THE DESI ARCHITECT

The 90-Day Backend to AI Engineer Roadmap

One project. Twelve weeks. Built the way production actually forces you to build it.

This is the companion checklist to the video. It does not ask you to collect technologies. It gives you **one application** and breaks it, week by week, the same way a real product breaks - so that RAG, evals, observability and tool calling arrive as **answers to problems you already hit**, not as items on a list somebody else wrote.

`.NET` `Java` `Node / TS` `Python`

Every task mapped to all four stacks

Version 1.0 · August 2026

00 How to use this document

Read this page properly. It is the difference between finishing in 90 days and abandoning this in week three like most roadmaps.

> **THE ONE RULE**
>
> You are building **one** application for 90 days. Not ten tutorials, not five side projects. Every week adds one layer to the same codebase. If at any point you are tempted to start something new because it looks more interesting, that is exactly the failure this roadmap exists to prevent.

WHAT EACH WEEK GIVES YOU

Why this week	The production problem that makes this week necessary. If you cannot explain this, do not start the week.
Day plan	Five working days of concrete tasks. The days are a sequence, not a schedule - if a week takes you nine days, it takes nine days.
Your stack	The equivalent tool in .NET, Java, Node and Python. Pick your column and ignore the rest.
Deliverable	What must exist and work at the end of the week. Tick every box before moving on.

ON PACE, HONESTLY

This roadmap is scoped by outcome, not by hours. Some people will do a week in four evenings, others in two weekends, others in three weeks because work got busy. None of those are wrong and none of them mean you are behind. The order matters, the deliverables matter, the calendar does not. A person who finishes this in five months has the same portfolio as a person who finishes in three.

THREE WAYS PEOPLE FAIL THIS ROADMAP

- **Framework-first.** Starting with a framework in week one means the framework does the engineering and you learn nothing. Frameworks come after you have felt the problem they solve.
- **Skipping to the interesting part.** Agents and RAG are the fun words. Timeouts and structured output are the reason your version will actually run. Weeks 1 to 4 are the ones that separate this project from every other portfolio chatbot.
- **Never deploying.** A project that only ever ran on localhost has not met real latency, real secrets management, or real failures. Week 11 is not optional.

> **A NOTE ON THE TOOLING IN THIS DOCUMENT**
>
> Every library named here was checked against its current state in August 2026, and where something has recently changed hands or been superseded, this document says so rather than repeating an older recommendation. Tooling moves faster than concepts. When a name here is out of date, the surrounding engineering reasoning will still be correct - use that to pick the replacement.

01 The project you are building

A support assistant for a food delivery company. In the video it is called Tadka. Use any domain you like - the point is that it must be a domain with real documents, real user intent, and real actions.

WHY THIS PROJECT AND NOT SOMETHING MORE IMPRESSIVE

Because it fails in all the right places. A support assistant needs private company knowledge, so it forces retrieval. Its answers are subjective, so it forces evaluation. It gets hit by traffic spikes, so it forces cost control. Users eventually want it to actually do things, so it forces the action boundary. Those four pressures are the entire discipline. A flashier project that only ever calls one model teaches you none of them.

WHAT IT DOES BY DAY 90

- Answers customer questions about orders, refunds, offers and delivery policy, grounded in your own documents rather than the model's memory.
- Says "I do not know" when the documents do not support an answer, instead of inventing one confidently.
- Has a measurable quality score that moves when you change a prompt, so improvement is a number and not an opinion.
- Reports its own latency, token usage and cost per conversation.
- Runs on the public internet, with secrets managed properly and failures handled.

SCOPE DISCIPLINE

In scope	One channel (HTTP API is enough), one language, one knowledge base, one model provider with a fallback.
Out of scope	A polished frontend, user accounts, multi-tenancy, mobile apps, fine-tuning, training anything.
Optional stretch	A minimal chat UI in week 12 if and only if everything else is done. Recruiters look at the architecture write-up, not your CSS.

> **GET YOUR DOCUMENTS READY NOW**
>
> You will need 30 to 60 pages of text that belongs to a domain you understand. Use public documentation, your own written notes, or a knowledge base you write yourself for a fictional company. Do not use your employer's internal or proprietary documents for a portfolio project you intend to show people.

02 Pick your stack and stop thinking about it

Build in the language you are already strong in. Your advantage in this transition is backend judgement, and you throw that advantage away by also fighting an unfamiliar language. Python matters, but not the way roadmaps claim.

DECISION	.NET	JAVA	NODE / TYPESCRIPT	PYTHON
Base LLM abstraction	`Microsoft.Extensions.AI` (IChatClient) over a provider SDK	Spring AI `ChatClient`, or LangChain4j if not on Spring	Vercel AI SDK (`ai`), or the provider SDK directly	Provider SDK directly (openai, anthropic)
Web layer	ASP.NET Core Minimal API	Spring Boot	Fastify or Express	FastAPI
Where the ecosystem is thinnest	Eval tooling - run it as a separate step	Eval tooling - run it as a separate step	RAG evaluation metrics	Nothing. Python has everything.
How you cover the gap	Use a language-neutral eval runner against your HTTP endpoint (see week 9). Your app stays in your language.			Use the native libraries.

SO HOW MUCH PYTHON DO YOU ACTUALLY NEED

Enough to read it and run it. Most AI engineering reference material, eval libraries and research code is written in Python, and you will be at a permanent disadvantage if you cannot follow a Python example and translate the idea into your stack. That is a few evenings of syntax, not a career change. Learn it in parallel, in the gaps - never as a blocker to starting week one.

