Laws of Development

Teams

Conway's Law

Organizations design systems that mirror their own communication structure.

- Software architecture naturally follows the communication paths of the organization that created it.
- **Inverse Conway Maneuver**: Structuring teams to match the desired architecture.

Brooks's Law

Adding manpower to a late software project makes it later.

- New members require training and coordination, temporarily reducing overall productivity.
- Communication paths increase exponentially: $\frac{n(n-1)}{2}$. Adjust scope or schedule instead.

Dunbar's Number

The cognitive limit to the number of people with whom one can maintain stable social relationships is about 150.

- Above 150, formal hierarchies and processes become necessary as informal communication fails.
- **Two-Pizza Team**: Effective collaboration happens in small units of 5–10 people.

The Ringelmann Effect

Individual productivity decreases as group size increases.

- Known as "social loafing." Small teams tend to have higher per-person output.

Price's Law

Half of the work is done by the square root ($\sqrt{n}$) of the total number of participants.

- Dependence on a few high-performers intensifies as the organization grows.

Putt's Law

Technology is dominated by two types of people: those who understand what they do not manage and those who manage what they do not understand.

- Highlights the gap between technical expertise and management roles.

Peter Principle

In a hierarchy, every employee tends to rise to their level of incompetence.

- People promoted for being good at one role may be incompetent in a new role (e.g., management).
- Need for dual-ladder systems separating IC and management tracks.

Bus Factor

The minimum number of team members that can leave before a project stalls.

- A Bus Factor of 1 indicates a single point of failure. Increase it through knowledge sharing.

Dilbert Principle

Companies tend to systematically promote incompetent employees to management to limit their damage.

- A satirical observation where management is seen as a place to minimize practical impact.

Planning

Premature Optimization

Premature optimization is the root of all evil. (Donald Knuth)

- In 97% of cases, small efficiencies should be ignored.
- **Order**: Make it work → Make it right → Make it fast (if needed).

Parkinson's Law

Work expands so as to fill the time available for its completion.

- Importance of short, clear milestones. Agile sprints are a practical response to this.

The Ninety-Ninety Rule

The first 90% of the code accounts for 90% of the development time, and the remaining 10% accounts for the other 90%.

- The final 10% (edge cases, polishing, bug fixes) takes much longer than expected.

Hofstadter's Law

It always takes longer than you expect, even when you take into account Hofstadter's Law.

- Expresses the recursive difficulty of estimating software schedules.

Goodhart's Law

When a measure becomes a target, it ceases to be a good measure.

- Using code coverage as a KPI leads to the creation of meaningless tests.

Gilb's Law

Anything you need to quantify can be measured in some way that is superior to not measuring it at all.

- Even rough measurements are more useful than no measurement at all.

Architecture

Hyrum's Law

With a sufficient number of users of an API, all observable behaviors of your system will be depended on by somebody.

- Users rely on informal behaviors (timing, error formats) beyond the official spec.

Gall's Law

A complex system that works is invariably found to have evolved from a simple system that worked.

- Designing for complexity from the start often leads to failure. Basis for MVP approach.

The Law of Leaky Abstractions

All non-trivial abstractions, to some degree, are leaky.

- ORMs hide SQL, but performance issues eventually force you to look at the generated queries.

Tesler's Law

Every application has an inherent amount of complexity that cannot be removed; it can only be moved.

- The question is who handles the complexity: the user or the system.

CAP Theorem

A distributed system can only provide two of three guarantees: Consistency (C), Availability (A), and Partition tolerance (P).

- Real-world choice is usually between Consistency (CP) vs. Availability (AP).

Second-System Effect

The tendency of small, successful systems to be followed by bloated, over-engineered successors.

- Confidence from the first success leads to feature creep and over-generalization.

Fallacies of Distributed Computing

8 false assumptions made by beginners: Network is reliable, latency is zero, bandwidth is infinite, etc.

Law of Unintended Consequences

Changes in complex systems always have side effects in unexpected places.

Zawinski's Law

Every program attempts to expand until it can read mail. (Satire on feature bloat)

Quality

Boy Scout Rule

Always leave the code cleaner than you found it. (Robert C. Martin)

- Continuous, incremental refactoring prevents technical debt accumulation.

Murphy's Law

Anything that can go wrong will go wrong. (Basis for defensive programming)

Postel's Law (Robustness Principle)

Be conservative in what you do, be liberal in what you accept from others.

Broken Windows Theory

Neglecting bad design or low-quality code leads to further degradation of overall quality.

Technical Debt
Choosing a shortcut in code borrows time from the future. Interest (productivity loss) must be paid.

