The 20 Software Engineering Laws

Why software projects fail, systems rot, and teams slow down.

Most engineers learn these laws the hard way. When you try to rewrite something and it doesn’t deliver, or when a project is already late, adding engineers to the team will just make it fail faster. Sometimes, when you start using a metric to measure progress, the whole team will start trying to manipulate it. Then, six months later, someone mentions a 1975 law that addresses exactly what happened.

I paid a price to learn this, too: I spent half my career learning these lessons the hard way, as many others probably did.

The twenty laws listed below are the ones I refer to most often, although there are more (more on this later). Software development laws explain what is happening, what is about to happen, and what will not work no matter how hard you try. Some of these laws are sixty years old. They still apply to software development in 2026, and they will still apply in 2036 because they are not really about software. They are about people working together to build things under time pressure (basically, a lot of them are just laws of human nature).

These laws are not rules that tell you what to do. They tell you what is already happening, but you still have to make the decisions. These laws just help you understand what is going on.

Each one of these laws made the list because I have seen it happen to

me. My book covers all fifty-six laws. If you only have time to remember twenty software development laws, these are the ones that I think are important.

In particular, we will talk about the following laws:

1. **Gall's Law**: A complex system that works is always built from a simple system that worked first.
2. **KISS**: Keep it simple. Anything beyond that is overhead.
3. **Conway's Law**: Organizations design systems that mirror their communication structure.
4. **Hyrum's Law**: With enough users, every observable behavior of your API becomes someone's dependency, no matter what the contract says.
5. **CAP Theorem**: A distributed system can guarantee only two of: consistency, availability, and partition tolerance.
6. **Zawinski's Law**: Every program expands until it can read mail. The ones that cannot are replaced by ones that can.
7. **Brooks's Law**: Adding people to a late software project makes it later.
8. **Ringelmann Effect**: Individual output drops as team size goes up.
9. **Price's Law**: Half the work is done by the square root of the people.

10. **Dunning-Kruger Effect**: The less you know about something, the more confident you tend to be.

11. **Hofstadter's Law**: It always takes longer than you expect, even when you account for Hofstadter's Law.

12. **Parkinson's Law**: Work expands to fill the time available.

13. **Goodhart's Law**: When a measure becomes a target, it stops being a good measure.

14. **Gilb's Law**: Anything you need to quantify can be measured in some way that beats not measuring it.

15. **Knuth's Optimization Principle**: Premature optimization is the root of all evil.

16. **Amdahl's Law**: The speedup from parallelism is limited by the sequential part.

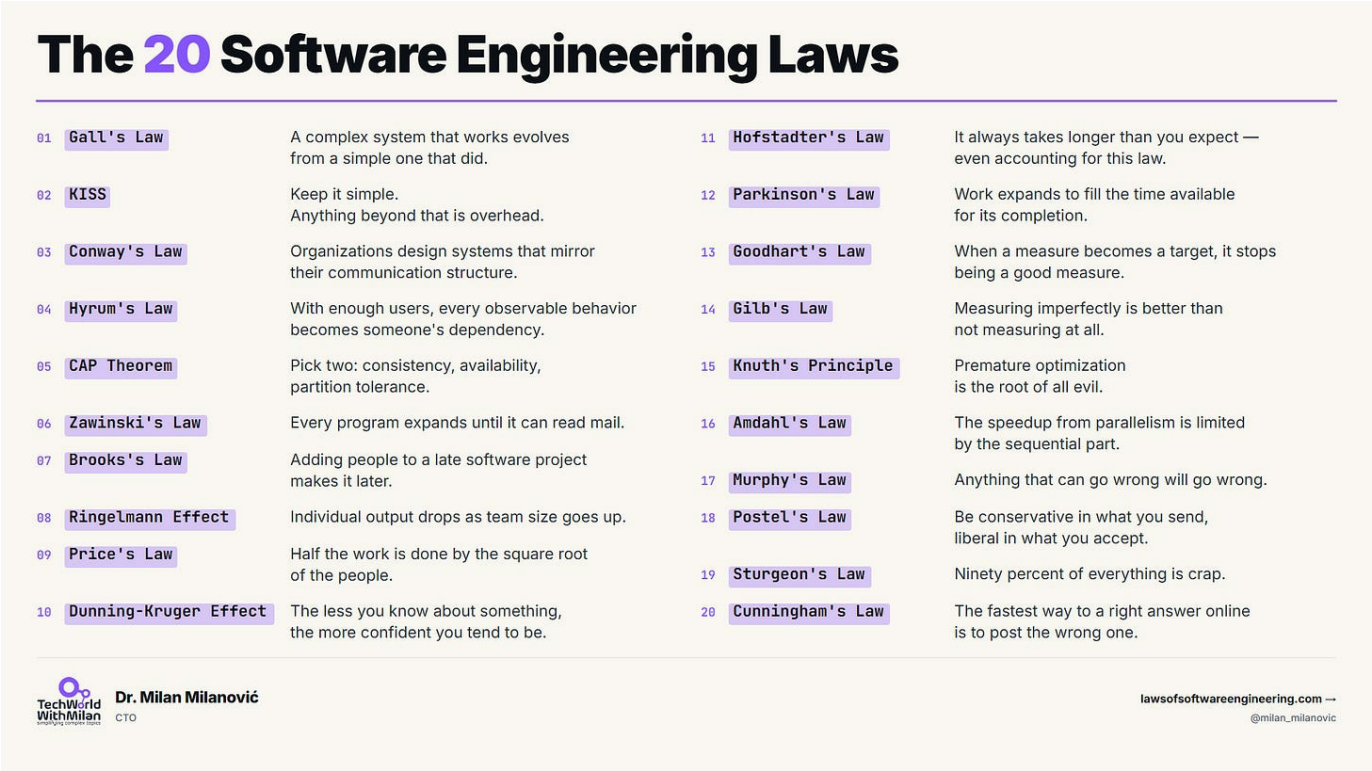
17. **Murphy's Law**: Anything that can go wrong will go wrong.

18. **Postel's Law**: Be conservative in what you send, liberal in what you accept.

19. **Sturgeon's Law**: 90% of everything is crap.

20. **Cunningham's Law**: The fastest way to get the right answer online is to post the wrong one.

So, let’s dive in.



> What to learn more? [The Laws of Software Engineering book]()
>
> is out.

This issue is brought to you by Microsoft

[Azure Copilot Migration Agent (Sponsored)]()

Most cloud migration plans stall in the planning phase. Microsoft’s new Azure Copilot Migration Agent generates one automatically from your VMware inventory, compares lift-and-shift against modernization, and hands landing zone templates to GitHub Copilot. It’s one of six Copilot agents now covering the full Azure ops cycle.

The free Introduction to Azure Copilot Agents module on MS Learn walks

through each. Check it out.