> **.NET DEVELOPERS, ONE IMPORTANT NOTE**
>
> If you search for .NET AI tutorials you will mostly find Semantic Kernel. As of 2026, Microsoft has shipped Agent Framework 1.0 and Semantic Kernel is in maintenance - supported and patched, but no longer where new investment goes. For this roadmap, build on Microsoft.Extensions.AI for the model abstraction, and reach for Agent Framework only in week 12, and only if your project genuinely needs agent orchestration. Do not start new work on Semantic Kernel in 2026.

EVERYTHING BELOW THIS LINE IS STACK-INDEPENDENT

Chunking strategy, retrieval quality, eval design, the propose-validate-execute boundary, cost control, failure handling. These are the parts that transfer between jobs and between languages, and they are the parts an interviewer will actually probe. The library names in the tables are conveniences. The reasoning is the skill.

MONTH 1 · DAYS 1 TO 30

BUILD

Direct model integration, and all the unglamorous backend engineering that decides whether this thing survives contact with a real user.

W1 First call, config, and the logging you will live on

W2 Timeouts, retries, circuit breakers, rate limits

W3 Structured output and a contract that never leaks raw model text

W4 Tokens, cost, caching, and streaming

WEEK 01

First call, config, and the logging you will live on

Days 1-5

WHY THIS WEEK

Almost everyone's first LLM project is a script with an API key pasted into it. That script cannot be debugged, cannot be deployed and cannot be shown to anyone. This week you build the boring shell that every later week plugs into - and you will spend the next eleven weeks grateful that you can actually see what your application did.

DAY PLAN

Day 1 **Project skeleton.** New service in your language. One health endpoint, one POST endpoint that will eventually take a customer question. No AI yet.
Deliberate: prove your build, run and test loop works before adding a network dependency you do not control.

Day 2 **Configuration and secrets.** Provider key, model name, base URL and timeouts all read from configuration. Nothing hardcoded, nothing committed. Fail loudly at startup if a required setting is missing.

Day 3 **The first real call.** Wire the model into your endpoint. Send a question, get an answer back. Keep it deliberately naive - no retries, no streaming, no cleverness.

Day 4 **Structured logging.** Log every model call as structured data: a correlation id, the model, the latency, the finish reason, and the token counts the provider returns. Not a print statement.
Log prompts and responses behind a flag that is off by default. Real user messages are real user data.

Day 5 **Break it on purpose.** Use a wrong API key, an invalid model name, and a deliberately unreachable base URL. Write down what your application does in each case. It will be bad. Week 2 is that list.

YOUR STACK

TASK	.NET	JAVA	NODE / TS	PYTHON
Service	ASP.NET Core Minimal API	Spring Boot Web	Fastify + TypeScript	FastAPI + uvicorn
Model call	`Microsoft.Extensions.AI` IChatClient	Spring AI `ChatClient` / LangChain4j	AI SDK `generateText`	Provider SDK client
Config	`IOptions<T>` + user-secrets	`@ConfigurationProperties`	env parsed and validated with Zod	`pydantic-settings`
Logging	ILogger with structured scopes, or Serilog	SLF4J + Logback, JSON encoder	`pino`	`structlog` or stdlib JSON formatter

DAY 5 DELIVERABLE

A service that answers a question through a model, and produces one structured log line per call that tells you exactly what happened.

- [ ] No secret appears anywhere in the repository, including in the git history
- [ ] Every model call logs correlation id, model, latency in ms, tokens in and out, finish reason
- [ ] Startup fails immediately and clearly when configuration is missing
- [ ] You have written down, in the repo, what currently happens on each of the three failure cases from day 5

WEEK 02

Timeouts, retries, circuit breakers, rate limits

Days 6-10

WHY THIS WEEK

A model provider is a third-party network dependency with variable latency, rate limits, and outages you do not control - and it is slower and less reliable than any database you have ever called. If you already know how to make a flaky downstream service safe, this week is you applying skills you have, which is precisely the argument this whole roadmap is making.

DAY PLAN

Day 6 **Timeouts.** Set an explicit request timeout. Pick a number and be able to defend it. Then confirm that when it fires, your caller gets a clean error rather than a hung connection.
Streaming changes what a timeout means: time-to-first-token and total duration are two different budgets.

Day 7 **Retries with backoff and jitter.** Retry only what is safe to retry: timeouts, 429s, 5xx. Never retry a 400. Exponential backoff with jitter, and a hard cap on attempts.
Write the cost note in your README now: every retry is a second bill for the same answer.

Day 8 **Circuit breaker.** When the provider is clearly down, stop hammering it. Open the circuit, fail fast, return your degraded response, and recover on a half-open probe.

Day 9 **Rate limiting and concurrency.** Cap the number of in-flight model calls your service will make. Add a client-side limit that sits under the provider's quota, and decide what happens to requests over it - queue or reject, but choose deliberately.

Day 10 **Degraded mode.** Decide what your product does when the model is entirely unavailable. A canned message and a support handoff is a legitimate answer. A 500 to the customer is not.

YOUR STACK

CONCERN	.NET	JAVA	NODE / TS	PYTHON
Resilience	`Microsoft.Extensions.Http.Resilience` (Polly v8)	Resilience4j	`cockatiel` or `p-retry`	`tenacity` or `stamina`
Circuit breaker	Polly resilience pipeline	Resilience4j `@CircuitBreaker`	`opossum` or cockatiel	`purgatory` / `aiobreaker`
Timeout	HttpClient timeout + CancellationToken	Resilience4j `@TimeLimiter`	`AbortSignal.timeout()`	`httpx.Timeout`
Concurrency cap	`SemaphoreSlim` / rate limiter middleware	`Bulkhead`, `@RateLimiter`	`p-limit`	`asyncio.Semaphore`

DAY 10 DELIVERABLE

A service that stays up and behaves predictably while the model provider is timing out, rate limiting you, or completely down.

