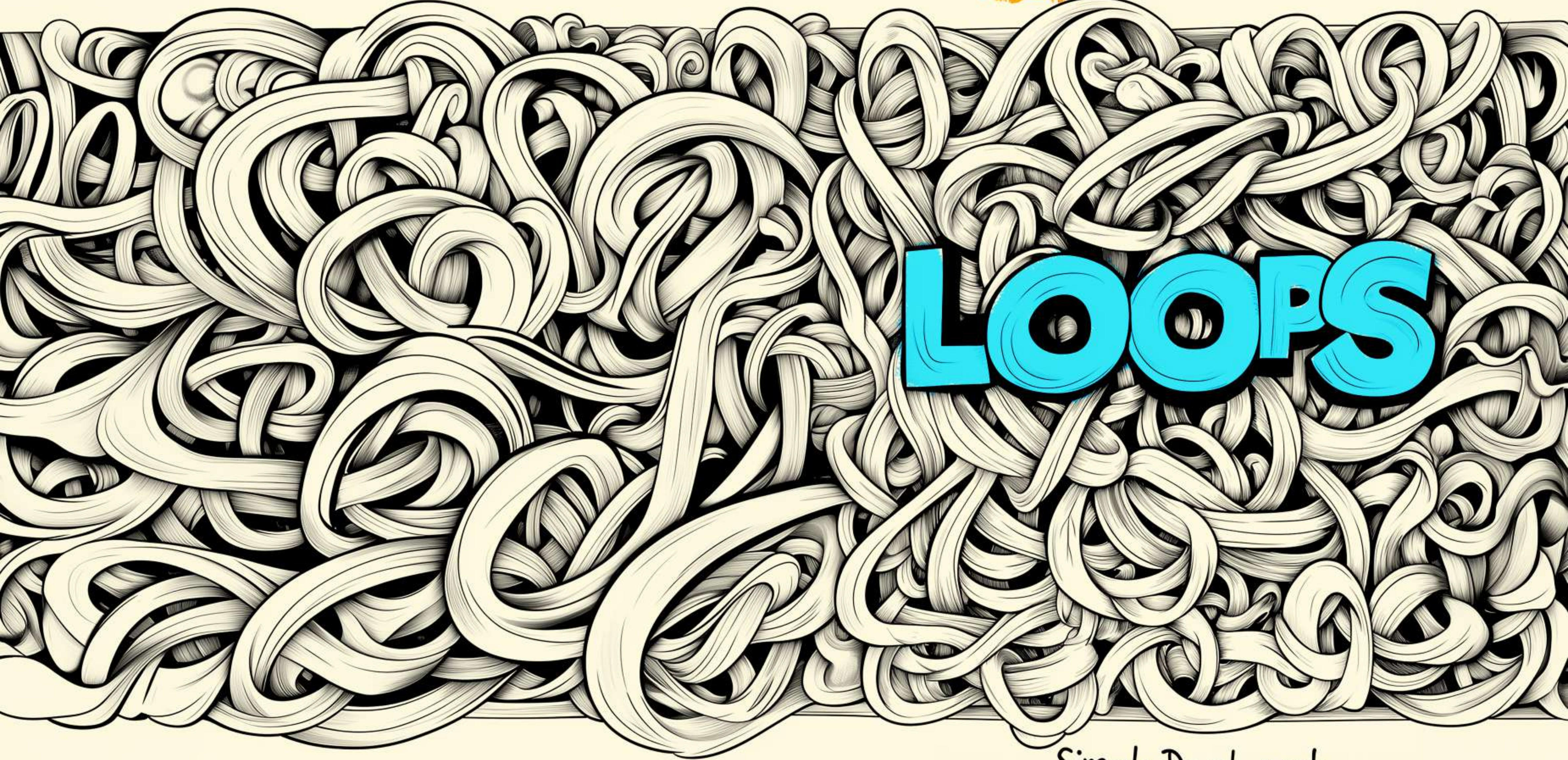
Clinton David Skakun



LOOPS

Simple Development Concepts to Managing Code Complexity

#CLINTON DAVID SKAKUN

###LOOPS

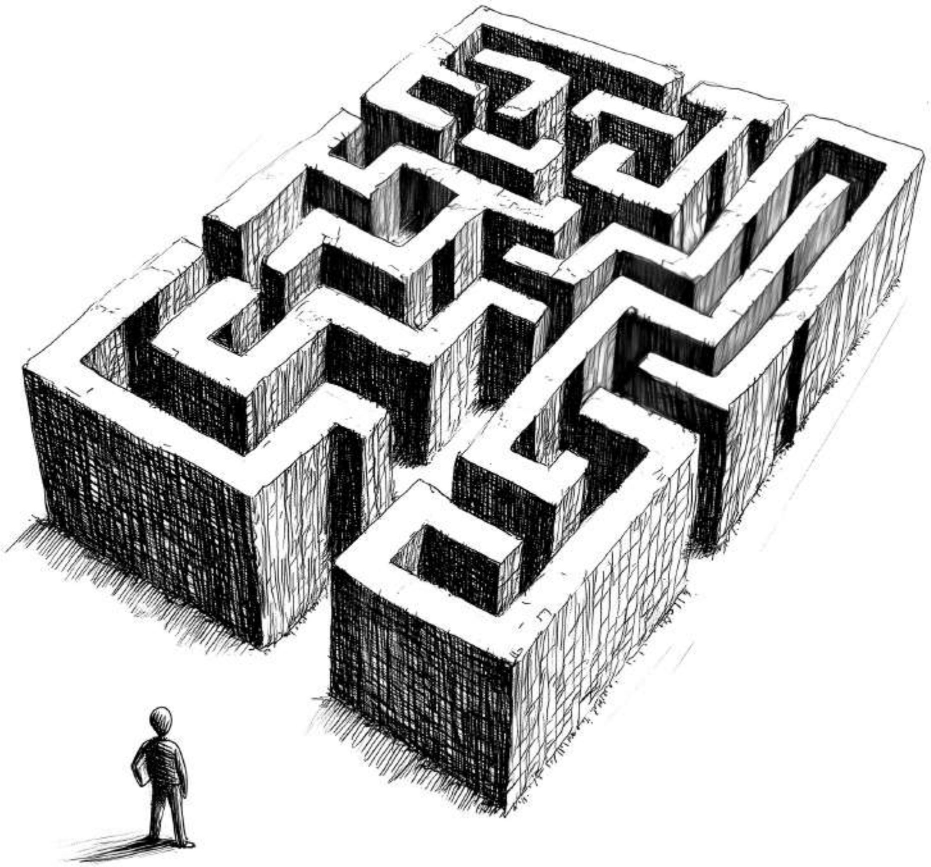
## SIMPLE DEVELOPMENT CONCEPTS TO MANAGING CODE COMPLEXITY

### Introduction

Welcome to LOOPS!, a collection of philosophies refined over two decades of coding across various languages and frameworks. This isn't a textbook of strict rules; it's a set of guidelines rooted in experience, focused on productivity, maintainability, and solving real-world problems. The core idea is simple: Software development is complex, and our job isn't to eliminate complexity but to manage it effectively.

We prioritize clarity over artificial simplicity, end-to-end type safety, and getting things done quickly, securely, and reliably. This book will challenge some popular beliefs, advocate for pragmatism over dogmatism, and help you build software that lasts. And more importantly, even though my ideas are opinionated, feel free to completely disagree. Why? What works great for me might just work completely the opposite for you. ###

###Chapter 1. Manage Complexity, Don't Eliminate It



"Code bases are always going to become complex once you start to change and add code."

The urge to simplify is often misguided in software development. The more we try to eliminate complexity the harder we make it. Instead of aiming for a mythical state of "simplicity," we need to accept and manage the inherent complexity that comes with building real-world applications. Trying to force simplicity in every corner often creates obscurity: black boxes that developers don't understand, and abstractions that leak, causing unpredictable behavior and headaches. We believe that code can be understood, changed, and maintained. To make that happen we need clarity.

