SAMPLE

Building Tomorrow's Blockchain Developers

The Invariant's Keeper in the Machine Era

Aiman JoeHanz

Free sample: Introduction & Chapter 1

putrabooks.github.io

Buy the full book:

Carousell carousell.sg/p/building-tomorrow-s-blockchain-developers-ebook-pdf-1471383414

Ko-fi

Payhip

Contents

Introduction

Chapter 1 — The Promise in the Code

Chapter 2 — The Territories

Chapter 3 — The Path

Chapter 4 — The Working Day

Chapter 5 — The Invariant

Chapter 6 — The Burnout

Chapter 7 — The Conversation

Chapter 8 — The Instruments

Chapter 9 — The Year Whole

Chapter 10 — The Long Path

Introduction

Ask what a blockchain developer does and you will hear one of two answers, neither of which is true.

The first answer comes from the enthusiast. A blockchain developer, on this telling, is building the replacement for banks, governments and lawyers, one line of code at a time. The technology abolishes trust, or requires it, or distributes it; the sentences vary and the excitement does not. The second answer comes from the sceptic. A blockchain developer is running a spreadsheet that costs too much electricity, or writing the plumbing for a casino, or inventing problems to sell a solution nobody asked for. Both answers are about the technology's reputation. Neither is about the work.

The work is narrower and more interesting. A blockchain developer maintains a system whose entire value depends on a property it must never lose. The property differs from chain to chain, but its character does not: it is the thing the system promises to do no matter what any participant tries. It might be that no one can spend the same coin twice. It might be that an entry, once accepted, cannot be quietly rewritten. It might be that a computation produces the answer the code specifies rather than the answer somebody with enough access would prefer. Strip away the slogans and this is what remains: a guarantee, held in software, against people who have reason to break it.

I call that guarantee the invariant, and I call the person who holds it the keeper. Both words come from engineering rather than from marketing, which is one reason they are worth using. An invariant is a condition that must remain true across every execution, every upgrade, every strange input that arrives at three in the morning. It is not a feature. It is not a roadmap item. If it breaks, the product has not degraded; it has ceased to be the thing it claimed to be. A developer who understands this is doing a trade. A developer who does not is writing code near a promise somebody else made.

The trade has three commitments, and they fit in a sentence. Hold the invariant, because a guarantee that fails once has failed retroactively and no amount of later correctness repairs the history that was already written. Listen to the adversary, because a system like this is defined not by what a well-behaved user can do but by what a determined and well-resourced opponent cannot. And build only what the participants will actually run, because a chain nobody validates is a database with extra steps and a committee of people pretending otherwise.

Those three are harder than they sound, and the difficulty is not primarily mathematical. The mathematics is settled enough that most working developers rely on audited components rather than proving things themselves. The difficulty is that the three commitments pull against each other in practice. Holding the invariant perfectly usually means refusing the feature the users want. Listening to the adversary means delaying the launch that the money needs. Building what participants will run means accepting that the elegant design with no users is worth less than the clumsy one with real ones. Every hard decision in this trade is some version of that triangle, and the developer who has not noticed the triangle is making one of its three corners into a religion.

This book concerns the people who hold that triangle together. It describes where the work happens, how people get into it, what the day actually looks like, and what the craft consists of when you stop describing tools and start describing judgment. It spends a chapter on exhaustion, because this trade is unusually good at producing it and unusually bad at admitting it. It spends a chapter on conversation, because most of the trade's real decisions are made in review threads, incident calls and arguments with people who have a financial interest in a particular answer. It describes the instruments, which are duller than the conferences suggest and more decisive than the conferences admit. It follows one engineer through a full upgrade year, invented and clearly labelled as such. And it follows the trade's long arc, which is short enough to see and long enough to matter.

What this book is not is a tutorial. It will not teach you to write a contract, and it will not make you employable. There are excellent free materials for both, and they will be more current than anything printed. What a tutorial usually omits is the reason any of it is hard: not the syntax, but the judgment about what must never be allowed to happen, and what you are willing to refuse in order to prevent it. That judgment is the trade, and it can be described even when it cannot be installed.

One more thing belongs in an introduction. This is a young trade with an unusually loud reputation, and the gap between its reputation and its daily reality is wide enough to mislead anyone choosing whether to enter it. The people doing the work are mostly not at the conferences. They are reading diffs, arguing about gas costs, writing incident reports, and explaining to a product manager for the fourth time why the thing that would make the demo beautiful would also make the guarantee false. If that sounds like work you

would find satisfying, this is the trade for you. If it sounds like the opposite of the excitement you were promised, better to learn that now.