Start the free module

---

1. How systems get built

1. Gall's Law

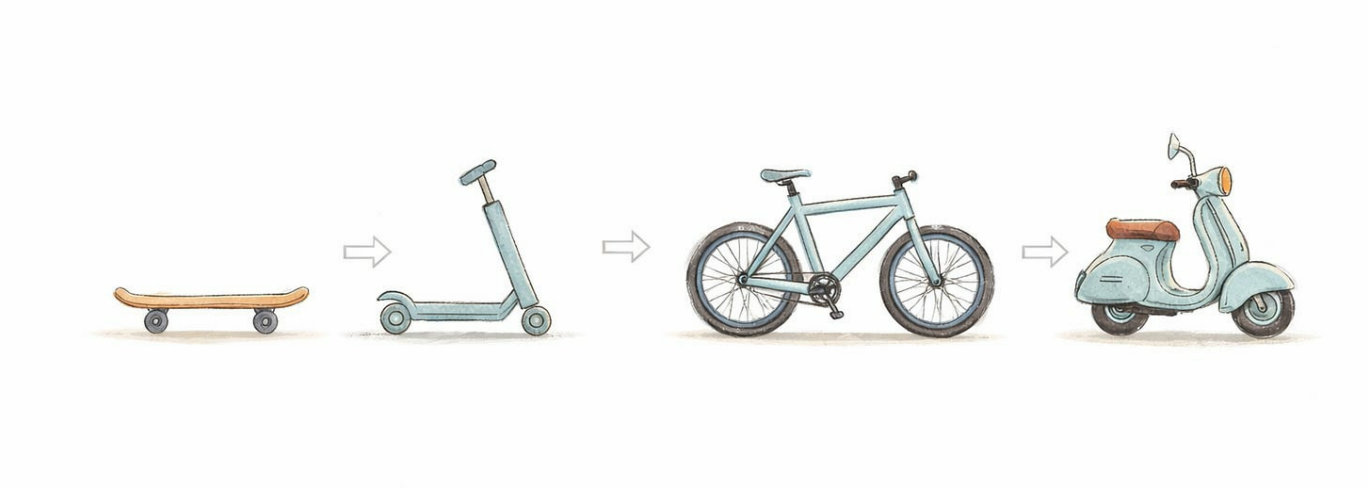
> **A complex system that works is always built from a simple system that worked first.**

Systems do not work well in real life as they do on paper because many problems do not show up until they hit the real world. These problems only appear when real users use systems, and by then, they either work or they do not. Every complex system that works got that way one step at a time. The systems that try to be perfect, from the start, usually fail.

This is why most new versions of systems being rewritten from scratch do not work out, as teams keep all the features they had before, but they lose the simple things that made the old systems good.

Examples. Let's take an example of Instagram. At the start, it was something else, but not a picture-sharing platform. The app was called Burbn, and it had: check-ins, gaming, photo sharing, all stuck together. Then, the founders cut everything except photo sharing, and the stripped-down core became the product.

Google Wave went the other way. It launched with chat, email, a forum, and a document editor, all at once. Nobody could tell you what it was for, and it was dead in 15 months.



Gall's Law

2. KISS (Keep It Simple, Stupid)

> **Keep it simple. Anything beyond that is overhead.**

The KISS principle is a reminder that simplicity should be our key goal. If

you can solve a problem with a 50-line script vs a complex 500-line solution, KISS favors the simpler solution because each line of code has the potential to cause an error.

Why is simplicity so important? Software, in general, is complex to build and must be understood by humans. A simple design is much easier to maintain: new team members can get up to speed faster, bugs are easier to localize, and modifications cause fewer ripple effects.

The KISS principle encourages developers to resist “clever” code that does too much at once, and to avoid architecting solutions that address future problems at the cost of current complexity.

Example. Let’s say that we have a startup that needs a feature-flag system and decide to build a custom solution. They built it as a separate microservice with its own database, cache, admin UI, WebSocket notifications, and A/B testing support. It introduces a lot of complexity and takes a lot of time to build, which, if something goes wrong, can cause a lot of trouble.

What they needed was a JSON config file. This would have taken an afternoon.

KISS (Keep It Simple, Stupid)

3. Conway's Law

> **Organizations design systems that mirror their communication structure.**

Your app architecture is already defined and essentially the same as your organization chart. For example, if you have four teams working on a project, you will probably end up with an app that has four parts. If the teams that work on the frontend, the backend, and the data do not communicate, your application will have three parts that do not work well together.

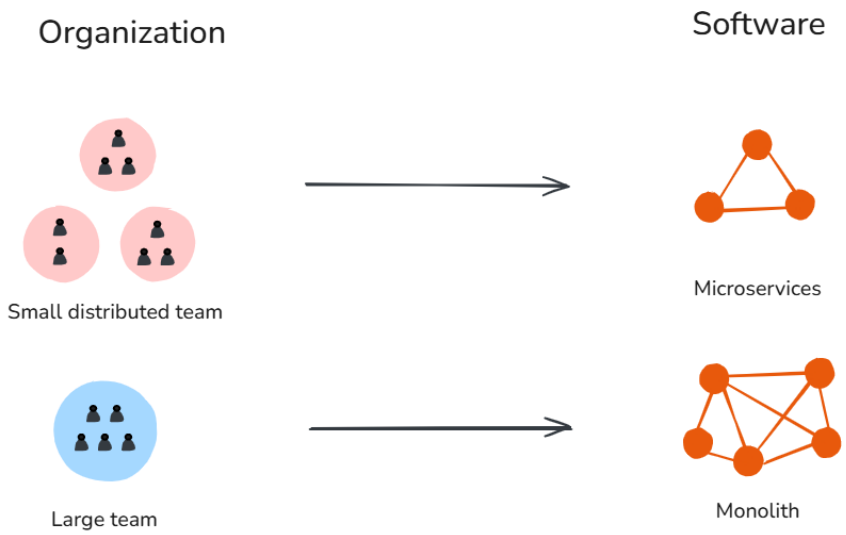
If you rewrite your system without changing how your company is organized, you will still have the system, just written in a different language.

The other way around works too. You can pick the architecture you want and then create teams that would naturally produce that kind of system. Amazon did this back in the 2000s. They broke their system down into

smaller services managed by small teams, which changed how the system and the company worked together. This is called Inverse Conway's Maneuver.

Examples. Many modern AI organizations often split research from application engineering. Then, research optimizes benchmarks, while product ships apps against real users. The output is a model that scores well and a product that doesn't work, because each side is optimizing for its own communication boundary.

The pattern shows up at a small scale, too. A three-person team almost always ships a monolith because the cost of breaking it up is higher than the cost of keeping it together.



Conway's Law

4. Hyrum's Law

> **With enough users, every observable behavior of your API becomes someone's dependency, no matter what the contract**

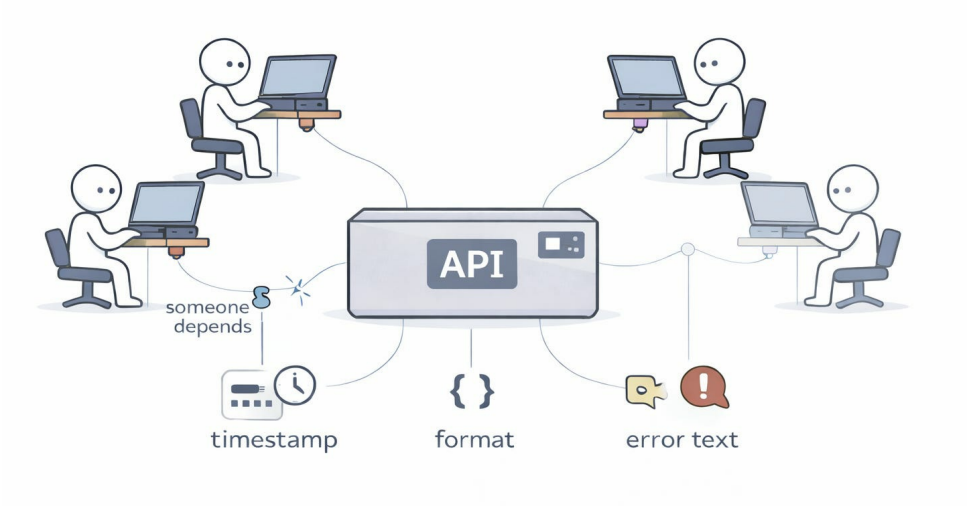
says.

