SAMPLE

Building the Computing of Tomorrow

Designing the Machines, Fleets, and Estates Beneath Every Application

Aiman JoeHanz

Free sample: Introduction & Chapter 1

putrabooks.github.io

Buy the full book:

Carousell carousell.sg/p/building-the-computing-of-tomorrow-ebook-pdf-1471312042

Ko-fi

Payhip

Contents

Introduction: The Layer Beneath Every Application

Chapter 1: The Instruction Is the Unit of Work

Chapter 2: One Machine Faster Meets a Wall

Chapter 3: The Memory Hierarchy Decides the Real Speed

Chapter 4: One Machine, Many Owners

Chapter 5: Machines That Are Made of Software

Chapter 6: Compute Purchased as a Metered Service

Chapter 7: Order Across Machines

Chapter 8: The Partition Chooses for You

Chapter 9: Failure as a Design Input

Chapter 10: Governing the Compute Estate

Introduction: The Layer Beneath Every Application

Every digital service that a person touches rides on a machine they never see. The booking confirms, the payment clears, the video plays, and somewhere below all of it a processor walks through a list of stored instructions, one at a time, at a pace no human can follow. That layer, the compute layer, is the subject of this volume. It is the forty-first book of the Building of Tomorrow series, and it arrives at the layer that the surrounding volumes depend upon: applications are its guests, networks are its couriers, and data centers are its buildings.

Compute is easy to treat as a commodity, because it is sold like one. Machines are purchased by the unit, rented by the hour, and benchmarked like lumber grades. Yet the difference between a well-governed compute estate and a neglected one is not performance. It is whether anyone can say what the machines are doing, what they will do when a part fails, what they cost to keep, and what they are permitted to forget. A processor obeys its program exactly, which means every surprise in the estate was placed there, knowingly or not, by a person. That is the quiet premise of this book: the machines of tomorrow will be exactly as trustworthy as the decisions made around them.

The chapters follow a single arc. The first chapters establish what a computing machine is and what one machine can and cannot do, from the instruction cycle that defines its work to the memory hierarchy that decides its real speed. The middle chapters follow the machine outward: how one machine is shared among many owners, how software imitates machines so that compute becomes countable, and how a metered service turned the machine room into a purchase order. The later chapters take up the hardest truth of the layer, that most serious systems are many machines acting as one, and that ordering events, surviving partitions, and failing gracefully are design disciplines rather than purchases. The final chapter gathers the operator's obligations into a governing view of the whole estate.

The book also continues a series habit that the reader of earlier volumes will recognize, and that the newcomer deserves to have explained. Each volume of Building of Tomorrow takes one layer of the world's technical infrastructure and treats it the way an engineer treats a machine: what it is made of, what it promises, where it fails, and how an operator governs it. Volume 40 took up the systems that carry messages. This volume takes up the layer that makes messages matter, because a message is only useful if there is, at its

destination, a machine that can act on it, a machine that was designed to know what acting means. The two volumes are companions, and the reader who holds both will find that the boundaries between them are drawn exactly where the layers hand work to each other: the transport says a delivery occurred, and the compute layer decides what the delivery does.

There is also a deliberate economy in the book's choice of foundations. The compute layer's fundamentals were settled by a small number of documents from the 1930s to the present: the paper that defined what a machine can compute, the report that put the program in the machine's memory, the standard that lets programs survive their machines, the definition that lets capacity be rented in plain words, the paper that orders events across distances, the proof that divides the promises a divided estate can keep, and the guideline that ends a machine's life honestly. The manuscript cites these foundations where they carry weight, and it treats everything else, every figure, every ratio, every timeline, as illustrative arithmetic, chosen so the reader can check it with a pencil and disagree with it out loud.

Finally, a word to the reader who operates one of these estates for a living. This is not a book of exhortations, and it does not believe in maturity models. It believes in pages: short written instruments, the contract table, the shared machine's agreement, the order page, the partition page, the failure page, the ledger, that convert an estate's folklore into decisions a person can read. Every chapter ends by assembling one of them, and the book's final chapter shows how they are read together on a calendar. If the book does its work, the reader will finish it with the modest and radical conviction that a compute estate is not a pile of machinery with opinions attached, but a governed object, knowable at any hour, whose behavior is the sum of written decisions that somebody can point to.