"I PREFER TO MANAGE COMPLEXITY OVER MANAGING OBSCURITY"

TypeScript is a great example of this idea. TypeScript doesn't make our code less complex; it gives us the tools and the clarity to manage the complexity that comes with Javascript, as the code changes, instead of sweeping it under the rug. It's about creating a system where complexity is organized, predictable, and manageable rather than hidden and chaotic.

This means we prefer code that's clear and well-structured, even if it's not the simplest possible solution. Clarity is more valuable than false simplicity.

### Chapter 2. Single Source of Truth Modeling

At the very core of every application lies its data. These data models must be the ultimate source of truth, dictating data structure, type safety, and runtime validation. We should not rely on different places to define our data. This eliminates errors caused by having half-truths spread across the app. Consider the common scenario:

You have database models, input validation schemas, and form schemas, and if you make a change, you have to update each and every one. This isn't just tedious—it's a recipe for inconsistencies and bugs. This is the problem we are trying to solve with our source of truth.

"EMBED ALL DATA-RELATED CONCERNS DIRECTLY WITHIN THE MODEL DEFINITIONS"

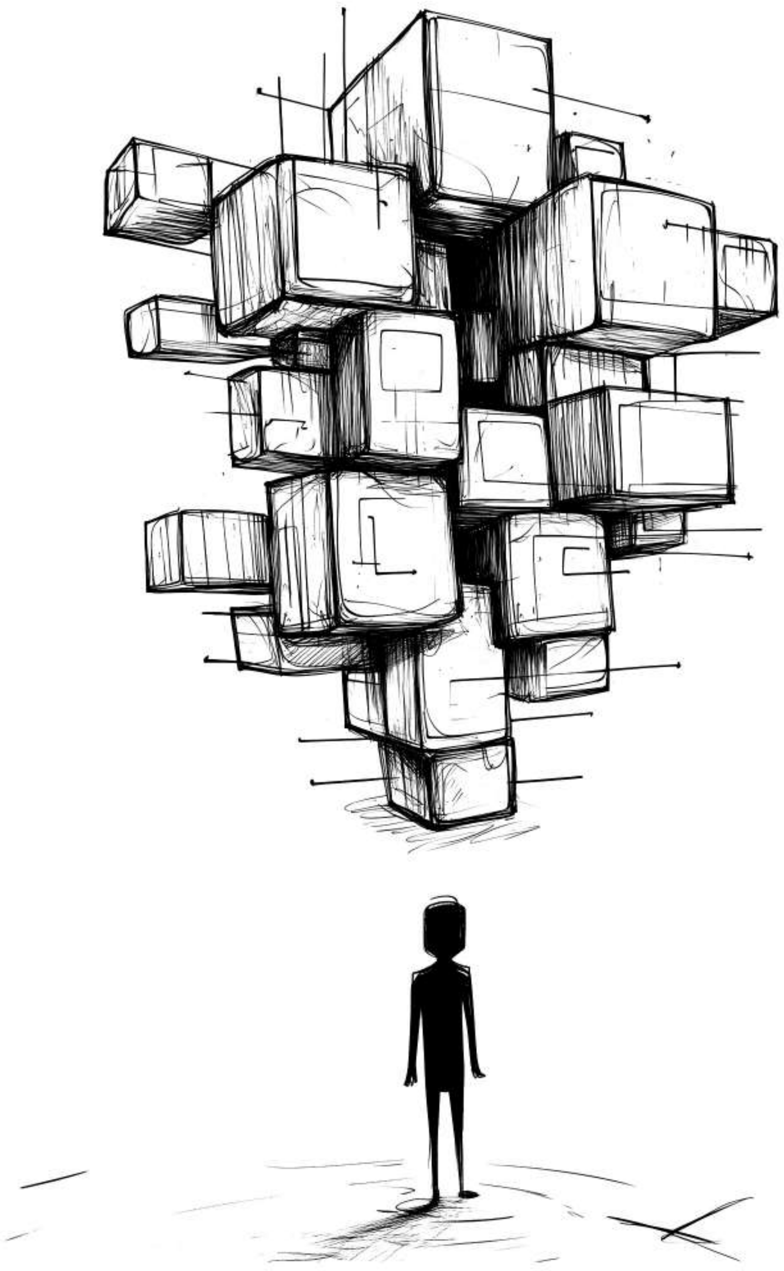
Our solution is simple: Embed all data-related concerns directly within the model definitions. Whether it's field lengths, data types, or validation rules, we should declare it once, and let all other layers use that source of truth.

