What If Learning Software Engineering Felt Like Joining a Real Team?

> *Watching someone build software is not the same as learning how to work inside software. I started building noetherkin to make learning engineering feel more like joining a real team. Updated for noetherkin's current CLI and skills architecture.*

AUTHOR

Prince Agyei Tuffour

PUBLISHED

September 20, 2026

READING TIME

24 min

IN THIS PIECE

01 The problem I'm trying to solve
02 Then AI complicated things
03 The idea: simulate a tech company around a real codebase
04 The company simulation
05 Debugging should actually feel like debugging
06 The simulator remembers what you actually demonstrated
07 Levels, without turning it into a video game
08 Real open source projects as the curriculum
09 The same skills keep showing up
10 A typical day with it
11 How the workspace is laid out
12 Why not just ask ChatGPT to be my manager?
13 Built to work with any AI tool
14 Where I want this to go
15 What this isn't
16 What I'm building first
17 Questions I don't have good answers to yet
18 Building in public

A note before this one: noetherkin has moved a lot since I first wrote it, from a private concept sketch to a public, released CLI. The motivation below is unchanged, but some of the mechanics are not: the five roles I describe became 21 separate, installable Agent Skills, and a trusted local CLI now mediates everything instead of a chat session. The companion piece covers the current architecture in two parts.

I've been thinking a lot lately about how I want to keep learning software engineering.

There's a long list of things I want to get seriously good at over the next few years: advanced Python, Java and Spring, C++, CUDA (the language and toolkit for programming GPUs), cloud infrastructure, Docker and Kubernetes (the tools that let you package software and run it reliably across many machines), distributed systems, machine learning systems, LLM serving, maybe Go.

The obvious way to tackle that list is familiar to anyone who's tried to learn to code. Find a course. Watch twenty hours of video. Follow along while someone else builds an app. Do a few exercises. Move to the next course.

There's nothing wrong with that. Courses have taught me a lot. But I kept running into the same nagging thought:

Watching someone build software is not the same as learning how to work inside software.

And working inside software is increasingly what I want to get better at. I don't just want to know how to spin up another REST API from scratch. I want to be able to open a repository I've never seen before and figure out what it does, how it's organized, where requests come in, how data moves through it, why certain design choices exist, where it tends to break, how its tests are structured, how its pieces talk to each other, how it gets deployed, how people debug it when something goes wrong, and how to change it without breaking everything else.

In short, I want the experience to feel less like another programming course and more like joining an engineering team.

That thought turned into an open source project I've started building, called **noetherkin**: an agent-agnostic apprenticeship protocol and CLI for learning software engineering by working inside real codebases. It's public now, released under Apache 2.0, currently at v0.2.0.

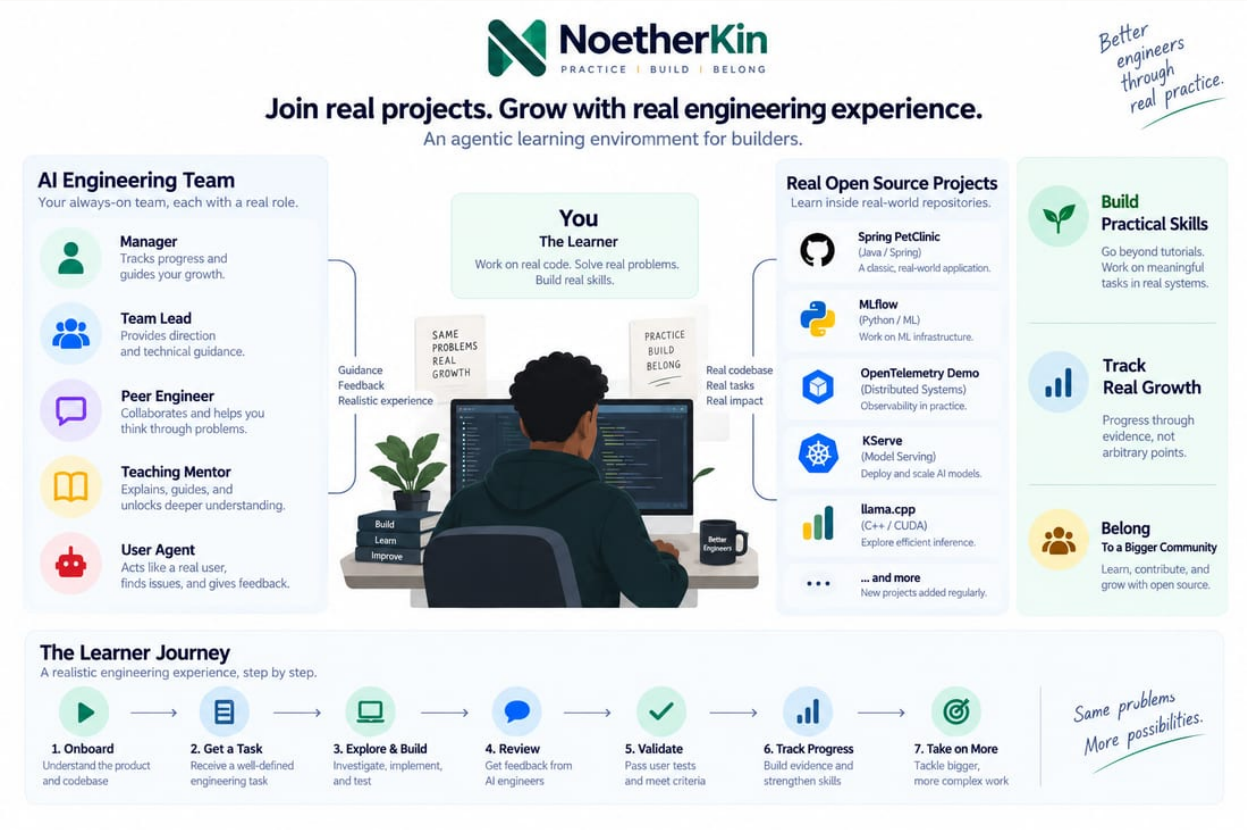


Figure. A simulated engineering team around you, working inside a real open source project.

01

The problem I'm trying to solve

Most programming education starts from a blank file. You learn a concept, see an example, and eventually build something. That's useful, but it's not how professional engineering usually works.

You join a company and inherit a repository with hundreds or thousands of files, abstractions you didn't design, decisions from years ago that nobody explained to you, tests written by people you've never met, dependencies you've never touched, and bugs whose causes aren't obvious.

Your first real task probably isn't "build a new application." It's more likely to be something like: *users occasionally get duplicate notifications, figure out why.* Or: *add support for this workflow without breaking the integrations that already depend on the old one.* Or: *this service gets slow under load, go find out why.* Sometimes it's even vaguer than that: *we're not sure where the problem is yet.*

Those are very different learning experiences from following a tutorial, and they exercise a

different set of muscles: navigating an unfamiliar codebase, debugging, reading documentation, understanding architecture, writing meaningful tests, reasoning about performance, explaining tradeoffs, and changing systems you didn't build. Those are exactly the skills I want to practice on purpose instead of picking up by accident.



Figure. Courses start from a blank file. The apprenticeship starts from a codebase you did not write.