- [ ] Provider unreachable: caller gets a clean, fast, documented error - not a hang
- [ ] Retries are capped, jittered, and never fire on a client error
- [ ] Circuit opens under sustained failure and recovers without a restart
- [ ] You can state your worst-case latency, and your worst-case cost for a single user request including retries

THE QUESTION THIS WEEK ANSWERS IN AN INTERVIEW

"What happens to your application when the model provider has an outage?" Most candidates have never thought about it. You will have a rehearsed, specific answer with a number attached, and that single answer signals more seniority than any framework on your resume.

WEEK 03

Structured output and a contract that never leaks raw model text

Days 11-15

WHY THIS WEEK

A model returns text. Your API must return a contract. The gap between those two facts is where most AI features quietly become unmaintainable - callers start string matching on model prose, and every prompt tweak becomes a breaking change. This week you put a real boundary in.

DAY PLAN

Day 11 **Define the response type.** Decide what your endpoint returns as a typed object: the answer, a confidence or a can-answer flag, the intent you detected, and later your sources. Design it before you prompt for it.

Day 12 **Get structured output from the model.** Use your provider's structured output or JSON schema mode rather than asking politely in the prompt and hoping. Bind straight into your type.

Day 13 **Validate anyway.** Schema mode reduces malformed output, it does not eliminate the need for validation. Validate on the way in, and decide what happens when validation fails - one repair attempt, then a safe fallback.

Day 14 **Error taxonomy.** Classify what can go wrong: provider error, timeout, validation failure, refusal, content filter, empty answer. Map each to a distinct API response and a distinct log signal. This taxonomy becomes your dashboard in week 10.

Day 15 **Prompt lives in the codebase.** Move prompts out of inline strings into versioned files or constants with an identifier. Log which prompt version produced each response.

Without this, week 9 evals cannot tell you which change moved the score.

YOUR STACK

TASK	.NET	JAVA	NODE / TS	PYTHON
Typed output	M.E.AI structured output into a C# record	Spring AI `entity()` / BeanOutputConverter	AI SDK `generateObject` + Zod	Pydantic model + provider schema mode
Validation	System.Text.Json + DataAnnotations	Jackson + Bean Validation	Zod `safeParse`	Pydantic `model_validate`
Prompt storage	Versioned files in the repo with an id such as `answer.v3`. A prompt management service is a week 10 decision, not a week 3 one.			

DAY 15 DELIVERABLE

An endpoint with a typed, validated contract that is stable even when you rewrite the prompt underneath it.

- [ ] The API response is a defined type, and raw model text is never passed through untouched
- [ ] Malformed model output is caught, counted, and handled - not returned to the caller
- [ ] Every failure mode maps to a distinct, documented API error
- [ ] Every stored response records which prompt version produced it

WEEK 04

Tokens, cost, caching, and streaming

Days 16-20

WHY THIS WEEK

Token cost is the one operational property of AI systems that has no equivalent in the backend work you have done before. A retry loop against a database is a performance bug. The same loop against a model provider is an invoice. Engineers who can talk about cost per conversation are treated very differently in design reviews.

DAY PLAN

Day 16 **Count tokens before you send.** Add token counting for your prompt. Now you know your input size before the bill arrives, and you can enforce a ceiling.

Day 17 **Cost per request.** Put provider pricing in configuration and compute an estimated cost for every call from the token counts. Log it. Aggregate it per conversation.
Configuration, not hardcoded - prices change and you do not want to redeploy for that.

Day 18 **Budget guards.** Enforce a maximum input size, a maximum output length, and a per-conversation ceiling. Decide what happens at the ceiling: truncate, summarise, or refuse.

Day 19 **Caching.** Cache identical requests. Then read your provider's prompt caching documentation and restructure your prompt so the stable part comes first - this is a real and often large saving that most people never claim.

Day 20 **Streaming.** Stream tokens to the caller. Perceived latency drops dramatically. Then handle the parts everyone forgets: what a mid-stream failure does, and how you count tokens when the response arrives in pieces.

YOUR STACK

TASK	.NET	JAVA	NODE / TS	PYTHON
Token counting	`Microsoft.ML.Tokenizers` or SharpToken	`jtokkit`	`js-tiktoken` / `gpt-tokenizer`	`tiktoken`
Metrics	`System.Diagnostics.Metrics`	Micrometer	OpenTelemetry metrics API	OpenTelemetry metrics API
Cache	`IDistributedCache` (Redis)	Spring Cache + Redis	`ioredis`	`redis-py`
Streaming	IAsyncEnumerable + SSE	Flux / SSE	AI SDK `streamText`	FastAPI StreamingResponse

DAY 30 DELIVERABLE

A working backend AI API with baseline resilience, a structured contract, and cost visibility. This is the milestone the video calls the end of month one.

- [ ] You can state the average and worst-case cost of one customer conversation
- [ ] Input size, output length and per-conversation spend all have enforced ceilings
- [ ] Repeated identical questions do not produce repeated bills
- [ ] Responses stream, and a mid-stream failure is handled rather than silently truncating
- [ ] Month 1 review: you have written zero AI-specific framework code so far, and the service is production-shaped. That is the point.

