SAMPLE

Building the Software of Tomorrow

The Instructions Beneath Every Machine

Aiman JoeHanz

Free sample: Introduction & Chapter 1

putrabooks.github.io

Buy the full book:

Carousell carousell.sg/p/building-the-software-of-tomorrow-ebook-pdf-1471313135

Ko-fi

Payhip

Contents

Introduction: The Machine That Obeys

Chapter 1: The Instruction and the Machine

Chapter 2: The Tower of Abstractions

Chapter 3: Languages and Their Discipline

Chapter 4: Algorithms and the Honest Limits

Chapter 5: The Operating System and the Sharing of Time

Chapter 6: The Persistence Layer

Chapter 7: The Network and the Contract

Chapter 8: The Engineering Discipline

Chapter 9: The Attack Surface

Chapter 10: The Economics and the Decade Ahead

Introduction: The Machine That Obeys

This volume earns its place on the shelf the way every volume in this series earns one: by refusing the marketing sentence and building the technology from its parts upward, until the reader can see the seams where it works and the seams where it fails. The chapters are built for two readings - one straight through, to acquire the argument, and a hundred partial ones afterward, opened at need before a decision, a hire, a purchase, or a debugging session. The introduction compresses the whole case into a few pages; the chapters assemble it piece by piece.

The Claim on the Box

Software is a set of instructions that a machine executes without judgment. That sentence sounds insultingly simple and is in fact the strangest fact of the modern world. A lever obeys a force, a motor obeys a current, but neither of them can be told anything - their behavior is their construction. Software split those apart. A general-purpose machine is built deliberately incomplete: it is a device for following instructions, and what it does is supplied after the fact, written down, changed without touching the metal. The same physical object that balanced a ledger in the morning can render a map at noon and simulate a bridge at midnight, because the behavior lives in the instructions and the machine only guarantees the obedience. Everything else in this book is a consequence of that one separation.

The separation has a cost, and the cost is where the whole discipline lives. Because the machine obeys exactly, it obeys stupidly: it does what the instructions say, not what the writer meant. The entire craft of software - the languages, the abstractions, the tests, the reviews, the maintenance calendars - is a set of prosthetics for the gap between human intent and machine execution. That gap never closes. It is managed, budgeted, and occasionally patched at three in the morning, and every practice in the engineering chapters exists because someone was burned by it.

The reader should also hear the honest economic claim early, because the decade ahead is mostly an economic story. Software is the only industrial artifact in history whose unit cost of reproduction is effectively zero and whose unit cost of maintenance never falls. The first half of that sentence built the largest fortunes of the era; the second half is the invoice the era keeps forgetting to read. The chapters on economics and on the decade ahead keep both halves on the table at once, because the sales literature quotes only the first.

This book treats software the way an engineer treats any mechanism worth understanding: the parts first, then the assumptions the parts rest on, then the standard failure paths, the real costs, and a workable practice to keep. The romantic framing - software eating the world, code as magic, the app that changes everything - is marketing shorthand for instructions that either execute as written or do not; the honest way to build on such a thing is to know the machine and the craft, not the slogan.

The Anatomy in One Paragraph

The chapters that follow walk a deliberate route. First the foundation: what an instruction physically is, and how a machine made of switches finds, decodes, and executes it billions of times a second. Then the tower of abstractions - the ladder from electricity to application - and the reason each rung exists, along with the reason every rung leaks a little. Then the disciplines the working engineer actually exercises daily: the notation called a programming language, the small catalog of algorithms and structures that carry almost every system, the operating system that legislates sharing among many programs, the persistence layer that lets data outlive the process that made it, and the network that turns one obedient machine into a civilization of them. After the anatomy comes the craft: how software is actually engineered, tested, debugged, and maintained - and why maintenance, not creation, is the majority of the work. Then the adversarial half of the discipline: the attack surface, and what correctness means when the inputs are hostile. The book closes with the economics of the decade - the zero-cost copy, the rented datacenter, the commons of open source, the switching costs called lock-in, and the machines that now draft instructions of their own - and an operating program a reader can actually run.

A note on what deliberately does not appear. There are no vendor names, no product names, no version numbers, no market-share figures, and no forecasts dressed as facts. They would all expire before the ink dried, and the series holds that structure outlives every product cycle. Where a number appears in the chapters, it is illustrative - a scale of magnitude, a count of rungs - or structural, a property of the design itself, and the text says which.