02

Then AI complicated things

AI coding tools are genuinely useful. I use them constantly. But they create an interesting problem for anyone trying to *learn*, as opposed to just ship.

If every time I hit a wall I hand it to an AI agent and ask it to write the fix, I'll finish projects much faster while understanding a lot less. That's a bad trade if the point of the project is to become a better engineer.

There's a real difference between "I used AI to understand why this service uses dependency injection" and "I asked AI to implement the service." Between "help me read this stack trace" and

"fix this bug for me." Both are legitimate uses of AI, but when my goal is to grow as an engineer, I want the AI acting like a senior engineer sitting next to me, not an autonomous one replacing me.

That became one of the central rules behind noetherkin: **the learner stays the engineer.** The AI can surround you with guidance, feedback, review, and a simulated engineering process, but it shouldn't quietly take the work off your plate.



Figure. The learner still investigates, designs, and ships. The AI team coaches, reviews, and remembers.

03

The idea: simulate a tech company around a real codebase

Instead of building toy projects specifically for the simulator, noetherkin uses existing open source software as the training ground.

Say you want to learn Java and Spring. Instead of spending weeks on isolated tutorial projects, you'd run `noetherkin project select spring-petclinic-microservices --clone-to spring-petclinic`, which clones Spring PetClinic Microservices[2] into your workspace after you confirm, pinned to a specific commit, and the simulator treats you like a developer who just joined

the team. If you'd rather build something from nothing than inherit someone else's repository, noetherkin also ships "forge" projects, specifications for real software (an accessible data table, an eval ledger, a batch ingestion pipeline) that you implement yourself in an empty directory, with nothing cloned and nothing to diff your answer against.

You'd start with something like: *welcome to the team. Before touching production code, let's understand what this product does, how to run it, and how a single request travels through the system.* Then you'd explore the repo, piece by piece rather than all at once, until eventually you get your first real assignment. Maybe something like:

> ***TASK-014*** *Invalid owner registration requests are reaching deeper layers of the system than they should. Investigate the existing validation flow and add proper validation without breaking the behavior that currently works.*

The task comes with context and acceptance criteria, but ideally not a recipe for solving it. You investigate. You read the docs. You dig through the relevant code. You run the tests, form a hypothesis, implement a change, and submit it for review. Then the simulator switches hats: a peer engineer might help you think something through, a team lead reviews your approach, a simulated user tries to break the feature, and eventually a manager looks at the pattern across everything you've done so far.

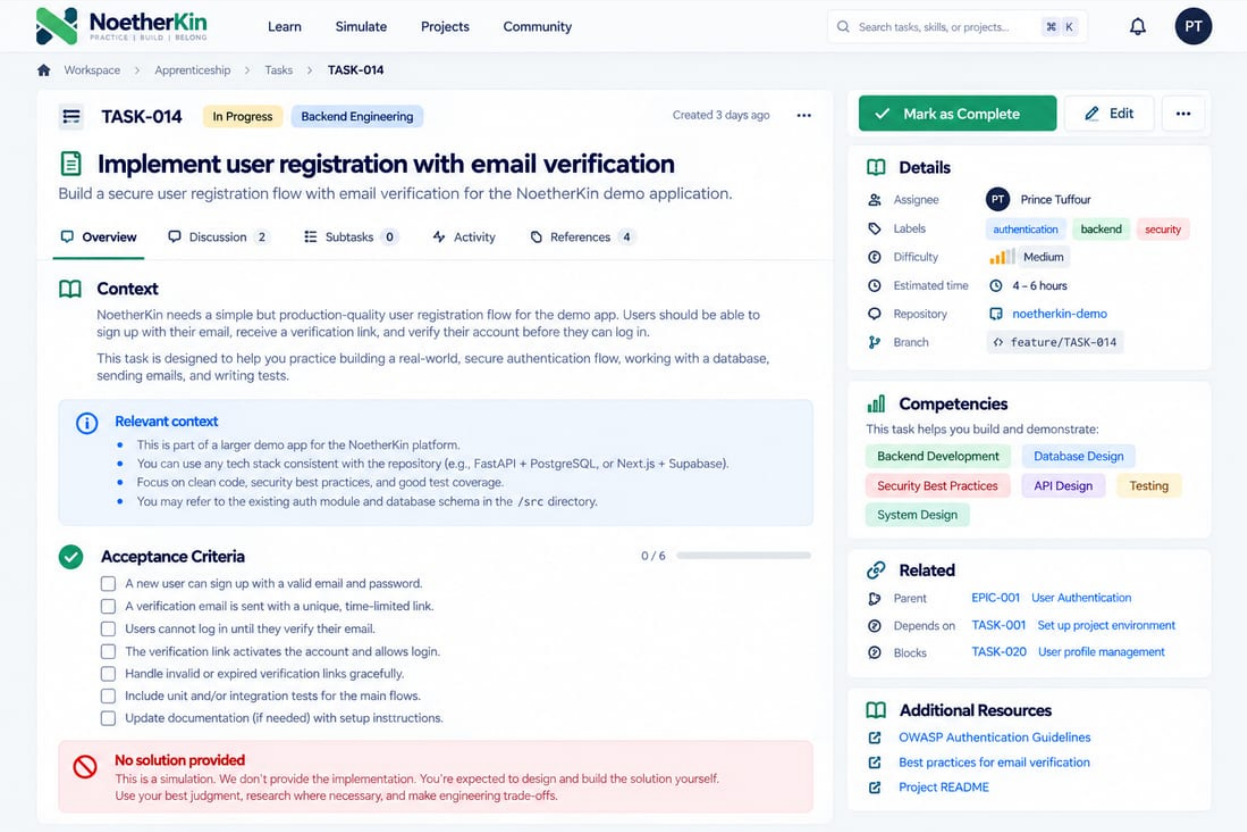


Figure. A first assignment with context and acceptance criteria, but no recipe for the solution.