Two promises about the writing. First, every organization in these pages is hypothetical, standing in for a pattern rather than a company, and every number is an illustrative assumption, chosen so the arithmetic can be checked with a pencil. Second, the book names no vendors and recommends no products, because the mechanisms described here outlast the brands that sell them. What remains when the brands fade is the layer itself: instruction cycles, memory walls, interface contracts, rented capacity, ordered events, and failures met with rehearsed answers. Build that layer deliberately and the applications above it inherit the discipline. Neglect it and no application can buy the discipline back. A reader who wants the layer's whole argument in one

breath can have it here: the machine is obedient, so write what it obeys; the machine is bounded, so measure what it can reach; the machine is shared, so govern what it shares; the machine is describable, so keep the description; the machine is rentable, so price the rent and the exit; the machines are many, so order their events and rehearse their divisions; and the estate is a living thing, so read its pages on a calendar, and correct them while the room still remembers. Every chapter that follows is one clause of that sentence, tested against a hypothetical organization and a pencil.

Chapter 1: The Instruction Is the Unit of Work

A machine that does exactly what it is told

Picture a mid-sized agricultural cooperative that has just bought its first server. The board expects the machine to settle accounts with its member farms each month. The machine arrives, the staff feed it the closing figures, and within a season the members notice that their statements add up correctly but never agree with the handwritten ledger at the cooperative's own shop. An inspection finds nothing broken. The machine was executing a rounding rule written by a programmer who assumed that leftover cents should be discarded, while the shop's clerk had spent a decade giving every leftover cent to the member. Both parties had followed instructions. Only one set of instructions was written down where anyone could read it.

The example is hypothetical, and it is deliberately mundane, because the compute layer is a mundane layer. A computing machine is not an oracle that understands intentions. It is a device that carries out stored instructions with complete fidelity, including the ones nobody meant. This chapter establishes what such a machine is, where the design came from, and what the instruction, the smallest unit of its work, implies for every organization that depends on one.

The consequences of getting this foundation right are not small. A processor that obeys exactly is the most predictable tool an organization can own, and simultaneously the most efficient way to repeat a mistake at a scale no clerk could reach. Everything later in this book, from shared machines to rented capacity, inherits its character from the cycle described here. The compute of tomorrow is built from components that never ask what you meant.

What a computation can be

The modern definition of what a machine can compute was written before any of the commercial industry existed. Alan Turing's 1936 paper, On Computable Numbers, describes a device of remarkable poverty: an unbounded tape of cells, a head that reads and writes one cell at a time, a table that says what to do for each combination of current symbol and internal state, and nothing else. Out of this poverty comes a definition. The numbers that such a device can produce are the numbers that can be computed by finite means. The paper goes further and describes a universal machine, one that can imitate any described machine when given that machine's table, which is to say that the instructions are data. The paper is the reference for this boundary of the computable.

Two consequences of that definition carry through this whole book. First, the power of a computing machine does not come from cleverness in its parts but from the table of instructions it is given. The same hardware, given different instructions, becomes a different instrument. Second, the boundary of the computable is real. There are well-posed questions about programs that no program can answer for every possible input, and any operator who has waited for a checker that never finishes has brushed against that boundary in a practical form. Testing can never be exhaustive, because exhaustive checking of a general machine is not itself computable. The honest answer to a request for a tool that will prove any program safe is that the mathematics does not permit it, and the practical substitute is a combination of design discipline, measurement, and rehearsed recovery.

The universal machine also explains something that procurement discussions often miss. When an organization buys compute, it is not buying a machine that does bookkeeping or routing or imaging. It is buying a machine that does whatever its instruction table says, which means the purchase is never about the capability in the brochure, and always about the software it will run and who is accountable for that software's behavior.

The stored program is the founding decision

The machines of the 1940s could in principle do this work, but most had to be re-instructed by hand, by rewiring plugboards or re-reading decks that described the calculation. The step that produced the modern machine was architectural rather than mechanical. John von Neumann's 1945 report on the EDVAC describes a computing machine organized into organs: a central arithmetic part, a central control part, a memory, and the organs of input and output. The decisive property is that the memory holds both the numbers being worked on and the instructions directing the work, in the same medium, readable by the same apparatus.

