My Learning Roadmap

My learning roadmap

This is the route I am following.

I arrived at it after comparing 2 earlier roadmaps, auditing 48 books in my local library, and looking at the dependencies between the subjects I want to learn. It keeps the broad range that interests me, but puts the work in an order I can defend. The rules that keep this roadmap mine protect the choices that must survive future revisions.

The rule I am following

I will complete 1 stage before moving to the next. Inside the current stage, I can choose whichever current book suits my energy.

```
read -> predict -> implement -> compile/run -> test -> inspect -> explain -> co
```

I can keep 2 or 3 technical books active inside a stage. I do not need permanent lanes, percentage allocations, daily page quotas, or equal progress across them.

How I will use the books

- **Primary textbook:** I will follow the teaching sequence and practise the central material.
- **Project book:** I will build the projects and change them so I am not only copying.
- **Deep reference:** I will read the central explanations and consult the rest when the work calls for it.
- **Pocket or manual resource:** I will make a structured pass, try unfamiliar material, and keep it for lookup.

- **Light companion:** I will read it normally without turning it into another technical obligation.
- **Optional read:** I can read it if I want, but it will never block the next stage.

The complete route

```
flowchart LR
    subgraph foundations["Foundations"]
        direction TB
        s1["1. C, shell, Python DSA"] ⟶ s2["2. Repeated building"]
        s2 ⟶ s3["3. Modern C and Linux internals"]
    end

    subgraph systems["Systems"]
        direction TB
        s4["4. C++ and Linux APIs"] ⟶ s5["5. Larger programs and de
        s5 ⟶ s6["6. Memory, kernel, embedded"]
        s6 ⟶ s7["7. Larger C++ systems"]
        s7 ⟶ s8["8. Linux security"]
    end

    subgraph ai["ML and AI"]
        direction TB
        s9["9. ML and deep-learning foundations"] ⟶ s10["10. Build
        s10 ⟶ s11["11. Native ML and CUDA"]
        s11 ⟶ s12["12. AI performance"]
        s12 ⟶ s13["13. Reasoning and post-training"]
    end

    s3 ⟶ s4
    s8 ⟶ s9
```

Stage 1: C, shell, Python DSA, and Unix context

The books

1. **C Programming: A Modern Approach, 2nd Edition** by K. N. King
2. **The Linux Command Line, 3rd Edition** by William Shotts
3. **A Common-Sense Guide to Data Structures and Algorithms in Python, Volume 1** by Jay Wengrow
4. **UNIX: A History and a Memoir** by Brian Kernighan, as a light companion

Why I am starting here

C gives me the programming and memory foundation. The shell gives that work a real environment where I can compile, run, inspect, and debug it. Python lets me build algorithmic judgement without turning every DSA exercise into a second lesson in C memory management. The Unix memoir gives me the history connecting these tools without adding another technical burden.

Where I am

I am at Chapter 6, Loops, in King. I have implemented and verified all 12 programming projects in that chapter. My next step is to review the chapter and solve its exercises independently.

What I will do

- Complete the King exercises and projects that introduce a new idea, while skipping only obvious repetition.
- Type and use Shotts's commands rather than reading them passively.
- Implement Wengrow's main structures and algorithms in Python.
- Read the memoir normally.

What I need before Stage 2

I should be able to write, compile, test, and debug small multi-file C programmes from the terminal. I should use the shell comfortably and explain the behaviour and cost of the main DSA Volume 1 topics.

Stage 2: Repeated building in C, shell, and Python DSA

The books

1. **Tiny C Projects** by Dan Gookin
2. **The Ultimate Linux Shell Scripting Guide** by Donald A. Tevault
3. **A Common-Sense Guide to Data Structures and Algorithms in Python, Volume 2** by Jay Wengrow

King, Shotts, and Wengrow Volume 1 give me the pieces. This stage makes me use them often enough that they stop feeling borrowed.

What I will do

- Build the useful Tiny C projects, then rebuild several central ideas without following the book line by line.
- Turn command-line knowledge into shell scripts that solve real tasks.
- Continue the Python implementations and exercises from DSA Volume 2.
- Solve matching LeetCode problems as the topics appear.
- Complete **LeetCode 75 in Python** after Volume 2.