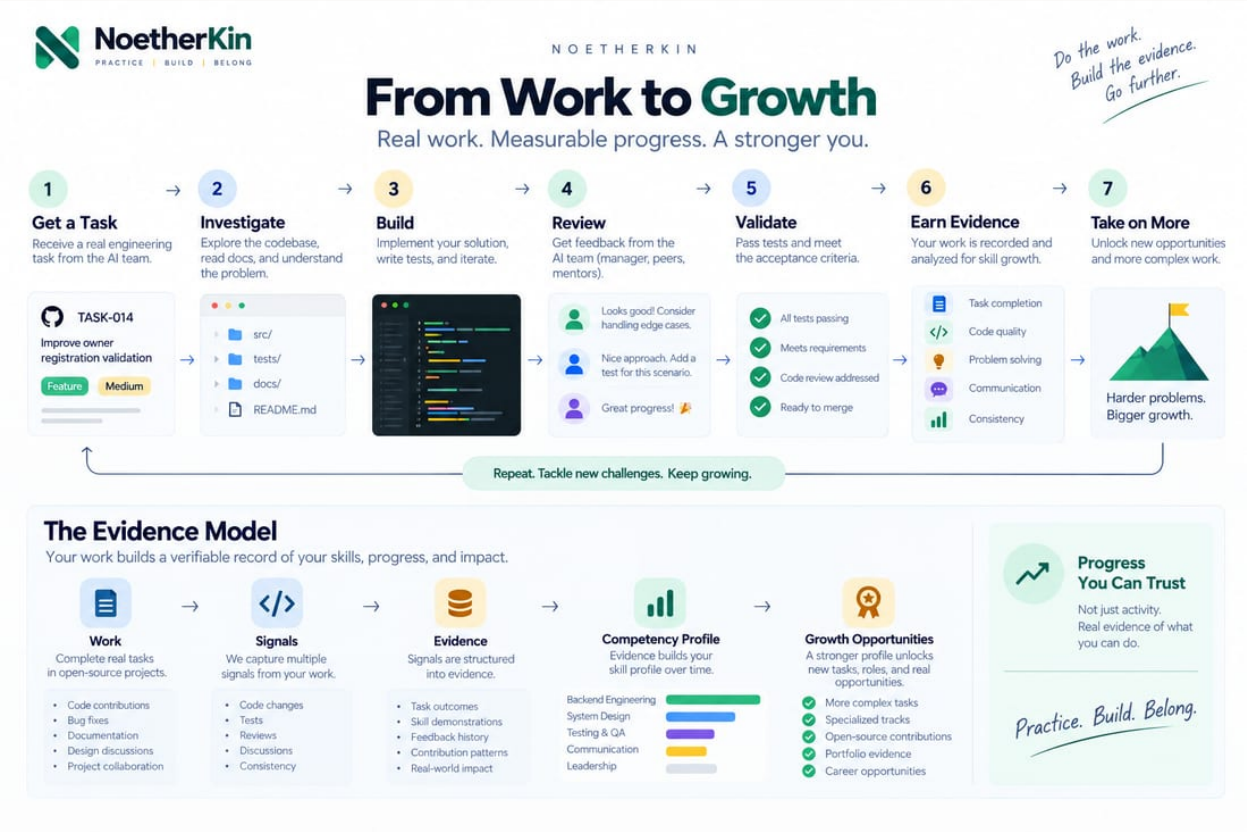


Figure. Work, review, evidence, then a harder assignment. The loop is the curriculum.

The point is that learning happens through repeated cycles of real engineering work, not by consuming lessons in order.

04

The company simulation

The whole thing is built around a simple idea: different AI skills play different roles in an engineering organization. That idea now ships as 21 separate, installable Agent Skills[3], because once I actually built the thing, task assignment, teaching, code review, design review, debugging, incident response, performance review, and promotion review all turned out to need their own contracts, not just their own persona. They install into Claude Code or Codex with one command, and a trusted local CLI, not the chat session, is what actually owns and publishes any of it.

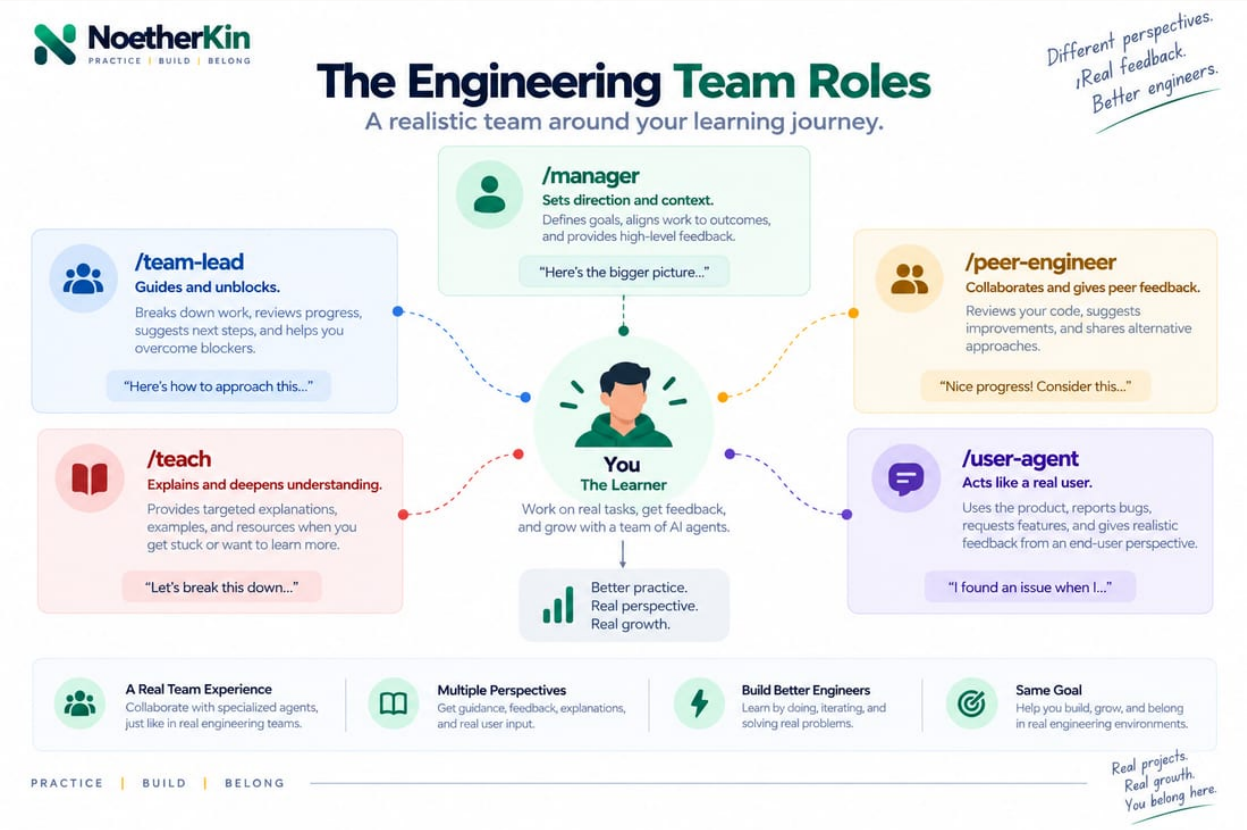


Figure. Manager, team lead, peer, teacher, and user agent, each with a different job.

Onboarding. Works like the onboarding process at a real job. It helps you figure out what the product does, who uses it, how the repo is organized, how to build and run it, how the tests work, and which pieces matter first. It shouldn't dump the entire architecture on you in one giant AI-generated summary. Real understanding builds up gradually, the same way it does when you actually join a team.

Task assignment. Picks or creates an assignment that fits where you currently are. Over time these fall into three buckets: training tasks made specifically to exercise one skill, product tasks that improve your own fork, and upstream tasks, meaning real issues from the actual open source project. That last category is the one I'm most excited about: you could start with simulated work and eventually graduate into contributing to real, live software that other people use.

Teaching. Acts like a mentor rather than a code generator. Instead of handing you the answer, it might say: *start by finding where this interface is implemented, then check the official docs for this API, and pay attention to what happens when the call fails.* If you need more help, the assistance ramps up in stages, roughly: investigate on your own, get pointed at documentation, get a conceptual hint, get asked leading questions, see pseudocode, see a small isolated example, get a partial implementation, and only then a full one. Asking for help isn't failure. Real engineers ask for help

constantly. What matters is what the finished work actually demonstrates you can do.