Storing the program in memory changes the economics of change. A machine whose program lives in its memory can be pointed at a different program as cheaply as pointing it at different data. It also changes the economics of danger, because instructions and data share a medium and nothing in the medium itself distinguishes them. Every practical security boundary described in later chapters, from protection rings to isolated virtual machines, exists because the founding design made instructions and data indistinguishable to the machine that holds them, and therefore made the boundary a matter of discipline rather than of hardware separation.

The report's organ language also survives in the way this book will describe machines. A machine has parts that calculate, parts that decide what happens next, parts that remember, and parts that reach the outside world. Keep those four verbs in view, calculate, decide, remember, reach, and most architectural debates in the compute layer reduce to arguments about how much of each part an organization needs and where each part should sit.

The cycle that never tires

Beneath every marketing claim about intelligence or acceleration sits a loop. The processor fetches an instruction from memory, decodes what the instruction asks, executes that one small action, writes back whatever the action produced, and moves to the next instruction. Fetch, decode, execute, write back. The loop has no opinion about the work. It will run a payroll, render a sunset, or corrupt a ledger with the same indifference and the same reliability.

That indifference is worth internalizing, because it defines the unit of work at this layer. The instruction is the atom. Everything an application appears to do is a collapsing of millions of such atoms into a pattern that people finally see as a result. When someone asks why a task took thirty seconds, the honest engineering answer begins by counting atoms. Suppose a task completes in thirty seconds and the machine retires roughly one billion simple instructions per second, which is an illustrative assumption rather than a measurement of any particular chip. The task then consumed on the order of thirty billion instructions. Whether that number is admirable or scandalous depends entirely on how much of the work was real and how much was pattern around it, and the arithmetic gives an operator a first, crude question to ask of any slow system: where did the atoms go?

The loop also defines the machine's relationship with time. Each instruction completes in a span measured in fractions of a billionth of a second. The machine does not wait for meaning. If the data needed by an instruction is absent, the loop does not improvise; it either stops on a fault or reads whatever is there, including whatever stale or wrong value was left behind. Correctness at this layer is therefore not a property the machine provides. It is a property the surrounding design must arrange, by ensuring that the values present when the loop reaches for them are the values that were intended.

A program is a process written down

An organization's processes, the ones its staff know how to perform in a

corridor and cannot always explain on paper, become software by being written down as instruction tables. This is why software projects that begin without an agreed description of the process produce systems that encode somebody's guess. The cooperative's rounding rule from the opening section is a small instance of a general law: the machine will hold the organization to whatever process it was given, including the parts the organization did not know it had.

Writing a process down changes the process. Handwritten work tolerates judgment calls, and the ledger clerk's cent policy was a judgment call that the handwritten month tolerated. A stored program cannot tolerate anything that was never encoded, so the act of automating a process forces the decisions that the process had been quietly avoiding. Teams that fight about requirements are usually fighting about judgments that the paper process had silently absorbed for years. That fight is useful. It is the discovery phase of the compute layer, and it is cheaper than discovering the same facts from a machine that has been applying the wrong rule at speed.

The instruction loop's fidelity makes this discovery expensive in one direction only. A human clerk who is given a bad rule will apply it unevenly, complain, and occasionally rescue the organization from it. A machine applies the bad rule evenly, silently, and forever, until somebody changes the instruction table or the inputs. The remedy is procedural rather than technical: rules that matter should be written where non-engineers can read them, and the tables that encode them should be examined by the process owners themselves, not only by the programmers who wrote them.

The register is the workbench

At any instant, the immediate state of a computation is small. The values being combined this cycle, the address of the next instruction, and a handful of flags about what just happened live in the processor's own working storage, its registers. Everything else the program will ever touch sits outside the processor, in memories that are larger and slower, arranged in the hierarchy that Chapter 3 examines in detail.

The narrowness of the register set explains a habit of the layer that surprises newcomers. Programs spend their working lives moving values toward and away from the workbench, and most of the atoms in any large computation are moves rather than arithmetic. An operator reviewing a slow system is therefore rarely looking at a machine that is calculating too slowly. More often

the machine is ferrying data around a poorly arranged workshop, and the remedy is arrangement, not speed.

