

A SELF-DIRECTED CURRICULUM • REVISED SEPTEMBER 2026

The Engineer's Syllabus

Sixteen stages from your first line of code to the depth that makes you the person a team calls when the system is on fire. Every stage names what to learn, what to read, and what to ship.

STAGES	DIRECTED STUDY	AT 20 H / WEEK	PROJECTS SHIPPED
16	~2,720 h	2.6 yr	15

Contents

How to use this	3
The prerequisite spine	4
Time budget	5
Where to start if you have a degree	6
Milestone A · You can build things	8
Stage 00 · Tools, the shell, and how to learn	9
Stage 01 · Programming foundations	10
Stage 02 · Data structures and algorithms	12
Stage 03 · Mathematics for computing	14
Stage 04 · Computer architecture	16
Stage 05 · Operating systems and concurrency	18
Stage 06 · Networking and the web	20
Stage 07 · Languages, interpreters and compilers	22
Stage 08 · Databases and data modelling	24
Milestone B · You understand the machine	26
Stage 09 · Software craft and engineering practice	27
Stage 10 · Building and shipping real products	29
Stage 11 · Distributed systems and architecture	31
Stage 12 · Production, infrastructure and reliability	33
Stage 13 · Security engineering	35
Stage 14 · AI engineering	37
Milestone C · You can be trusted with a system	39
Stage 15 · Expert practice	40
Where to specialise	42
The ways this goes wrong	43
The shelf	44

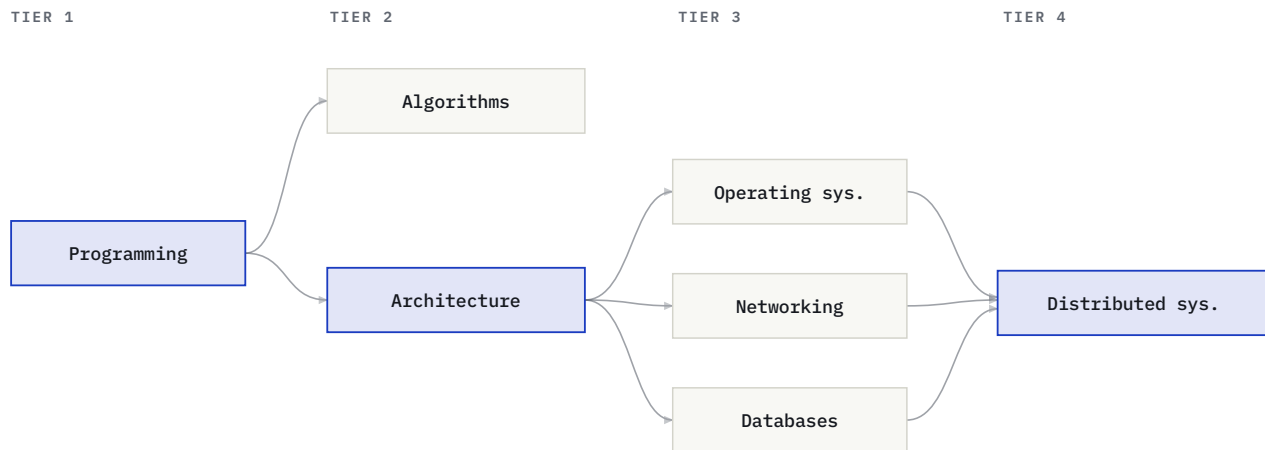
This is a map, not a schedule. The order is chosen so that each stage rests on something you already understand, which is why it is worth following even when a later stage looks more exciting. Seven rules decide whether you finish it.

If you already have a degree or a few years of work behind you, skip ahead to where to start and take the diagnostic first.

- 01 **Build more than you read.** Every stage below ends with something you ship. A stage is not finished when you have watched the lectures. It is finished when the project runs and someone other than you has used it.
 - 02 **Budget 100 to 200 hours per major subject.** This is the figure the Teach Yourself Computer Science authors settled on, and it holds up. Anything under that produces recognition without understanding.
 - 03 **One language, all the way down, before a second.** Depth in one language transfers. A shallow acquaintance with six does not. Pick Python or JavaScript and stay there until stage 07.
 - 04 **Never run more than two stages at once.** Mathematics pairs well with algorithms. Craft pairs well with anything. Three concurrent subjects means three abandoned subjects.
 - 05 **Write about what you learn, in public.** A learning log forces you to articulate the thing you half-understand. It is also, in practice, the strongest hiring signal a self-taught engineer can produce.
 - 06 **Let the assistant write code only after you can read it.** Coding agents are leverage on a foundation and a liability without one. The rule that keeps you honest: you must be able to explain every line you ship.
 - 07 **Consistency beats intensity.** Ten focused hours a week for three years outruns forty sporadic hours for six months. Protect the streak, not the session.
-

THE PREREQUISITE SPINE

Most of the curriculum is reorderable. These dependencies are not. Architecture is what makes operating systems and databases legible instead of memorised, and distributed systems are incoherent until you know what a process and a packet actually are.



TIME BUDGET

Hours below are study hours, not calendar time. They assume focused work with a project attached, not lectures playing in a browser tab.