The interface contract you wrote is not a proper contract. The real one is what your system actually does, including the parts you never expected to be important. For example, it could be timing, error message text, key order in JSON responses, and the exact bytes of a hash. Someone, somewhere, is depending on all of it.

This is why backward compatibility costs so much in mature systems. This means that you actually don’t maintain the API you designed, but the accidental one.

Examples. A good example is the SimCity game. I remember well that it had a use-after-free bug that worked fine on Windows 3.x because memory was never actually reclaimed. Then, Windows 95 reclaimed it, and SimCity crashed. Microsoft shipped Windows 95 with a special memory-allocator mode that was activated only when SimCity was running, so the bug would continue to work.

Browsers do this at internet scale. Every quirk that web developers built into the platform effectively becomes part of it. The browser can’t change the quirk without breaking half the web.



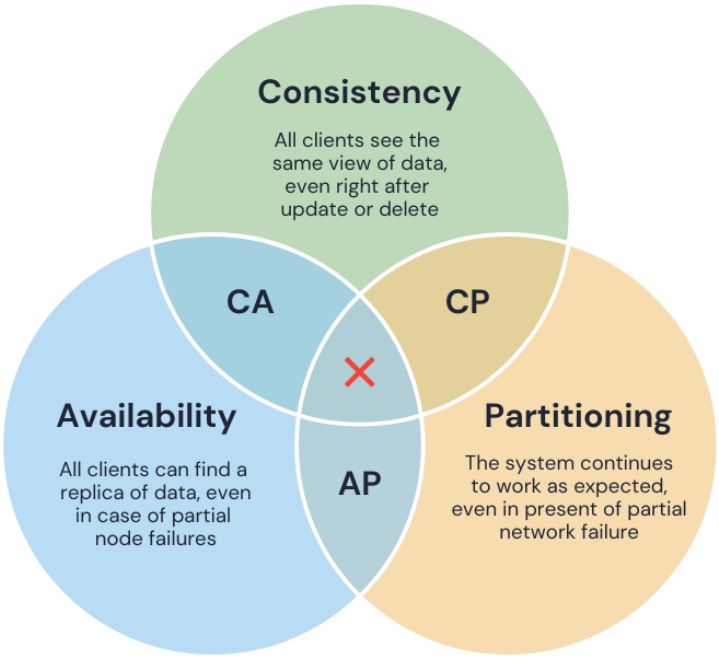
Hyrum’s Law

5. CAP Theorem

> **A distributed system can guarantee only two of the following: Consistency, Availability, and Partition tolerance.**

Networks fail. In a distributed system, that's not something you design around. It's something you accept. Once a partition happens, you have to pick: block writes to keep data consistent, or keep serving traffic and let replicas drift. Every distributed database makes this call. Most just don't tell you which one. They hide behind labels like "eventually consistent" or "highly available" and leave you to find out during an incident.

Examples. MongoDB favors consistency, meaning that when a partition problem occurs, some MongoDB replicas will not accept any data until the entire system is working properly again. On the other hand, Cassandra will keep answering queries even when the replicas do not agree, and it will later fix the inconsistencies. Neither MongoDB nor Cassandra is wrong. They are just making choices about what your system can afford to lose.



CAP Theorem

6. Zawinski's Law

> **Every program expands until it can read mail. The ones that cannot are replaced by ones that can.**

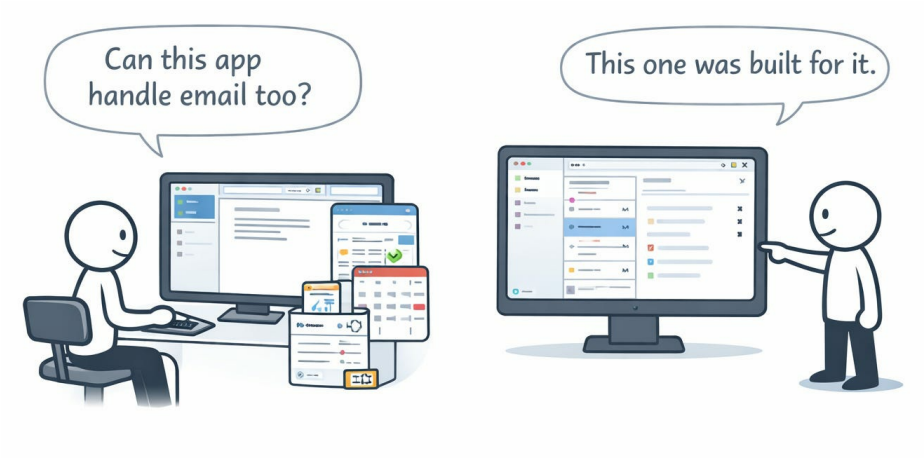
Feature creep is not something that happens during the process. It is actually the process itself. When a tool is good at what it does, and people like it, they start using it all the time. The people in charge of the product want to keep the users engaged and stay on the platform. So the tool begins to take on tasks that are related to it. Over time, the tool becomes really slow and has a lot of unnecessary extra features.

Then a new competitor comes along with a simpler version that does exactly the same thing. As the app's popularity grows, more and more unnecessary features are added.

Examples. A famous example is Netscape, which started as a browser

and ended as a suite with email, news, and a web editor. Firefox came as a fix and stripped it down, got popular, but then added plugins and a developer toolchain.

We also remember Slack, which was launched to kill email and now has voice, video, bots, and an app directory. All of this is possible if the product doesn't have the right north star metrics.