Peer engineer. Plays the role of a coworker you can talk something through with, like "I'm deciding between adding another abstraction here or extending the one that exists, can you look at this with me?" The peer can push back on assumptions, help you debug, or explain something unfamiliar, but it shouldn't quietly become the one doing your assigned work for you.

Team lead. Gives technical direction: reviewing your designs, assessing code quality, spotting weak points, suggesting your next assignment, and judging whether a task was actually hard enough to prove you can do the thing it was meant to test.

Manager. Looks at the longer arc rather than any single piece of code. Are you getting more independent over time? Can you handle ambiguous work and own it? Are you consistently struggling with testing? Are you strong at building things but weaker at designing them? Are you ready for more scope? The manager can flag you as ready for a performance or promotion review, but it never gets to just declare "you've been here three months, congratulations, you're senior now." That would defeat the entire point.

User agent. This might be my favorite piece. After you implement something, a simulated user tries it, not as a programmer testing edge cases on purpose, but as an actual user: strange inputs, repeated clicks, interrupted requests, empty states, sequences nobody planned for. Instead of quietly fixing the bug, it files something that looks like a real bug report:

> ***BUG-042: Checkout allows duplicate submission*** *Expected: the payment should only be created once. Actual: rapidly submitting the checkout form can create duplicate charges. Impact: users may accidentally be charged more than once.*

Now you have to go investigate. That's a much more interesting way to learn than "Exercise 14: implement idempotency."

05

Debugging should actually feel like debugging

One thing I care a lot about is how bugs get handled. Say a service that normally responds in 180 milliseconds suddenly starts taking 2.8 seconds. The simulator shouldn't immediately tell you "the Redis connection pool is misconfigured, change this setting." It should look more like a real incident:

> ***SEV-2 Incident*** *Checkout latency has increased significantly since the latest deployment. Error*

> *rates are normal. Users are reporting a slow checkout. Investigate.*

You'd have access to logs, metrics, traces, recent deployments, CPU and memory usage, and database stats. From there the process is: observe what's happening, gather evidence, form a hypothesis, design a small experiment to test it, check whether the evidence supports it, and if not, go back and form a new hypothesis. Once you find the actual root cause, you implement a fix, verify it worked, and add something to prevent it from happening again.

That builds a genuinely different engineering muscle than reading about debugging in a course.

06

The simulator remembers what you actually demonstrated

One problem with AI mentorship in general is that it's easy for the whole thing to evaporate into vibes. A model might tell you "you're doing really well," but what does that actually mean? Did today's version of it even remember what you did last week?

noetherkin is built around keeping actual evidence instead of a vague impression. Something like this gets recorded after a task:

> ***EVID-0042** — competency: debugging Observation: learner independently traced a checkout failure across the gateway and payment service and identified incorrect retry behavior. Verified by: reproducing the failing case, analyzing traces, and adding a regression test. Highest level of help needed: pointed to documentation. Signal: strong evidence toward the "independent engineer" level.*

That means a later review can ask "what evidence do we actually have that this person can debug across services," instead of relying on a model's memory of how the conversation felt.

07

Levels, without turning it into a video game

I want the simulator to have a sense of progression, but not the "+20 debugging XP, level 7 engineer unlocked" kind. That's fun, but it doesn't mean much.

Instead, the current design uses behavior-based levels. **Apprentice** works with heavy guidance. **Junior** can complete well-scoped work independently. **Independent Engineer** can navigate

unfamiliar territory, debug across components, make reasonable design calls, and own features with limited help. **Product Engineer** can own bigger chunks of a system and handle ambiguity. **Senior/Systems Engineer** can reason across architecture, reliability, and performance. **Lead Engineer** can set technical direction and help other engineers improve. These are internal to the simulator, not an attempt to copy how any real company levels its engineers.

More importantly, moving up a level should require evidence, not a single lucky win. If I debug one hard problem successfully, that's meaningful, but it doesn't prove I'm consistently good at debugging. So progression weighs things like repetition (did I do it more than once), recency, variety of contexts, how complex the problem was, and how much help I needed. It can also record the honest downside: "strong at debugging backend request flows, still needs help with concurrency issues" is a far more useful sentence than "debugging: 82/100."



Figure. Levels track demonstrated behavior, not experience points.

08

Real open source projects as the curriculum

This might be the part I'm most excited about. Rather than inventing a separate toy repo for every skill, the catalog leans on real, existing open source projects, 87 of them now, organized into 34 learning tracks: Spring PetClinic Microservices[2] or MLflow[4] early on. For distributed and cloud native systems, the OpenTelemetry demo[5] app. For ML systems, KServe[6] and vLLM-[7]. For systems programming, OpenTelemetry's C++ SDK[8] and llama.cpp[9]. For infrastructure, Prometheus[10], Argo CD[11], and etcd[12]. Further out, things like Triton[13] and larger GPU inference platforms. Early on, before a learner is ready to inherit a real codebase's constraints, the seven built-from-empty "forge" projects carry the same first two or three levels instead: you write every line yourself against a shipped specification and acceptance criteria, with no upstream repository to lean on.