How to Read This Book

The chapters are ordered as one continuous ascent and descent: the first half climbs the machine - the instruction, the tower, the notation, the structures, the sharing constitution, the persistence layer, the network - and the second half turns to the practice that lives on it: the engineering loop, the adversary,

and the money. A reader new to the subject should take the chapters in order, because every chapter spends trust it earned in the one before. A reader with a specific decision pending may enter directly: the buyer auditing a vendor will find the boundary and interface doctrine spread across the tower, security, and economics chapters; the practitioner preparing for a system design will find the load-bearing sections in the structures, persistence, and network chapters; and the manager sizing an effort should read the engineering and economics chapters first, then the maintenance sections again, more slowly. Each chapter closes with a single page that compresses its argument to keepable size, and the reader who retains only the ten closing pages retains the book.

Two reading conventions are worth stating once. First, the chapters speak of no products by name, so the reader will find no shopping list here - only the questions a shopping list should have to answer. Second, where the chapters name a number, it is a scale of magnitude or a property of structure, never a specification to quote; the industry's printed figures expire annually, and the book's shapes do not. The reader who wants today's numbers should measure their own systems - the operating program's closing section explains how.

The Method Behind the Chapters

The series builds each volume the same way, and the method deserves a sentence of disclosure before the machinery starts. Every chapter begins with the technology's definition - stated plainly, and falsifiable, so that a wrong definition can be caught rather than admired - and then walks the same five stations in the same order: the working parts, named and located; the load-bearing assumptions, exposed where the marketing copy prefers euphemism; the failure modes, told as physics rather than as scandal; the economics, accounted on both ledgers; and the practice, compressed to the closing page. The discipline behind the method is the promise made above: definitions and structures over products and figures, because the product cycle is short and the anatomy is not. The reader who finishes this volume and disagrees with a definition will find the argument constructed to make that disagreement productive - each chapter states what would have to be true for its claims to fail, which is the honesty a technical book owes.

Who This Is For

The book is written for four readers at once, and the order matters. For the practitioner - the developer, the tester, the operator - it offers the connective tissue between daily practice and first principles: the reasons behind habits

that are usually inherited rather than examined. For the technical manager and the founder, it offers a map of where the money actually goes, which is rarely where the diagrams draw it. For the student, it offers the skeleton that a curriculum of fragments rarely assembles: one continuous argument from transistor to industry. And for the curious builder - the person who commissions software, lives inside it, and pays for it - it offers the vocabulary that separates an honest technical conversation from a sales call.

None of these readers need prior depth. The chapters assume the reader has used software, argued with it, and wondered why it behaves the way it does. That is the only credential required. Where a concept needs machinery to explain, the chapter builds the machinery first and uses it second, the way this series has always taken its subjects apart.

The Promises This Series Keeps

Every volume in this series makes the same five promises, and this one makes them about the invisible machine that runs the others. The definition will be honest: what software is, stated plainly enough to be wrong if it is wrong. The anatomy will be real: the parts named, the seams located, the abstractions identified as abstractions rather than revealed truths. The failures will be told without euphemism: the leak, the race, the deadlock, the breach, the abandoned dependency - not as scandals but as ordinary physics of the system. The economics will be accounted on both ledgers: the zero-cost copy and the never-ending maintenance invoice. And the operating program at the end will be small enough to keep, because a practice that cannot fit on one page does not survive a quarter.

The last promise is the quiet one, and it belongs at the front. Software is the most consequential human craft that almost nobody can see, and the decade ahead will be negotiated, in part, between the people who understand it and the people who merely purchase it. This volume exists to move readers across that line. The machine obeys; the chapters that follow are about everything that obedience costs, and everything it makes possible, from the first instruction to the decade's balance sheet.

Chapter 1: The Instruction and the Machine

Before software can be engineered, bought, secured, or maintained, it has to be stated honestly, and the honest statement is stranger than most working definitions admit: software is a text that a machine reads as physical behavior. Not a metaphorical text, not a legal one - an actual sequence of numbers stored in a physical medium, where each number instructs the hardware to move a small amount of electricity in a particular way. This chapter builds that statement from the bottom up, because every practice in the later chapters - every abstraction, every test, every security boundary - is an attempt to manage the consequences of this one fact.