MONTH 2 · DAYS 31 TO 60

GROUND

Connect the model to your own data, and build the test list that turns "it feels better" into something you can actually defend.

W5 Prove the failure, then build ingestion and chunking

W6 Embeddings and vector search

W7 Retrieval wired in, with citations and honest refusals

W8 The test list, and a failure taxonomy that splits retrieval from generation

WEEK 05

Prove the failure, then build ingestion and chunking

Days 21-25

WHY THIS WEEK

Do not build retrieval because a roadmap told you to. Build it because you spent day 21 watching your own application answer confidently and wrongly about your own company's policies. That experience is what makes the rest of this month stick, and it is the difference between knowing what RAG stands for and knowing when it is the right answer.

DAY PLAN

Day 21 **Watch it fail.** Ask your current application ten questions that can only be answered from your documents - a refund window, a specific offer, a delivery policy. Record the answers verbatim. Some will be confidently wrong. Keep this file; it is your first before-and-after evidence.

Day 22 **Ingestion pipeline.** A separate command that reads your source documents, extracts clean text, and stores it with metadata: source name, section, and a stable id. Make it re-runnable without duplicating.

Day 23 **Chunking, written by you.** Split documents into passages. Write the splitter yourself first - split on structure such as headings and paragraphs, with a size limit and a small overlap.

Doing this by hand once teaches you why chunk boundaries decide retrieval quality. Then compare against a library and keep whichever is better.

Day 24 **Chunk quality review.** Print 20 random chunks and read them. If a chunk cannot be understood standing alone, retrieval will surface it and the model will misuse it. Fix the splitter until chunks are self-contained.

Day 25 **Metadata that will matter later.** Attach document title, section heading, last-updated date and source URL to every chunk. Week 7 uses this for citations, and stale-content filtering depends on that date field.

YOUR STACK

TASK	.NET	JAVA	NODE / TS	PYTHON
Text extraction	PdfPig, HtmlAgilityPack	Apache Tika, jsoup	`unpdf`, `cheerio`	`pypdf`, `BeautifulSoup`
Chunking	Write your own, then compare with M.E.AI / SK text chunkers	Write your own, then compare with Spring AI `TokenTextSplitter`	Write your own, then compare with LangChain.js splitters	Write your own, then compare with LangChain / LlamaIndex splitters
Ingestion job	Console project or hosted service	Spring Batch or a CommandLineRunner	A CLI script in the same repo	A CLI script (typer / argparse)

DAY 25 DELIVERABLE

A re-runnable ingestion command that turns your documents into clean, self-contained, metadata-rich chunks.

- [ ] Ten recorded examples of your application answering wrongly without context
- [ ] Re-running ingestion updates rather than duplicating
- [ ] You read 20 random chunks and each makes sense in isolation
- [ ] Every chunk carries source, section, date and a stable id

WEEK 06

Embeddings and vector search

Days 26-30

WHY THIS WEEK

You need to find the right three passages out of two thousand, using a question that shares almost no keywords with them. That is the actual problem embeddings solve. Approach it as a search and indexing problem - which is a backend problem you already have instincts for - not as machine learning.

DAY PLAN

Day 26 **Generate embeddings.** Embed every chunk in batches. Store the vector next to the chunk text and metadata. Record which embedding model and dimension you used.
Write this down. Changing embedding model later means re-embedding everything, and you will forget which one you used.

Day 27 **Set up the store.** Postgres with pgvector. You almost certainly already run Postgres, and for a project of this size a separate vector database is operational overhead with no benefit.

Day 28 **Similarity search.** Embed the incoming question, retrieve the top k chunks by cosine distance. Start with k of 5. Look at what comes back for your ten questions from day 21.

Day 29 **Index and filter.** Add an HNSW index and measure the latency difference. Add metadata filtering so you can restrict retrieval by source or recency.

Day 30 **Hybrid search.** Pure vector search is weak on exact terms - order ids, product names, policy codes. Combine it with Postgres full-text search and merge the two result sets. Measure whether it actually helped on your ten questions.

YOUR STACK

TASK	.NET	JAVA	NODE / TS	PYTHON
Embeddings	M.E.AI `IEmbeddingGenerator`	Spring AI `EmbeddingModel`	AI SDK `embedMany`	Provider embeddings API
Vector store	Npgsql + `pgvector-dotnet`	Spring AI `PgVectorStore`	`pgvector-node` + `pg`	`pgvector-python` + psycopg
Full-text half	Postgres `tsvector` in all four. Merge with Reciprocal Rank Fusion - about fifteen lines of code, and worth writing yourself.			

WHEN A DEDICATED VECTOR DATABASE IS THE RIGHT CALL

pgvector comfortably covers projects of this size and far beyond. The honest threshold where teams move to Qdrant, Milvus or similar is tens of millions of vectors, or hard p99 latency requirements combined with heavy metadata filtering. Knowing that threshold - and that you are nowhere near it - is a better interview answer than having used the fancier tool for a 2,000-chunk project.

DAY 30 DELIVERABLE

Working semantic search over your own documents, with an honest measurement of whether hybrid search beat pure vector search.

- [ ] Embedding model and dimension recorded in the repo
- [ ] Top-k retrieval returns visibly relevant passages for your ten questions
- [ ] You measured search latency before and after adding the index
- [ ] You can name one question where hybrid search beat pure vector search, and one where it did not

WEEK 07

Retrieval wired in, with citations and honest refusals

Days 31-35

WHY THIS WEEK

Retrieval that finds the right passage and then lets the model ignore it has solved nothing. This week is about grounding: making the answer traceable to a source, and making the system willing to say it does not know. The refusal path is the part that separates a demo from something a support team would actually let near customers.