Zawinski's Law

2. How teams lose speed

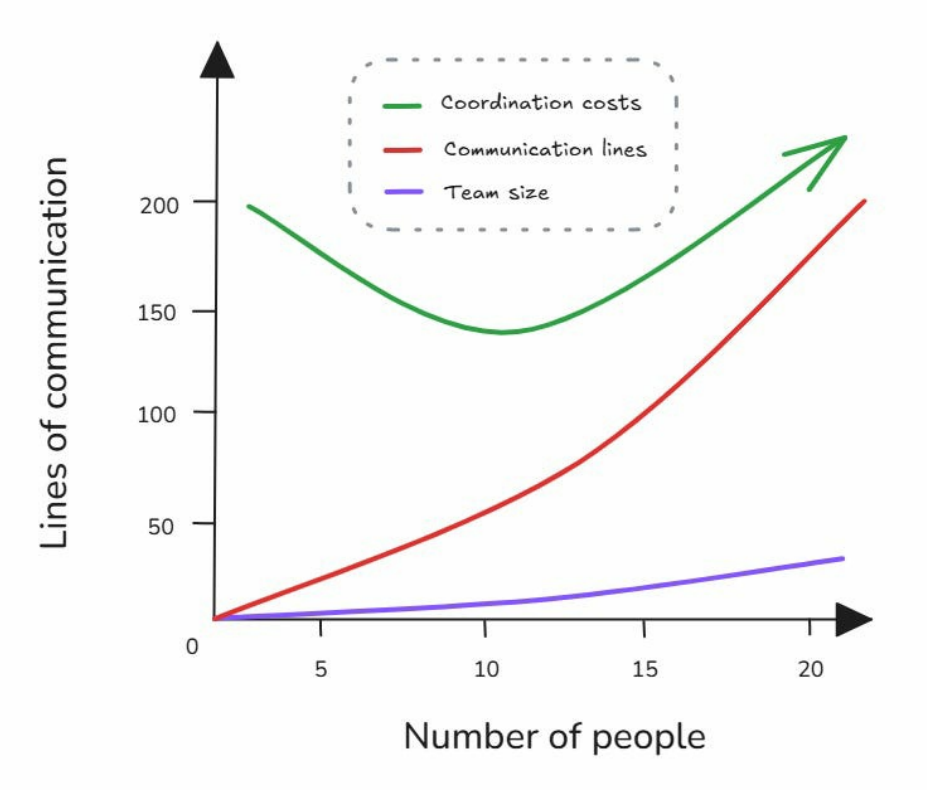
7. Brooks's Law

> **Adding people to a late software project makes it later.**

Software work is not easy to split among team members. When you bring someone new onto the project, it takes them a while to get up to speed, which means your experienced people have to stop what they are doing to help the new person learn. If your project is already behind schedule, adding more people won't make it go faster. It will just make things worse.

Frederick P. Brooks said it well: you cannot have a baby in one month just because you have nine women pregnant. Software work is, like that, too. Software work does not get done faster just because you have people working on it.

Example. Once, I was a team lead of eight people, and we were always behind schedule. My first thought was to hire two engineers to help us catch up. But in the meantime, while we were searching for new people, two people left us. It seemed that everything was now working better, communication was easier, and we managed to do more than before. So, obviously, the solution was to make the team smaller, not bigger.



Brooks's Law

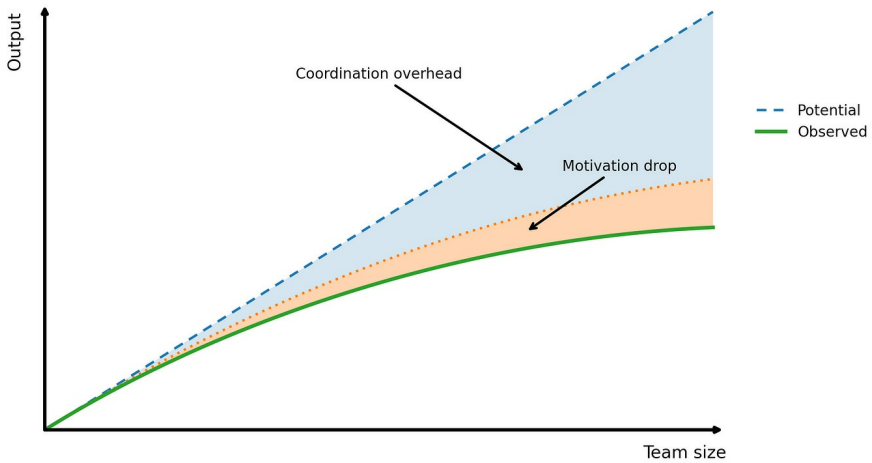
8. Ringelmann Effect

As teams grow, output per person falls.

When many people pull on the rope, each person does not pull as hard. Some of this is because it is hard to work smoothly, and some of it is because people think someone else will do the part. Either way, this pattern is real. It is more extreme than most people think.

Examples. A large GitHub study measured this directly. Developers on teams of 2-5 people averaged around 1,850 lines of code a month, while a team of 10 dropped to 1,200. At 50 or more, it was 450. Output per person fell 75%.

This is why small teams ship faster than big ones, and why Amazon's two-pizza rule holds true. It's a defense against Ringelmann. This is especially true in today's AI-driven world, where productive teams have fewer members than before, as AI is driving up personal and team productivity.



The Ringelmann Effect

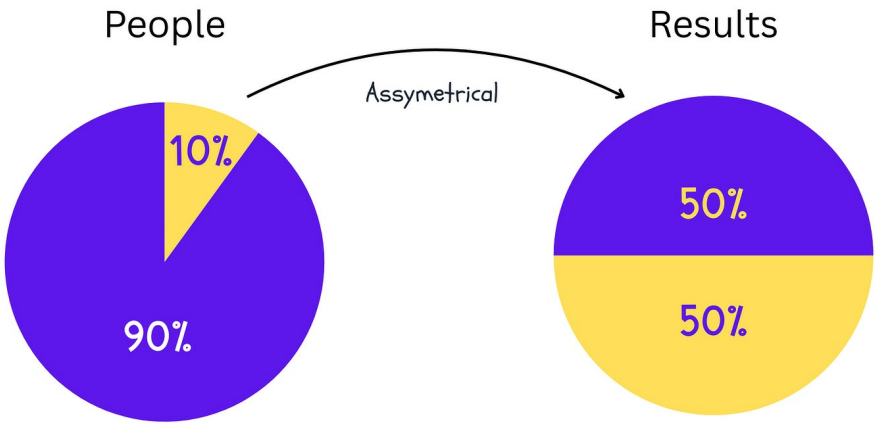
9. Price's Law

Half the work is done by the square root of the people.

In a group of 100 people, about 10 people actually do half of the work that matters. If you have a group of 16 people, it is likely that 4 people do most of the work. This is true for every creative field.

The people in the group who do most of the work are really important, but the others are important too, because they do what needs to be done to support everyone else. They make sure everything runs properly (sometimes called glue work). So we need both groups, but the problem is that if the top people in your group leave, the group will lose a lot of its ability to get things done.

Example. We all know that when Musk took over Twitter, it cut its staff by roughly 50%, and the site kept running. Price's Law predicted that. What the law did not predict was what the layoffs removed: depth in trust and safety, SRE coverage, and incident response. The top performers kept the lights on. The organization lost the ability to handle the next hard problem, and Twitter quietly asked some laid-off people to come back.



