SAMPLE

Building the Smart Contracts of Tomorrow

A Practical Guide to the Technology Reshaping Agreements, Automation, and Trust

Aiman JoeHanz

Free sample: Introduction & Chapter 1

putrabooks.github.io

Buy the full book:

Carousell carousell.sg/p/building-the-smart-contracts-of-tomorrow-ebook-pdf-1465869375

ElevenReader

Ko-fi

Payhip

Contents

Introduction

Chapter 1: The Current State of Smart Contracts

Chapter 2: Limitations — the Failure Modes of Today's Contracts

Chapter 3: Common Mistakes in Smart Contract Projects

Chapter 4: Foundational Principles of Smart Contract Design

Chapter 5: Building Smart Contract Systems — the Step-by-Step Guide

Chapter 6: Implementation Strategies for Production Smart Contract Systems

Chapter 8: Case Studies: Real-World Smart Contract Deployments

Chapter 9: Evaluating and Testing Smart Contract Systems

Chapter 10: The Future of Smart Contracts

Connect

About this Series

This free sample contains the Introduction and Chapter 1. The complete book contains the full contents above.

Introduction

An agreement used to be a promise with witnesses. Then it was a document with signatures. Then, for the last fifty years, it has mostly been a database row inside somebody else's computer — a balance at a bank, a subscription at a platform, a ticket at an airline — and every one of those rows is guarded the same way: by an institution that promises to keep it true, a legal system that promises to un-make it if the institution lies, and the residual, permanent hope that both promises hold at the same hour.

The smart contract is the newest answer to the oldest question in commerce: what if the agreement enforced itself? A smart contract is a program, deployed to a public network, whose code — not a company, not a court, not a clerk — decides what the agreement does next. It holds balances, it checks conditions, it moves value, it executes remedies, and it does all of it exactly as written, at machine speed, for any caller with an address, with no off switch that any single party controls. When it is written well, it is the most reliable counterparty a stranger has ever had. When it is written badly, it is a vault whose combination is published in advance — and the decade's record contains both kinds at industrial scale.

This volume is the series' treatment of that machine. It is written to the same standard as its siblings: no vendor names, no statistics that expire, no forecasts dressed as facts — only the structure of the thing, drawn plainly enough that the reader can reason about any contract they will ever meet, on any network, in any season of the decade ahead.

The ten chapters run in a deliberate order. Chapter 1 draws the landscape: the virtual machines that run contracts, the languages people write them in, the standards they inherit, and the layers built on top of them. Chapter 2 catalogs the machine's honest limits — the failure modes that every deployment lives with, from the famous reentrancy bug to the irreversibility that turns every bug into a potential catastrophe. Chapter 3 turns the record's tuition into a catalog of recurring project mistakes, priced so the reader never pays them at retail. Chapter 4 assembles the design principles that the surviving contracts share, from checks-effects-interactions to the discipline of writing for the adversarial reader.

Chapter 5 walks the build itself, step by step, from specification through phased deployment. Chapter 6 covers the operation of contracts in production — upgrade governance, key management, incident response, and the pause authority that mature teams install before they need it. Chapter 7 goes under the hood: gas metering, storage layout, proxy patterns, formal verification,

and the plumbing that connects code to reality. Chapter 8 reads the record through composite case studies — launches done responsibly, governance in practice, and the canonical exploits with the pattern lesson each one encoded. Chapter 9 builds the verification stack: unit and invariant tests, fuzzing, audits, and the monitoring that catches what everything else missed.

Chapter 10 maps the decade ahead — account abstraction, better languages, cross-chain execution, the institutional arc — and closes the volume the way the series always closes: with the reader holding the instrument, not a prediction.

One warning before the first chapter, printed early because it matters most here: a smart contract is not a judge, not a lawyer, and not a source of trust. It is a machine that does exactly what it says, to everyone, forever, whether or not that is what anyone meant. The entire discipline of this subject — and the whole of this book — is the practice of making sure that "what it says" and "what everyone meant" are the same text, verified before the machine is born, because after the birth, the machine answers to no one. That is the subject's whole gravity, its whole difficulty, and — when the discipline holds — its whole value: the agreement, at last, that keeps itself.