DAY PLAN

Day 31	**Assemble the context.** Retrieved passages go into the prompt with clear delimiters and their source ids. Enforce a context budget - retrieval must never blow the ceiling you set in week 4.
Day 32	**Ground the instruction.** Instruct the model to answer only from the provided context. Extend your week 3 response type with the source ids it used.
Day 33	**Citations.** Return the sources with the answer, resolved to real titles and links via your week 5 metadata. Now every answer is auditable, which is what makes this defensible in a regulated or support context.
Day 34	**The refusal path.** When retrieval returns nothing above a relevance threshold, do not call the model for an answer at all - return a documented "I do not have that information" with a handoff. Tune the threshold against your day 21 questions. Interviewers probe this. A system that cannot refuse is a system that will confidently mislead a customer.
Day 35	**Re-run day 21.** Ask the same ten questions. Put the before and after answers side by side in your repo. This becomes a slide in your interview story.

DESIGN DECISIONS TO RECORD, NOT JUST IMPLEMENT

Chunk size and overlap	What you chose and what you observed at other sizes.
k	How many passages you retrieve, and what happened when you raised it - usually more cost and more distraction, not more accuracy.
Relevance threshold	The score below which you refuse. The single most important number in the system.
Conflicting sources	What the system does when two documents disagree. Recency wins is a defensible default, if you say so out loud.

DAY 35 DELIVERABLE

Grounded, cited answers with a working refusal path.

- [ ] Every answer returns the sources it used, with real titles
- [ ] Questions outside the knowledge base produce a refusal, not a guess
- [ ] Retrieved context can never exceed your token budget
- [ ] A written before-and-after on the same ten questions lives in the repo

WEEK 08

The test list, and a failure taxonomy that splits retrieval from generation

Days 36-40

WHY THIS WEEK

This is the week almost every portfolio project skips, and it is the week that makes yours credible. In the video, this is the CEO asking whether the bot got better or worse after the last deployment. Right now you cannot answer that. By Friday you can.

DAY PLAN

Day 36	**Write 20 to 30 questions.** Cover four categories deliberately: straightforward lookups, questions needing two passages combined, questions the documents cannot answer (the correct answer is a refusal), and adversarial or ambiguous phrasings.
Day 37	**Write the expected outcome.** For each question record what a correct answer must contain, which source it should cite, and whether refusing is the right behaviour. Store it as data in the repo, not in your head.
Day 38	**Score the current system by hand.** Run all of them and grade each one. Yes, manually. You need to see your own failures before you automate the measuring of them.
Day 39	**Split the failures.** For every failure decide: did retrieval fail to find the passage, or did the model have the right passage and still answer badly? These have completely different fixes and conflating them is the most common RAG debugging mistake.
Day 40	**Fix the top failure class, re-score, record.** One change, measured. You now have a baseline number and a documented improvement, which is exactly the loop week 9 automates.

THE TWO FAILURE CLASSES

SYMPTOM	RETRIEVAL FAILURE	GENERATION FAILURE
What you see	Correct passage is not in the retrieved set at all	Correct passage was retrieved, answer is still wrong or invented
Usual causes	Chunk boundaries split the answer, k too low, question and document vocabulary differ, exact term needed hybrid search	Weak grounding instruction, contradictory passages, context too long and the passage got lost, model ignored the constraint
Where you fix it	Ingestion, chunking, k, hybrid search, filters	Prompt, context ordering, budget, refusal threshold

DAY 60 DELIVERABLE

A Tadka-style contextual RAG system with a representative test set and a measured baseline. This is the milestone the video calls the end of month two.

- [ ] 20 to 30 questions with expected outcomes, versioned in the repo
- [ ] A baseline score you can state as a number
- [ ] Every failure classified as retrieval or generation
- [ ] One improvement made and measured against the baseline
- [ ] Refusal cases are in the test set and are scored as successes when the system refuses

MONTH 3 · DAYS 61 TO 90

HARDEN

The layer that is missing from almost every portfolio project, and the reason companies pay for this skill rather than for prompt writing.

W9 Turn the test list into an automated eval harness

W10 Observability: traces, cost, latency, quality signals

W11 Deploy it, and meet the problems only deployment creates

W12 Actions if the product needs them, and the write-up that gets you hired

WEEK 09

Turn the test list into an automated eval harness

Days 41-45

WHY THIS WEEK

A manual test list gets run twice and then abandoned. An automated one runs on every change and becomes the thing that lets you refactor a prompt without fear. This is the closest equivalent to a test suite that AI systems have, and treating it with the same seriousness is what makes you an engineer rather than a prompt tinkerer.

DAY PLAN

Day 41 **Automate the deterministic checks first.** Did it refuse when it should have? Did it cite the expected source? Is the response schema valid? These need no model to judge them and they catch a surprising share of regressions.

Day 42 **Add retrieval metrics.** Measure whether the correct passage appeared in the retrieved set, and how far down. This isolates retrieval quality from answer quality - the split you established in week 8.

Day 43 **Add model-graded checks.** For open-ended answers, use a model to grade faithfulness to the retrieved context and relevance to the question. Use a strong model as the judge and pin the judge model version.
Model-graded scores are noisy. Trust the direction of a change across the whole set, never a single question's score.

Day 44 **One command, one report.** Running the suite must be a single command that prints a scoreboard and writes a report file. If it is harder than that you will stop running it.