This gives us a maintainable system where a single source of truth automatically generates our type safety and runtime validation. This means that our Zod schemas and Prisma schemas are in sync whenever a change is done.

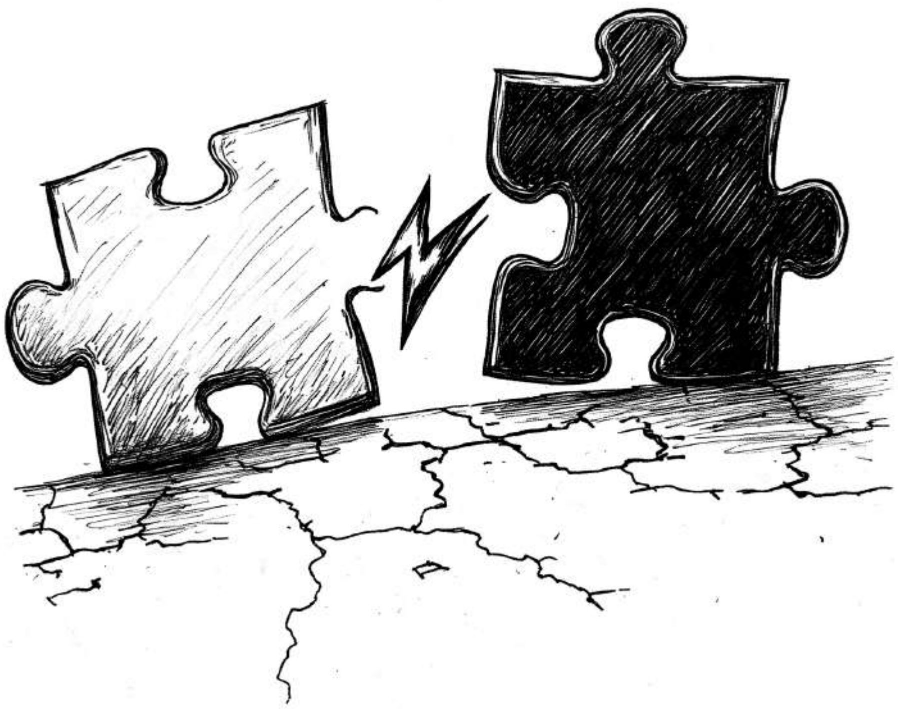
```
// Model definition
model User {
    /// @zod.string.min(5).max(90)
    username String
    /// @zod.string().min(8).regex( ... )
    password String
    created Date
}


// Zod schema derived from model
const usernameChangeForm = z.object({
    username: UserModelZod.shape.username
});
```

## SIMPLE DEVELOPMENT CONCEPTS TO MANAGING CODE COMPLEXITY



### Chapter 3. End-to-End Type Safety



Types are the safety rails of our code. Without them we are driving without a seatbelt. They keep our code consistent, help us prevent errors, and enhance our productivity. To unlock the true power of type safety, we need to ensure that type definitions don't get re-written in multiple locations. Instead, we should use the original type definition across different parts of the application. For example in our backend and frontend.

End-to-end type safety means that our types are always consistent, no matter where they're being used. This applies to different models, endpoints, or across different environments, or between our front and backends.

"THE IDEA IS TO HAVE A PUZZLE THAT BREAKS IF ONE RELATED PART BREAKS."