Chapter 1: The Current State of Smart Contracts

Strip every marketing layer away and a smart contract is this: a program whose execution and storage are maintained by a network of mutually distrusting computers, so that no single operator — not its author, not its callers, not the computers themselves — can alter its behavior once it is running. Every word in that sentence carries weight, and the chapter opens by unpacking each one, because the decade's confusion about this machine is mostly confusion about the sentence. "A program" — plain, testable logic, written once, born with a public address. "Maintained by a network" — not hosted by a vendor but replicated across thousands of independent machines that each verify every step.

"Mutually distrusting" — the defining condition: the computers do not trust each other, which is exactly why everyone else can trust the result. "Cannot alter its behavior" — the property that separates the contract from every database and every API that came before it: the guarantee is not a promise about future conduct but a fact about present code.

Why it took a decade to arrive at this definition is the subject's first honest history lesson. The phrase itself predates the technology by two decades — coined by a computer scientist in the 1990s to describe automated agreements in general, from vending machines upward — and the decade between the first practical networks and the present was spent discovering, mostly by collision, which parts of the original dream the engineering could actually deliver. What survived the collisions is narrower and stronger than the dream: not "contracts that understand law," and not "code that argues about intent," but programs that hold value and enforce their own state transitions, running on networks where the ledger's history cannot be rewritten. This volume is about the machine that survived.

The machines that run contracts — the virtual machines — cluster into families, and the landscape has a familiar shape: one dominant family and a periphery of challengers, the same shape operating systems took, the shape databases took, the shape browsers took. The dominant family descends from the first general-purpose contract network: a stack-based virtual machine with a simple instruction set, a gas-metered execution model, and a storage model of 32-byte words — an architecture so austere it became a standard by accident. Its descendants and imitators dominate because network effects compound: the tooling, the auditors, the libraries, the search results, the stack-exchange answers, and the graduates all concentrate where the

bytecode is best understood, and every year of concentration widens the moat.

The challengers matter and are worth naming structurally. One family compiles broader languages — general-purpose languages with familiar syntaxes — into machine code executed by a WebAssembly-class runtime, betting that onboarding friction, not expressiveness, is the bottleneck. Another family optimizes for raw throughput with parallel execution, betting that contracts are a performance problem before they are a trust problem. A third bets on provable computation — the machine checking its own execution with cryptographic receipts — hoping to remove "trust the network" from the sentence entirely. None of these bets is foolish; several are winning their niches; and the reader should note the structural lesson before the chapter details any of them: in platform markets, the dominant family's moat is never its instruction set, which anyone can copy — it is everything the instruction set attracts, which no one can copy.

What the families share, across every schism, is more consequential than what divides them. Every family runs deterministic computation: same input, same state, same result, on every machine, every time — the property that makes thousands of mutually distrusting computers able to agree without a referee. Every family meters execution with some fee mechanism, because unmetered computation on a shared network is a denial-of-service invitation. Every family stores state publicly and charges for it, because public verifiability is the product. And every family faces the same wall at the edge of its world: the network cannot see anything outside itself — no prices, no weather, no court rulings, no delivery confirmations — without some mechanism for importing the fact, and every such mechanism reintroduces, in miniature, the trust the network exists to remove. That wall, and the machinery built against it, returns in chapters 2, 7, and 9; it is the subject's deepest structural fact.

Contract languages are a small, strange genre, shaped by three pressures no ordinary language faces. The first pressure is adversarial execution: the program runs in a world where every input may be hostile, every caller may be an attacker, and any bug may cost real, unrecoverable value — so the language must be small enough to reason about completely, and its dangerous corners must be few and well-lit. The second pressure is cost transparency: every operation has a metered price, so the language must make costs visible — a language that hides the meter teaches developers to write programs nobody can afford to run. The third pressure is immutability: deployed code cannot be patched in place, so the language's compile-time guarantees are not conveniences but the only pre-publication review the code will ever get.

The dominant language of the dominant family descends from two older languages — one bracketed and C-like, one Python-like — and reads, deliberately, like a small scripting language with a storage layer: contract declarations, state variables with visibility annotations, functions with modifiers, events for logging, and an inheritance mechanism that carries a fair share of the ecosystem's footguns in its constructor ordering. Its critics are correct about everything they say — it permits patterns it should prohibit, its integer types default to unsafe, its historical string and array handling was expensive — and its critics are also correct about the deeper point: none of that has displaced it, because the moat is not the language but everything attached to it, and because the language has hardened under fire.

