code (hint: check Git blame and bug reports)
- Refactor slowly and surgically don't rewrite the whole thing (that's a trap)
- Add tests where you fear to tread. You'll thank yourself later.

Pro tip:

Use tools like SonarQube or CodeClimate (https://codeclimate.com/) to spot the high-risk 20%.
Then fix one thing at a time not everything at once. This isn't Dark Souls.

Because the bugs aren't everywhere.
They're just hiding in that 20%… mocking you.

Section 10: the law of leaky abstractions your nice wrapper will betray you

> ***"All non-trivial abstractions, to some degree, are leaky."**Joel Spolsky, dev philosopher and frequent abstraction victim*

Abstractions are supposed to **hide complexity**.
Instead, they often **hide landmines** that blow up when reality leaks through.

You write code like `fetchData()` and expect a nice response.
Then one day, a DNS failure upstream leads to a partial payload and a “cannot read property of undefined” moment in production during a live demo.

The leak in action:

- A framework hides HTTP details… until you hit CORS.
- ORMs abstract SQL… until you get an N+1 query problem.
- Cloud providers make storage simple… until you realize latency spikes are due to cold starts.

Why this law matters:

- No abstraction is perfect they **trade control for convenience**.
- The more layers you add, the further you are from what’s actually breaking.

And when it breaks? You dive down the rabbit hole with Wireshark, Stack Overflow tabs, and tears.

Dev tips to stay dry:

- Learn the layer **under** your abstraction. Even a little helps.
- Log and monitor at the raw level, not just through the abstraction.
- When in doubt, blame DNS (you’ll be right more often than not).

Because one day, you’ll be staring at a bug thinking,
“I thought this was handled by the framework.”
It was. Until it wasn’t.

section 11: the law of diminishing returns when more tests ≠ more value

> ***“Beyond a certain point, additional effort yields progressively smaller improvements.”**Economist logic, but also your test coverage reality*

You’ve hit 80% test coverage nice.

You aim for 90% solid.

You push for **100%**?

Welcome to hell, population: you and your mocking library.

Real dev situation:

- You spend 4 hours writing a test for a one-line condition that throws only if the API returns a 418 “I’m a teapot.”
- You create mocks of mocks, mock the mocks’ responses, then mock your own sanity.

And in the end?
Your tests pass, but **real bugs** still sneak into production because no test caught the actual logic bug hidden under the “happy path.”

What the law looks like:

- First 50% coverage: catch critical bugs, improve confidence
- Next 30%: useful for edge cases and regression
- Final 10–20%: fighting the framework, testing getters, and mocking time itself

Dev tip:

- Focus on **high-impact** code: business logic, data flows, integrations
- Don’t test what the language or framework already guarantees e.g., TypeScript getters… seriously?
- Aim for *“enough coverage to sleep at night”* not perfection

Want a perfect system? Write no code.

Want a working one? Test smart, not obsessive.

Because 100% test coverage might look great in a CI badge but it won’t save you from an angry 2 a.m. call when prod breaks.

section 12: the second law of thermodynamics (but for your codebase)

> ***“Entropy always increases.”**Physics, but also your repo after three sprints*

In physics, this law explains why the universe moves toward disorder.
In software, it explains why your once-beautiful, well-structured codebase now looks like it was written during a caffeine-fueled panic attack.

No matter how perfect your architecture starts, **entropy creeps in**:

- A last-minute hotfix
- A rushed new feature before launch
- That “temporary” workaround from six months ago (that’s still there)

Real-life decay:

- A clean folder structure becomes `components_v2_final_rewrite`
- Code comments go from “clarifying” to “crying for help”
- CI pipelines get so convoluted that no one dares to update them

Entropy triggers:

- New devs unfamiliar with original patterns
- Tight deadlines forcing messy merges
- Business logic duct-taped across three services and a cron job

Dev tip:

- Accept entropy. Fight it **proactively**, not reactively.
- Schedule regular refactoring treat it like debt interest.
- Write documentation. Seriously. Even if just for Future You.