Price's Law

3. Why plans drift

10. Hofstadter’s Law

> **It always takes longer than you expect, even when you account for Hofstadter’s Law.**

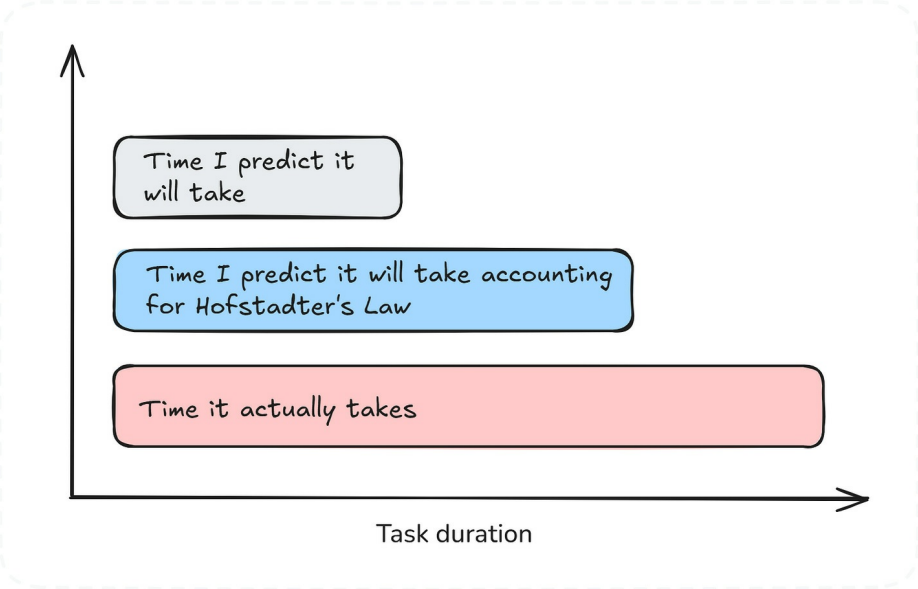
Let’s say you need to estimate how long something will take. You think four weeks is an estimate, but then you remember that your guesses are usually too optimistic, so you double it to eight weeks, just to be sure. But in the end, it takes sixteen weeks.

Now you think, the next time you will be better, aren’t you? You think it will take sixteen weeks because that's what happened the last time. No, it now takes thirty-two weeks, because things you don’t know about surprise you. These are tasks such as unplanned integration issues or requirement changes.

In practice, Hofstadter’s Law explains why techniques like padding estimates, awareness of Parkinson’s Law, and the use of historical data are essential, yet surprises still occur.

Example. A good example of the Hofstadter law is the Berlin Brandenburg Airport project. The software integration process was taking much longer than expected, as it involved 75,000 sensors and 50,000 light fittings. The plan was to take 18 months to finish, but they later realized this was not possible and extended the timeline to 30 months. In the end, it took 7 years to complete, with a final cost of €7 billion. This was 2.5x higher than

planned, and the airport opened 9 years late.



Hofstadter's Law

11. Dunning-Kruger Effect

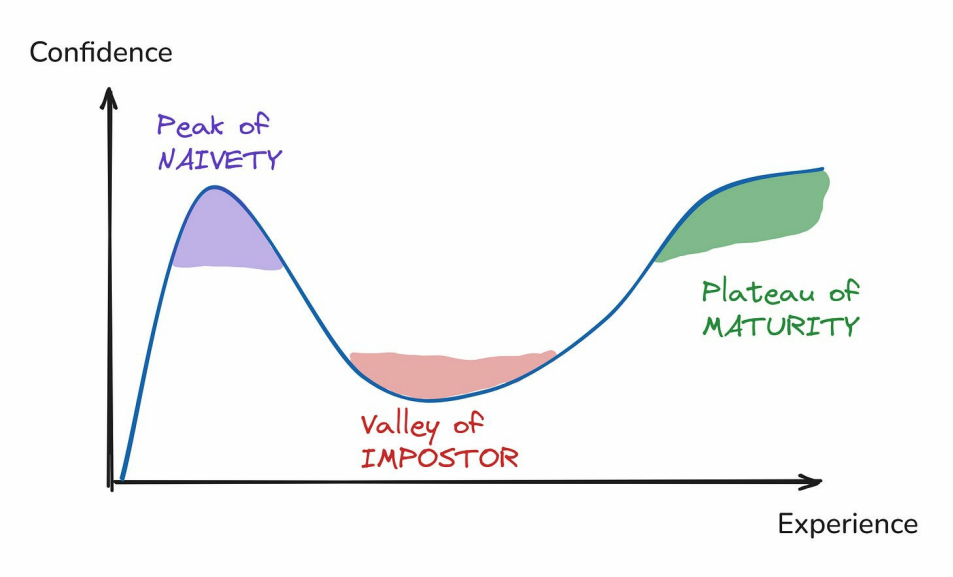
> **The less you know about something, the more confident you tend to be.**

Here is the uncomfortable part. The skill you need to do something is the same skill you need to judge how well you did the thing, and this is the problem. People who are not very good at something cannot see what they are doing wrong, so they think they are better at the thing than they really are. Yet, people who are good at it see all the things they are still getting wrong, so they think they are not as good at it as they really are.

Examples. When asked when something will be done, new developers often give confident, precise estimates, while experienced developers give ranges (the famous "it depends" answer). The juniors aren't wrong to be convinced. They simply don't yet know what they don't know (unknown-

unknowns).

People usually get really excited about new technology at first. This is because they have not used it a lot yet. We are seeing this happen with Artificial Intelligence now. The people who say AI can do anything are usually the ones who do not use it every day, like managers.



Dunning-Kruger Effect

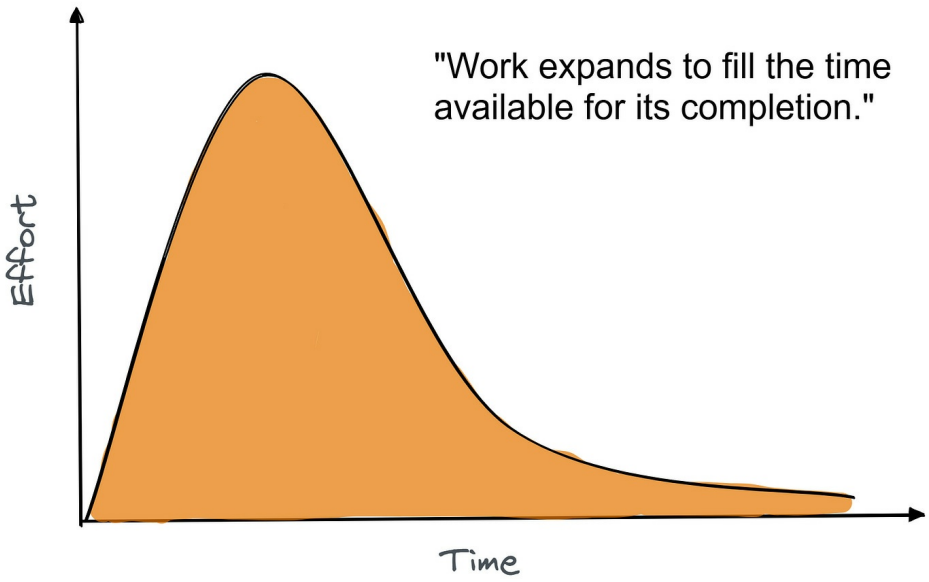
12. Parkinson's Law

Work expands to fill the time available.

If you give a developer two weeks to do a task that can be done in two days, it will take two weeks to finish. This does not mean the developer is lazy or puts things off. People tend to fill up the time they have. Over the two weeks, the developer will likely spend time making plans, trying things, and adding extra tasks that do not need to be done (gold-plating). But if there was a deadline to have this done in a day, it would probably be done on that day.

The thing about Parkinson’s Law is that it says if you give people a certain amount of time to do something, they will probably take all the time to do it. So, teams should set clear and realistic time limits (aka deadline-driven development). However, managers must use it judiciously, combining Parkinson’s insight with realistic scheduling. If you compress timelines too much, you risk running into Hofstadter’s Law, which reminds us that work often still takes longer than expected, even with buffers.