COMMITMENT	FIRST JOB (~1,070 H)	SOLID MID-LEVEL (~1,940 H)	SENIOR DEPTH (~2,720 H)
10 h / week	2 years	3 yr 9 mo	5 yr 3 mo
20 h / week	1 year	1 yr 11 mo	2 yr 7 mo
40 h / week	6 months	11 months	1 yr 4 mo

One honest caveat. The last column buys you the *knowledge* of a senior engineer, not the title. Seniority also requires years of shipping alongside other people, absorbing production incidents, and being wrong in public. No reading list shortens that part.

WHERE TO START IF YOU HAVE A DEGREE

A computer science degree covers stages 01 through 08 and largely skips 09 through 15. You were taught what a B-tree is. You were probably never made to refactor someone else's code under test, deploy it, or be woken up when it failed. That gap is the whole of the applied half of this document, and it is what hiring screens for now.

So the short answer is *stage 09*, after a one-week sweep of stage 00. But a degree certifies attendance, not retention, and the course you took in second year has decayed. Verify rather than assume. Each test below is one question you can answer honestly in an afternoon, and failing one means that stage is not behind you.

STAGE	CAN YOU DO THIS TODAY, UNAIDED?	TYPICAL GRADUATE
00 Tools	Rebase through a conflict, then explain what it did to the commit graph	GAP
01 Programming	Build a tested command-line tool in an afternoon with no framework	SOLID
02 Algorithms	Solve an unseen medium graph problem in thirty minutes with no reference	RUSTY
03 Mathematics	Prove an algorithm correct by induction	SOLID
04 Architecture	Explain why two loops of identical complexity differ fourfold in speed	RUSTY
05 Operating systems	Find and fix a race condition in threaded code you did not write	RUSTY
06 Networking	Narrate a request end to end, including the TLS handshake	RUSTY
07 Compilers	Write a recursive descent parser from memory	VARIES
08 Databases	Read an execution plan and name the index that is missing	GAP
09 Software craft	Refactor a large unfamiliar codebase under test without changing behaviour	GAP
10 Shipping	Name something a stranger depends on that you built and still maintain	GAP
11 Distributed	Design for ten million users and defend your choice of	GAP

shard key

12 Production	Point to an incident you were paged for on something you deployed	GAP
13 Security	Describe a real broken access control bug you found yourself	GAP
14 AI engineering	Show a model-backed feature you shipped with an automated eval suite	GAP

□ Solid – skip the stage □ Rusty – refresher only, about a fifth of the hours

□ Gap – the full stage at full hours

The compressed track

A stage you once passed takes roughly a fifth of the listed hours to restore. Re-read your notes, build the one project, and move on. Do not re-study from the top. That turns 2,720 hours into something closer to 1,640.

SWEEP 00 – 08	FULL EFFORT 09, 10, 12	THEN 11, 13, 14	GRADUATE TOTAL
400 h	680 h	560 h	~1,640 h
Refresher pace. Projects only, skip the lectures.	Craft, ship a product, then operate it in production.	Distributed systems, security, AI engineering.	1 yr 7 mo at 20 h per week.

One warning specific to your position. Graduates tend to jump straight to stage 11, because distributed systems feels like the prestigious part and the coursework flattered it. It does not stick before you have operated something real. Kleppmann reads as trivia until you have been woken at three in the morning by a replication lag alert.

Stages 09, 10 and 12 are the whole distance between a computer science graduate and an employable engineer. Spend the 680 hours there first, in that order.

You can build things

Roughly 1,070 hours. You can take a vague idea, decompose it, build it, test it, and put it in front of a user. This is the point where a junior role becomes realistic.

Tools, the shell, and how to learn

4 weeks 60 hours No prerequisites

Almost nobody teaches this, and it quietly costs beginners hundreds of hours. Before you learn to program, learn to operate the machine you will program on. Every stage after this runs faster because of it.

WHAT YOU LEARN

- The shell as a language: pipes, redirection, globbing, exit codes, **grep**, **find**, **sed**, **awk**
- The filesystem model: paths, permissions, symlinks, environment variables, **PATH**
- Git as a data structure, not a list of commands: commits, trees, refs, branching, rebase, merge conflicts, bisect
- Collaborating on GitHub: forks, pull requests, review comments, issues
- One editor to fluency, with its keybindings, search, and multi-file navigation
- Reading documentation and error messages instead of pasting them somewhere
- Asking a question well: minimal reproduction, what you tried, what you expected
- Study technique: spaced repetition for facts, deliberate practice for skills, retrieval over rereading

RESOURCES

COURSE The Missing Semester of Your CS Education MIT · the single highest-leverage two weeks in this document

BOOK Pro Git Chacon & Straub · read chapters 1–3 and 7

PRACTICE Learn Git Branching visual, interactive, fixes rebase anxiety

BOOK Make It Stick Brown, Roediger & McDaniel · how learning actually works

SHIP THIS

A public dotfiles repository, plus one shell script that automates a chore you genuinely repeat. Write the README as if a stranger will use it.