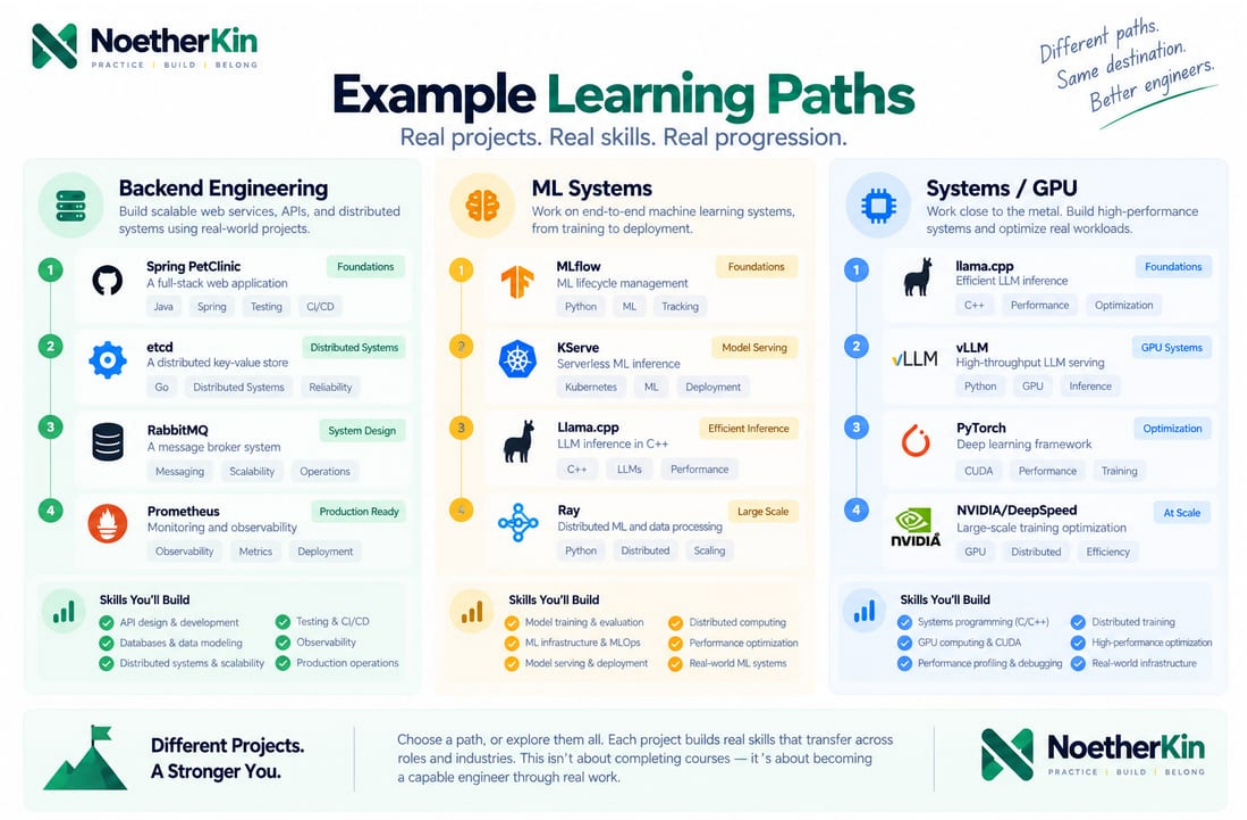


Figure. Different tracks through real projects, not one required ladder.

There's no single required path. Someone interested in backend work might go Spring, then distributed services, then Docker, then Kubernetes, then observability, then Go, then

infrastructure. Someone chasing ML systems might go Python, then MLflow, then PyTorch[14], then model serving, then KServe, then vLLM, then C++, then CUDA. Someone into systems programming would probably take a completely different route.

09

The same skills keep showing up

This is part of why I think the project could end up useful beyond just me. Technologies aren't really things you "finish." Nobody finishes Docker, or Kubernetes, or Python. You keep running into them, and you understand one more layer each time.

MLflow touches Python, API design, databases, and general ML systems knowledge. Spring PetClinic touches Java and Spring, API design, databases, and observability. The OpenTelemetry demo touches distributed systems, observability, and Kubernetes. KServe touches Kubernetes, Go, and ML systems. vLLM touches Python, GPU systems, and model serving. llama.cpp touches C++, GPU systems, and model serving too.

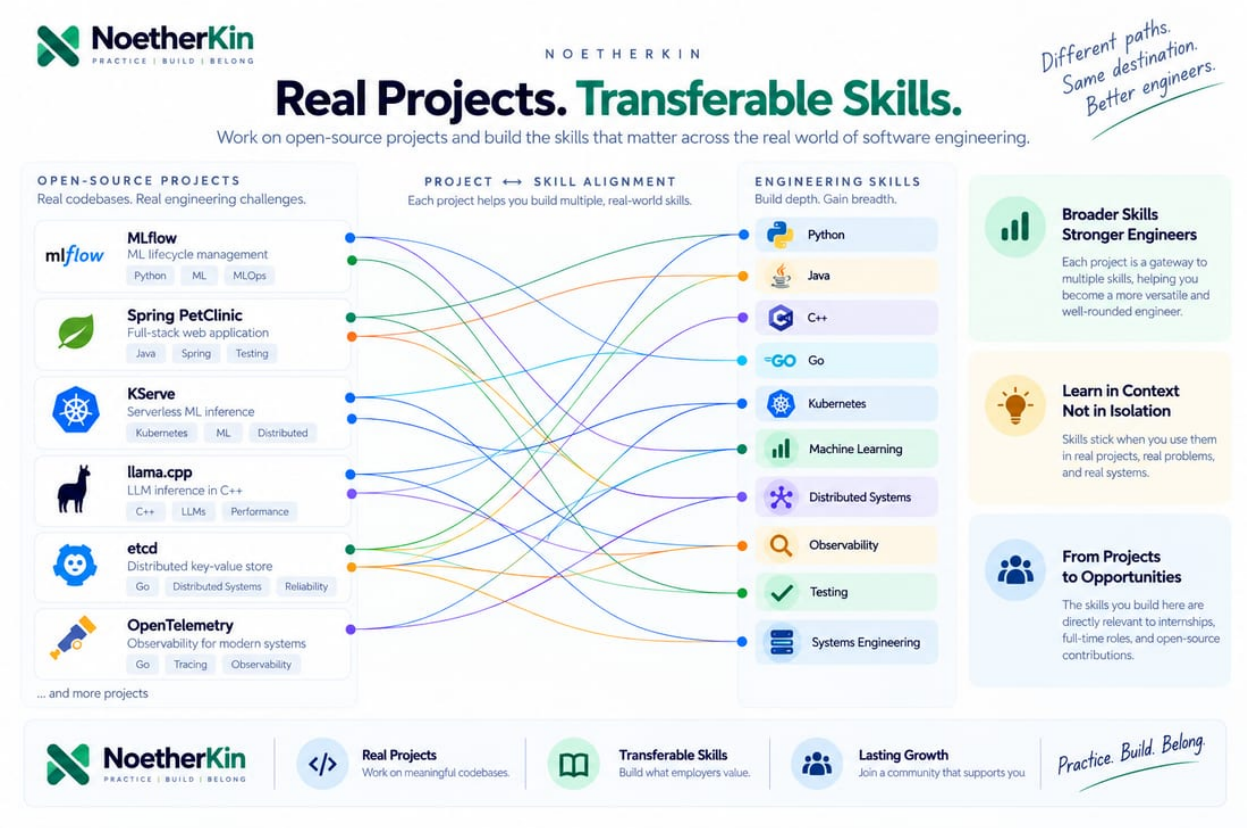


Figure. The same underlying skills keep showing up from a slightly different angle.

That repeated exposure, from a slightly different angle each time, is really the curriculum.

10

A typical day with it

Say I'm currently working through Spring PetClinic. I start the session with my active task. Instead of asking "how do I implement this in Spring," I'd ask something like "I think this request reaches the repository through this controller, but I'm not sure where validation happens, help me investigate without giving me the answer."

The mentor might point me toward a package, an interface, some Spring validation docs, or an existing test. I go dig through the code myself, then implement something, then run the tests. Something breaks. I ask for debugging help and instead of getting handed the fix, I get asked "what changed between the passing path and the failing one?" I look at the stack trace, form a guess, test it, and eventually track down the actual problem. Then I ask for a code review, which might flag an unnecessary abstraction, missing error handling, or an untested edge case. I fix those, a simulated user tries the feature, and once it meets the acceptance criteria the task closes and the evidence gets recorded. Then I move to the next one.

Any single task might only take a few hours. But repeat that cycle hundreds of times across different systems, and it starts adding up to something closer to a real apprenticeship.

11

How the workspace is laid out

One design goal is keeping the simulator cleanly separate from the actual open source repo, so the code you're contributing to stays untouched by simulation state. In practice that now looks like this:

```
TEXT

~/apprenticeship/                    your workspace: open your agent here
├── .apprenticeship/                 your record; only the CLI writes it
├── accessible-data-table/           a forge project: your own new Git repository
└── spring-petclinic-microservices/  an open-source clone, if you attach one
```

A hidden `.apprenticeship/` folder holds your profile, your competencies, your work in progress, past reviews, and recorded evidence, and every project you attach or build, cloned or forge, sits beside it as its own clean folder. Noetherkin never gets installed into the project itself. That

separation means the upstream repository stays clean enough to genuinely contribute back to, and one workspace can carry you through several projects in a row without your record splintering across them.