The register also sharpens the definition of state. When a program is interrupted, by another program's turn or by a failure, what must be preserved is the workbench and the location in the instruction table, together with whatever parts of the wider memory the program has claimed. Save the workbench and the claim, and the program can continue where it stopped. This is the fact that makes the sharing of one machine among many owners possible, and it returns in Chapter 4 as the mechanics of the process.

Determinism is a promise the machine keeps

Given the same instruction table and the same inputs, a machine in good order produces the same result, cycle after cycle. This determinism is the compute layer's greatest gift to design, because it turns behavior into something that can be tested, replayed, and reasoned about. It is also conditional. The machine keeps the promise only as far as its periphery. A clock that ticks at the wrong rate, an input device that returns different values to the same question, a sensor that occasionally reports noise, or a second program writing into the claimed memory will each break the appearance of determinism from the inside.

The design discipline that follows is to draw the boundary of determinism explicitly. Inside the boundary, the machine is a proof: replay the same inputs and the same outputs will appear, which is the basis of every checkpoint and replay scheme described later in this book. Outside the boundary lies the world, which promises nothing. Systems that blur the two, reading volatile outside values as if they were stable inside facts, inherit the world's disorder at machine speed. The chapter's checklist question is simple to ask and hard to answer honestly: which values does this system treat as law, and where did each of them come from?

Speed changes scale, not correctness

It is tempting to believe that a fast machine is a correct machine in a way that a slow one is not. The instruction loop offers no such comfort. A machine that executes a wrong rule a thousand times faster does not approach correctness through volume. It simply distributes the wrong result with greater efficiency, to more people, in less time, which is precisely what happened to the cooperative's cent before anyone read the program's rounding step.

Speed does change one thing materially: the size of the repair. Errors that a

handwritten process produces one at a time can be found, confessed, and corrected locally. Errors that a machine applies to every account at once must be corrected by other machines, which is why later chapters insist on reversibility, audit records, and staged changes as first-class requirements rather than luxuries. The faster the estate, the more the design must assume that any mistake will be large, and the more valuable the humble habit of keeping the recipe for undoing a change next to the recipe for making it.

Numbers obey their own conventions

The instruction works on representations, not on quantities. A price, a weight, and a countdown are all stored as patterns of bits that the machine combines according to fixed conventions. The conventions are exact, which makes misunderstandings between people and representations exact as well. A signed integer stored in four bytes of memory has a fixed ceiling, and the value of that ceiling follows from the convention itself: thirty-one bits of magnitude permit the count to reach two million one hundred forty-seven thousand four hundred eighty-three thousand six hundred forty-seven before the pattern turns over on its sign. The figure here is arithmetic of a convention, not a measurement of any machine, and it is chosen because the story it illustrates is common in operating estates.

Consider a hypothetical loyalty program that awards a point for each purchase and stores a member's balance in such a four-byte integer. Marketing plans a promotion awarding ten million points at once. Nothing about the promotion is unusual, and nothing about the machine fails. A member with a balance near the ceiling receives the award, the pattern overflows, the balance wraps to a negative value, and the program begins sending apology letters to a customer the business was trying to delight. The machine obeyed its convention. The failure was a decision made in ignorance of the convention, and it was detectable in advance by asking one question: what is the largest value this representation admits, and how does the largest plausible business event compare to it?

Overflow is only the loudest member of a family of representation surprises. Division of integers discards the remainder, so a program that averages counts in whole numbers quietly loses a fraction of every result. Sorting text orders characters by their numeric codes, which places capital letters before lowercase ones and can make a catalog appear disordered to people who read it. Clock values stored as seconds since an agreed origin will eventually exhaust their allotted bits. None of these behaviors is a malfunction. Each is a

convention that the instruction loop applies without comment, and the operator's protection is a written habit: for every quantity the estate keeps, record its representation, its range, and the largest event that could plausibly arrive.

The program is a translation, not a birth

Almost no working system contains its instructions in the form its authors first wrote them. Authors compose in languages convenient for people, and a chain of translating tools renders the composition into the instruction tables the loop consumes. The translation is deterministic, so the same composition yields the same machine program, and the chain also carries the result through layers of reusable libraries, which are simply other people's instruction tables bound into the same program.