Programming foundations

3–4 months

300 hours

After stage 00

The longest single stage, and the one people rush hardest. The goal is not syntax. It is the ability to hold a problem in your head, break it into parts, and express those parts as code that someone else can read. *Do not be the person who never quite understood recursion.*

WHAT YOU LEARN

- Values, types, variables, expressions, statements, scope
- Control flow, iteration, and the loop invariants that make it correct
- Functions as the unit of abstraction: arguments, return values, purity, side effects
- Recursion, base cases, the call stack, and why deep recursion blows up
- Collections: lists, dictionaries, sets, tuples, and the cost of each operation
- Strings, encodings, and why UTF-8 exists
- Errors and exceptions: raising, catching, and failing loudly at the right boundary
- Files, streams, serialisation, JSON and CSV
- Objects and classes, composition, interfaces, and when not to use inheritance
- Automated testing from week one, with a real test runner
- Debugging with a debugger: breakpoints, stepping, inspecting state
- Modules, packages, dependencies, virtual environments, lockfiles
- Memory: the stack, the heap, references, mutation, and aliasing bugs
- Reading unfamiliar code and changing it without breaking it

RESOURCES

COURSE	Berkeley CS 61A abstraction, recursion, interpreters · do this or SICP
BOOK	Structure and Interpretation of Computer Programs Abelson & Sussman · chapters 1–3 with exercises
PRACTICE	Exercism ~100 small problems with human mentorship
COURSE	The Odin Project if the web is your destination, use this as the spine

SHIP THIS

Four programs, each in a different shape. A command-line tool with arguments and a help screen. A program that reads messy real-world data and produces a clean report. A small game with state. A script that talks to a public API and caches its results.

Each with tests, a README, and no framework doing the thinking for you.

Data structures and algorithms

3–4 months

250 hours

After stage 01

This is the vocabulary of the field. Not because interviews ask for it, though they do, but because a problem you can name is a problem you can look up. Implement every structure once, from nothing, before you ever use the library version again.

WHAT YOU LEARN

- Asymptotic analysis: big-O, big-theta, amortised cost, space complexity
- Arrays, dynamic arrays, and why appending is amortised constant time
- Linked lists, stacks, queues, dequeues, circular buffers
- Hash tables: hashing, collision resolution, load factor, resizing
- Trees: binary search trees, balancing, heaps, priority queues, tries
- Graphs: adjacency lists and matrices, BFS, DFS, topological sort, cycle detection
- Shortest paths: Dijkstra, Bellman–Ford, A*; minimum spanning trees
- Union-find with path compression
- Sorting: merge, quick, heap, insertion, counting and radix; stability; when each wins
- Binary search and its off-by-one traps, including search over answer space
- Recursion to divide-and-conquer to dynamic programming: memoisation, tabulation, state design
- Greedy algorithms and the exchange argument that proves them
- Backtracking, pruning, constraint problems
- Two pointers, sliding window, prefix sums, monotonic stacks
- Bit manipulation and its idioms
- NP-completeness and reductions: recognising when no efficient answer exists
- Probabilistic structures: Bloom filters, count-min sketch, HyperLogLog

RESOURCES

BOOK	The Algorithm Design Manual Skiena · the real stories are the reason to pick this over CLRS
COURSE	MIT 6.006 Introduction to Algorithms full lectures, notes and problem sets
COURSE	Stanford Algorithms Tim Roughgarden · four-part specialisation
PRACTICE	NeetCode 150 a curated set beats grinding 500 random problems
BOOK	Introduction to Algorithms CLRS · a reference to look things up in, not to read cover to cover

SHIP THIS

A library implementing every structure above from scratch, with tests and complexity documented per method. Then solve around 150 curated problems, and for each one you fail, write down which pattern you missed.

Mathematics for computing

2–3 months 150 hours Run in parallel with stage 02

Computer science is applied mathematics that got out of hand. You do not need a mathematics degree, but the discrete half is not optional, and proof by induction is the same muscle as recursion. Run this alongside algorithms, not before them.

WHAT YOU LEARN

- Logic and proof: implication, quantifiers, contradiction, and induction
- Sets, relations, functions, equivalence classes, partial orders
- Combinatorics: permutations, combinations, pigeonhole, inclusion–exclusion
- Discrete probability: conditional probability, Bayes, expectation, variance, linearity of expectation
- Common distributions and tail bounds; randomised algorithm analysis
- Graph theory as mathematics: connectivity, colouring, matchings, planarity
- Number theory: modular arithmetic, GCD, primes, and the road to public-key cryptography
- Recurrence relations and the master theorem
- Linear algebra: vectors, matrices, transformations, rank, eigenvectors
- Calculus to the level of derivatives, gradients, and optimisation
- Statistics you will actually use: sampling, confidence intervals, A/B tests, Simpson's paradox

RESOURCES

- | | |
|--------|---|
| COURSE | MIT 6.042J Mathematics for Computer Science Tom Leighton · lectures plus the free 1,000-page text |
| VIDEO | Essence of Linear Algebra 3Blue1Brown · watch before any linear algebra course |
| COURSE | MIT 18.06 Linear Algebra Gilbert Strang |
| BOOK | Think Stats Downey · statistics through code rather than notation |

SHIP THIS

Written solutions to two full 6.042J problem sets, including the proofs. Then a short simulation that demonstrates a counter-intuitive probability result, with the maths written up next to the code.

Computer architecture

2–3 months 150 hours Gateway to stages 05, 06, 08

Without a solid mental model of how the machine works, every higher abstraction you learn stays brittle. This is the stage that turns performance from superstition into arithmetic, and it unlocks operating systems and databases behind it.

