PROTOVATE / WHITE PAPER / 2026

How to Become an AI Engineer

The roadmap I actually walked. Sixteen years of shipping software, two years of shipping AI, and everything that broke in front of real users along the way.

7 STAGES	90-DAY PLAN	NO PHD REQUIRED

Prateek Sharma | Technical Lead, Protovate
protovate.com

Most AI engineer roadmaps are a watchlist. Linear algebra, then transformers, then a vector database, then LangChain - congratulations, you're an AI engineer.

That isn't how it happened for me.

I've been writing software for over sixteen years. Swift and Objective-C on iOS, Kotlin and Java on Android, Node and React when the work called for it. Two years ago I had shipped exactly zero AI features.

Today, in production: a conversational facility-management agent that takes real phone calls for an enterprise client, a suite of AI consumer apps on the Play Store, an internal Slack bot with an analytics dashboard, and an agricultural advisor that answers farmers in three languages.

None of it came from a course. It came from shipping, breaking things in production, reading logs - and a CEO who kept pushing me toward a better answer than the one I'd brought him.

So this is the roadmap I actually walked. If you're an experienced engineer wondering how to make the jump, this is for you.

AI engineering is not machine learning

This is the single most expensive misunderstanding in the field right now.

Training models

Data pipelines, feature engineering, loss curves, GPUs. A real discipline that needs a real math background.

Building products on models

Systems work. Designing what the model can see, deciding what tools it can call, handling the case where it returns garbage, keeping latency down, tracking token spend, and surviving contact with actual users.

If you can already reason about state, network failures, caching, race conditions and API contracts, you are much closer to being an AI engineer than a fresh graduate who has read three papers on attention.

Your existing engineering judgment is the moat - not the thing you need to replace.

- Do you need a PhD?
- Do you need to fine-tune anything in your first year?
- Do you even need a vector database on day one?

Not necessarily. And I'll come back to that last one, because it's the one that costs teams the most time.

Stop treating the model like a chatbot

The first real shift is treating a model as a component in a system rather than a thing you chat with. Three things, in this order.

Prompting as specification

Not tricks. A production system prompt is a spec document: what the assistant is, what it must never do, what order it asks questions in, what it does when information is missing.

On the BGIS project, the rewrites that stuck were the ones that replaced *"use your judgment"* with an explicit decision framework.

Structured output

The moment the model's answer drives UI, you need reliable JSON, not prose. Specify the schema hard, strip stray formatting on the way back in, parse defensively.

Assume one response in fifty will surprise you. Make sure that one doesn't crash the screen.

Tool use

This is the door into everything else. Once the model can call your functions, you're not building a chat feature anymore.

You're building an agent.

Give yourself one weekend on each. Build something small and disposable for all three.

Learn where the intelligence belongs

My biggest lesson, and I owe it to Brian Pollack, our CEO, who led the client relationship on the project where I learned it.

We were building **Bradley** , a conversational assistant for BGIS, a large facilities management company. The goal: replace a five-screen service request form with a conversation. Somebody reports a water leak, the assistant works out which building, which floor, which area, which service category - and files the work order.

My first architecture was the one any experienced developer would reach for. Load building data into memory, run fuzzy search over it with Fuse.js, rank the top five candidates in JavaScript, hand those five to the model. Clean, fast, predictable. I was quietly proud of it.

Brian's feedback rebuilt the whole thing. His point was that I had put the thinking in the wrong place. When he ran the same task by hand with a general-purpose model and nothing but a search tool, the model didn't need my ranked shortlist. It worked out on its own that *"the court building"* could mean court, courthouse, court house, judicial centre and half a dozen other things - generated all of those variations, and searched for them itself.

My ranking function was a very small and very stubborn brain, sitting between a much better brain and the data.

Tools should be dumb executors. The model does the reasoning. Every piece of clever logic you write between the user and the model is a ceiling you're installing on your own product. The one rule I now apply everywhere

That's worth internalizing early, because it's the opposite of the instinct that sixteen years of traditional development builds in you. We're trained to constrain, validate and pre-process. **In AI engineering, over-constraining is the most common way to build something mediocre.**

The corollary matters just as much, and Brian was equally firm on it: **some decisions belong to the user, not the model.**

Floor selection

Guessing wrong is expensive. Confirming is cheap. So we ask.

Service category

The model reads a description better than a user reads a dropdown of 208 options. So we don't ask.

Knowing which decision belongs to which side is the actual craft.

Retrieval, and why your "model bug" usually isn't one

Everyone reaches for a vector database first. Here's what actually happened.

A huge share of real business retrieval is keyword and attribute search over structured records, and a good search engine handles that better, faster and more cheaply than embeddings. We ended up on Elasticsearch, then Meilisearch, for typo-tolerant fuzzy matching across buildings, floors, areas and services. **No embeddings involved.**

Certain buildings were simply unfindable. We spent real time on it. The cause was a post-search filter that only ever checked the building *name* field - so a perfectly good address match got thrown away before the model ever saw it.

> **When your agent behaves stupidly, the model is usually innocent.**

Ask "what did the model actually see?" before you touch the prompt. Log every tool call and every tool result - you cannot debug an agent you can't observe.