Linus's Law
Given enough eyeballs, all bugs are shallow. (Importance of code reviews and open source)

Kernighan's Law
Debugging is twice as hard as writing the code in the first place. Therefore, write simple code.

Testing Pyramid
Strategy of having many unit tests, fewer integration tests, and very few UI/E2E tests.

Pesticide Paradox
Repeating the same tests reduces their effectiveness over time. Continuously update test cases.

Lehman's Laws of Software Evolution
Software that reflects the real world must evolve and be continuously updated to remain useful.

Sturgeon's Law
Ninety percent of everything is crap. Maintain high standards and focus on the valuable 10%.

Scale

Amdahl's Law
Speedup from parallelization is limited by the portion of the task that must remain sequential.

Gustafson's Law
Significant speedup in parallel processing can be achieved by increasing the problem size.

Metcalfe's Law
The value of a network is proportional to the square of the number of users (n^2).

Design

DRY (Don't Repeat Yourself)
Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

KISS (Keep It Simple, Stupid)
Systems work best if they are kept simple. Simplicity reduces maintenance costs.

SOLID Principles
SRP (Single Responsibility), OCP (Open-Closed), LSP (Liskov Substitution), ISP (Interface Segregation), DIP (Dependency Inversion).

Law of Demeter
A module should not know about the innards of the objects it manipulates. (Principle of Least Knowledge)

Principle of Least Astonishment
Software should behave in a way that is least likely to surprise its users or other developers.

YAGNI (You Aren't Gonna Need It)
Do not add functionality until it is actually needed. Prevents over-engineering.

Decisions

Dunning-Kruger Effect
The tendency for people with low ability at a task to overestimate their ability.

Hanlon's Razor
Never attribute to malice that which is adequately explained by stupidity or carelessness.

Occam's Razor
The simplest explanation is usually the most accurate one.

Sunk Cost Fallacy
Continuing an endeavor as a result of previously invested resources (time/money).

The Map Is Not the Territory
Models (UML, diagrams) are abstractions and approximations of reality, not reality itself.

Confirmation Bias
The tendency to search for, interpret, and favor information that confirms one's beliefs.

Hype Cycle & Amara's Law
We overestimate the short-term impact of technology and underestimate its long-term impact.

The Lindy Effect
The future life expectancy of non-perishable things (like technology) is proportional to their current age.

First Principles Thinking
Breaking down complex problems into basic elements and rebuilding them from fundamental truths.

Inversion
Solving problems by thinking about them in reverse—identifying risks and failure modes first.

Pareto Principle (80/20 Rule)
80% of consequences come from 20% of causes. Focus on the high-impact 20%.

Cunningham's Law
The best way to get the right answer on the internet is not to ask a question; it's to post the wrong answer.

Developer Philosophy

Avoid Ground-up Rewrites
Rewriting from scratch should be the absolute last resort, even when technical debt is high.
- Catch complexity warning signs early (difficulty in fixing, poor documentation) and refactor.
- Always include an integration phase to reduce complexity after expansion.

The 90% Completion Trap
Getting the code to work is only half (50%) of the total job.
- The rest of the time goes to edge cases, testing, deployment, docs, and optimization.
- Use a schedule buffer to bridge the gap between "working" and "finished."

Automate Best Practices
Enforce guidelines with automated tools (Linters, CI) rather than just documenting them.
- "Build failure on rule violation" is more effective than verbal reminders.

Consider Pathological Data
Assume the worst: high latency, massive data sets, and bizarre strings, not just the "Golden Path."
- Handling extreme edge cases determines the final quality of the code.

Attitude of Simplicity
Most code can be written more simply.
- Make it work first, then take the time to refactor for clarity and conciseness.

Design for Testability
Minimize side effects and clean up interfaces to make automated testing easier.
- Untestable code is often a sign of poor encapsulation.

Obvious Safety
Code shouldn't just be "provably" correct; it should be "obviously safe" and robust against future changes.
- Ensure security and core logic remain solid even if calling patterns change.

Beware of Off-by-one Errors
Always watch for errors at boundary conditions (n vs $n-1$) in loops and index calculations.

Single Source of Truth for Knowledge
Centralize wikis and documentation to prevent fragmented information.
- Link documentation with source control and comments to keep it up to date.

Reproducible Build Environments

Keep build scripts simple (`cd project; ./build`) and shared across the entire team.

- Every developer should have easy access to the full build and test environment.

Friction for Behavioral Change

If you want people to follow a specific path, design the system so that the wrong path is "inconvenient."

- Systemic guidance is more powerful than memorization.

Laws of Growth

Adaptation through Progressive Overload

Growth is not just the accumulation of effort; it's the balance of systematic stimulus and recovery.