WHAT YOU LEARN

- Number representation: binary, hex, two's complement, overflow
- IEEE 754 floating point, and why $0.1 + 0.2$ is not 0.3
- Boolean logic, gates, multiplexers, adders, the ALU
- Sequential logic: flip-flops, registers, the clock
- Instruction set architecture, machine code, and reading assembly
- The fetch-decode-execute cycle; a datapath and a control unit
- The memory hierarchy: registers, L1/L2/L3, RAM, disk, and their real latencies
- Cache lines, locality, associativity, and cache-friendly data layout
- Pipelining, hazards, branch prediction, speculative execution
- Instruction-level and data-level parallelism, SIMD
- How a program becomes a process: compiling, assembling, linking, loading
- The stack frame, calling conventions, and how buffer overflows happen
- I/O, interrupts, DMA, the system bus

RESOURCES

COURSE Nand2Tetris, parts I and II build a working computer from a NAND gate upward

BOOK Computer Systems: A Programmer's Perspective Bryant & O'Hallaron · if you read one systems book, this

COURSE Berkeley CS 61C Great Ideas in Computer Architecture

BOOK Computer Organization and Design Patterson & Hennessy · the RISC-V edition

SHIP THIS

The full Nand2Tetris stack: gates, ALU, CPU, assembler, virtual machine, and a compiler for its language. It is the first project most people build that is complex enough to make software engineering feel necessary.

Then take one slow program and make it four times faster using only cache locality.

Operating systems and concurrency

2-3 months

150 hours

Requires stage 04

Every line of code you write is run by an operating system, and most production incidents are arguments with it. Concurrency lives here too, and it is the subject where confident guessing does the most damage.

WHAT YOU LEARN

- Processes: address space, fork and exec, process states, signals
- Threads, context switching, kernel versus user mode, system calls
- Scheduling: round robin, priority, multi-level feedback, fairness
- Concurrency primitives: mutexes, semaphores, condition variables, atomics
- Race conditions, deadlock, livelock, starvation, and the four Coffman conditions
- Concurrency models beyond threads: event loops, async/await, actors, CSP channels
- Virtual memory: paging, page tables, the TLB, page faults, swapping
- Memory allocation, fragmentation, garbage collection strategies
- File systems: inodes, directories, journalling, durability and fsync
- I/O models: blocking, non-blocking, multiplexed, asynchronous; epoll and kqueue
- Interprocess communication: pipes, sockets, shared memory, message queues
- Virtualisation and containers: namespaces, cgroups, what Docker actually is

RESOURCES

- | | |
|----------|---|
| BOOK | Operating Systems: Three Easy Pieces Arpaci-Dusseau · free, excellent, the default choice |
| <hr/> | |
| COURSE | Berkeley CS 162 lectures that pair with OSTEP |
| <hr/> | |
| PRACTICE | MIT 6.1810 and the xv6 labs a real teaching kernel you modify |
| <hr/> | |
| BOOK | The Little Book of Semaphores Downey · synchronisation puzzles that build intuition |

SHIP THIS

A shell that supports pipes, redirection, and job control. A thread pool with a work

queue. Then complete the xv6 labs, adding at least one system call of your own.

Networking and the web

6–8 weeks

120 hours

Requires stage 04

Almost everything you build will be reached over a network, and almost every mysterious latency problem lives in this stage. Learn it top-down, starting from HTTP, which is where your work touches it.

WHAT YOU LEARN

- The layered model and encapsulation; what each layer is actually responsible for
- HTTP in depth: methods, status codes, headers, caching, cookies, content negotiation
- HTTP/2 multiplexing and HTTP/3 over QUIC; head-of-line blocking
- TCP: the handshake, sequence numbers, retransmission, flow control, congestion control
- UDP and when losing packets is the right trade
- IP addressing, subnets, routing, NAT, IPv6
- DNS: resolution, record types, TTLs, and why it is always DNS
- TLS: handshake, certificates, chains of trust, certificate authorities, mTLS
- Sockets: the API underneath every client and server you have used
- Real-time transports: WebSockets, server-sent events, long polling, WebRTC
- RPC over the network: gRPC, protocol buffers, schema evolution
- CDNs, edge caching, anycast, and load balancers at layers 4 and 7
- Network diagnosis: **dig**, **traceroute**, **tcpdump**, Wireshark, browser dev tools

RESOURCES

BOOK Computer Networking: A Top-Down Approach Kurose & Ross · do the Wireshark labs, they are the course

COURSE Stanford CS 144 the lab where you implement TCP is worth the whole stage

BOOK High Performance Browser Networking Ilya Grigorik · free online

REFERENCE MDN HTTP reference the working reference you will return to for years

SHIP THIS

An HTTP/1.1 server written directly on TCP sockets, handling keep-alive, chunked encoding and concurrent connections, with no web framework. Then implement a working TCP stack in the CS 144 labs.

Languages, interpreters and compilers

6–8 weeks

120 hours

Requires stages 01, 04

Understanding how a language is implemented makes every language easier to pick up and makes your own code better, because you stop guessing what the runtime does. It is also the most enjoyable stage in this document.