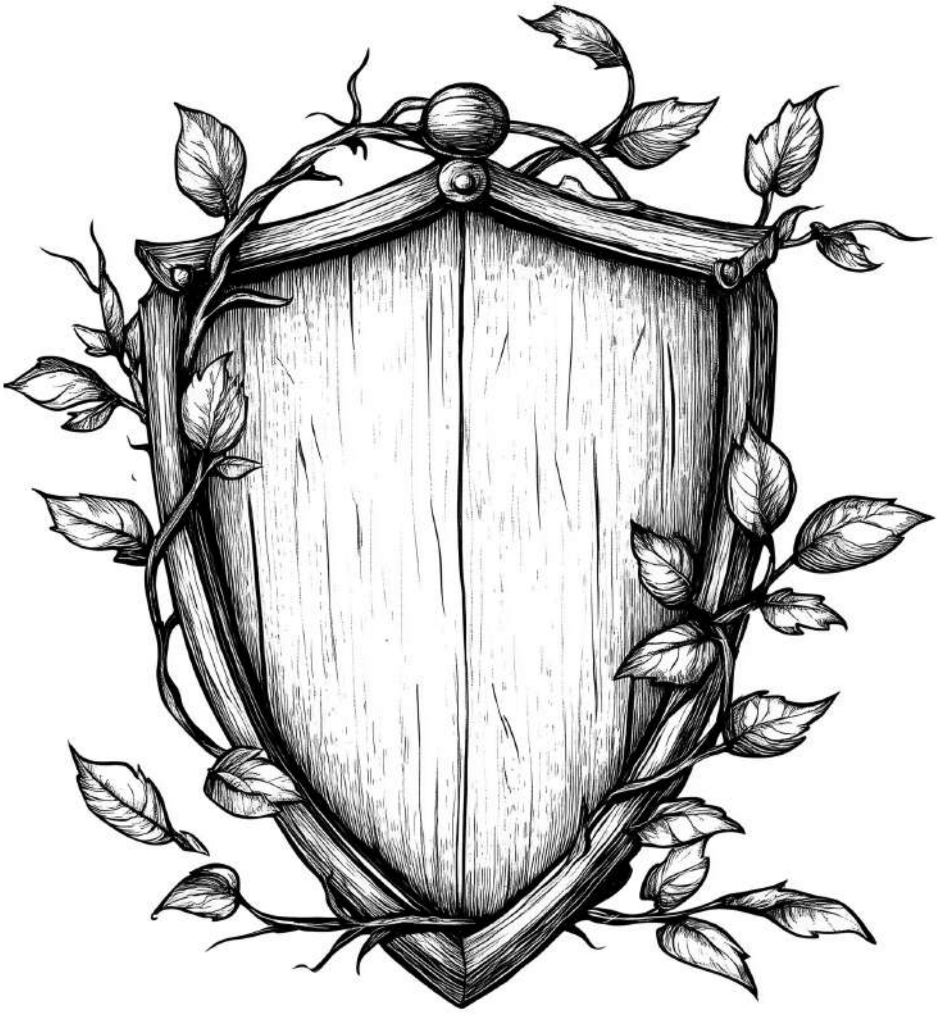
Think of it like a puzzle. Each piece must fit correctly for the whole puzzle to be complete. End-to-end type safety ensures that each piece (type) is consistent and fits its place. If any piece is wrong, the puzzle breaks, and we immediately know that there is an error. This approach greatly enhances the overall robustness and reliability of our systems.

```
// Backend endpoint return {
userMe: await db.users.get(session.id) // type
User
}

// Frontend
let { userMe } = $props(); // User is
automatically typed here

userMe.x = 1; // Error, x does not exist on
User console.log(userMe.username); // Works,
username exists on User
```

### Chapter 4. Security by Design



"For the most part security will be a default."

Security isn't an afterthought; it's a fundamental part of the software development process. We shouldn't rely on developers to always remember to apply security measures manually. Instead, we should design systems that are secure by default, where security becomes the status quo.