Day 45 **Wire it into CI.** Run on every pull request. Fail the build if the score drops more than your chosen tolerance. Now prompt changes are governed by evidence.

YOUR STACK

APPROACH	.NET	JAVA	NODE / TS	PYTHON
Recommended runner	**promptfoo** - a language-neutral CLI driven by YAML that tests your HTTP endpoint. Your application stays in your language; only the eval harness is JavaScript tooling.			**Ragas** for RAG metrics, **DeepEval** for CI-style assertions, or promptfoo
Deterministic checks	xUnit against the eval dataset	JUnit against the eval dataset	Vitest against the eval dataset	pytest against the eval dataset
RAG-specific metrics	Ragas is the specialist here (context precision and recall, faithfulness, answer relevancy). Non-Python teams can run it as a separate step against exported traces rather than rewriting it.			
CI	GitHub Actions on pull request, with the score report published as a build artifact.			

KEEP THE EVAL HARNESS OUTSIDE YOUR APPLICATION

It talks to your service over HTTP like any other client. This keeps your production code free of test-only dependencies, lets you use the best tool regardless of your stack, and means the harness still works if you later rewrite the service.

DAY 45 DELIVERABLE

A single command that scores your system, and a CI gate that blocks regressions.

- [ ] One command runs the full suite and writes a report
- [ ] Retrieval quality and answer quality are reported separately
- [ ] CI fails when quality drops beyond your tolerance
- [ ] Judge model and prompt versions are pinned and recorded in the report

WEEK 10

Observability: traces, cost, latency, quality signals

Days 46-50

WHY THIS WEEK

In the video this is the Monday morning finance call about a bill nobody can explain. Nothing was down, no customer saw an error, and lakhs of tokens were quietly burned over a weekend. Uptime monitoring would not have caught it. This week you build the monitoring that would.

DAY PLAN

Day 46	**Tracing.** Instrument with OpenTelemetry. One trace per user request, with spans for retrieval, embedding, the model call and any tool call. Now you can see where the seconds actually go.
Day 47	**The four metrics that matter.** Latency (p50, p95, p99, and time-to-first-token separately), tokens in and out, estimated cost per request and per conversation, and error rate broken down by your week 3 taxonomy.
Day 48	**Prompt tracing.** Capture which prompt version, which model version, which retrieved sources and which judge scores produced each response. This is what makes a production incident debuggable three weeks later.
Day 49	**An LLM observability tool.** Add Langfuse (self-hosted via Docker Compose) or Arize Phoenix. Send your traces there and use the session view to inspect a real conversation end to end.
Day 50	**Alerts that would have caught the bill.** Alert on cost per hour, token usage per conversation, p99 latency and refusal rate. Then simulate the incident: force a retry loop, and confirm your alert fires. A rising refusal rate is an early warning that your knowledge base has gone stale. Almost nobody monitors it.

YOUR STACK

TASK	.NET	JAVA	NODE / TS	PYTHON
Tracing	OpenTelemetry .NET + `ActivitySource`	OpenTelemetry Java agent	`@opentelemetry/sdk-node`	`opentelemetry-sdk`
Metrics	System.Diagnostics.Metrics to OTLP	Micrometer to OTLP	OTel metrics to OTLP	OTel metrics to OTLP
LLM observability	Langfuse (MIT core, self-hostable, ingests OpenTelemetry traces) or Arize Phoenix. Both accept traces from any language.			

AN HONEST NOTE ON THE GENAI TRACING STANDARD

OpenTelemetry's `gen_ai.*` semantic conventions are still marked Development, not Stable, and in June 2026 they moved into a dedicated repository with no versioned release to pin against yet. Use them - they are the direction the industry is heading - but expect attribute names to change, keep your dashboards cheap to rebuild, and do not let a vendor tell you this part is settled. Being able to say this accurately in an interview is itself a signal.

DAY 50 DELIVERABLE

You can answer "what did this system do last Tuesday, and what did it cost" without guessing.

- ☐ One trace per request, spanning retrieval through model call
- ☐ Cost, token, latency and error-rate metrics all recorded
- ☐ Any stored response can be traced back to its prompt version, model version and sources
- ☐ A cost alert exists, and you have proven it fires by simulating the incident

WEEK 11

Deploy it, and meet the problems only deployment creates

Days 51-55

WHY THIS WEEK

A live URL is not production, and this document will not pretend otherwise. What deployment does give you is the first honest encounter with real network latency, real secret management, cold starts, and failures you did not simulate. You cannot reason credibly about production until you have run something in public, even something small.

DAY PLAN

Day 51	**Containerise.** A Dockerfile that builds and runs your service, plus Postgres with pgvector. Make it start from nothing with one command.
Day 52	**Deploy.** Push it to a managed platform. Render, Railway and Fly.io are all straightforward; Azure App Service or Container Apps suit .NET; Cloud Run suits anything containerised. Cheapest tier is fine.
Day 53	**Secrets and configuration properly.** Platform secret storage, not environment variables committed anywhere. Different keys for local and deployed. Confirm rotating a key needs no code change.
Day 54	**Measure the difference.** Run your eval suite against the deployed URL. Compare latency with local. The gap is network, cold start and provider region - explain each part.
Day 55	**Break it in public.** Revoke the API key while it is running. Fill the database connection pool. Send twenty concurrent requests. Confirm your week 2 resilience actually holds outside your laptop, and fix what does not.