The Switch That Counts

The physical atom of computing is a switch, and the switch used in digital machines has one unusual property: it is not operated by a finger but by voltage. A transistor conducts or refuses to conduct based on the electrical level at its control terminal, which means one switch can operate another, and that recursion is the entire secret. Logic built this way does not merely evaluate - it can be wired so that its own outputs select which of its inputs to listen to next, and a circuit arranged that way is no longer just a calculator but a machine that can direct itself.

To the question of why computers count in binary rather than in ten fingers' worth of levels, the engineering answer is margin. In a two-symbol alphabet the gap between the symbols is as wide as the circuit can make it; every additional symbol narrows the gaps and multiplies the ways noise can turn one symbol into another. The machine's world is therefore an alphabet of two symbols - call them zero and one, off and on, low and high - and the remarkable fact is that this impoverished alphabet suffices. With enough positions, any number can be written; with enough positions and an agreed code, any letter, any color, any sound pressure can be written. The alphabet carries no meaning at all, which is exactly why it can carry every meaning.

Representation is the first abstraction and the first place where the obedience gap appears. The machine does not know that a particular pattern of bits is a person's name, a pixel of teal, or a bank balance. It knows voltages. Meaning is supplied by convention - by the agreement among the humans and the programs that produce and consume the pattern - and every convention is a potential seam. When a program reads a pattern expecting one convention and the pattern was written under another, the machine executes flawlessly and the result is nonsense. Nothing crashed. Nothing erred. The gap between

representation and meaning is the site of a whole family of failures that later chapters will revisit with names like injection, encoding, and endianness, and the reason they exist at all is visible here at the bottom: the hardware obeys perfectly and understands nothing.

The Anatomy of One Instruction

An instruction is a small packet of bits, conventionally divided into fields: an operation code naming what is to happen, and one or more operands naming where. Where is itself a number - an address into the machine's storage, or the name of one of a small set of extremely fast storage cells called registers. The repertoire of operations in any one processor family is small and prosaic: load a value from memory into a register, store a register back to memory, add one register to another, compare two values, branch to a different instruction depending on the comparison, jump unconditionally. There is no instruction called "invoice" and none called "autumn." Everything a modern application appears to do - render typefaces, rank a feed, steer a car - is at the bottom this same prosaic handful of moves, executed at an almost indecent rate.

The machine's loop is worth naming once in full, because it is the heartbeat under every layer built later. Fetch: the processor presents, at a dedicated register called the program counter, the address of the next instruction, and the memory system hands back its bits. Decode: control logic reads the operation code and selects the circuitry it names. Execute: that circuitry runs - an addition occurs, a load moves a value, a branch updates the program counter. Then the loop repeats. A modern processor performs this cycle for multiple instructions at once, guessing ahead about which branches will be taken and discarding the wrong guesses, but the loop itself is unchanged: fetch, decode, execute, fetch, decode, execute. Software in every language on every platform is, at the bottom, a choreography for this loop.

Two consequences of the loop deserve their own light. The first is the conditional branch, the instruction that lets behavior depend on data. It is the difference between a calculator and a computer: with branching, the program can examine the world it is processing and choose. Every decision any application has ever made - every discount applied, every route chosen, every login refused - bottoms out in a comparison and a branch, and the reader who keeps that in mind will find that the impressive vocabulary of policy and workflow resolves into an enormous tree of tiny forks. The second consequence is the indirect reference - the instruction whose operand is the

address of an address. Indirection lets programs build structures in memory and walk them, and it is also, stated plainly, the mechanism behind every pointer bug in the craft's history. The power and the failure mode are the same mechanism; the engineering chapters are largely about discipline around exactly this kind of dual-natured power.

The Stored Program

The idea that separates the modern computer from the calculating machines that preceded it is deceptively small: store the instructions in the same memory as the data. Earlier machines were programmed by re-wiring panels or re-arranging gears; the program was the machine's construction. The stored-program design demotes the program to data - a sequence of numbers in a readable, writable medium - and the consequences arrive in a cascade. Programs can be loaded, which means one machine can be a different machine every hour. Programs can be written by other programs, which is the origin of compilers, interpreters, and eventually the whole tower of the next chapter. Programs can inspect and modify themselves, which is the origin of both the most powerful techniques in the craft and the most effective attacks in the security chapters. And programs can be copied, which is the origin of the software industry's entire economics - the zero-cost reproduction that makes software unlike any previous industrial artifact.