Even the cleanest architecture **rots over time** if no one maintains it.
Your job isn’t to stop entropy it’s to manage it before your repo becomes archaeological.

Because at some point, you stop building new features and start **translating ancient commit messages.**

section 13: hanlon’s razor (never attribute to malice what

tired devs can explain)

> ***"Never attribute to malice that which is adequately explained by stupidity."**Hanlon's Razor, or: why your coworker isn't trying to ruin your sprint on purpose*

When something in your system **breaks in a deeply confusing and painful way**, it's easy to assume sabotage:

> ***"Who wrote this? Were they trying to destroy us?"***

But the truth is far more tragic:

They were probably just **tired**, rushing, or didn't understand the full context.

Real-world examples:

- A `===` was accidentally a `==`
- A `setTimeout` with a delay of `0` was assumed to be *instantaneous*
- Someone copy-pasted code from Stack Overflow and hoped for the best

No one *meant* to destroy your production pipeline with that one-liner.

They just needed to close their laptop and get to lunch.

Why this mindset matters:

- It fosters **empathy** over blame
- It encourages **code reviews** as learning opportunities
- It stops flame wars in PR comments

Dev tip:

- Assume good intentions, then verify with logs
- Use comments to explain weird decisions ("Yes, this looks cursed here's why")
- When reviewing code, ask: "What was the goal?" before judging the method

Because most bugs aren't evil.

They're just tired. Like the dev who wrote them.

section 14: the boy scout rule (leave the code cleaner than you found it)

> *“Always leave the campground cleaner than you found it.”*
> *— Boy Scouts, but also the world’s most underappreciated coding standard*

The **Boy Scout Rule** is simple:
When you touch a piece of code, make it better.
Not perfect. Not refactored from scratch. Just… better.

Fix a confusing variable name.
Extract that 20-line method into a helper.
Delete the `console.log('debug')` that’s been there since 2021.

Why this rule works:

- Small improvements compound over time.
- You reduce tech debt *while shipping features.*
- It saves your teammates (and future you) from pain.

The alternative:

- “I’ll clean this up later.” (Spoiler: you won’t.)
- “It works, don’t touch it.” (Until it doesn’t.)
- “Let’s just ship it.” (Welcome to bug city.)

Real life scenario:

You’re working in `InvoiceService.ts`.

You spot a typo in a method name: `calulateTotal()`.

You fix it. No ticket. No ceremony.

That’s the rule in action.

Dev tip:

- Refactor as you go but set a time limit.
- Add missing types, fix warnings, update outdated comments.
- If you're not confident in the change, leave a `TODO` with context at minimum.

Because if everyone leaves the code just 1% better…
eventually, we might not be terrified of opening `legacy-utils.js`.

section 15: greene's law (every line of code is a liability)

> ***"Every line of code you write is a liability, not an asset."**Ward Cunningham and every dev who's ever screamed into a terminal*

You might think writing code is the goal.
But in reality? The **best code** is the code you **didn't** have to write.

Every line you ship:

- Needs testing
- Can break later
- Must be understood by someone else (or future you at 2 a.m.)

⚠ The harsh truth:

More code = more surface area for bugs, regressions, and existential dread.
You're not building value you're accumulating **maintenance debt**.

Real-world pain:

- You write a custom solution when a 5-line library would've worked
- You build your own logger, scheduler, or state machine
- You forget: even "clever" code needs to be supported forever

Greene’s mindset:

- Write only what’s *necessary*
- Reuse existing tools and abstractions
- Prefer *deleting* code over adding more

Dev tip:

- Treat new code like new expenses justify it
- Practice “code dieting”: can you solve it in fewer lines?
- Refactor ruthlessly, document fearlessly, and **delete bravely**

The codebase is not your art museum. It’s your toolbox.
You don’t get points for cleverness. You get peace for clarity.

Because in the end, the best feature you ever shipped…
might be the one you **never had to write**.