The optional book

I may read **Algorithmic Thinking, 2nd Edition** by Daniel Zingaro after this stage if I want more C-based algorithm practice. Its selective, problem-first structure does not replace the Python DSA sequence, and it will not block

Stage 3.

What I need before Stage 3

I should be able to finish small C programmes, automate useful Linux work, implement the advanced topics from the Python DSA sequence, and solve judged problems while stating the time and space cost of my approach.

Stage 3: Modern C and Linux internals

The books

1. **Modern C, 3rd Edition** by Jens Gustedt
2. **How Linux Works, 3rd Edition** by Brian Ward
3. **Linux Pocket Guide, 4th Edition** by Daniel J. Barrett, as a structured pass and reference

King gives me a strong C89 and C99 foundation. *Modern C* carries that foundation through C23, modern types, failure handling, memory, threads, and atomics. Ward explains the machine around the language: devices, boot, processes, filesystems, networking, libraries, services, and virtualisation. The Pocket Guide fills out my command knowledge without pretending to be another textbook.

A deliberate omission

I am not using **Computer Systems: A Programmer's Perspective** in this stage. I dislike its textbook style, and owning it does not override that decision.

I accept the cost. My coverage of assembly, cache architecture, linking internals, and processor pipelines will be less systematic. Parts of those subjects will still appear in modern C, Linux system programming, kernel

work, embedded work, CUDA, and performance profiling. If the gap later becomes a real obstacle, I can look for a different architecture resource that teaches in a way I can sustain.

What I will do

- Move quickly through familiar C syntax and slow down for C23, object lifetime, generic programming, memory-model, thread, and atomic material.
- Inspect `/proc`, `/sys`, `/dev`, boot logs, processes, mounts, libraries, routes, sockets, and services on a Linux system.
- Try unfamiliar Pocket Guide commands without trying to memorise the book.

What I need before Stage 4

I should be able to explain the major changes from C99 to C23 and the structure of a running Linux system, including boot, devices, processes, filesystems, services, libraries, networking, and virtualisation.

Stage 4: Modern C++, Linux userspace APIs, and security-oriented Linux

The books

1. **Introducing C++** by Frances Buontempo
2. **Linux System Programming, 2nd Edition** by Robert Love
3. **Linux Basics for Hackers, 2nd Edition** by OccupyTheWeb

I want to learn C++ as its own language rather than write C with classes. At the same time, I will begin using the Linux APIs beneath ordinary command-line work and turn my existing Linux knowledge towards security.

What I will do

- Learn references, RAII, classes, lifetime, containers, algorithms, move semantics, smart pointers, templates, and modern vocabulary types.
- Compile Love's examples involving files, processes, signals, threads, memory, timers, and event APIs.
- Move quickly through repeated Linux basics in *Linux Basics for Hackers* and practise its security-specific networking, scripting, logging, and Kali material.

What I need before Stage 5

I should be able to build an ordinary multi-file modern C++ programme, understand the main Linux userspace APIs at tutorial depth, and operate Linux from a security-oriented perspective.

Stage 5: C++ projects, deep Linux APIs, and Kali methodology

The books

1. **Learn C++ by Example** by Frances Buontempo
2. **The Linux Programming Interface** by Michael Kerrisk
3. **Learning Kali Linux, 2nd Edition** by Ric Messier

This stage turns the introductions into larger work. The C++ projects make the language concrete. Kerrisk takes the userspace API work much deeper than Love. Messier adds a method for reconnaissance, traffic analysis, vulnerability work, evidence, and reporting.

What I will do

- Complete the C++ projects and make small original changes to them.

- Use TLPI as the deep continuation of Robert Love by reading the central explanations, compiling clarifying examples, and checking current man pages.
- Learn the Kali testing process inside a lab I own and can safely break.

What I need before Stage 6

I should be able to build practical C++ programmes, write Linux systems programmes using the right userspace APIs, and perform a documented introductory security assessment inside an isolated lab.