Figure. Apprenticeship state stays out of the source repo you might one day contribute back to.

12

Why not just ask ChatGPT to be my manager?

That's a fair question. I could open any chatbot and say "pretend to be my engineering manager" and it would probably do a decent job in the moment. The problem is consistency. Today it might call me a junior engineer. Tomorrow, senior. One conversation thinks a task is finished, another has no idea it existed. A code reviewer and a manager end up applying completely different, invented standards because nothing ties them together.

noetherkin tries to fix that by giving each role an explicit contract. A manager has specific responsibilities that differ from a team lead's, which differ from a peer engineer's permissions. Reviews draw on the same evidence. Competencies use shared definitions. Progression follows one shared rubric. Every role operates against the same underlying record instead of improvising

its own version of the company every time you open a new conversation.

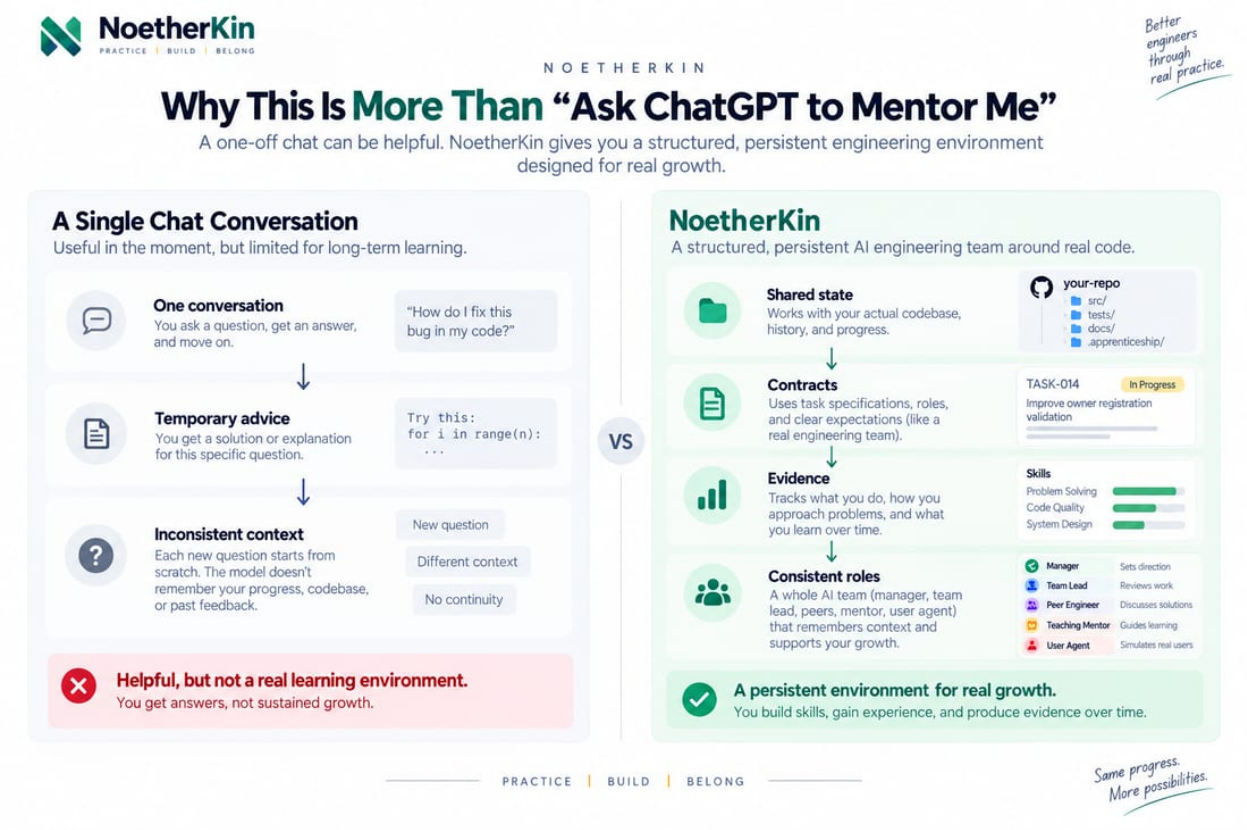


Figure. Shared contracts and a persistent record, instead of a new company every conversation.

13

Built to work with any AI tool

Portability matters to me too. I don't want this to only work inside one specific AI coding product. The underlying design uses plain, portable building blocks: Markdown, YAML, JSON, versioned schemas, and open agent instructions, so the core doesn't care whether you're using Codex or Claude Code today, or something else tomorrow. That's no longer just intent: `noetherkin skills install` installs the same 21 skill bundles into `~/.claude/skills` and `~/.agents/skills`, and generic, Codex, and Claude Code capability adapters project the same portable capability, like onboarding, into each host's own invocation surface, `/onboarding` in one, `$onboarding` in the other. Either way, the underlying behavior stays the same, and anything that actually changes your record still has to go through the CLI and a confirmation you type yourself.