DEPLOYMENT CHOICES BY STACK

OPTION	.NET	JAVA	NODE / TS	PYTHON
Simplest path	Azure App Service or Container Apps	Render or Fly.io container	Render, Railway or Fly.io	Render, Railway or Fly.io
Managed Postgres	Neon, Supabase, or your platform's managed Postgres. Confirm the pgvector extension is available before you commit to one.			
Watch out for	Cold starts inflating first-request latency, connection pool limits on small database tiers, and platform request timeouts shorter than your model timeout.			

DAY 55 DELIVERABLE

A live, publicly reachable system with managed secrets, that you have deliberately broken and repaired.

- [ ] Anyone with the URL can call it, and the eval suite passes against it
- [ ] No secret exists in the repository or in the container image
- [ ] You can explain the local-to-deployed latency gap component by component
- [ ] Revoked keys, pool exhaustion and concurrent load all degrade gracefully

WEEK 12

Actions if the product needs them, and the write-up that gets you hired

Days 56-60

WHY THIS WEEK

Two jobs. First, whether your product needs to perform actions at all - and if it does, the boundary that keeps that safe. Second, the documentation, which is not admin work at the end; for a hiring manager it is frequently the only part they read carefully.

PART A: ACTIONS, ONLY IF THE PRODUCT NEEDS THEM

In the video this arrives when a customer says "just cancel my order" and the bot tells them to click a button themselves. No equivalent moment in your project? Skip to part B - an excellent RAG system with real evals beats a half-built agent.

PROPOSE, VALIDATE, EXECUTE

The **model** proposes an action. Your **application** validates permissions and business rules. Your **business service** executes it. The model never touches the database or the payment provider directly, and never decides on its own authority that an action is allowed. The more autonomy you give the system, the stricter this boundary and its guardrails have to be - autonomy raises the importance of validation, it never removes it.

Day 56 **Expose two narrow tools.** Something like `getOrderStatus` and `cancelOrder`. Scoped, typed, and doing exactly one thing each.

Day 57 **Validate in your code, not in the prompt.** Ownership checks, eligibility windows, state checks. A prompt instruction is not an authorisation mechanism and must never be treated as one.

Day 58 **Audit and idempotency.** Log every proposed action, every validation result and every execution with a correlation id. Make execution idempotent so a retry cannot cancel an order twice.

PART B: THE DOCUMENTATION

Day 59 **Architecture diagram and README.** One diagram covering request, retrieval, model, validation, execution and observability. A README that gets a stranger running locally in ten minutes.

Day 60 **Trade-offs and limitations.** The document that does the real work. Every significant decision, the alternative you rejected, and why. Then an honest list of what your system does badly.

Writing your own limitations reads as senior. Claiming you have none reads as inexperienced.

TOOL CALLING BY STACK, IF YOU ARE DOING PART A

TASK	.NET	JAVA	NODE / TS	PYTHON
Function calling	M.E.AI tool calling; Agent Framework (`Microsoft.Agents.AI`) only if you need orchestration	Spring AI `@Tool` / ToolCallback, or LangChain4j `@Tool`	AI SDK `tools` with Zod schemas	Provider function calling, or the MCP Python SDK

MCP has official SDKs for all four stacks and is worth a day of reading given its 2026 adoption, but plain function calling is enough here - MCP is the integration layer above it, not a replacement.

DAY 90 DELIVERABLE

A live, evaluated, observable AI system, documented well enough that someone can judge your engineering without you in the room.

- [ ] Architecture diagram and a README that works for a stranger
- [ ] A trade-offs document with rejected alternatives and honest limitations
- [ ] Eval scores published in the repo, including the ones that are not flattering
- [ ] If you built actions: every action is validated in code, audited, and idempotent
- [ ] You can give the four-minute walkthrough on the next page from memory

03 How to talk about this in an interview

Ninety days of work is worth very little if you describe it as "I built a chatbot with RAG." Describe it as a sequence of production problems and the engineering that answered them, and it becomes evidence of judgement.

THE FOUR-MINUTE WALKTHROUGH

The problem	"A support assistant over company documents. The interesting part was not the model call - it was everything that had to sit around an unreliable, non-deterministic dependency to make it safe to expose to customers."
The failures, in order	"It answered confidently from stale knowledge, so I added retrieval. I could not tell whether changes made it better, so I built an eval set. Cost was invisible until it spiked, so I instrumented tokens and spend. Then it needed to perform actions, which forced a hard boundary between what the model proposes and what my application is willing to execute."
The numbers	"Baseline eval score X, currently Y. p95 latency Z. Roughly N paise per conversation. Refusal rate R percent, which I monitor because a rise in it usually means the knowledge base has gone stale."
The limitations	"It handles conflicting sources badly - it prefers recency, which is a default I chose rather than a solved problem. And my eval set is 30 questions, which is enough to catch regressions and not enough to claim general quality."

QUESTIONS YOU SHOULD BE ABLE TO ANSWER COLD

- **What happens when the provider is down?** Week 2. Give the specific behaviour and your worst-case latency and cost.
- **How do you know a prompt change improved anything?** Weeks 8 and 9. This is the question that separates candidates.
- **The retrieved passage was correct and the answer was still wrong. What now?** Week 8's split. Say the words "that is a generation failure, not a retrieval failure" and explain how you tell them apart.
- **How do you stop the model doing something destructive?** Week 12. Propose, validate, execute - with validation in code, never in the prompt.
- **Why not fine-tune?** Because the problem was missing current context, not missing capability. Fine-tuning would not have fixed a stale offer, and retrieval did.
- **Why pgvector and not a vector database?** Week 6. Name the scale threshold where the answer would change.