The translation chain matters to an operator for two reasons. First, the running program is evidence. When a defect is observed, the behavior belongs to the translated artifact, and reproducing it requires the same translation, which is why disciplined estates keep the recipe: the exact composition, the exact translation tools, and their settings. An estate that cannot rebuild what it runs is guessing about what it runs. Second, the chain widens authorship. A payroll program contains fragments written by strangers, translated and bound into the artifact, and the organization inherits those fragments' behavior under the same exactness as its own. The practical response is modest and worth the effort: keep an inventory of the fragments, record their origins and versions, and treat a silent change in any fragment as a change to the whole program.

The instruction as unit of work therefore has an ancestry as well as a body. A compute estate that can name what runs, rebuild what runs, and account for who wrote what runs has converted the founding decision of 1945 from a curiosity into a controlled supply chain. An estate that cannot has simply arranged for its processes to be decided, in small part, by whoever next updates a fragment.

Counting atoms is an instrument of governance

Because the instruction is countable, the compute layer can be governed with a unit. Nothing else in an operating estate comes with so natural a denominator, and the habit of using it converts vague arguments about performance into arguments about work. Take a hypothetical billing service that completes four million transactions in a day. Suppose each transaction executes three million instructions, which is an illustrative assumption chosen

for round numbers, not a measurement of any real system. The estate then retires twelve trillion instructions daily for billing alone. If an engineer removes one instruction from the per-transaction path, the arithmetic of the saving is four million instructions a day, and if an engineer removes a tenth of the path, the saving is one point two trillion, or roughly a tenth of the service's entire atom budget.

The point of the arithmetic is not precision. It is that a tenth of the path has a size, and the size can be weighed against the cost of removing it. Estates that never count atoms argue about speed in adjectives. Estates that count them argue in atoms, and the argument ends in a decision that both sides can audit. The count need not be exact to be useful, in the same way that a rough inventory is more useful for governing a warehouse than a strong opinion about its tidiness.

The count also disciplines requests for capacity. A proposal to double the fleet for a service whose atoms are unmeasured is a proposal to spend money on an unknown. The same proposal for a service whose atom count is tracked, discussed, and known to be growing with transaction volume is a proposal that can be argued on its merits. The chapter's counsel is therefore brief: count the atoms, publish the count, and let every later chapter in this book use it.

Obedience at the boundary of the estate

The chapter's arithmetic closes with the boundary where obedience becomes visible to people. The compute layer does not deliver results to other machines in the abstract; it delivers them to people through edges, the screens and stations and printouts where the machine's exactness meets the world's exceptions. A stored program cannot hesitate at the edge case, but the estate that surrounds it can, and the boundary discipline is to decide, in writing, which edge cases the program should refuse rather than process. A rounding rule was the cooperative's undoing not because it existed but because nobody had asked whether the machine should be its author, and the question generalizes: for every rule the machine applies at full speed, name the human who owns the rule, and the naming is the estate's quiet protection against the obedience that nobody ordered.

The ownership of rules composes with the translation chains of an earlier section. A rule that lives in a fragment written by strangers is a rule whose owner is absent, and the estate's inventory of fragments is therefore also an inventory of authority: whose judgment runs here, and would the organization

endorse it today? The question sounds heavy and is asked quickly, in an afternoon per service, and the estates that ask it hold a property the brochures never sell, an estate whose every rule has a name beside it, answerable at the speed of a meeting rather than the speed of an incident.

The boundary no purchase crosses

The chapter closes where it began, at the boundary of automation. The universal machine defined in 1936 reaches any computation that can be expressed as a finite table of operations on symbols, and no further. Within that reach, machines are tireless, exact, and cheap to copy. Beyond it, three obligations fall back on people, and no chip purchase lifts them.

First, machines execute processes, and processes must be written down before they can be trusted at speed. Second, the boundary of what a program can be shown to do in general is fixed by mathematics, so assurance comes from discipline, measurement, and rehearsal rather than from any final proof. Third, the machine's exactness amplifies whatever it is given, so the accountability for results stays with the organization that chose the instructions.

The compute of tomorrow will obey its programs exactly as the compute of today does. The operators who profit from that obedience will be the ones who treat the instruction as the unit of work: counted, reviewed, and owned. The chapters that follow trace what happens as one such machine becomes many, as ownership becomes tenancy, and as tenancy becomes a metered service. At every step, the atom of the layer remains the same stored instruction, and the discipline that surrounds it remains the difference between an estate and an accident.

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.