- Signal the body/mind to adapt by providing consistent and targeted stressors.

Avoiding the Dead Zone

Training at moderate intensity often fails to stimulate any system effectively.

- Clearly separate "low and slow" high-volume (foundation) from high-intensity intervals (limit-breaking).

The Compound Effect of Consistency

Showing up every day is simple but difficult, especially after the initial passion fades.

- Once consistency gains momentum, stopping actually requires more effort than continuing.

The Magic of Mileage (Volume Matters)

When facing a plateau, one of the most reliable solutions is simply to increase the "Volume" of input.

- Massive time investment leads to baseline shifts in performance and intuition.

Evaluate Performance in Context

Don't compare yourself to experts; evaluate your progress against cohorts at a similar stage.

- Everyone has their own path, and initial struggles are not a final judgment of potential.

Imperceptible Growth and Baseline Shifts

Growth is rarely felt day-to-day; it reveals itself through a gradual shift in what you consider "normal."

- Yesterday's "hard" becomes today's "normal" through a slow, non-linear process.

Beware of Over-measurement

Obsessing over metrics leads to anxiety and poor judgment.

- Once a strategy is chosen, focus on the process and use intermittent check-ins rather than daily tracking.

Master the Basics

High performance comes from executing fundamental principles with extreme precision and high intensity.

- In programming, this means deep understanding of infrastructure, performance, and complexity.

Desirable Difficulty

Growth occurs at the edge of uncertainty—doing tasks that are "just a bit too hard" to be fully confident.

- Too comfortable leads to stagnation; too much panic prevents learning. Aim for the threshold.

Connecting the Dots

Small, seemingly unrelated experiences from the past often connect in unexpected ways to create value later.

- Trust that every bit of learning and experience will become an asset in the future.

Focus and Margin

Focus Means 'Stopping'

The practical meaning of focus is not doing more, but stopping most things.

- Only by stopping the unimportant can you create the space and energy for high-impact work.

The 30% Improvement Rule

Stop trying to improve everything by 1%, and you will have the margin to improve the most important thing by 30%.

- Do not dilute your resources; focus on the single opportunity with the highest potential upside.

The Fig Tree Analogy

By trying to keep every option open, you risk deciding on nothing until every opportunity rots and falls.

- You cannot have everything; you must choose a few vital things and let the rest go.

'Fanatics' over Satisfied Customers

Instead of trying to satisfy everyone, focus on creating "fanatics" who will never leave and will spread the word.

- Saying no to the wrong customers creates margin to deeply serve your core audience.

Leverage Strengths over Fixing Weaknesses

Stop trying to fix every minor weakness and instead use that energy to leverage your unique strengths.

Autonomy over Control

Stop trying to control every detail. Create margin for your team to grow autonomously and for you to focus only on what you can do.

Disconnecting for Flow

Stop the habit of checking emails or social media every few minutes to secure margin for creative flow.

- Frequent context switching prevents deep thought and entry into a state of flow.

The North Star Metric

Stop chasing every metric; focus on the one single metric that can truly transform your organization.

Hell Yeah or No

When you have only one priority, decisions become simple. If it's not a "Hell Yeah!", it's a "No."

- A clear goal on the horizon, like a mountain peak, should be the standard for every choice.

Practical Advice

Fix the Gun

If the team repeatedly makes the same mistakes in a specific area, fix the "system" that causes those mistakes.

- Improving the system to prevent errors is far more effective than just pointing out individual slips.

Balance Quality and Speed

Ask yourself, "How okay is it to ship a bug right now?" and adjust your rigor based on risk.

- For non-critical web apps, it's often better to ship fast and fix quickly than to delay for perfection.

Sharpen the Saw

Mastering your tools is a core engineering skill.

- Editor shortcuts, shell commands, browser dev tools, and typing speed are almost always worth refining.

Identify Accidental Complexity

If you can't explain a difficulty simply, it's likely "accidental complexity."

- Solve the structural complexity first to make the underlying problem as trivial as possible.

Solve for the Root Cause (One Layer Deeper)

Don't just fix the surface symptoms (like adding a null check); go one layer deeper to resolve the root cause.

- Fixing the source leads to a cleaner system and prevents entire classes of future bugs.

The Value of History (Git History)

For elusive bugs, dig into the commit history.

- Use tools like `git bisect` to find exactly when and where a bug started; it's often more accurate than intuition.

Feedback from "Bad" Code (Perfect is the Enemy)

Perfect code provides no feedback. Quickly written, functional code—even if imperfect—provides learning and direction.

- Don't waste time on all best practices if it prevents you from validating core value early.