WHAT YOU LEARN

- Lexing, parsing, grammars, abstract syntax trees
- Recursive descent and precedence climbing; parser generators and their trade-offs
- Semantic analysis, scope resolution, symbol tables
- Type systems: static and dynamic, strong and weak, inference, generics, variance
- Tree-walking interpreters versus bytecode virtual machines
- Intermediate representations, optimisation passes, code generation, register allocation
- Runtime memory management: reference counting, tracing collectors, generational GC
- Closures, first-class functions, tail calls, continuations
- Paradigms compared: imperative, object-oriented, functional, logic
- Immutability, algebraic data types, pattern matching, and what they buy you
- Foreign function interfaces and language interoperation

RESOURCES

BOOK **Crafting Interpreters** Bob Nystrom · free, beautifully written, do both halves

COURSE **Stanford Compilers** Alex Aiken

BOOK **Compilers: Principles, Techniques and Tools the Dragon Book** · cherry-pick chapters, do not read it straight through

PRACTICE **A second paradigm** learn Haskell, OCaml, Elixir or Clojure until it changes how you write your first language

SHIP THIS

A complete language: scanner, parser, resolver, tree-walking interpreter, then a

bytecode virtual machine with its own garbage collector. Add one feature the book does not cover, so you have to design it yourself.

Databases and data modelling

2-3 months

180 hours

Requires stage 04

Data sits at the centre of almost every program worth writing, and very few engineers understand how the database actually executes their query. This stage pays for itself the first time you read an execution plan instead of adding an index at random.

WHAT YOU LEARN

- The relational model, keys, referential integrity, normal forms, and when to denormalise
- SQL properly: joins, subqueries, CTEs, window functions, grouping sets, set operations
- Indexes: B-trees, LSM trees, composite and covering indexes, selectivity, index-only scans
- Query planning and execution: reading **EXPLAIN ANALYZE**, join strategies, statistics
- Transactions and ACID; isolation levels; the anomalies each one permits
- Concurrency control: two-phase locking, MVCC, optimistic concurrency, deadlock handling
- Durability: write-ahead logging, checkpoints, crash recovery, replication lag
- Storage engines: row stores versus column stores, page layout, compression
- Schema design and safe online migrations under live traffic
- NoSQL families: document, key-value, wide-column, graph, time-series, and the fit for each
- Caching: Redis, cache-aside and write-through, TTLs, stampedes, invalidation
- Full-text search and inverted indexes
- Analytical data: OLTP versus OLAP, warehouses, star schemas, ETL and ELT
- The object-relational mismatch, and where ORMs help and where they mislead

RESOURCES

COURSE [CISC 120 Database Systems](#) Amy Davidson · lectures and projects, all public

BOOK [Use The Index](#), Luke Markus Winand · indexing explained for developers

COURSE [Stanford Databases](#) Jennifer Widom · the modelling and SQL modules

BOOK [Designing Data-Intensive Applications](#), chapters 2–4 Kleppmann · storage, encoding and data models

SHIP THIS

A key-value store with a write-ahead log, crash recovery, and a B-tree or LSM index. Then take a slow query in a real dataset of a million rows and bring it under fifty milliseconds, documenting each plan change.

You understand the machine

Roughly 1,360 hours. You can reason about a program from the transistor to the HTTP response, and debug it at any layer in between. Most working engineers never reach this point. It is what separates competence from seniority later.

Software craft and engineering practice

3–4 months 220 hours Start early, never finish

This is where a programmer becomes an engineer. Everything before this stage is about making code work. This stage is about making code that a team can change safely two years from now, which is the actual job.

WHAT YOU LEARN

- Naming, function size, and comments that explain why rather than what
- Managing complexity: deep modules, narrow interfaces, information hiding
- SOLID principles, read critically rather than as scripture
- Composition over inheritance, dependency inversion, the seams that make code testable
- Design patterns, and the far more useful skill of recognising when you do not need one
- Testing: unit, integration, end-to-end; the test pyramid and its critics
- Test-driven development, property-based testing, snapshot tests, mutation testing
- Test doubles: fakes, stubs, mocks, and why over-mocking makes tests useless
- Refactoring in small safe steps with tests as the net
- Working with legacy code: characterisation tests, seams, strangler fig migrations
- Code review: what to look for, how to disagree, how to receive it
- Branching strategies, trunk-based development, feature flags, semantic versioning
- Systematic debugging: bisect, reduce, hypothesise, verify. Profiling before optimising
- Technical writing: READMEs, design documents, architecture decision records, RFCs
- Estimation, decomposition, scope negotiation, and working in iterations honestly
- Error handling strategy: where to fail, where to retry, what to log, what to surface

RESOURCES

BOOK **A Philosophy of Software Design** Ousterhout · short, dense, the best book on complexity

BOOK **The Pragmatic Programmer** Hunt & Thomas · the 20th anniversary edition

BOOK Refactoring Martin Fowler · second edition

BOOK Working Effectively with Legacy Code Michael Feathers · for the job you will
actually have

READING Google's code review guidelines short, practical, free

SHIP THIS

Take your worst earlier project and refactor it under a full test suite without changing behaviour. Then make three real pull requests to open-source projects and absorb the review comments without arguing first.

Building and shipping real products

3-4 months

240 hours

Requires stages 06, 08, 09

Everything so far has been components. This stage is the whole machine, with users, payments, mistakes, and things that break at three in the morning. The hiring market has moved decisively from "I helped on a feature" to "I built this and shipped it."