The stored program is also where the obedience gap becomes a formal structure. Because the machine cannot distinguish instruction bits from data bits except by context, a program that writes data into a place the machine will later fetch as instructions has changed the machine's behavior without changing its construction. Every language-level construct later in this book - every function call, every dispatch table, every sandbox - is a set of conventions for managing that power safely, and every memory-safety failure in the security chapters is a convention failing under pressure.

Why General-Purpose Is Worth the Cost

A general-purpose machine is deliberately worse than a dedicated one at any single task. A circuit built only to compress video will outrun, per watt and per dollar, a general processor doing the same work. The general machine wins anyway, for one reason: behavior is cheap to change. When the task changes, the dedicated machine is scrap and the general machine needs new instructions. The economics of hardware - expensive to design, cheap to replicate - couple with the economics of software - cheap to replicate, expensive to design - into a single industrial strategy: freeze the obedience in

silicon, and let the behavior live in text. The reader who grasps that pairing understands why the industry looks the way it looks - why processors are conservative and long-lived, why software churns, and why the skill of writing instructions became the lever that moves the whole arrangement.

There is a second-order cost that belongs on the same ledger. A general-purpose machine obeys arbitrary instructions, which means its behavior is bounded by nothing but the instructions it is given. Dedicated machinery is safe by construction - it physically cannot do most harmful things. A general machine can do anything the instruction set allows, and the instruction set allows everything, including instructions its designers would rather it never followed. The entire security profession is the ongoing attempt to rebuild, in software, the safety-by-construction that general-purpose hardware deliberately gave away. That is not a scandal; it is the bill for the flexibility, and it is presented again in every chapter that mentions a boundary.

The Gap, Stated as Physics

The chapter opened with a claim worth restating as a rule, because the rest of the book depends on it. The machine executes; it never intends. Between the intent in the writer's mind and the execution in silicon there is a permanent gap, and every artifact of the discipline - language design, type systems, code review, testing, formal methods, style guides, the entire maintenance culture - is a prosthetic for that gap. No prosthetic closes it. Good practice shrinks it, documents it, and instruments it, and bad practice is simply practice that forgets the gap is there.

This is also the honest answer to the layperson's most reasonable question: why is software broken so often, when bridges mostly stand? A bridge's loads are bounded by physics and its tolerances are sized by law; a program's inputs are bounded by nothing but the imagination of everyone who can reach it, and its tolerances were sized by whoever wrote it, under whatever deadline they were under. The craft's failure rate is not a mystery of incompetence; it is the predictable sound of an artifact that obeys exactly, being asked to guess intent, at a scale of input situations no writer can hold in mind. The chapters on engineering and on the attack surface are about the institutions builders have erected around that fact, and the chapter on the decade ahead is about how those institutions are changing as the writing itself becomes partly automated.

One more floor-level fact belongs here because it shapes every budget conversation later: speed has a shape. Fetching a value from a register is nearly instantaneous; fetching from the nearer tiers of memory takes longer by a factor best pictured as a short walk; fetching from the farther tiers takes longer by a factor best pictured as a cross-country errand; and reaching for a distant disk or a network is, at the processor's timescale, effectively overnight mail. The absolute numbers improve every year, which is exactly why this book refuses to print them - but the shape, the ratios of the tiers, is stable enough to design against. A program that respects the shape feels instant; a program that ignores it feels broken on hardware that is, by any printed specification, fast. The whole art of performance, as the algorithm chapters will show, is arranging work to fit the shape of waiting.