A Note on the Two Vocabularies

This trade speaks in two registers that are difficult to reconcile, and a reader who notices the difficulty is already thinking well. The first is the register of engineering: state transitions, invariants, threat models, failure modes, blast radius. It is precise, unemotional, and it assumes that the speaker's concern is being understood rather than being accepted. The second is the register of the movement around the technology: sovereignty, empowerment, permissionlessness, the language of liberation and of institutions being superseded.

Both registers are used by the same people, often in the same hour, and the mismatch is not hypocrisy. It reflects a genuine double claim: that the work is ordinary engineering, and that the work matters historically. The claim can be true in both parts. Steel construction was ordinary engineering and it changed the shape of cities. The difficulty is that the two registers answer different questions badly, and a practitioner who uses the movement's vocabulary to settle an engineering question will make a mistake, usually in public.

This book uses the engineering register, with two deliberate exceptions. The introduction and the final chapter speak partly in the other one, because the question of whether the work is worth doing is not an engineering question and cannot be settled in that vocabulary. Everything between those chapters is flat on purpose. If the flatness reads as scepticism, that is a reasonable reading, and it is the correct register for describing a discipline whose entire value consists in not being persuaded by enthusiasm.

Chapter 1 — The Promise in the Code

Two Stories About the Same Work

The decentralisation zealot and the token huckster look like opposites and are cousins. The zealot believes the mechanism settles questions that used to require institutions, and therefore that any objection to a decentralised design is either ignorance or bad faith. The huckster believes the mechanism is a story that makes a token valuable, and therefore that the technical detail matters only insofar as it survives a superficial read. One is sincere and one is not, and both have arrived at the same place: the technology's reputation has become more important than the property the technology is supposed to hold.

The zealot does real damage because the zealot is often technically competent. A developer who has internalised the ideology will resist the boring safeguard, because the safeguard implies that the mechanism alone is insufficient. Adding an admin key, a pause function, a rate limit, an upgrade path: each of these is a local admission that the guarantee needs human judgment somewhere, and the zealot reads that admission as betrayal rather than as engineering. The result is systems that are philosophically pure and operationally fragile, and their failures are spectacular precisely because nobody was allowed to plan for them.

The huckster does a different kind of damage, and does it faster. A developer working for a huckster learns to optimise for the audit summary, the documentation that photographs well, and the launch date. The invariant becomes a slide. Nothing teaches the trade's judgment in that environment, because the environment rewards the appearance of the judgment and punishes the cost of having it. People leave those projects with impressive titles and no craft, and some of them go on to found projects of the same shape.

Between the two stands the actual trade, which is unglamorous and specific. It consists of people who can answer a question like this: if an attacker with a hundred million dollars, a year of patience and the ability to bribe one of your node operators decides to break your promise, what happens, in what order, and what will you notice first? Anyone who can answer that question carefully is doing the work. Anyone who cannot, however loudly they speak about decentralisation, is not.

What the Invariant Is

An invariant is a statement about a system that must remain true at every

observable moment. In ordinary software, invariants are matters of quality: an account balance should not go negative, a queue should not lose a message, an identifier should be unique. They matter, they are tested, and when they are violated the team fixes the bug and moves on. In a system whose promise is the invariant itself, the failure is different in kind. The bug may be fixed. The history it produced cannot be.

That asymmetry is the whole reason this trade exists as a separate discipline. If an ordinary application double-credits an account, the operator reverses the entry, the customer is made whole, and the ledger is corrected by an authority both parties accept. If a chain's rule is violated, there is no authority above the chain to correct it. Either the community agrees on a repair, which is a political act and sometimes an impossible one, or the violation stands as part of the record forever. The developer is therefore building in a medium where the ordinary escape hatch of "we will fix it in production" has been removed.

The invariant is also, and this is the part outsiders miss most often, a promise about who cannot do things. A payment network that can reliably transfer money between two willing parties is a fine product and not this trade. The distinction arrives when you ask who is not able to stop it. If the answer is a compliance team, a foreign ministry or the company's own administrator, then the system is a very good database with a governance structure, and that is a legitimate thing to build. It is simply a different promise, and conflating the two is where a great deal of confusion enters, both in marketing and in engineering.