Stage 6: Memory, kernel programming, and bare-metal execution

The books

1. **C++ Memory Management** by Patrice Roy
2. **Linux Kernel Programming, 2nd Edition** by Kaiwan N. Billimoria
3. **Bare-Metal Embedded C Programming** by Israel Gbati

This is where memory, privilege, and hardware stop being distant concepts. I will study ownership and allocation in C++, cross into the Linux kernel inside a recoverable VM, and bring up a microcontroller without an operating system underneath me.

What I will do

- Study object lifetime, ownership, RAII, allocation, arenas, containers, allocators, and undefined behaviour in C++.
- Build and test kernel modules only inside a disposable, recoverable VM.
- Use the exact supported embedded board and work through registers, memory-mapped I/O, startup code, linker scripts, interrupts, timers,

serial interfaces, and peripherals.

What I need before Stage 7

I should be able to reason about memory and lifetime, build and inspect basic Linux kernel modules safely, and bring up a small bare-metal programme from reset to peripheral operation.

Stage 7: Larger and asynchronous C++ systems

The books, in order

1. **C++ Software Design** by Klaus Iglberger
2. **Asynchronous Programming with C++** by Javier Reguera-Salgado and Juan Antonio Rufes

By this point I should have enough C++ code behind me for software design to mean something. I can apply the ideas to programmes I have already built, then add asynchronous behaviour without treating concurrency as a collection of isolated syntax examples.

What I will do

- Apply dependency management, abstraction, value semantics, and design principles to an existing programme.
- Add threads, futures, promises, coroutines, cancellation, error propagation, and clean shutdown to real code.
- Use tests, sanitizers, and repeated runs to expose concurrency failures.

What I need before Stage 8

I should be able to design, implement, test, and maintain a medium-sized modern C++ application with clear ownership and asynchronous behaviour.

Stage 8: Offensive and defensive Linux security

The books

1. **The Ultimate Kali Linux Book, 3rd Edition** by Glen D. Singh
2. **Mastering Linux Security and Hardening, 3rd Edition** by Donald A. Tevault

I want to understand both sides of Linux security. Finding a weakness without knowing how to harden, monitor, and recover a system gives me half the picture.

What I will do

- Build the prescribed isolated Kali and vulnerable-target lab.
- Perform reconnaissance, vulnerability assessment, exploitation, post-exploitation, web, wireless, and Active Directory exercises only where I have explicit permission.
- Apply authentication, permissions, firewalling, auditing, logging, isolation, hardening, and detection to Linux systems.

What I need before Stage 9

I should be able to explain how Linux weaknesses are found and how systems are hardened, monitored, and recovered.

Stage 9: Practical machine learning, statistics, and deep-learning foundations

The books and their jobs

1. **Machine Learning with PyTorch and Scikit-Learn** by Sebastian Raschka, Yuxi Liu, and Vahid Mirjalili, as the main practical text
2. **Understanding Deep Learning** by Simon J. D. Prince, for theory
3. **Statistics Every Programmer Needs** by Gary Sutton, for mathematical support
4. **Why Machines Learn** by Anil Ananthaswamy, as a lighter intuition companion
5. **The Little Book of Deep Learning** by Francois Fleuret, as a compact reference

The earlier version of my roadmap reached LLMs too quickly. I need enough machine learning, statistics, and deep learning to know whether an experiment is sound before I start building larger models.

What I will do

- Build classical ML pipelines and understand validation, leakage, metrics, preprocessing, optimisation, and model selection.
- Train and inspect neural networks in Python and PyTorch.
- Use the theory and statistics books to explain what the experiments are doing instead of treating notebooks as recipes.

What I need before Stage 10

I should be able to design and evaluate a sound ML experiment, explain neural-network training and optimisation, and diagnose basic statistical, modelling, and implementation failures.

Stage 10: Build a large language model from scratch

The book

Build a Large Language Model (From Scratch) by Sebastian Raschka

I will build the tokenizer, embeddings, attention mechanism, transformer, training loop, checkpointing, evaluation, and fine-tuning path at a scale my hardware can support.

What I need before Stage 11

I should be able to explain and implement the central transformer and LLM pipeline rather than treat a pretrained model as a sealed box.