Three Instructions, Walked Slowly

The loop deserves one slow walk, because a reader who has watched it once never again mistakes software for magic. Take a fragment of a program that totals a column of figures: the first instruction loads into a register the value stored at a memory address - the location holding the current figure. The processor fetches it: the program counter names the address where the instruction itself lives, the memory hands back its bits, the program counter advances. Decoding shows an operation meaning read-from-memory, with two operands - the register to fill, the address to read - and the execution stage performs the read, so the register now holds the figure. The second instruction adds the register holding the figure into another register holding the running total: fetch, decode, the arithmetic circuitry runs, and the sum has grown. The third instruction is a comparison and a branch: the counter of remaining figures is tested against zero, and because figures remain, the branch is taken - the program counter is loaded not with the next address in sequence but with the address of the first instruction again, and the loop begins another turn.

Everything the reader has ever argued with on a screen is that fragment. A word processor's cursor blinking is a version of the third instruction, testing a keyboard register hundreds of millions of times a second, branching when a key finally shows up. A thermostat's cycle, a payment batch's progression, a game's heartbeat of read-inputs, update-world, draw-world - all of it is the same three moves, elaborated and stacked. And the walk also shows where discipline enters: the program counter can be made to point anywhere, and the loop has no opinion about where. Control flow is a promise kept by

convention, and the entire apparatus of structured programming in the next chapters - the function, the loop, the guarded boundary - is the craft's long negotiation over which uses of the jump are legible enough to survive contact with maintenance.

The walk also explains, in passing, two words the reader will meet constantly and rarely sees defined. The stack is the convention that uses one end of memory as a spike where return addresses and local values are parked in the order they will be needed in reverse; the function call is the instruction pair that parks the return address and jumps, and the return is its mirror image, and so the pile of paused calls - deep for recursive work, shallow for flat work - becomes a live map of what the program is doing at this instant. And the interrupt is the machine's arrangement for changing the program counter from outside the program: hardware - a disk finishing, a network card receiving, a timer expiring - can force the processor to park the current program and jump to a handler the operating system registered in advance. Interrupts are how the obedient machine takes orders from the physical world, and they are also why a program can be paused anywhere, at any instruction boundary, without its knowledge - a fact the operating-system chapter builds on, and that a generation of subtle defects has leaned on.

Where Hardware Stops Being Hard

The border between hardware and software is not a place on a diagram but a budget decision, and it moves. The instruction set itself is usually wired - the fetch-decode-execute machinery is circuitry, fixed at the factory. Yet even that statement needs a footnote: modern processors run a translation layer of their own, a small private program that converts the published instruction set into an even lower private one, so that successive generations of silicon can change their internals radically while keeping the published face identical, letting decades of existing software run on machines its authors never imagined. Below the visible machine, then, there is always another machine - and the principle generalizes upward through the whole industry. A video card carries firmware that is software wearing a hardware costume; a disk drive's controller is a computer of its own, running instructions its purchaser never sees; even the keyboard runs a tiny program. The reader who opens any modern device and looks for the computer will find several, each with its own loop, its own stored program, and its own maintenance obligations that nobody budgeted.

The moving border explains a recurring strategic pattern in the industry, worth

naming because it recurs at every scale. When a function is unstable - changing with product fashion - it lives in software, where change is cheap. When a function stabilizes - everyone agrees what it should do - it begins to migrate downward: into the libraries, then into the silicon, where the same behavior runs faster, cooler, and cheaper than the general machine could do it. Encryption arithmetic, network packet inspection, video decoding, the operations of the machine-learning moment: each began as expensive software and became circuitry, freeing the software layer to climb to the next unstable problem. The pendulum swings both ways across the decades - periods where capabilities consolidate into dedicated circuitry, and periods where the general processor, grown fast enough, absorbs the dedicated circuits back into software. The reader should notice the pattern's economic engine rather than its current phase: dedicated hardware is a loan against future stability, and like all such loans it is only sound when the stability arrives on schedule.

The moving border has one more consequence that belongs in every negotiation about platforms: hardware that can be reprogrammed after shipment is software wearing a costume, with software's entire obligation structure attached - patches, vulnerabilities, end-of-life dates, and the moral question of who owes maintenance after the sale closes. A device whose firmware cannot be updated is not more reliable than one that can be; it is simply one whose defects are permanent, a distinction the security chapter turns into a buying criterion.