Precision about the promise pays immediately. A team that has written down, in one paragraph, exactly what its system guarantees and to whom, will argue less about features and will discover the dangerous requests earlier. A team that has not written it down will discover the promise only when a customer asks for something that would violate it, usually under time pressure, usually in a contract negotiation. The paragraph is not bureaucracy. It is the artifact that makes every later decision cheaper.

What the Invariant Is Not

The invariant is not the same as immutability, and treating the two as synonyms causes real design errors. Immutability is one way of securing a guarantee: if nothing can be changed, then nothing can be changed maliciously. It is a strong tool and it is expensive, because it also prevents correction of genuine mistakes, and human institutions have spent centuries

building correction into their systems for good reasons. A developer who cannot distinguish the guarantee from one implementation of it will keep choosing immutability in places where a narrower, cheaper, more reversible mechanism would protect the actual promise.

Nor is the invariant the same as decentralisation. Decentralisation is a property of who participates and how many independent parties would have to collude. It is often the best available way to make a promise credible between strangers. But a small consortium of banks can hold a strong invariant among themselves with modest decentralisation, and a large public network can hold a weak one if the participants are honest but the code is not. Counting nodes is easier than measuring the promise, which is why so much public discussion counts nodes.

Finally, the invariant is not a moral claim. There is nothing inherently liberating about a hash function, and nothing inherently sinister about a permissioned system. The technology is a set of techniques for making certain guarantees cheaper to verify, and those techniques can serve a remittance corridor for migrant workers, a settlement layer for securities, a supply chain that documents provenance, or a scheme to separate strangers from their savings. The trade has no immunity to being used badly. Its only defence is that the people doing the work can tell the difference and choose, which requires them to have thought about it rather than absorbed a slogan.

The First Commitment

Hold the invariant. This sounds like a slogan until you notice what it requires in practice, which is a willingness to lose arguments that you would otherwise win. The product manager wants the feature that shortens the flow by one step, and the feature happens to make a transfer conditionally reversible by the operator. The sales team has promised a client that entries can be removed on request. The founder would like the ability to freeze a wallet in an emergency. Each request is reasonable in isolation. Each one, granted, changes what the system guarantees, and the change is invisible to everyone who reads the marketing rather than the code.

Holding the invariant therefore has a social cost that the trade underestimates when it recruits. You will be the person who says no, repeatedly, to people who are more senior and more commercially motivated, about a distinction they cannot see and which you may struggle to explain in a meeting. The developers who survive this are not the ones with the strongest opinions. They

are the ones who can make the distinction legible: here is the property, here is the request, here is which of the two we are choosing, and here is who will be harmed if we choose wrongly. That is a communication skill as much as an engineering one, and it is the reason this book spends a chapter on conversation.

There is a harder version of the same commitment, which is the willingness to stop. Occasionally the honest answer is that the system cannot be made to hold the promise under the constraints on offer, and the correct move is to say so and accept the consequences. That answer is rare, unpopular, and the mark of a practitioner rather than a participant. A developer who has never once concluded that a design should not ship has probably not been paying attention to the designs.

The Second Commitment

Listen to the adversary. The phrase is not an invitation to paranoia; it is a statement about what the specification actually is. In most software, the specification describes what the system does for a cooperating user. In this trade, the specification is best read as a description of what the system refuses to a hostile one. A design review that only walks through the happy path has not reviewed the design. It has reviewed the demo.

Listening means building a habit of asking who benefits from a failure, and how they would arrange it. It means reading your own code as an attacker with the same information you have, then again as an attacker with information you do not have, such as the state of a pool at a particular moment or the contents of a pending transaction. It means taking seriously the failure classes that the ecosystem has already paid for: reentrancy, oracle manipulation, key compromise, governance capture, bridge assumptions, and the quieter class where nothing is technically broken and the incentives simply point somewhere awful.

The commitment also implies a specific posture toward audits, which is where many teams get this wrong. An audit is an adversary hired for a fortnight. It is valuable and it is bounded: the auditors see a snapshot, they work to a budget, and their report describes what they found rather than what exists. A team that treats the audit as the adversary has finished working has misunderstood the exercise. The adversary is continuous, and the team is the only party present for all of it.