> **ONE THING NOT TO DO**
>
> Do not claim the project is production-grade or that it serves real traffic. Say what it is: a deliberately hardened portfolio system, deployed publicly, evaluated, with known limitations you can list. Overclaiming is the fastest way to lose a technical interviewer, and the honest version is already strong.

04 What is deliberately not in this roadmap

A roadmap is defined as much by what it refuses to include. Here is what was left out, and the honest reason for each. If your product later needs one of these, you will learn it then - which is the entire argument.

LEFT OUT	WHY
SKIP **Training or fine-tuning a model**	You did not need it once in 90 days, and neither do most AI engineering roles. Companies hire overwhelmingly to integrate and operate existing models, not to build them. Fine-tuning is a later specialisation, not an entry requirement.
SKIP **Advanced ML mathematics**	Not required for this transition. It matters if you move toward research or model work; it is not what stands between a backend developer and shipping a reliable AI product.
LATER **Agent frameworks and multi-agent systems**	Reach for these when a single validated tool call genuinely cannot express what your product needs. Starting here means orchestrating something you have not yet made reliable.
LATER **Fancy RAG variants**	Re-ranking, query rewriting, GraphRAG and the rest are real techniques. Adopt them when your week 8 eval set shows the specific failure they address - and then you will be able to prove they helped.
LATER **Self-hosting open-weight models**	A serious infrastructure discipline of its own. Worth doing when cost, latency or data residency actually demands it. Learning it before you have a working product is a detour.
CORE **Prompt engineering, in proportion**	Not a separate study block. It is a skill you develop continuously across weeks 3, 7 and 9, measured against evals rather than vibes. Treating it as the discipline itself is the mistake the video opens with.
CORE **Tokens, context windows, temperature**	Understand what they are and how they affect cost and behaviour - that is an afternoon. Do not turn foundational concepts into a months-long study block that delays building.

> **THE PATTERN TO TAKE AWAY**
>
> Every technology in this document arrived because a specific thing broke. Stale answers brought retrieval. An unanswerable question from a CEO brought evals. A surprise invoice brought observability. A frustrated customer brought tool calling. That order is not a teaching device - it is the order these problems genuinely appear in production. Learn the next thing when the problem arrives, and you will never need somebody else's list again.

05 Resources

Primary documentation over tutorials, wherever possible. Tutorials go stale quietly; provider and library documentation does not.

MODEL PROVIDERS – READ THE DOCS, NOT A COURSE

RESOURCE	WHERE	USE IT FOR
OpenAI Platform docs	`platform.openai.com/docs`	Structured outputs, function calling, prompt caching, streaming
Anthropic docs	`docs.anthropic.com`	Tool use, long-context patterns, prompt caching
Google AI for Developers	`ai.google.dev`	Gemini API, embeddings, structured output
OpenAI Cookbook	`cookbook.openai.com`	Working reference implementations, mostly Python - read them and port the idea

YOUR STACK

RESOURCE	WHERE	USE IT FOR
Microsoft.Extensions.AI (.NET)	`learn.microsoft.com/dotnet/ai`	The base abstraction to build on in 2026
Microsoft Agent Framework (.NET)	`devblogs.microsoft.com/agent-framework`	Only for week 12, only if you need agents
Spring AI (Java)	`docs.spring.io/spring-ai/reference`	ChatClient, structured output, VectorStore, advisors
LangChain4j (Java)	`docs.langchain4j.dev`	Alternative if you are not on Spring
Vercel AI SDK (Node / TS)	`vercel.com/docs/ai-sdk`	generateText, generateObject, streaming, tools
Resilience4j / Polly	`resilience4j.readme.io` · `github.com/App-vNext/Polly`	Week 2, and worth reading properly once

RETRIEVAL, EVALUATION, OBSERVABILITY

RESOURCE	WHERE	USE IT FOR
pgvector	`github.com/pgvector/pgvector`	Indexes, distance operators, and the client library for your language
promptfoo	`promptfoo.dev`	The language-neutral eval runner recommended in week 9
Ragas	`docs.ragas.io`	RAG-specific metrics: context precision and recall, faithfulness
DeepEval	`deepeval.com`	Python-native eval assertions in a familiar test-suite shape
Langfuse	`langfuse.com`	Self-hosted tracing, prompt management, and eval storage
Arize Phoenix	`arize.com/docs/phoenix`	Alternative to Langfuse, strong local experimentation workflow

OpenTelemetry GenAI conventions	`github.com/open-telemetry/semantic-conventions-genai`	Track the standard - and see for yourself that it is still pre-stable
Model Context Protocol	`modelcontextprotocol.io`	Specification and SDKs, if you go down the MCP route in week 12

WORTH READING PROPERLY, ONCE

RESOURCE	WHERE	WHY
Building effective agents	`anthropic.com/engineering`	The clearest published argument for using the simplest thing that works, from people who build these systems
Your provider's rate limit and pricing pages	`(provider specific)`	Genuinely. Most cost incidents are caused by engineers who never read them.

Day 91

You have a system that retrieves, refuses, measures itself, reports what it costs, and runs where other people can reach it. You never trained a model. You never needed the mathematics. What you did was take the engineering judgement you already had and extend it over a new kind of dependency - one that is probabilistic, expensive, and occasionally confidently wrong.

> *"I architected a system around a non-deterministic LLM using production-grade engineering practices, and quantified its reliability through evals, observability and cost metrics."*

That sentence is only worth saying because you can now be questioned on every clause of it. That is the whole difference, and it is why you were already most of the way there.