Successive compiler generations now refuse to compile what the older ones accepted silently: explicitness about function mutability, checked arithmetic by default, restrictions on self-destruct, warnings on uninitialized storage pointers. A language that cannot be fixed in place, it turns out, can still be fixed at the compiler, generation by generation — the ecosystem's most underrated act of collective self-defense.

The newer languages are built from the crash record. One descends from a formally specified intermediate representation and was designed so that common bug classes — reentrancy, unbounded loops, arithmetic overflow — are hard to express or provably absent; one compiles a general-purpose language under a capability-discipline model; one embeds the domain in a proof assistant so the contract is written alongside its theorem. The structural lesson for the reader: language choice in this genre is risk posture, not taste. The dominant language buys the ecosystem's tooling and auditors at the price of the crash record; the designed-for-safety languages buy provable corners at the price of the smaller ecosystem; and a decade of post-mortems says both prices are real and neither is wrong — what is wrong is choosing without knowing which risk is being bought.

The single most consequential artifact in the contract world is not a chain, a language, or a compiler: it is a token standard — a published interface of about a dozen functions (name, balance, transfer, approval and transfer-on-behalf) that turned "my program holds value" into "my program holds the value everyone's tooling already understands." Its importance is structural:

interfaces, not implementations, are where network ecosystems compound. Once wallets, exchanges, explorers, block-scanners, accounting software, and other contracts all speak the interface, every new implementation inherits the entire integration surface for free — and the reader who grasps that one dynamic understands why the "boring" standards layer of this ecosystem is

where its deepest moats live.

The standards family has since grown, and the chapter inventories it the way a mechanic inventories fasteners: interchangeable parts with published tolerances. There is the original fungible interface; a non-fungible companion with a metadata extension that made the collectibles-and-certificates economy possible; an interface for semi-fungible mixes; a security-token superset with transfer restrictions and identity hooks; an identity standard that stores a pointer from an address to its owner's data; a metadata discovery convention; a signature-validation standard; a multi-token batch interface. The inventory matters less than its consequences, which are three. First: standardization creates composability — a new token inherits custody flows, listings, and accounting on day one.

Second: standardization concentrates risk — a bug in a widely-shared implementation library becomes a bug in thousands of contracts at once, a dynamic the record has priced repeatedly. Third: standardization creates the upgrade temptation — every standard has a well-known ergonomic gap, and every gap breeds "improved" variants that splinter the very network effect the standard exists to create. The mature reading: implement the standard, extend by interface, and treat every "improvement" as a fork in a network, not a feature in a product.

Above the tokens sits the reusable-agreement layer — the open-source contract libraries, the audited vault and governance templates, the factory patterns — and it deserves an honest paragraph of its own. This layer is the ecosystem's genuine gift: a solo developer can deploy a custody system whose components have survived more adversarial attention than any private team could buy. But the gift has a twist that the record has priced at full retail: reuse is safe only when the reuse is understood. The most expensive failures in the decade's ledger include teams that copied an audited template, edited the four lines they did not understand, and shipped a system whose audit no longer applied. The template's audit covers the template; the deployment's audit is the deployment's alone — and a chapter of this book (the third) exists because that sentence keeps appearing, verbatim in structure, in the record's post-mortems.

The working developer's day, this decade, is spent inside a toolchain, and mapping it early prevents a confusion that costs real money: the belief that "the blockchain" is hard, when the hard part is the pipeline. The pipeline's stations, in order: a development environment that compiles the language, runs a local single-node network, executes test suites, and logs the deployment; a testing layer of unit tests, fork tests that replay real network

state, and invariant tests that assert global properties under random call sequences; a static-analysis layer that scans the source for the known bug patterns and dead code the compiler cannot refuse; a coverage layer that measures which branches the tests actually visit; a formal-methods layer — bounded model checkers, symbolic execution, proof assistants — for the invariants worth proving; a security-review layer that ranges from automated scanners to professional audits; and a deployment layer that treats the deployment itself as a critical operation: scripted, reproducible, verified on a test network, executed through a multisignature process, and confirmed byte-for-byte before the announcement.