The Third Commitment

Build what the participants will actually run. This is the clause that separates engineers from theorists, and it is the one most often violated by people who are very good at the other two. A design can hold its invariant perfectly under a set of assumptions that nobody will satisfy in practice. If validating requires hardware no hobbyist owns, the assumption of many independent validators quietly becomes an assumption of a few well-funded ones. If the honest behaviour depends on participants watching a dashboard, the promise holds only as long as the dashboard is interesting.

The test is empirical and slightly brutal: look at what is running, not at what is specified. How many independent operators, with what hardware, under what economic pressure, and what happens to the guarantee when the least enthusiastic of them stops. A chain with a beautiful design and three operators in one jurisdiction has a different promise from the one on its website. Saying that out loud is part of the trade, and it is unwelcome in rooms where the funding depends on the other story.

The clause has a second meaning that is closer to craft. Participants run what they can afford and what they can understand. A protocol that is elegant and undocumented will be run in one configuration, badly. Documentation, sensible defaults, tooling that makes the safe path easy, and client diversity are not marketing overhead; they are how the invariant survives contact with operators who are not the people who wrote it. The engineering that makes a guarantee real at scale is usually unglamorous infrastructure work, and it is where a surprising amount of the trade's genuine expertise lives.

The Triangle and Its Temptations

The three commitments fight each other, and understanding how they fight is most of the judgment this trade requires. Holding the invariant perfectly pushes toward refusing features, which reduces adoption, which weakens the third commitment because fewer participants run the thing. Listening to the adversary pushes toward conservatism and delay, which can lose the window in which a system becomes used enough to matter. Building what people will run pushes toward convenience, which is exactly how a promise gets quietly softened.

The temptations follow the corners. A team that has made the invariant into an idol will ship something flawless that four people use, and will call the emptiness a failure of the market. A team that has made adoption into an idol will add the operator override, then the emergency pause, then the curated

validator set, and will discover one day that it is operating a permissioned database while its documentation still uses language written three years earlier. Neither team is dishonest. Both have stopped holding the triangle.

The workable position is not a compromise in the middle. It is a sequence: decide what must never happen, then design the smallest system that prevents it, then make that system as easy to run as possible without touching the first decision. Written in a sentence it sounds obvious. Held across a year, under commercial pressure, with a team whose members disagree about which corner matters most, it is the actual job.

Why the Word Decentralised Does So Much Work

The word carries at least four meanings, and most arguments about this technology are really arguments between people using different ones. It can mean the absence of a controlling operator, a property of the deployment. It can mean geographic or organisational spread among participants, a property of the set of people involved. It can mean that no small group can change the rules, a property of governance. And it can mean that users have an exit, a property of the market around the system.

These are not the same claim, they do not have the same strength, and they are routinely used interchangeably. A system with fifty validators in six countries is decentralised in the second sense and may be entirely centralised in the third if those validators share a funding source or a client implementation. A system with exit is decentralised in the fourth sense even if a single company operates everything, provided the users can genuinely leave with their assets. Precision about which sense is meant converts a shouting match into a specification question, and specification questions can be answered.

The reason the ambiguity persists is that the word does emotional work. It signals that the system is on the side of the user against the institution, and a builder who wants that signal will use the word broadly. A developer who wants to be useful should be the person who asks which sense, in writing, whenever the word appears in a proposal. That question is unpopular in marketing meetings and decisive in design reviews, and a practitioner's reputation for asking it is usually established in the first month.

The Reader Who Is Deciding

This book assumes three kinds of reader, and they need different things from the chapters that follow. The first is someone considering the trade: a student,

a career changer, an engineer in an adjacent field wondering whether the reputation matches the work. For them the useful chapters are the path and the day, because those two describe the life rather than the discipline, and the decision they are making is about the life.

The second is someone already in the trade who suspects that their environment has stopped teaching them. For them the useful chapters are the craft and the instruments, because those describe what competence looks like when it is not being modelled by the people around them. Many practitioners are the only person in their organisation doing this work, and a written description of expert practice substitutes for a colleague they do not have.

The third is someone from outside the trade who must make decisions involving it: a manager, an investor, a supervisor, a journalist, a procurement officer. For these readers the useful chapters are the territories and the conversation, because those describe who does what and where the arguments actually happen. This reader does not need to write a contract. They need to be able to tell whether the person across the table is giving them an answer or a slogan.

A book that tried serving all three audiences on every page would satisfy none of them well, so the chapters are arranged for the first reader and cross-referenced for the others. The introduction and this chapter can stand alone. The path and day chapters can be read without the craft chapter. The craft and instruments chapters assume the definition has been accepted and do not re-argue it. A reader who jumps straight to the section they need will miss no argument they require, which is the utmost a book of this kind can promise.