Stage 11: Native ML, CUDA, and GPU performance

The books, in order

1. **Hands-On Machine Learning with C++, 2nd Edition** by Kirill Kolodiazhnyi, used selectively as the native-ML bridge
2. **CUDA Programming Guide, Release 13.3** by NVIDIA, used through guided sections and as a reference
3. **Understanding Latency Hiding on GPUs** by Vasily Volkov, read after I have basic CUDA profiling experience

The C++ book connects the Python ML work to native runtimes. The CUDA guide gives me the programming model. Volkov's work belongs later, when latency, occupancy, and scheduling describe problems I have already seen in a profiler.

What I will implement

- vector operations,
- reductions and scans,
- tiled matrix multiplication,
- convolution,
- softmax,
- layer normalisation,
- and 1 fused operation.

For every kernel, I will:

1. establish a correct CPU or framework baseline,
2. validate the numerical result,
3. benchmark it,
4. profile it,
5. classify the bottleneck,
6. change 1 thing,
7. and measure again.

I will study memory transfers, coalescing, shared memory, divergence, occupancy, register pressure, synchronisation, launch overhead, and end-to-end speedup.

What I need before Stage 12

I should be able to write, debug, profile, and improve CUDA kernels while explaining whether compute, bandwidth, latency, occupancy, synchronisation, launch overhead, or host-device transfer is limiting them.

Stage 12: AI systems performance engineering

The book

AI Systems Performance Engineering by Chris Fregly

By this point I should understand the model, the native runtime, and the GPU kernels well enough to profile the whole system rather than optimise whichever layer happens to be familiar.

I will profile 1 model across Python, the framework, compiler, runtime, CPU, GPU, memory, batching, caching, quantisation, serving, and distributed layers.

```
define workload
-> measure baseline
-> profile the full stack
-> identify the bottleneck
-> change one layer
-> repeat the benchmark
-> keep or reject the change
```

I will build at least 1 native C++ and CUDA extension, or an equivalent custom operator.

What I need before Stage 13

I should be able to diagnose and improve an AI workload using latency, throughput, memory, accuracy, warm-up, variance, and cost measurements rather than slogans.

Stage 13: Reasoning models, reinforcement learning, and RLHF

The books, in order

1. **Mathematical Foundations of Reinforcement Learning** by Shiyu Zhao, using the foundations relevant to the work ahead
2. **Build a Reasoning Model (From Scratch)** by Sebastian Raschka, as the main applied book
3. **Reinforcement Learning from Human Feedback** by Nathan Lambert, for post-training depth

I will study evaluation, inference-time scaling, verification, search, reward modelling, policy optimisation, distillation, preference data, and RLHF. I will reproduce experiments at a scale my hardware can support.

What this stage should leave me with

I should be able to explain and implement the main reasoning and post-training ideas while separating improvements to the model from improvements to the system running it.

Books I left outside the default route

The complete 48-book audit records every decision. These are the choices most likely to be questioned later:

- *Computer Systems: A Programmer's Perspective* is not selected because I dislike its style.
- *Algorithmic Thinking* is optional.
- *Algorithms, 4th Edition* and Goodrich's DSA text repeat the role of the Python DSA sequence I chose.

- *Linux: The Comprehensive Guide* overlaps heavily with Shotts, Ward, Love, TLPI, the shell-scripting book, and the selected security books.
- *Modern C++ Programming Cookbook* repeats reference material already covered by the structured C++ route.
- *Hacking and Security* lost its place to the more focused Kali and hardening books.
- The Rust and database books are outside my present goal.

How long I expect this to take

This is a large, multi-domain curriculum. My current estimate is roughly **2,500 to 4,000 productive hours**. TLPI, kernel work, embedded hardware, security labs, ML experiments, CUDA profiling, and reasoning-model work create most of the uncertainty.

- With near-full-time sustained study, I estimate 15 to 24 months for a strong working foundation.
- With serious part-time study, I estimate 3 to 5 years.
- Broad mastery will continue well beyond the reading plan.

I cannot shorten the calendar by skipping implementation, debugging, testing, or measurement. That only moves the missing work into the future.

When I will install things

I will set up tools when their stage begins. Kernel work needs a recoverable VM. Embedded work needs the supported board. Security work needs an isolated lab. CUDA and performance tooling can wait until Stage 11.