The Machine's Pulse, and Why Identical Hardware Feels Different

The last floor-level idea is rhythm. The processor's work is divided into ticks of a clock - billions per second - and the honest mental picture is not a sprinter but an assembly line: an instruction is fetched in one tick and, while it is decoded, the next is already being fetched, so that several instructions are in flight at once, each at a different station. When the flow guesses right - the branch taken, the data already in the fast memory near the processor - the line runs at full rate. When it guesses wrong - the branch went the other way, the needed value lives in a farther tier - the line drains and refills, and ticks are lost. The same hardware, running two programs with the same instruction counts, can differ in felt speed by an order of magnitude purely on how often the guesses and the near memories hold. This is why the industry's benchmark arguments are so stubborn: the numbers are real, but they are

measurements of the match between a workload's habits and a machine's plumbing, not of a single speed.

The pulse explains one more everyday mystery worth taking upstairs to the later chapters: the machine never has one speed, so software never has one performance either. A request that finds its data near feels instant; the same request, one tick later, with one different access pattern, waits. Systems under load degrade in staircase steps rather than slopes - a component's internal line floods, everything downstream queues behind it - and the operating instinct that matters is to find which station stalled rather than to demand a faster factory. The engineer who can read a stall, in the language of this chapter's assembly line, will recognize it again at every larger scale the book reaches, because the shape of waiting, first seen here at the register, is the same shape at every tier: it is always a queue, someone is always waiting in it, and the engineering question is always whether the wait was chosen or merely suffered.

Numbers Have Furniture

The machine's arithmetic holds two surprises every professional meets eventually, and both are representation seams of the purest kind. Integers are stored with a sign convention - a leading flag in the negative's clothing - whose cleverness is that the circuit for subtracting is the circuit for adding with a wrapped operand, and whose danger is the silent wraparound: a sum that exceeds the type's ceiling reappears from the floor, so a program that counts past its largest representable value suddenly reports a catastrophic negative. The classic incidents of the craft - the counter that misbehaves only at the billionth event, the date arithmetic that fails on the far boundary of the type - are all this seam, wearing different costumes. The professional disciplines are unglamorous and absolute: size the counter for the total it must reach, not the daily rate; refuse the silent wrap by requesting the failing behavior explicitly where the notation allows it; and treat any integer that might plausibly touch its ceiling as a defect waiting for its calendar date.

The second surprise is the floating point, and its honest description dissolves a whole genre of bewilderment. Real quantities are stored in a scientific notation of base two - a significand times a power of two - which means that fractions whose denominators are not powers of two cannot be represented exactly, in the same way that a third cannot be written exactly in decimal. The famous demonstration - a sum that should be a half and prints as something disconcertingly near it - is not a malfunction but the notation working as

specified, and the working disciplines follow: never test fractional equality directly, because near is the honest best; accumulate money in the smallest integer units rather than in fractions of a unit, because money is a count wearing a decimal costume; and let the tolerance live in the problem's own terms rather than in the machine's. The lesson is the first chapter's in miniature and the whole book's in one line: the machine obeys the notation, the notation was chosen for the hardware, and the meaning was supplied by a human who did not attend the meeting.

The Bus, and the Machine's Own Traffic

One floor-level fact completes the picture: the processor is not the machine's only bottleneck, because the parts of the machine talk to each other over shared roads - the buses - and the roads have rush hours. A request that must cross from the processor to a far tier shares the road with every other crossing, so a system's felt speed is set less by how fast any one part computes than by how well the work avoids the roads: keeping near data near, batching crossings, and refusing to make distant trips for values that could have been carried in one. The professional's first instinct on a slow system - check the roads before blaming the factories - is the assembly-line lesson of the pulse section, and the reason the instinct works at every tier of the book: a bus, a memory channel, a network link, and a service's request queue are the same object wearing different widths.

The First Principles Page

One page from this chapter, kept small on purpose:

• Software is a text a machine reads as behavior; the machine is a fast, obedient, and comprehension-free executor of stored instructions.
• Binary is an engineering choice about noise margin, not a mystical one; representation is convention, and every convention is a seam.
• The loop is fetch, decode, execute; the branch makes it a computer, and indirection makes it powerful and dangerous in the same gesture.
• Storing programs as data is the single idea that created the industry's economics, its tooling, and its attack surface in one stroke.
• The gap between intent and execution never closes; every practice in the craft is a prosthetic for it.
• Speed has a shape - registers, memory, storage, network - that is stable long after the numbers change; design against the shape, not the digits.

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.