The stations are all standard software practice, which is the point: the discipline of this field is not occult. What distinguishes the contract pipeline from an ordinary one is the consequence gradient. In an ordinary pipeline, a missed test is a bug ticket; in this one, a missed test can be an unrecoverable loss. The field's mature response to that gradient is not superstition but exaggeration of the ordinary: more testing than seems reasonable, more review than seems affordable, more ceremony around deployment than seems necessary — because the costs of exaggeration are paid in hours and the costs of understatement are paid in principal. The reader should hold this calibration early, because the chapters ahead keep returning to it: every practice in this book is ordinary practice, exaggerated one notch past comfort, and the exaggeration is the entire discipline.

The chapter closes with the human landscape, because the machine's state is people. Four builder populations share the ecosystem, each with a distinct center of gravity. Protocol engineers build the settlement layer itself and its core clients — the smallest population, the heaviest burden, watched by everyone. Contract engineers build the agreements — the audience most of this volume addresses — living inside the toolchain of the previous section and shipping value-bearing code under time pressure. Integrators build on top of contracts — the wallets, the interfaces, the aggregators, the enterprises — and inherit every interface the standards layer ever standardized, usually without reading the underlying code.

And auditors, formal-methods specialists, and security researchers make their living breaking what the first three build — the ecosystem's immune system, paid by the healthy to attack them. The four populations distrust each other productively, and the reader should notice the system property that produces: every artifact in this ecosystem, at every layer, is written by someone who expects a hostile expert to read it — which is why the ecosystem's best artifacts read like they were written for a hostile expert, and its worst artifacts read like they were written for a demo. The nine chapters that follow are, in

one long sense, a manual for writing the first kind.

State-of-the-field chapters in fast fields age like milk, so this one measures the field by gauges that age like tools. The first gauge: what breaks. The year's headline losses still cluster in the same three bins they have clustered in for years — logic bugs in hand-rolled code, failures of governance and key management around otherwise-correct code, and the economic exploits that treat a correctly-running contract as a machine to be gamed rather than broken. The bins' relative sizes have shifted — the ecosystem's tooling has genuinely thinned the first bin, while the third bin's losses have grown as attackers moved up the stack from code to economics — but the bins themselves have not changed in a decade, and a reader can date any field's maturity by how stable its failure taxonomy is. This taxonomy is stable; the field is no longer young.

The second gauge: what is contested. Three arguments organize the field's public debate, and each is a real engineering argument wearing a culture-war costume. The upgradeability argument — should contracts be immutable, as the rhetoric claims, or upgradable, as the practice demands — is really the question of how a fixed artifact survives a changing world, and the current settlement is a mature one: minimal privileged surfaces, timelocked changes, and upgrade paths designed in daylight rather than denied in public. The throughput argument — the network cannot yet run the world's commerce — is really the question of what belongs on the base layer at all, and the settlement is the architecture itself: heavy execution moving to secondary layers that settle to the base, exactly as mainframes yielded to terminals.

The institution argument — enterprises want permissioned, regulators want accountable, while the public networks want neither — is really the question of which properties buyers will pay for, and the current answer is quietly decisive: the same firms that piloted closed systems keep shipping to public ones, because the closed systems deliver the same engineering with none of the settlement guarantees. The reader should treat every future headline about all three as weather, not climate.

The third gauge: what is boring. The strongest signal in any technical field is the arrival of dullness, and this field now has it: custody systems that run for years without incident; token standards processed as clerical work; upgrade ceremonies executed on schedule and noticed by no one; audit findings that repeat known patterns rather than reveal new ones; insurance and bug-bounty markets that price contract risk in basis points. Boring was never this field's brand — the decade's first half ran on spectacle, launches, and front-page exploits — but boring is what a working infrastructure looks like from the

inside, and the spread of dullness through the ecosystem is the chapter's quietest and most important fact.

The chapters that follow are, in this sense, a tour of the boring parts: the failure catalog (chapter 2), the mistake catalog (chapter 3), the design principles (chapter 4) — all of them instruments for a field that has decided to survive, written for the builders who intend to be here when it does.