WHAT YOU LEARN

- API design: resources, HTTP semantics, status codes, idempotency, pagination, filtering
- Versioning and backward compatibility; deprecation without breaking clients
- GraphQL and gRPC, and the honest trade-offs against REST
- Authentication and authorisation: sessions, tokens, OAuth 2.0, OIDC, refresh flows
- Permission models: role-based, attribute-based, multi-tenancy and data isolation
- Frontend fundamentals: semantic HTML, the CSS box and layout models, the DOM, the event loop
- Accessibility as a requirement: keyboard navigation, focus, contrast, screen readers, ARIA
- One component framework well, plus state management and data fetching patterns
- Rendering strategies: client-side, server-side, static, streaming, hydration cost
- Build tooling, bundling, code splitting, and the performance budget
- Background work: job queues, workers, schedulers, retries, dead letter queues
- Files and media: uploads, object storage, signed URLs, image pipelines, CDNs
- Integrations: webhooks, third-party APIs, payments, transactional email
- Rate limiting, pagination at scale, idempotency keys, and defensive input validation
- Feature flags, analytics, and shipping to a subset of users first
- Product thinking: what the user is trying to do, and what you can refuse to build

RESOURCES

COURSE Full Stack Open University of Helsinki · modern, free, project-driven

COURSE The Odin Project the closest free thing to how a working developer learns

REFERENCE roadmap.sh a checklist of the ecosystem, not a curriculum · use it to find gaps

REFERENCE WCAG 2.2 quick reference accessibility as a specification, not an opinion

SHIP THIS

One real product with real users. Accounts, a database, background jobs, payments or another irreversible action, and a deployment you maintain for at least three months.

Three to five polished projects beat twenty abandoned ones. Delete the rest.

Distributed systems and architecture

4–6 months

280 hours

Requires stages 05, 06, 08

Once a second machine is involved, everything you assumed about time, order and failure stops being true. This is the deepest stage in the curriculum and the one that most defines a senior engineer. Do not start it before you have built something real, or none of it will stick.

WHAT YOU LEARN

- Why distribute at all: throughput, availability, latency, and the cost of each
- The eight fallacies of distributed computing, and how each one bites
- Replication: single-leader, multi-leader, leaderless; synchronous and asynchronous trade-offs
- Partitioning and sharding, rebalancing, consistent hashing, hot keys
- Consistency models: linearizability, sequential, causal, eventual; read-your-writes
- CAP and PACELC stated correctly, rather than as a slogan
- Consensus: leader election, Raft, Paxos, quorums, split brain
- Time and ordering: clock skew, logical clocks, vector clocks, hybrid clocks
- Distributed transactions: two-phase commit, sagas, outbox pattern, idempotent consumers
- Messaging: queues versus logs, Kafka, delivery semantics, and the exactly-once myth
- Event-driven architecture, CQRS, event sourcing, and their real costs
- Failure handling: timeouts, retries with jittered backoff, circuit breakers, bulkheads, backpressure
- Load balancing, service discovery, health checks, graceful degradation
- Batch and stream processing; windowing, watermarks, late data
- Architecture styles: monolith, modular monolith, services, serverless, event-driven
- Domain-driven design: bounded contexts, aggregates, ubiquitous language
- The system design method: requirements, estimates, API, data model, high level, deep dive, bottlenecks

RESOURCES

BOOK	Designing Data-Intensive Applications Martin Kleppmann · the central text of this stage. Read it twice
COURSE	MIT 6.5840 Distributed Systems lectures, paper list, and the Raft labs
PAPER	In Search of an Understandable Consensus Algorithm Ongaro & Ousterhout · the Raft paper
BOOK	System Design Interview, volumes 1 and 2 Alex Xu · for the interview framing specifically
BOOK	Fundamentals of Software Architecture Richards & Ford · trade-off analysis as a discipline

SHIP THIS

Implement Raft, including leader election, log replication and persistence, and make it survive a partition test. Build a distributed rate limiter. Then write full design documents for five well-known systems and have someone senior tear them apart.

Production, infrastructure and reliability

3–4 months 220 hours Requires stages 05, 06, 10

Code that is not running is not doing anything. This stage covers the distance between your laptop and a system that stays up, and it is where engineers who can operate what they build become disproportionately valuable.

WHAT YOU LEARN

- Linux in production: processes, systemd, users and permissions, resource limits, log locations
- Diagnosing a sick box: **top**, **iostat**, **ss**, **strace**, **perf**, flame graphs
- Containers: images, layers, Dockerfiles, multi-stage builds, Compose, registries
- Orchestration: pods, deployments, services, ingress, config maps, secrets, autoscaling
- Continuous integration and delivery: pipelines, caching, artefacts, gated merges
- Deployment strategies: rolling, blue-green, canary, and safe rollback
- Infrastructure as code: Terraform, state, modules, drift, plan review as code review
- One cloud provider in depth: compute, object storage, networking and VPCs, IAM, managed databases
- Observability: metrics with Prometheus, structured logs, distributed tracing with OpenTelemetry
- Dashboards that answer a question, and alerts that a human should actually wake up for
- Service level indicators, objectives and error budgets
- Incident response: severity, comms, mitigation before diagnosis, blameless postmortems
- Capacity planning, load testing, and reading a latency histogram rather than an average
- Cost as an engineering constraint; the bill is a design document
- Backups, restores you have actually tested, and disaster recovery objectives

RESOURCES

REFERENCE DevOps Roadmap Milan Milanovic · tool landscape with resources

BOOK Systems Performance Brendan Gregg · the reference for performance work on Linux

BOOK The Phoenix Project Kim, Behr & Spafford · why delivery practice is an organisational problem

SHIP THIS

Take the product from stage 10 and put it on real infrastructure defined entirely in code, with a deployment pipeline, metrics, traces, two meaningful alerts, and a runbook. Then break it deliberately and write the postmortem.