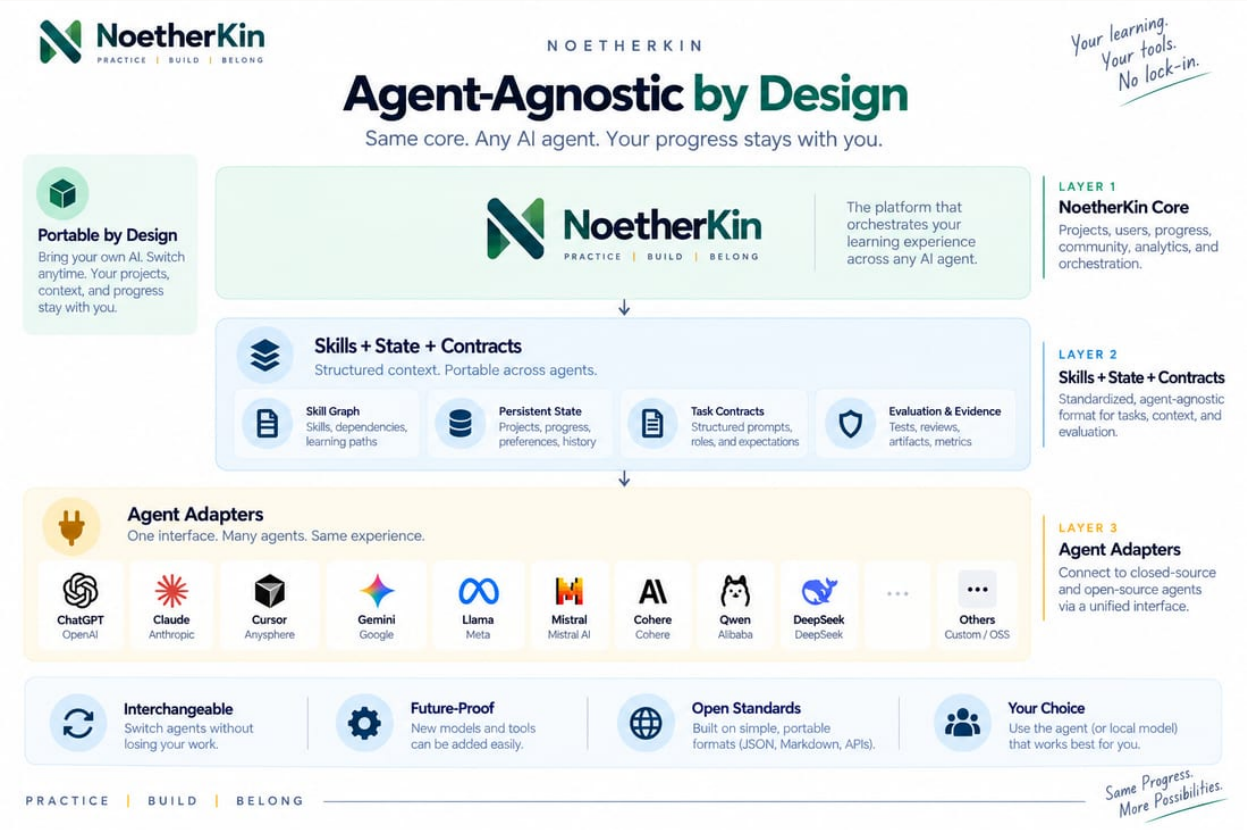


Figure. Markdown, YAML, JSON, and role contracts. The coding tool is an adapter.

14

Where I want this to go

The first version is scoped to my own goals, but there's a bigger version of this I keep thinking about. Instead of one fixed curriculum, noetherkin could eventually support entirely different engineering tracks: backend engineer, ML engineer, platform engineer, SRE, infrastructure engineer, GPU engineer, security engineer, data engineer, research engineer. Each would define its own relevant competencies, recommended projects, expectations, and progression path, while the core simulator underneath stays the same.

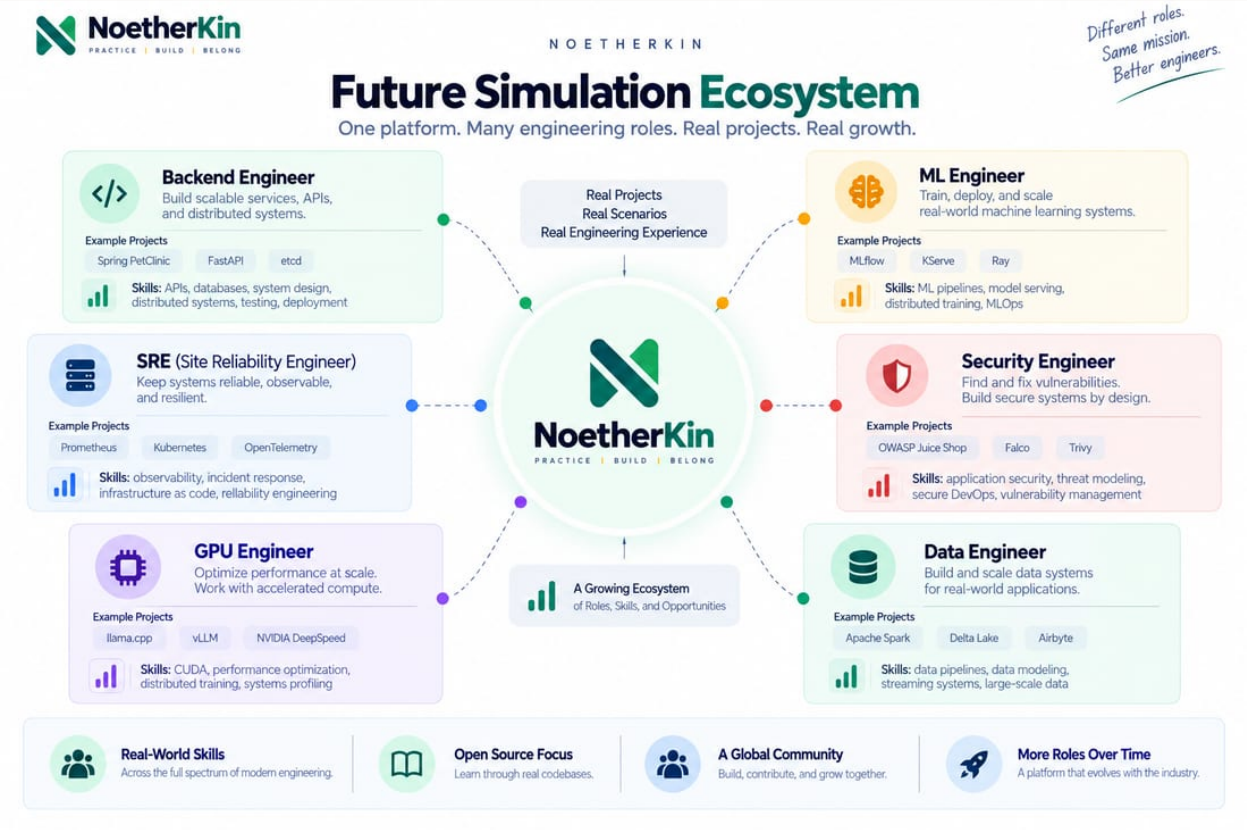


Figure. The same simulator underneath, with different tracks for different kinds of engineering.

I also want the project catalog itself to eventually be community driven. Someone who actually knows Kubernetes deeply could build a far better Kubernetes track than I could on my own. A Spring maintainer could design a realistic path through Spring's own projects. Someone working in GPU infrastructure could contribute exercises around inference optimization. Because the learner is ultimately working inside real, living open source projects, there's a nice loop hiding in there: you learn, you work inside a real project, you build real understanding, you fix bugs or add features, some of that work goes upstream, and you come out the other side a stronger engineer, ready to do it again somewhere harder.