Because the series teaches machines by hand, the volume opens with its exercise: a one-hour census of the reader's own exposure, run before any further reading. List every contract the reader's household currently touches — the wallet on the phone, the tokens inside it, the services it connects to, any position the household holds in any on-chain system — and for each one, note two things only: the address, and whether the reader could state, in one sentence, what that program is entitled to do with the household's value. Most readers discover the same two facts the census reliably produces: first, the list is longer than expected — the household has been a contract party for years, mostly without noticing; second, the one-sentence test fails almost everywhere — the reader has been signing approvals and holding positions whose contracts they have never read, the way most people once held bank accounts whose terms they never read.

Neither discovery is a verdict; both are the book's motivation, printed in the reader's own hand. The chapters ahead convert that census from a list of unknowns into a list of answered questions, and the reader should keep it — the final chapter will ask for it again, grown into something stronger.

The series' method, restated for this subject: every claim in this book is structural, and every case is composite. When the text says a class of bug has cost the ecosystem nine figures, it says so without the specific incident's name, date, or amount, because the pattern is the lesson and the details are the costume — next season's exploit will wear new details over the same pattern, and the reader who memorized the pattern will recognize it while the reader who memorized the details waits for a repeat. The method has one cost the reader should grant up front: it makes this book quieter than the field's headlines, and quieter than its folklore, and quieter than its marketing.

The method has one dividend that repays the cost: nothing in this book expires. The reader finishing chapter 10 will hold a working model of every contract they will meet for the rest of the decade — on today's networks or tomorrow's, written in today's languages or their successors — because the machine underneath the costumes does not change, and never has: code, born public, enforcing its own text, answerable to no one after the fact. Everything in these pages is about learning to live with that machine well; the

learning starts with its limits, which is where chapter 2 begins, and the limits are where every serious practitioner's respect for this subject was born.

One more orientation helps before the chapter closes, and it belongs here because the whole field makes more sense with the comparison in view. Before this machine, an automated agreement between strangers ran on a stack of five layers: the two parties' databases, each recording its own version of the truth; the reconciling middleware that compared them and surfaced disputes; the operators on each side with the standing authority to correct, reverse, or overrule any record; the legal framework that specified what the operators' promises meant and who paid when they broke; and the settlement rail that actually moved the money once everyone agreed.

Each layer worked, at maturity, remarkably well — the modern payments system is one of engineering's wonders — and each layer charged its rent: the reconciliation cost, the operator risk, the legal overhead, the settlement delay, and above all the standing possibility that any layer's authority could be turned against the transaction it existed to serve. The contract machine replaces that stack with one layer: the program, executing on a network whose operators are adversaries by design and therefore trustworthy by construction, holding the record, applying the rules, and settling the movement in a single step that no party can reverse.

The comparison's honest arithmetic explains the decade's enthusiasm better than any marketing: the machine does not remove the cost of trust — it removes the number of places trust must live, and the difference between five layers and one, priced over millions of transactions, is the field's entire economic case, printed as architecture rather than as a promise.

The chapter ends where the volume's map begins, with the whole landscape compressed to the paragraph the reader should be able to reproduce: the contract is a program, deployed to a network of mutually distrusting machines, running deterministic, metered computation over permanent public storage, enforceable by no operator and reversible by no one; the ecosystem around it is dominated by one virtual-machine family whose moat is its tooling, one language family hardened by its crash record, and a standards layer whose interfaces compound value and risk in the same breath; the field's builders are four populations watching each other productively; the field's failures cluster in three stable bins while its arguments settle into three mature patterns; and the whole machine is visibly, measurably, becoming boring — the terminal state of working infrastructure.

That paragraph, honestly held, is the current state of smart contracts: not a hype cycle, not a scam catalog, and not a panacea, but a settlement machine

in its second decade, with its anatomy mapped, its limits priced, and its discipline — the rest of this volume — already knowable.

End of free sample. The full book is available at putrabooks.github.io

ABOUT THE AUTHOR

Aiman JoeHanz writes the Building of Tomorrow series -- structural, vendor-neutral guides to the technologies reshaping industry, labor, and daily life. Each volume treats one technology the way an engineer treats a machine: definition first, then anatomy, failure modes, principles, build, deployment, and the honest decade ahead.