Examples. A developer given two months for a one-week task will spend a month prototyping alternatives, another week on architecture debates, and the last three weeks polishing details nobody asked for. If we give the same task, but this time with a clear one-week deadline, it will be shipped in one week.



Parkinson’s Law

4. How metrics distort work

13. Goodhart’s Law

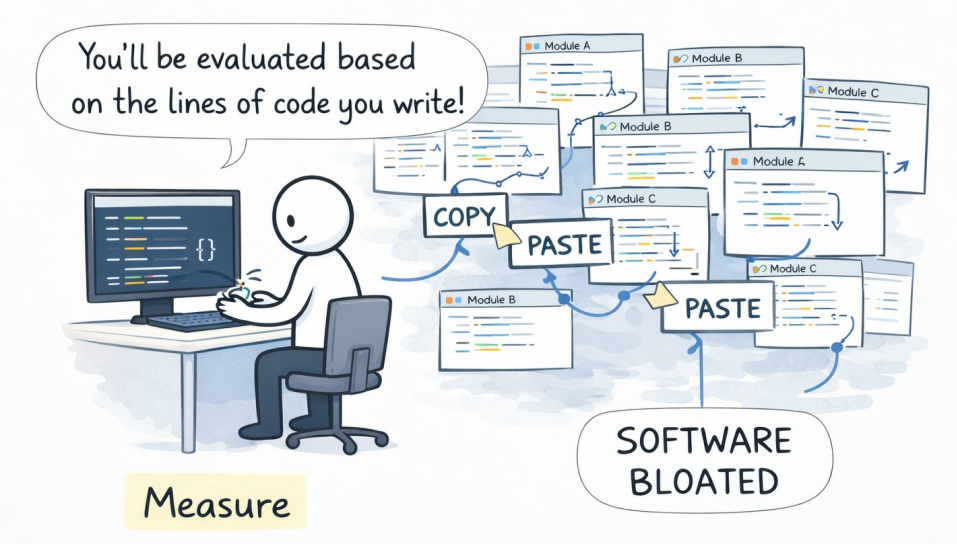
> **When a measure becomes a target, it stops being a good measure.**

We can use many different ways to measure our work, e.g., number of bugs closed, number of incidents, test coverage, or team velocity. When we start measuring people's performance based on these things, they will focus on making those numbers look good instead of actually doing good work.

The numbers will go up, but the work will not get any better. This is because when we give people incentives, they will do what gets them the reward, not what we really want. When we measure the wrong thing, people will do the wrong thing to get ahead.

Examples. I watched a team get rewarded for lines of code written at the start of 2000, and the number of PRs created some years later. Developers started copy-pasting instead of extracting shared logic. Some created PRs for almost every commit they made.

The modern version is AI tokens consumed per engineer (called tokenmaxxing). More tokens are being treated as a sign of productivity.



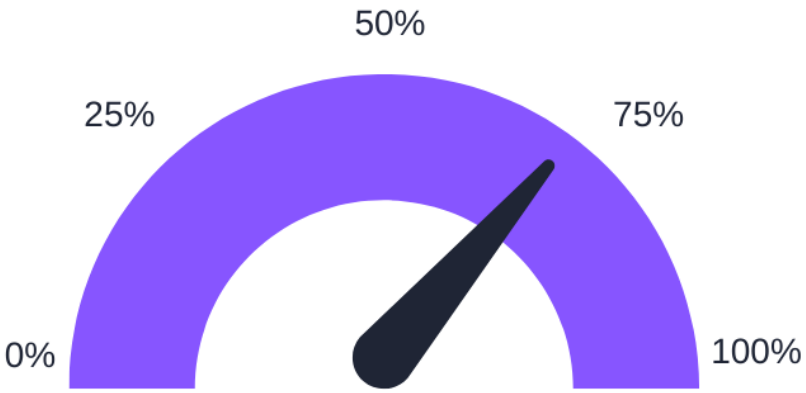
Goodhart's Law

14. Gilb's Law

> **Anything you need to quantify can be measured in some way that beats not measuring it at all.**

Gilb's Law is like the side of the coin to Goodhart's Law. You can say, when looking at Goodhart's Law, that having metrics is bad, but that is actually not true. Not having any metrics is even worse than that. If something is important to you, you should try to find a way to measure it, because we cannot improve what we don't measure (as Peter Drucker famously said).

Example. Developer productivity is usually a hard thing to measure, and it always has been. We had many bad metrics, from lines of code to token consumption. But deployment frequency and change lead time give you a signal (as in the DORA metrics for DevOps) as a proxy.



Gilb’s Law

5. What breaks under load

15. [Knuth’s Optimization Principle]

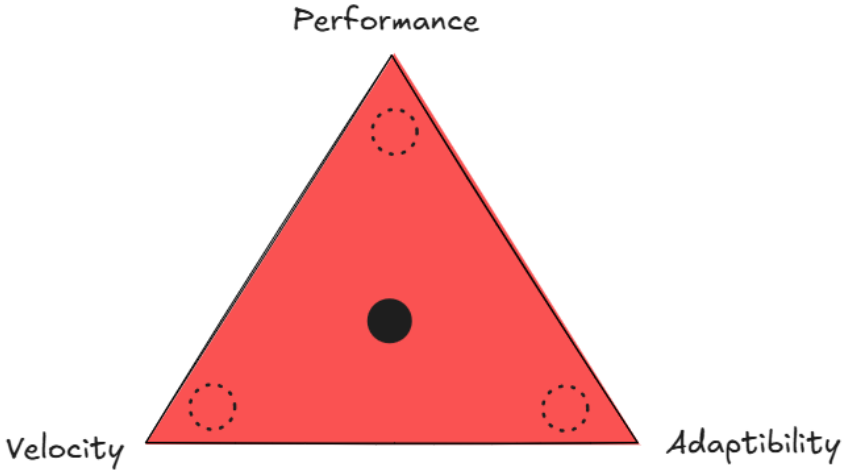
> **Premature optimization is the root of all evil.**

Most performance work happens too early and in the wrong place. Teams optimize code paths that never become hot, introduce complexity they never need, and burn time solving a scale problem they may never earn.

So the best way is to write the code that works, then check its performance. If there is a problem, a tool will show you where it is. If not, just move on.

Examples. I worked at a startup once, where we spent a lot of time setting up Kubernetes. The thing was that we did it to handle millions of users, and we didn’t even have 10 users yet. We were making our infrastructure ready for a load that didn’t exist. Our product features were not even finished.

One of my colleagues said that we should make sure 100 people even want our product before we worry about handling millions of users. He was right. We still launched late.



Premature Optimization

16. [Amdahl's Law]

The speedup from parallelism is limited by the sequential part.