"BY MAKING SECURITY A DEFAULT, WE SHIFT OUR MINDSET FROM "REMEMBERING SECURITY" TO "DESIGNING SECURITY""

One way we do this is by using a default-deny approach. New routes and features are denied by default until permissions are given to them. This approach will help prevent developers from accidentally exposing things they shouldn't.

Security should be something that's woven into the fabric of our code. By making security a default, we shift our mindset from "remembering security" to "designing security". This approach not only enhances security, but it also reduces errors.

### Chapter 5. Test After Writing

"Write unit tests when it's working."

The popular practice of Test-Driven Development (TDD) often leads to unproductive workflows. Writing tests before writing the code assumes things about the code that you can't assume before you begin writing it. The result is you are often rewriting more tests than code. It leads to a rigid mindset and takes away from the creative process that programming is.

"MANUAL TESTING, WHILE WRITING THE CODE, IS NECESSARY TO SEE IF YOUR ASSUMPTIONS ARE CORRECT."

Instead, we believe in writing the code first, getting it working, and then writing the tests. Manual testing, while writing the code, is necessary to see if your assumptions are correct. Once the code works and meets requirements, tests are written. Tests are there for change management and to eliminate bugs that you didn't see during development.

We believe that our tests should be there for bug prevention and make sure that we don't break things when we change the code in the future.

### Chapter 6. Productivity vs. Popular Belief

"We just care about productivity and longevity of OUR code."

There is a wide range of popular industry practices like Agile, TDD, and Clean code, that are often presented as the only way. While these concepts can sound good on paper, they often fail to take the real-world context into account. We aren't a multi-billion dollar company with thousands of developers. We are trying to ship working software and focus on productivity and longevity.

"THE KEY IS TO BE PRAGMATIC AND ADOPT PRACTICES THAT ACTUALLY BENEFIT US INSTEAD OF BLINDLY FOLLOWING WHAT OTHERS DO."

We care about speed, security, and getting code to the customers quickly. This means we care more about results than we care about adhering to a strict set of rules that might not apply to our context. The key is to be pragmatic and adopt practices that actually benefit us instead of blindly following what others do.

### Chapter 7. Minimize Casting, Coercion, and Serialization

"Casting, forcing, etc. makes code harder to change and maintain."

One of the key goals of end-to-end type safety is to reduce the amount of casting, coercion, and serialization that's happening in our code. Each and every time we cast or coerce we are adding a layer of complexity that's not necessary. These extra steps make the code harder to debug, maintain, and change. It also has performance implications.

"WE WANT OUR CODE TO HAVE CLEAR AND SMOOTH DATA FLOW, NOT A MINEFIELD OF TYPE TRANSFORMATIONS."

Ideally, types should only morph when they pass through serializers and deserializers. Even better is to automate this process and make it transparent or minimize it completely.

This is important when dealing with APIs, databases, and third-party libraries. For example, JSON fields in Postgres can become a nightmare when we have to cast and coerce Dates, BigInts, etc. That's why serializers like devalue that handle this problem better are preferred. These types of serializers do cost some processing power, but they save you from doing extra work on the other side.

### Chapter 8. Less Magic is Better

"Whenever we can we prevent magical code."

Magical code hides details and often introduces hidden side effects. It forces us to learn a new framework on top of the existing ones. This makes it hard to understand the code, and as a result, it is harder to debug. This is why we prefer process over abstraction, and it's why we should be wary of "magic".

"Prefer clarity over magic, and understand the code instead of hiding away details."

Every time we use magical code, it is like another black box that hides details we should know about. Why create a complicated ORM on top of the existing ORM when we can just write the code? Why add metadata annotations when we can just write the code instead? In short: Prefer clarity over magic, and understand the code instead of hiding away details.

### Chapter 9. Falling in Love with Types

"any is a type that can be anything. It's a bad practice because it's hard to know what type it is and it can cause bugs."

The any type is like a loophole in type safety. It can be anything. This makes it hard to know what type a variable actually is. If you're using any, you're not using types. We should never use any unless we have no other option. This will ensure we are relying on our types and catching bugs at compile time.