Security engineering

6–8 weeks 120 hours Woven through every stage above

Security is not a stage you complete, it is a lens you acquire. Concentrate it here once so that it becomes reflex, then apply it to everything you have already built. The exercises below are on deliberately vulnerable targets, which is the only place to practise.

WHAT YOU LEARN

- Threat modelling: assets, trust boundaries, attacker goals, STRIDE
- The OWASP Top Ten in practice: broken access control, injection, XSS, SSRF, misconfiguration
- Input validation and output encoding as distinct defences with distinct placement
- Authentication done properly: Argon2 or bcrypt, salting, MFA, secure session lifecycle
- Authorisation bugs, which are the most common serious flaw in real applications
- Applied cryptography: symmetric and asymmetric, hashing, signing, key exchange, TLS
- Why you use a vetted library and never design your own scheme
- Secrets management, key rotation, and keeping credentials out of source control
- Supply chain: dependency pinning, lockfiles, vulnerability scanning, provenance
- Browser security model: same-origin policy, CORS, CSP, cookie attributes, CSRF tokens
- Privacy engineering: data minimisation, retention, PII handling, GDPR obligations
- Secure defaults, least privilege, and defence in depth as design principles

RESOURCES

COURSE PortSwigger Web Security Academy free, hands-on labs · the best security training available at any price

REFERENCE OWASP Top Ten plus the OWASP Cheat Sheet Series

BOOK

Crypto 101 Laurens Van Houtven · cryptography for engineers who must use it

BOOK

The Web Application Hacker's Handbook Stuttard & Pinto · the attacker's view of your stage 10 project

SHIP THIS

Complete the PortSwigger labs for access control, injection and XSS. Then write a threat model for your own product, fix everything it surfaces, and add a dependency scan to the pipeline.

AI engineering

2-3 months 160 hours Requires stages 03, 09, 10

Language models are now a component you integrate, like a database or a payment provider, and integrating them well is an engineering discipline with its own failure modes. The scarce skill is not prompting. It is evaluation: knowing whether the thing you shipped is actually working.

WHAT YOU LEARN

- How a language model works to a useful depth: tokens, context windows, attention, sampling
- Embeddings and vector similarity; what they represent and where they mislead
- Prompt engineering as interface design: structure, examples, decomposition, constrained output
- Structured outputs and schema-validated responses your code can rely on
- Retrieval-augmented generation: chunking, indexing, hybrid search, reranking, citation
- Tool use and function calling; agent loops, and recognising when an agent is the wrong shape
- Evaluation: golden datasets, task-specific metrics, model-as-judge, regression suites in CI
- Guardrails: prompt injection, jailbreaks, PII redaction, output filtering, human review gates
- Cost and latency engineering: model routing, caching, batching, streaming, budgets per request
- Observability for non-deterministic systems: tracing, sampling, replay, drift detection
- Fine-tuning and distillation, and the narrow cases where they beat better prompting
- Working with coding agents: writing the spec, reviewing the diff, keeping ownership of the code
- Classical machine learning literacy: train and test splits, overfitting, metrics, baselines
- The ethics that are also engineering: bias, attribution, disclosure, informed consent

RESOURCES

-
- COURSE** **Neural Networks: Zero to Hero** Andrej Karpathy · build a transformer from scratch
-
- BOOK** **AI Engineering** Chip Huyen · the systems view of building on foundation models
-
- REFERENCE** **Anthropic documentation** tool use, structured output, prompt caching, agent patterns
-
- COURSE** **Machine Learning Crash Course** Google · enough classical ML to be literate

SHIP THIS

One AI feature inside the product from stage 10, with a golden dataset of at least fifty cases, an automated eval that runs in CI, tracing, a cost ceiling per request, and a documented fallback for when the model fails.

The eval suite is the deliverable. The feature is the excuse to build it.

You can be trusted with a system

Roughly 2,720 hours. You can design a system, build it, secure it, deploy it, operate it, and explain every decision in it. From here the curriculum stops being a sequence and becomes a direction.

Expert practice

Continuous The rest of it Requires everything above

Expertise is not one more subject. It is depth in a chosen area, taste built from having been wrong, and the ability to move other engineers. The technical half of this is the easy half.

WHAT YOU PRACTISE

- Pick one track below and go three levels deeper than your team needs
- Read primary sources: Dynamo, Bigtable, MapReduce, Spanner, Raft, The Google File System
- Read production source code of tools you use daily, and fix something in one
- Contribute to open source until you are trusted with review, not just patches
- Write design documents that survive contact with sceptical senior engineers
- Lead a project end to end: scoping, sequencing, risk, stakeholders, landing it
- Mentor deliberately. Teaching is the fastest way to find the holes in your own model
- Learn the business: what your company sells, what it costs, where engineering moves the number
- Develop taste in trade-offs, which means being able to argue both sides honestly
- Write publicly. A body of written work compounds more than any certification
- Cultivate the boring virtues: reliability, clear communication, finishing things

RESOURCES

- | | |
|-----------|---|
| BOOK | The Staff Engineer's Path Tanya Reilly · the most useful book about senior technical work |
| <hr/> | |
| BOOK | Staff Engineer Will Larson · guides free online |
| <hr/> | |
| COMMUNITY | Papers We Love a curated entry point into the literature |
| <hr/> | |
| BOOK | The Mythical Man-Month Fred Brooks · fifty years old and still describing your project |

SHIP THIS

A body of work rather than a project: a maintained open-source contribution, a written archive others cite, and at least one system in production that other

engineers depend on and you did not build alone.