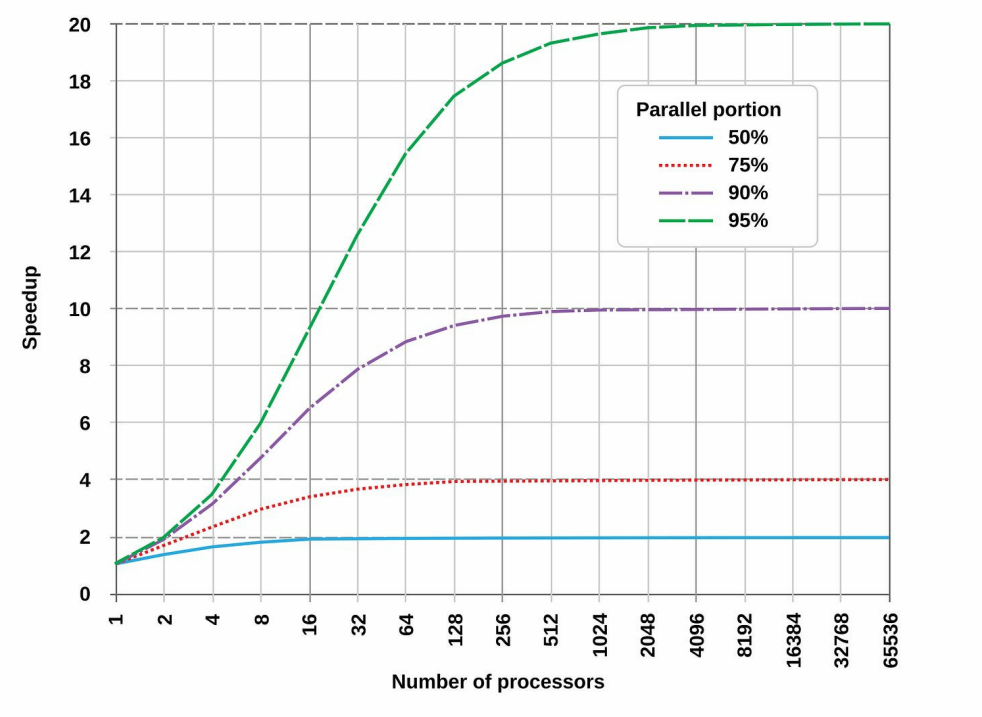
If 10% of your work has to be done in a sequential way, the work will only go 10x faster, no matter how many computers you use. If 50% of the work has to be done one thing at a time, the work will only go twice as fast.

The same thing happens with people. If one group of people has to say yes to every decision, about how something is built, that limits how fast your team can work, no matter how many engineers you have. If you add engineers, but they all have to wait for the same group of people to say yes, the line of people waiting just gets longer. Your team of engineers will still be slow because the group of people making decisions is a bottleneck. The work of your team of engineers will only go as fast as the

group of people making decisions.

Examples. Scaling web traffic by adding more app servers helps until every request hits one shared database or authentication service. Then adding more horizontal scaling doesn’t help.

The conversation about AI productivity is hitting the roof now. AI makes coding faster, but you still have to think, check, fix errors, and work together on those steps that can’t be done simultaneously. This sets the limit on how much you can gain in the end. That’s why some engineers see their work speed up by 10 times, and others see a 1.2 times increase.



Amdahl’s Law (Wikipedia)

17. Murphy’s Law

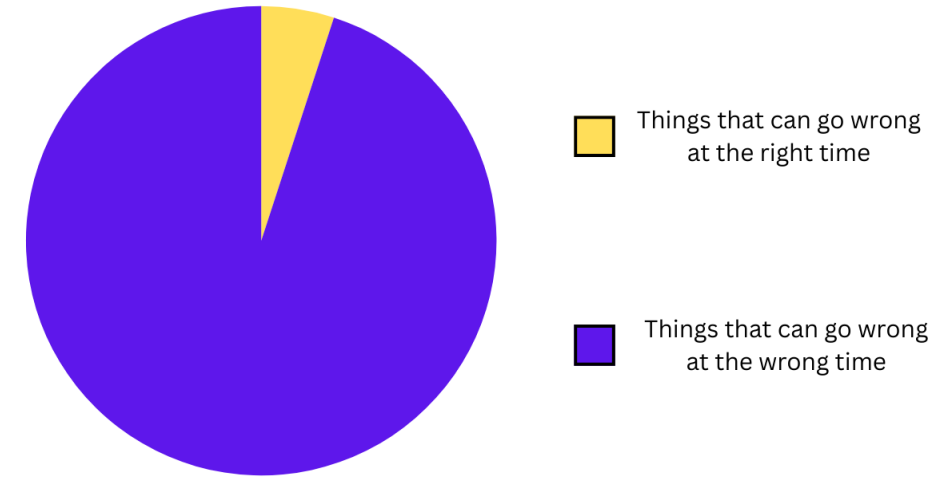
> **Anything that can go wrong will go wrong.**

In software, Murphy’s Law is often mentioned to explain bugs and

production incidents: whatever can go wrong in code (a null pointer, a race condition, a network outage) will eventually manifest, especially in large user bases or at the worst possible time (Friday evening).

In practice, this law encourages developers to write more defensive code. This means checking for nulls, handling exceptions, validating inputs, and failing gracefully when errors occur. It also reminds DevOps teams to anticipate failures by implementing monitoring, enabling rollbacks, and maintaining contingency plans.

Example. On July 19 2024, CrowdStrike made a change to the Falcon Sensor settings. This change caused a memory issue on Windows machines. It made 8.5 million Windows machines stop working and show a screen. To fix this problem, someone had to log in to each machine and apply the fix, because those machines could not start up. This could be done remotely. And this happened on a Friday morning when no IT staff members were working. It caused problems for airlines, hospitals, and banks. Everything that could go wrong did go wrong on the day, just like Murphy’s Law says.



Murphy's Law

18. Postel's Law

Be conservative in what you send, liberal in what you accept.

This law says that if your server sends HTTP responses, it should format headers exactly per spec. But if your server receives an HTTP request with an uncommon header order or an unusual format, you should still process it rather than drop the connection, as long as you can interpret it safely.

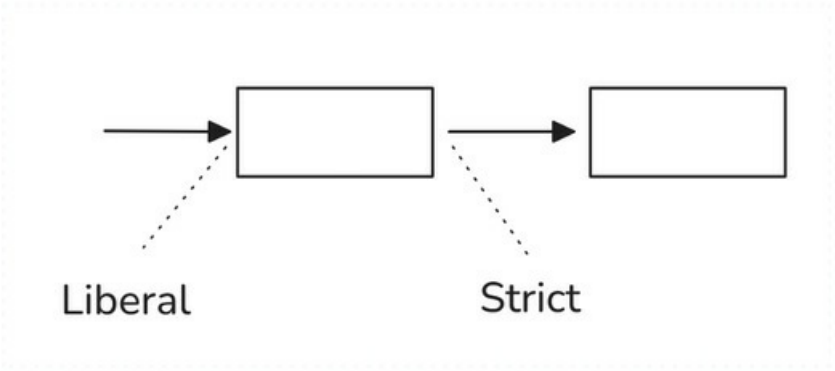
Browsers do this at a scale. Most of the HTML on the web is not written correctly, but modern browsers still render it. If they were strict, half the internet would not be found.

But there is one thing to consider. Being too liberal has a cost: if everyone accepts anything, problems will never be corrected. There will be just more mess.

In security-sensitive code, tolerating input can make it easier for attackers

to find. So, the basic idea still holds. You need to use judgment, as being lenient is not the same as being permissive.

Example. In APIs, say your service expects a timestamp. If it receives a timestamp without a time zone, instead of rejecting, maybe you assume UTC or try to parse it anyway, being liberal in acceptance. But when your service returns data, you always include the time zone to ensure the output is conservative and precise.