"INSTEAD OF ANY, CONSIDER USING UNKNOWN."

This type is also very flexible, but it forces you to perform type checks before using it. Unknown will remind you to validate the type before doing anything with it. This makes it safer than any in most situations. It's the perfect type when you really don't know what type it is.

### Chapter 10. Validate When in Doubt

"Static typing doesn't mean anything if you don't have confidence in the types."

Static types are great for catching errors at compile time. But, what happens if the type is not what you think it is? We should validate types when we are not sure what they are. In some cases we should assume our types are wrong to prevent unexpected behavior. We shouldn't trust types coming from external sources.

"We shouldn't trust types coming from external sources. "

These types include user input, API responses, JSON, XML, YAML, etc parsing. All of these types require a parse function that takes a string and returns a value of the type. The act of parsing, or decoding, means we should assume we don't have the correct type and validate before proceeding. We need to validate the data to make sure it is correct and that all data matches the types we expect. This improves overall safety of the code.

### Chapter 11. Creative Testing

"Go to crazy lengths to test things that are mission critical."

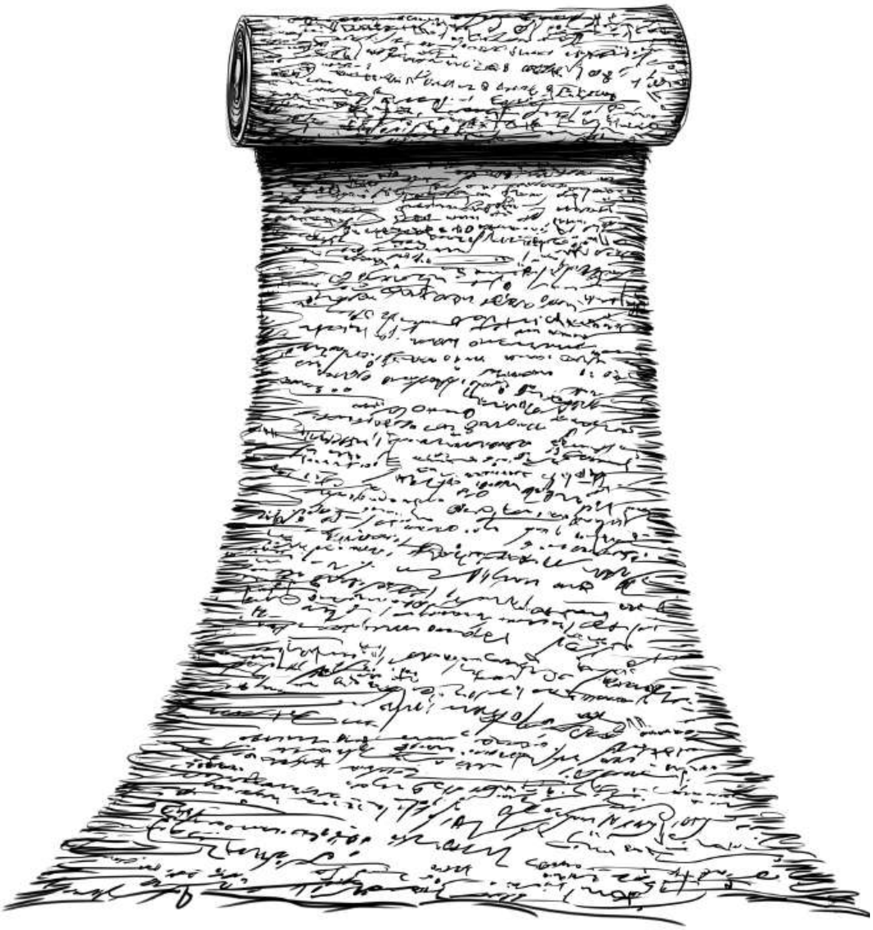
When it comes to mission-critical features, the approach to testing should be extensive and creative. It's not enough to just test the happy path. We need to be creative and explore every possible scenario to avoid unexpected bugs. This includes scenarios that you may not think of at first.