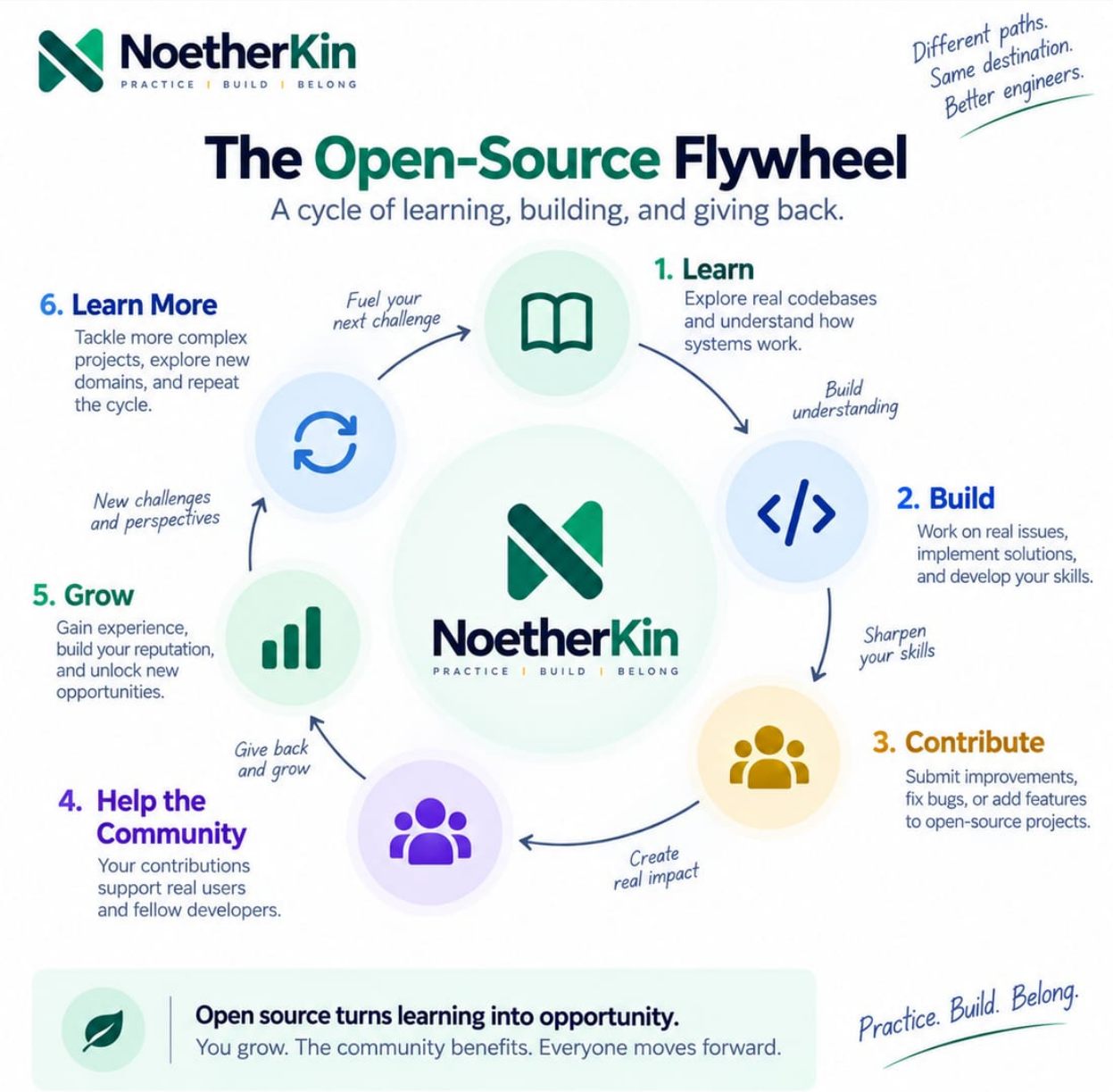


Figure. Learn in a living project, contribute, then take on something harder.

That's a genuinely different arc than "I am learning Java" turning into "I understand this codebase" turning into "I modified this subsystem" turning into "I fixed an actual upstream issue."

15

What this isn't

I want to be honest about what I'm not trying to build. This isn't another pitch for "build production apps without writing code." There are plenty of good projects exploring that direction already, and this is intentionally close to the opposite of it. The goal is to actually learn how the software works, not to skip that step.

It's also not trying to perfectly recreate what it's like to work at Google or Amazon. No simulated manager reproduces the human dynamics of a real team. No AI-generated incident really captures being on call at 2am. No open source repo perfectly stands in for a private production system serving millions of people. It's an educational environment, meant to expose people to the kinds of engineering behavior that are genuinely hard to develop just by reading and watching.

16

What I'm building first

I'm intentionally starting with the unglamorous part: the protocol underneath all the AI roles, before building the roles themselves. That means defining the simulation lifecycle, how a learner's state is represented, what actually counts as evidence, how competencies and levels are defined, and what each simulated role can and can't do. Without that, every role would end up inventing its own rules, and a manager could talk itself into declaring someone "promoted" just because a conversation sounded impressive.

Roughly, the plan is: lay the foundations (state, evidence, competencies, levels, role permissions), then try hard to break it (stress-test the promotion logic, evidence integrity, and role boundaries before building on top of a shaky base), then build the deterministic core (workspace setup, validation, schemas, the project catalog), then implement the first set of skills (onboarding, task assignment, teaching, peer engineering, review, team lead, manager), then get one full apprenticeship working end to end using Spring PetClinic before expanding, then add the rest of the engineering workflows (debugging, architecture review, incidents, production readiness, promotion review), then grow the project catalog, and eventually open it up for other people to actually use and extend.

17

Questions I don't have good answers to yet

This is part of why I'm writing about it this early rather than waiting for a polished launch. Some of this will change. I'm still chewing on things like: how much should the simulator automatically understand about a source repo versus have explained to it? How should competence transfer between projects? Should moving up a level require evidence from more than one project? How do conflicting reviews from different simulated roles get resolved? How much should tasks be generated on the fly versus hand curated, and how do you keep generated ones realistic? How do you stop the whole thing from turning into engineering theater and bureaucracy? Could private or company repositories work with this at all? Could maintainers eventually publish official tracks for their own projects? How do you track AI assistance honestly without discouraging people from asking for help when they should? And could this eventually be useful for teams onboarding actual junior engineers, not just solo learners?

Those all feel like genuinely interesting problems, and I don't think I'll answer most of them until I'm actually using the thing myself.

18

Building in public

This started from a fairly selfish problem: I wanted a better way to learn, and I was tired of hopping between courses instead of spending real time reading, debugging, building, and breaking actual systems. The more I sat with the idea, though, the more it felt like something that might be useful to other people too, so I'm building it in the open.

The architecture will keep changing. Some of what's in this post will probably turn out to be wrong, some pieces will disappear entirely, and others will show up once I actually start using this seriously myself. That's fine. The goal right now isn't to pretend this is finished. It's to build something that makes getting better at engineering the actual point, instead of just producing more code faster. If it ends up helping other people do the same, even better.

GitHub: github.com/nanaagyei/noetherkin[15] — public, with releases, CI, and CodeQL running on every change. **Status:** early (0.x), actively developed, v0.2.0 released, wire protocol 3.0. **Install:** `npm install -g https://github.com/nanaagyei/noetherkin/releases/latest/download/noetherkin.tgz`, then `noetherkin setup`. It isn't on the npm registry on purpose; see the architecture notes[1] for why.

I'll keep writing about the architecture, the mistakes, and what I learn as I actually build and use this.

• • •

That's the piece. If it earned a margin note, I'd love to hear it.

prince.agyei.tuffour@gmail.com

LINKS, FOR WHEN YOU'RE READING ON PAPER

[1] https://princeagyeituffour.com/writing/noetherkin-architecture-notes
[2] https://github.com/spring-projects/spring-petclinic
[3] https://agentskills.io/specification
[4] https://github.com/mlflow/mlflow
[5] https://github.com/open-telemetry/opentelemetry-demo
[6] https://github.com/kserve/kserve
[7] https://github.com/vllm-project/vllm
[8] https://github.com/open-telemetry/opentelemetry-cpp
[9] https://github.com/ggml-org/llama.cpp
[10] https://github.com/prometheus/prometheus
[11] https://github.com/argoproj/argo-cd
[12] https://github.com/etcd-io/etcd
[13] https://github.com/triton-lang/triton
[14] https://github.com/pytorch/pytorch
[15] https://github.com/nanaagyei/noetherkin

Typeset from the live article at princeagyeituffour.com/writing/what-if-learning-felt-like-joining-a-team. The web version may be newer and carries the interactive parts.