Pick one after milestone B, not before. Breadth first makes the depth mean something, and every track below assumes the whole foundation underneath it.

Backend and distributed systems

BUILD THE THING THAT STAYS UP

The default track, and the one with the most demand. Depth in stages 08 and 11.

- Storage engine internals
 - Consensus and replication in practice
 - Stream processing at scale
 - A systems language: Go or Rust
-

AI engineering

MAKE NON-DETERMINISTIC SYSTEMS DEPENDABLE

The fastest-growing track. Depth in stages 03 and 14.

- Retrieval and ranking quality
 - Evaluation and measurement design
 - Agent architectures and tool design
 - Inference cost and latency engineering
-

Product and frontend

OWN WHAT THE USER ACTUALLY TOUCHES

Underrated, and the track closest to the business. Depth in stages 09 and 10.

- Rendering and interaction performance
 - Accessibility and design systems
 - Offline, sync and local-first data
 - Experimentation and measurement
-

Platform and infrastructure

BUILD WHAT OTHER ENGINEERS BUILD ON

For people who like leverage more than features. Depth in stages 05 and 12.

- Kubernetes internals and operators
 - Developer experience and build systems
 - Networking, service meshes, eBPF
 - Reliability engineering and capacity
-

Low-level systems

BUILD THE LAYERS EVERYONE ASSUMES

Smallest job market, highest barrier, deepest satisfaction. Depth in stages 04, 05 and 07.

- Kernels and drivers
 - Database and storage engines
 - Compilers and runtimes
 - Embedded and real-time systems
-

Security engineering

FIND IT BEFORE SOMEONE ELSE DOES

Two distinct careers, offensive and defensive. Depth in stage 13.

- Application and cloud security
 - Detection and incident response
 - Cryptographic engineering
 - Supply chain and software provenance
-

These are the failure modes that end self-directed study. Each one feels like progress while it is happening.

Tutorial hell

Following along with a video produces the feeling of competence and none of it. The test is whether you can build a variation of the thing without the video. If not, you have not learned it.

Collecting languages and frameworks

Six languages at a shallow depth is worth less than one at depth, and hiring managers stopped being impressed by long tool lists some years ago.

Skipping the foundations because the assistant writes the code

An engineer who cannot evaluate generated code is a liability, and the market prices that correctly. Fundamentals are what make the leverage safe.

Reading the advanced book too early

Designing Data-Intensive Applications is transformative after you have built and operated something, and forgettable before. The same is true of most of the systems canon.

Optimising only for interviews

Grinding problems without building systems produces candidates who pass the screen and fail the job, which the industry has now adjusted for.

Never shipping to a real user

A portfolio of unfinished projects signals the one thing employers screen hardest for. Finish three things completely rather than starting twenty.

Learning alone the whole way

Code review, pair programming and public work correct errors you cannot see from the inside. Find people, even if that means open source rather than a job.

Treating this document as a queue

It is a map. Loop back, re-read, and let a real problem pull you into a stage early when it does.

THE SHELF

If you read nothing else, read the two in the first row. Between them they cover the machine underneath your program and the systems built on top of it, and they carry more return per hour than anything else on this list.

TIER	BOOK	AUTHOR	READ AT
Essential	Computer Systems: A Programmer's Perspective	Bryant & O'Hallaron	Stage 04
Essential	Designing Data-Intensive Applications	Kleppmann	Stage 11
Craft	A Philosophy of Software Design	Ousterhout	Stage 09
Craft	The Pragmatic Programmer	Hunt & Thomas	Stage 09
Craft	Refactoring	Fowler	Stage 09
Craft	Working Effectively with Legacy Code	Feathers	Stage 09
Theory	The Algorithm Design Manual	Skiena	Stage 02
Theory	Structure and Interpretation of Computer Programs	Abelson & Sussman	Stage 01
Systems	Operating Systems: Three Easy Pieces	Arpaci-Dusseau	Stage 05
Systems	Computer Networking: A Top-Down Approach	Kurose & Ross	Stage 06
Systems	Crafting Interpreters	Nystrom	Stage 07
Systems	The Elements of Computing Systems	Nisan & Schocken	Stage 04
Practice	Site Reliability Engineering	Google	Stage 12
Practice	Systems Performance	Gregg	Stage 12
Practice	AI Engineering	Huyen	Stage 14
Career	The Staff Engineer's Path	Reilly	Stage 15
Career	The Mythical Man-Month	Brooks	Stage 15

Sequence and hour budgets synthesised from Teach Yourself Computer Science, the Open Source Society University curriculum, published university course structures at MIT, Berkeley, Stanford and Carnegie Mellon, and 2026 hiring signal for entry and mid-level engineering roles.