Make Debugging Easy

Invest time in making your software easy to debug.

- If setup and reproduction take more than 50% of your debugging time, improve your tools or environment.

The Art of Asking Questions

Don't spend hours stuck alone; ask a colleague who knows the system or the technology better.

- Unless the answer is trivial to find in minutes, asking questions increases the entire team's speed.

Deployment Cycle as the North Star

Building an environment for fast and frequent deployment is key to a project's success.

- Slow CI, flaky tests, and rigid processes should be treated as seriously as production outages.

Laws of Productivity

Knowing What to Build

Speed is useless if you are building the wrong thing.

- Understanding customer needs, constraints from other teams, and past failures is more important than pure velocity.

Doing Less

True productivity is not just finishing tasks quickly, but identifying and stopping unnecessary work.

- Eliminate high-effort, low-value "busy work" and streamline processes to focus on real impact.

Responsiveness of Tools

Latency in editors, Git, and build systems does more than waste time—it shatters a developer's focus.

- Recurring 1-second delays lead to massive losses due to context-switching costs.

Maintaining Collective Knowledge

A 10x developer is often someone who simply knows the codebase the best.

- Keep the Bus Factor above 1 and share knowledge to prevent high turnover in ownership.

Clear Boundaries and Documentation

Clean interfaces and high-quality documentation drastically reduce the time developers spend researching a system.

- One missing documentation page can waste thousands of hours across a large team.

Infrastructure as an Enabler

Infrastructure should be a helper, not a hurdle.

- Aim for designs that align closely with actual use cases to avoid the "our infrastructure doesn't support this" bottleneck.

Minimizing the Surface Area of Tech Debt

Reducing technical debt minimizes the "surface area" of code you must understand before making a change.

- Never leave debt-repayment projects half-finished; incomplete migrations often leave the system worse than before.

Maintaining Low Failure Rates

Failures in builds, tests, or deployments waste the time of both the individual and the entire team owning the system.

- Lowering failure rates is a direct path to increasing organizational productivity.

Productive Practices are Practical

If the environment hinders prototyping, it hinders the most productive problem-solving approach.

- Tools must be easy to use to encourage measurement and data-driven decision-making.

Protecting Mental RAM (Focus)

Meetings, interruptions, and unresolved questions occupy a developer's "Mental RAM," reducing focus.

- Forcing a team to think about too many projects at once is a recipe for low productivity.

The Value of Finishing

Building 50% of a feature results in 0% productivity, not 50%.

- A consistent pattern of shifting priorities before completion destroys a team's output.

Sharpen the Saw

When you need to cut down a tree quickly, start by sharpening the saw.

- Spending hours on tools and documentation can save the company thousands of hours later.

Critical Thinking in the AI Era

AI is an 'Intern'

Treat AI answers and code as "intern drafts" that require verification, not as absolute truths.

- Focus on "what is the evidence" rather than "who (AI) said it," and ensure a human takes final responsibility.

Define the Real Problem (What)

Clearly define the actual problem you need to solve before rushing to a solution.

- Don't get lost in surface-level requirements; use 5W1H to systematically check your goals.

Awareness of Context and Environment (Where)

Recognize that a solution that works well in one environment may cause side effects elsewhere.

- Anticipate where the solution will be applied (client, server, DB) and its potential blast radius.

Choose the Depth of Analysis (When)

Distinguish between situations requiring emergency patches and those requiring root cause analysis.

- Even under time pressure, maintain rigor by flagging points that must be revisited later.

5 Whys and Root Cause (Why)

Don't settle for surface symptoms; repeat "Why" until you reach the core of the problem.

- Constantly question if there are other possible causes to avoid confirmation bias.

Evidence-Based Thinking (How)

Prioritize evidence over claims and experiments/measurements over intuition.

- As AI outputs become more plausible, direct verification through logs, tests, and reproduction becomes critical.

Avoid the Plunging-in Bias

Guard against the bias of jumping straight into coding without sufficient data collection and problem definition.

- Allocate more time to gathering specific evidence that supports your conclusions.

Avoiding Groupthink

Intentionally bring in diverse perspectives to explore opposing views and other possibilities.

- Use "fresh eyes" to view problems objectively and validate the soundness of data and assumptions.

Premortem

Assume the project has failed and write down the reasons why to identify risks early.

- Effective for revealing hidden assumptions and side effects overlooked during the planning stage.

Maintaining Unique Human Strengths

Even if AI generates the drafts, it is the human's role to "solve the right problem, for the right reason, in the right way."

- Teams that maintain humble curiosity and critical thinking will consistently deliver better results in the AI era.