There is a fourth reader this book is written against, and that one is worth naming. It is the person who wants the trade to be simpler than it is: a list of tools, a certification, a credential that converts a career changer into a practitioner in twelve weeks. That reader exists because the trade's compensation and reputation attract it, and it will find this book frustrating. The frustration is deliberate. A trade that can be entered without first learning to keep a promise is not the trade described here, and treating it as such produces practitioners whose first serious incident usually arrives as their last.

Three Tests Anyone Can Apply

A reader who deals with this trade without practising it needs a way to tell whether a project is holding a promise or performing one, and three questions

do most of the work. They are usable in a meeting by someone who cannot read the code, and the answers are informative even when the respondent is trying to be careful.

The first question is what the system guarantees and what it does not. A team that has an answer, in one or two sentences, that does not contain the word decentralised is a team that has thought about it. A team that answers with a list of properties, each of which is technically true and jointly meaningless, has not. The follow-up matters more than the first answer: ask what the guarantee would look like if it were violated, and listen for whether the answer is specific.

The second question is who can change the rules, and how. Every system can be changed by someone, whether through an upgrade mechanism, a governance vote, a multisignature, or a coordinated social process among operators. A team that can name the mechanism, the threshold and the delay is a team that has confronted the question. A team that says no one can change it is either describing an unmaintainable system or misunderstanding their own deployment, and both are worth knowing before you depend on them.

The third question is what has gone wrong before. A team with a history has incident reports, near-misses and changes made in response. A team that claims no significant issues is either very young or not recording, and the second is more common. The quality of the answer tells you more about the team's culture than any architecture diagram, because describing a failure accurately requires the honesty the whole trade depends on.

These tests do not establish that a system is safe. Nothing does. They establish whether the people operating it are doing the work, which is the only variable an outsider can actually assess and the one that determines most outcomes.

The Cost of Getting It Wrong

The abstract case for the invariant becomes concrete when you count what a violation actually costs, and the counting is more varied than the headlines suggest. Four kinds of cost recur, and a practitioner should be able to name which one a given failure produced.

The first is direct loss, which is the kind that gets reported. Funds removed from a contract, value extracted through a manipulated price, assets taken through a compromised key. The magnitude varies enormously and the

reporting is biased toward the largest, which distorts the trade's sense of its own failure rate. Most losses are small, and the aggregate is larger than the incidents suggest.

The second is loss of confidence, which is harder to measure and often larger. A user who loses once does not return, and a market that has been burned requires years to trust again. The second-order cost lands on systems that did nothing wrong, which is why the trade's practitioners take other people's failures personally even when they are not involved.

The third is the cost of the repair, which is frequently underestimated. Restoring a system after a serious violation can require a coordinated migration, a legal process, a reimbursement and months of a team's attention that was budgeted for building. Some projects never recover the momentum, not because the loss was fatal but because the repair consumed the capacity to continue.

The fourth is the cost carried by the people, which appears in no ledger. An engineer who shipped the vulnerable function, a founder who overruled a warning, a team that watched a loss happen in public: these are the trade's casualties, and the way a project handles them determines whether it keeps its experienced people. A trade that counts only the first three will keep making the same mistake with the same people.

The Keeper and the Trade

What the keeper holds is not a coin, a protocol or a company. It is the truth of a promise. That is why the trade's failures are so much louder than its successes: a working guarantee produces nothing to look at, and a broken one produces a headline, a lawsuit and a lesson that the next team relearns anyway.

The reader who is considering this trade should take one thing from this chapter. The interesting question is never whether the technology is overhyped, which is a debate for other people. The interesting question is whether you want to spend your working life on the specific and repetitive discipline of deciding what must never happen, and then being the person who insists on it. If yes, the rest of this book describes the life that follows from that answer. If no, that is useful too, and cheaper to learn here than in an incident review at four in the morning.

ABOUT THE AUTHOR

Aiman JoeHanz is the author of the Building Tomorrow's People series. Blockchain development drew him because the trade's promise is an invariant: a rule that holds because the code cannot be talked out of it. This volume examines what that invariant costs, from the smart contract audit to the upgrade key, the bridge that gets drained, the token that was never a security until a regulator said so, and the decentralised system that is, in practice, maintained by four people with a multisig.