Postel's Law

6. How to judge better

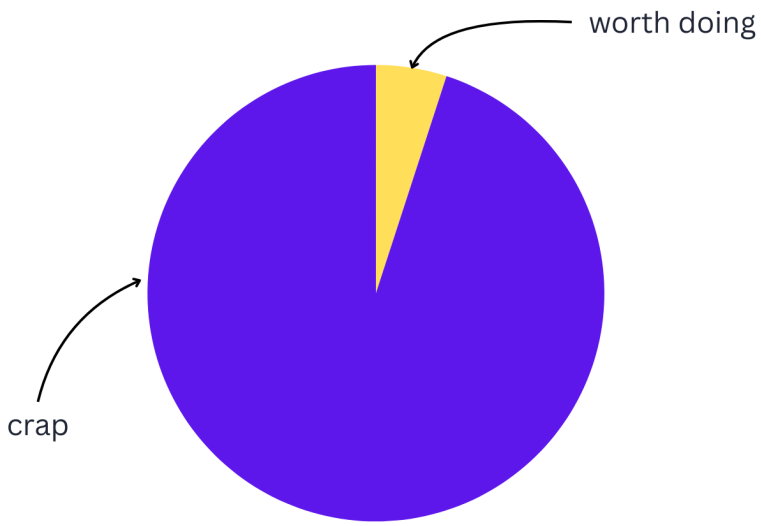
19. Sturgeon's Law

> **90% of everything is crap.**

Most things we make will go unused, and most of the code we write is not good. Most projects we start do not deliver the value that we thought they would. This is not a bad thing per se. This is how things are when we are trying to create something new. If we pretend everything is great, we will treat every project the same, which will make things too complicated.

The projects that really matter are the ones, like 10% of them. Finding these projects and getting rid of all the others is what really takes skill.

Example. WordPress has roughly 57,000 plugins in its directory. Over 34,000 haven't been updated in the past 2 years, and nearly 19% have zero active installs. A small number of well-maintained plugins powers 40%+ of the public web. That distribution is Sturgeon's Law in one screenshot.



Sturgeon's Law

20. Cunningham's Law

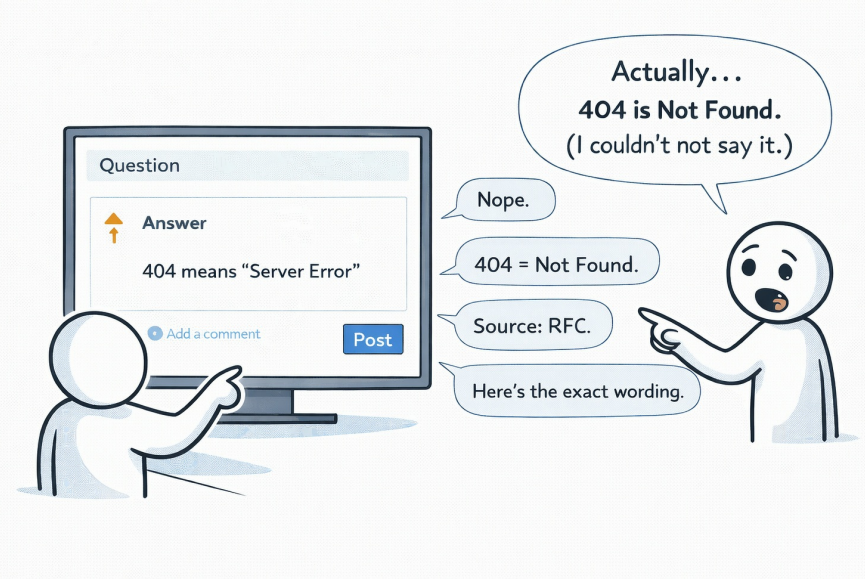
> **The fastest way to get the right answer online is to post the wrong one.**

When you ask a question on some online forum, you usually get no response. If you post something that is clearly incorrect, people will jump in to correct you. They might just walk by if they see a question, and then cannot help themselves when they see something that is wrong. You can

actually use this to your advantage. If you are having trouble with something, do not ask how you should do it. Instead, propose a solution you know is not very good, or share a draft, and then see what happens. The right answer might come to you without you even asking for it.

Note that this trick only works when the people around you know what they are talking about. If you are in a group where everyone's just as confused as you are, then a wrong answer can actually cause more harm than good. In that case, the wrong answer can just become information that people start to believe.

Example. The whole bet of wikis, and later Wikipedia, runs on this insight. People correct errors faster than they write articles from scratch. The bet paid off on a civilization scale.



Cunningham's Law

Conclusion

In this article, I shared some of the most impactful laws I saw in my career.

You do not have to memorize all of them. The top five or six laws will help you solve most of your issues. The rest are there for when a new problem arises.

What is more important is knowing when a law applies and when it does not. These twenty laws often conflict with each other. Knuth says do not optimize early. Amdahl says find and fix the part of your project that is slowing everything down. Both are correct at times. The key is to know which one to use now.

Also, this list is my list. Your list will be different. The laws that have caused you problems will be more important to you than the ones that have not. Over time, you will add your laws. Write them down when you notice them. One line per project, incident, or rewrite. Which law helped you? Which law gave you advice? What changed? Your personal list will be more helpful to you than any list I can give you.

Frameworks, platforms, and deployment models have changed since Brooks wrote his book in 1975. These laws have not changed. They describe the one thing that has not changed: **humans building things together under constraints they do not yet fully understand.**

That is why they are worth learning before the project, not after it causes problems.

📘 The Laws of Software Engineering book is out

The book began as a document I wrote over the years. During my 20+

year career in Tech, I saw the same things happening at companies with different technologies and teams. I wrote down what I saw. I learned about Galls Law from a project that did not work out. Brooks' Law was observed in a team that grew larger, but everything became slower. Goodhart's Law arose from a time when we met all our goals, yet the results were no better. They have been even worse.

Later, I met engineers who had figured out the same things. Most of them learned these lessons the hard way, as I did. They had a project that failed a team that got tired or a codebase that was a mess. This is how engineers usually learn these lessons because no one tells them. It is true, and it costs a lot.

This book is a list of what I learned.

This issue covers 20 software engineering laws. My book covers 56 laws across architecture, people, time, quality, scale, code, and decision-making.

Each chapter discusses what the law says, where it comes from, when it applies, and what it looks like in a project. Some chapters also include connecting ideas such as The Two-Pizza Rule, The Cobra Effect, and Impostor Syndrome.

This book is something you can keep at your desk and look at when you need help.

Forewords are written by **Dr. Rebecca Parsons**, CTO Emerita at

Thoughtworks, and **Addy Osmani**, Engineering Director at Google Cloud AI. Reviewed by 20 engineers and leaders from Google, Amazon, Uber, Oracle, Yelp, Nutanix, and CodeScene.

[Get the book]

Want to advertise in Tech World With Milan?

If your company is interested in reaching founders, executives, and decision-makers, you may want to **consider advertising with us**.

Love Tech World With Milan Newsletter? Tell your friends and get rewards.

Share it with your friends by using the button below to get benefits (my books and resources).

Share Tech World With Milan Newsletter

Track your referrals here.