Guardrails and evals, or you're just vibing

Two things separate an AI demo from an AI product.

Guardrails

On Bradley we ran three small, fast model calls in parallel with the main conversation: one for profanity, one for prompt-injection attempts, one for genuine emergencies that need to bypass the normal flow and escalate immediately. Small cheap models are perfect for this. **Run them concurrently so you don't pay for the latency.**

And then test your guardrails against real users, because they *will* fire when they shouldn't. Ours flagged people spelling out building names letter by letter as a hacking attempt. Nobody predicts that in design review. You find it by watching sessions.

Evals

This is the discipline most self-taught AI engineers skip, and it's the one that makes you credible.

> **If you can't say "version B resolves 12% more requests correctly than version A," you're not engineering. You're guessing with confidence.**

We built a browser-based prompt editor with a benchmark runner attached, so a non-engineer could edit the prompt, run it against a set of real historical service requests, and read the results. That let Kat, our prompt engineer, run statistical evaluation as a parallel workstream while I kept building. **Two tracks, no bottleneck.**

Learn to build that harness. It isn't glamorous, and it's the most valuable thing on this list.

Production is a different sport

Every hard-won lesson below came from something breaking in front of real users.

A search API with no timeout blocked our health check endpoint, so the platform kept killing a perfectly healthy app.

Cost tracking silently reported zero, because our pricing lookup table didn't recognize versioned model identifiers.

Calls dropped when the transcript came back as an empty string.

A Slack bot crash-looped in production for hours. The bug was upstream in a socket library, but nobody noticed because our process manager was running the dev file-watcher, which parks on a crash instead of restarting.

On the mobile side, one screen was executing roughly 17,000 sequential database queries against a 21 MB API response.

Add to that list: idempotency, retries, offline behaviour, and a real answer for what your app does when the model API is down.

Users don't care that the outage was OpenAI's or Anthropic's.

Build versus buy, decided with evidence

Bradley needed to answer phone calls. I built a custom telephony pipeline by hand: streaming audio, speech to text, text to speech, all wired together myself. Then I ran a timeboxed spike on **Vapi** , an off-the-shelf voice agent platform, wrapping our existing pipeline as its custom LLM endpoint so our search, work order logic and logging stayed on our infrastructure.

Vapi won. I filed a report saying so, including the parts that didn't work, and we shelved most of the custom build.

Writing the honest report on the thing you built yourself is a career skill, not a technical one. It's also what earns you the latitude to run the next spike.

Put AI into products people already use

Greenfield agents are the fun part. Shipping AI into existing products is where most of the money is - and it's mostly plumbing.

- **Remote config for API keys** , so you can rotate providers over the air without shipping a release.
- **A provider fallback chain** , so one vendor's outage isn't your outage.
- **Local persistence and migration** for AI-generated content.
- **Multilingual support** - for my farmer advisory app, English, Hindi and Punjabi. Far more about content and prompt design than translation strings.
- **Monetization that respects the user** , and app store policy compliance, which is stricter than you expect the moment children might use your app.

Unglamorous. Entirely learnable. Very hireable.

Four habits that did the work

I learned this by building real things, with Claude as a working partner, in a loop that looks nothing like a tutorial.

Log first, edit second

When something breaks, my default isn't to reason about the code. It's to add one targeted log, reproduce, read the actual value, then fix.

That habit caught bugs hours of reasoning had not: a base URL configured as HTTP instead of HTTPS, silently killing every POST. A job id arriving as -1 and blanking a screen. A parse failure hiding behind a scope mistake.

Ask for complete, drop-in files

No diffs. No refactor I didn't ask for. One method, replaced entirely, no side quests.

That's how you use AI on a mature codebase without setting fire to code that already works.

Treat it as an interlocutor, not an oracle

The most useful sessions are the ones where I get four sharp questions before any code appears, and where I push back when the answer is incomplete.

My best architectural decisions came out of argument, not autocomplete.

Write the report

End-of-day summaries, spike findings, estimate documents.

Forcing yourself to explain the work in plain language to a non-engineer is where you discover what you don't actually understand yet.

An AI assistant is very good at generating plausible theories. Evidence beats plausible every time - and the AI gets far more useful once you've handed it a real log line.

If you want one, here it is

Twelve weeks. One deployed system with observability and an eval number.

That portfolio piece beats any certificate on the market.

What matters less than the internet suggests

- **Training a model.** You won't, and you don't need to.
- **Fine-tuning in year one.** Better prompts, better tools and better retrieval will outrun fine-tuning for most business problems.
- **A framework.** Plain HTTP calls to a model API taught me more than any abstraction layer - and they're easier to debug at 2am.
- **Having started young.** Sixteen years of debugging other people's race conditions turned out to be excellent preparation.

What you do need is the willingness to put the intelligence in the right place, measure whether it worked, and ship it where real people will break it. Prateek Sharma / Technical Lead, Protovate

Let's Build What's Next

Have an AI, software, or product idea you want to validate or build? Schedule a consultation with the Protovate team and let's talk through what comes next.

SCHEDULE A CONSULTATION