"Background processes should be tested under every situation."

For example, if we are working with a webhook from Stripe, we should not just test it manually once and hope that it works in production. You should try things like multiple retries, invalid data, etc. Similarly, background processes should be tested under every situation.

We should not be thinking about different types of tests (unit, integration, etc.) and focus on creating good tests that provide coverage. They are all just different forms of "tests".

### Chapter 12. Don't Use console.log



"Use console.debug instead of console.log."

Console.log is very vague and should not be used in production. console.debug is much more specific and the better choice when you want to output information to the console. When you're using console.debug you're saying you're debugging.

"Console.log has no clear meaning and it doesn't provide a lot of value."

The only place where console.log should exist is in access logs. Every other place in the code should use console.debug if you want to use console logging.

In general, we should ask ourselves if we're debugging, tracing, or outputting as much info as possible to run up our cloud bill. The answer isn't always the same.

### Chapter 14. Use the Right Tool for the Job

"All they care about is if you can get the job done, on time to make money and give the customer what they promised them 6 months ago."

All languages are close enough to the metal. Modern computers are very powerful, so you don't need to worry too much about what to choose. You should be choosing what you are most comfortable with. The goal is to focus on getting things done and making money.

"THE MOST IMPORTANT THING IS TO FOCUS ON WRITING GOOD CODE AND SOLVING PROBLEMS, AND NOT CARING ABOUT THE LANGUAGE."

No one cares if you're a "real programmer" or a "JavaScript developer", what they care about is if you can get the job done on time. And the customer just wants the page to load. They don't care about Rust or Ruby.

Yes, you need to think about things like memory management, but most companies won't ever run into memory problems. Types matter, but some companies hate types and don't want to use them. Some companies can lose millions of dollars from a bug, and others can lose $0.

The most important thing is to focus on writing good code and solving problems, and not caring about the language. Don't let ego dictate what you use. Choose the right tool for the job and be creative in finding solutions to problems.

### Chapter 15. Know When to Optimize

"Focus on perception, setting expectations and experience... not benchmarks."

With modern internet speeds, a lot of optimizations are usually not necessary. Many companies rely on caching with Cloudflare and other techniques to make old websites load fast. So, how important is optimization?

"Optimization comes down to perception and user expectation. "

If a user expects something to happen right away and it takes 1 second, that feels slow. If something is expected to take 3 weeks and it only takes 3 days, that feels very fast. This means most of your "speed" optimization is pointless. for of vs forEach on a list of 100 entries will make zero difference. Even if one is 100x faster.

However, if you can optimize a process that takes one hour to 10 seconds, that's huge. It means you are providing real value to the user. If you can show the most interesting video clips on instagram while they're waiting, maybe they'll want it to take longer.

### Chapter 16. Code Matters!

"Removing a column from a database shouldn't be a 3 day ordeal. Changing a type shouldn't take a week."

Despite everything I've just said, code matters. Not the language, but the code itself. Thinking about the code, making decisions about it, and structuring it properly is essential for the long term growth of our application.

"If you're struggling with code it is probably a signal to take a second look."

Removing a column from a database should never take 3 days. Changing a type shouldn't take a week. Creating new features should be more about getting it right instead of working around bad system design. You should choose the right technologies, and hire people that know how to deal with it. Then you need to decide how much to dictate vs delegate. It takes dedication and work. But it's all worth it in the long run.

There it is!

Everything I've learned along the way. I hope you found it useful.

This book is focused on my development team at Dedupely. Instead of endless meetings or adding it to a how-to guide, hidden away in a wiki somewhere, I decided to make it fun and easy to read. I hope you enjoy it as much as I did writing it.

A lot of passion went into writing these few pages in this tiny book. However I'm not going to pretend this is the end-all of software development. Not even close. Hopefully we can re-write this book together in another ten years with your perspectives and experiences.

### Clinton David Skakun

Clinton is a software developer, entrepreneur, and the founder of Dedupely.

He has over 20 years of experience in building and launching software products.

Loops
Written by Clinton David Skakun