![](./15 unbreakable laws of software engineering that keep breaking us _ by Devlink Tips _ May, 2025 _ Medium_files/1_1hVvOCqq31dIVgMTvu2CnA.png)

Conclusion: software engineering laws are written in blood (and commits)

If you’ve made it this far congrats. You’ve just walked through **15 painful, hilarious, and oddly comforting truths** that every software engineer eventually learns (the hard way).

These laws aren’t here to scare you.
They’re here to remind you: **you’re not alone** in the chaos.

That feeling when:

- A “quick fix” takes all day?
- Your clean code rots in a sprint?
- You write 20 tests and still miss the bug?

That’s not failure that’s the job.
Every bug, late feature, and ugly workaround? It’s just another chapter in the universal dev story.

What to do now:

- Don't aim for perfection aim for progress.
- Don't fear rules break them *with understanding.*
- And please, stop making every app a Slack clone.

Whether you're a junior staring at legacy code or a senior praying before merging to `main`, just know:

> ***You are part of an ancient brotherhood of devs bound by caffeine, cursed by deadlines, and guided by memes.***

Helpful resources so you don't learn all these laws the hard way

Here's a curated list of legit resources, references, and articles to help you dive deeper into the chaos we just covered written by people who've been burned enough times to know better:

Foundational Reads

- The Mythical Man-Month by Fred Brooks (https://en.wikipedia.org/wiki/The_Mythical_Man-Month) where Brook's Law was born (and your project plan died).
- Code Complete by Steve McConnell (https://www.goodreads.com/book/show/4845.Code_Complete) practically a bible for writing clean, maintainable code.
- Refactoring by Martin Fowler (http://refactoring%20by%20martin%20fowler/) how to fight entropy and win.

Articles & Blogs

- Joel Spolsky on Leaky Abstractions (https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/) a must-read rant.
- Hofstadter's Law on LessWrong (https://www.lesswrong.com/posts/2cJnn8QgF86xAotYH/hofstadter-s-law) because time management is a myth.
- Zawinski's Law on Wikipedia (https://en.wikiquote.org/wiki/Jamie_Zawinski) for when your simple app starts sending email.
- Postel's Law in Internet Protocol Design (https://datatracker.ietf.org/doc/html/rfc791) for the protocol nerds.

Tools for Sanity

- SonarQube (https://www.sonarsource.com/products/sonarqube/) find bugs and code smells before they find you.
- Zod (https://zod.dev/) and Joi (https://joi.dev/) for keeping your APIs sane.
- CodeClimate (https://codeclimate.com/) see where the bad 20% lives in your repo.
- Git Blame (https://git-scm.com/docs/git-blame) aka: Who did this?

Because devs need memes too

- r/ProgrammerHumor (https://www.reddit.com/r/ProgrammerHumor/) cry-laugh between builds.
- CommitStrip (http://www.commitstrip.com/en/) comics for coders, by coders.

![](./15 unbreakable laws of software engineering that keep breaking us _ by Devlink Tips _ May, 2025 _ Medium_files/1_vVMXXx-SXWfd9RJo6aslsQ.png)

[

Technology

](https://medium.com/tag/technology?source=post_page-----d97663f63f2e---------------------------------------(https://medium.com/tag/technology?source=post_page-----d97663f63f2e---------------------------------------))

[

Programming

](https://medium.com/tag/programming?source=post_page-----d97663f63f2e---------------------------------------(https://medium.com/tag/programming?source=post_page-----d97663f63f2e---------------------------------------))

[

Software Development

](https://medium.com/tag/software-development?source=post_page-----d97663f63f2e---------------------------------------(https://medium.com/tag/software-development?source=post_page-----d97663f63f2e---------------------------------------))

[

Software Engineering

](https://medium.com/tag/software-engineering?source=post_page-----d97663f63f2e---------------------------------------(https://medium.com/tag/software-engineering?source=post_page-----d97663f63f2e---------------------------------------))