

// phase 4 deep dive

AI System Architecture



Design the whole system on paper first. This is where you make the real engineering decisions, and learn to defend them, that mark you as a top one percent candidate.

baswe.ai // become a software engineer

This is the in-depth guide to phase four of the Signal Project. Work through it once you have a chosen problem and a research brief from phase three. By the end you will have designed the entire system on paper, with a design document and an architecture diagram, without writing production code yet. In the next phase, you build it with a team.

Why architecture is where top candidates separate themselves

Anyone can wire together a tutorial. What makes a hiring manager see a top one percent engineer is the quality of the decisions behind the system and the ability to justify them. This phase is pure thinking. You design the whole system before writing a line of production code, and the design, along with your reasoning for it, is what proves you think like a real engineer rather than someone following steps.

At the senior level, engineering is not about typing. It is about tradeoffs and judgment: choosing between options that each have real costs, and being able to explain why you chose the one you did. This phase is where you demonstrate exactly that. It is also where you build the system mentally first, which makes the actual build fast and makes it possible to split the work across a team.

The difference between a good project and a top one percent project is not the code. It is the decisions. This phase is where you make them, and where you learn to defend them.

What you are producing in this phase

You are producing two artifacts, plus the reasoning underneath them. The first is a written design document. The second is an architecture diagram. Together they capture what you are building, how it works, and why you made the choices you did.

The design document itself is a portfolio asset that companies love, because most candidates cannot produce one, and it doubles as your interview script. You are not writing production code in this phase. You may run a tiny throwaway experiment to de-risk a single risky assumption, but the output is a design, not a product.

Frame the problem as a system

Before any AI decisions, define the boundaries of the system. Turn your problem and your research brief into a crisp specification by answering a few questions.

- ▶ Who is the user, and what job are they trying to get done with this?
- ▶ What are the inputs and outputs? What goes in, and what comes out?
- ▶ What does success look like, concretely? What does good actually mean here?
- ▶ What are the constraints? Latency, cost, available data, privacy, and scale.
- ▶ What is explicitly out of scope? Be ruthless, because you are building this in one to two months.

Write a one-paragraph problem statement and a short list of requirements. Split the requirements into functional requirements, which are what the system must do, and non-functional requirements, which are the constraints it must respect, like speed, cost, and reliability. Everything downstream is a response to this spec.

Decide whether you even need AI, and how much

One of the clearest signals of senior judgment is knowing when not to reach for a large language model. Many problems are best solved with simple heuristics, plain search, or classic machine learning, with generative AI used only where it genuinely adds value, such as understanding language, generating content, or reasoning over messy unstructured data.

Ask honestly whether your problem actually needs generative AI, or whether a simpler approach would work just as well. Use AI where the value is, and keep the rest of the system simple. Choosing the simplest thing that solves the problem is a mark of seniority. Reaching for AI everywhere is a junior tell, and experienced interviewers spot it immediately.

Choose your core AI pattern

This is the heart of the design. There are a handful of core patterns, and a top candidate picks the one that fits the problem and can explain why it beats the alternatives.

- **Direct prompting**

The simplest pattern: send a prompt to a model and use the response. It works when the task is self-contained and the model already knows enough to do it. It is cheap and fast to build, but it is limited by what the model knows and cannot use your private or current data.

- **Retrieval-augmented generation (RAG)**

Use this when the answer depends on specific, private, or current information the model does not already have. The system retrieves the relevant context first, then the model generates a grounded answer from it. It is the right fit for document question-answering, knowledge assistants, and search over a body of content. The tradeoff is that you now need a data pipeline and a retrieval step, and the quality of the whole system depends heavily on the quality of retrieval.

- **Agents and tool use**

Use this when the task requires the model to take actions, call tools or APIs, or reason across multiple steps, rather than just produce text. It fits automation and multi-step workflows. Agents are powerful but harder to make reliable, with more failure modes and higher cost and latency, so reach for them when the task genuinely needs to do things, not just when they sound impressive.

- **Fine-tuning**

Use this when you need a specific style, format, or behavior that prompting cannot reliably produce, and you have the data to train on. It is costly and data-hungry, and for most portfolio projects a good combination of prompting and retrieval will beat it. Know when it is the right tool, and know that it usually is not the first move.

- **Orchestration and combinations**

Real systems usually combine these. You might use retrieval to ground an agent, route easy requests to a cheap model and hard ones to a strong model, or chain several steps together. Pick the combination that fits your problem, and be ready to explain the flow.

Make the model and infrastructure decisions

- **Choosing a model**

The main decision is a hosted API, from a provider like OpenAI, Anthropic, or Google, versus an open-weight model you run yourself, like Llama or Mistral. The tradeoffs are capability, latency, cost, privacy and control, and how much you have to operate. For most portfolio projects a hosted API is the pragmatic choice, and the sophisticated move is to know exactly when self-hosting or an open-weight model matters, which is usually privacy, cost at scale, or control. Two more decisions signal maturity: choosing a smaller, cheaper, faster model where it is good enough instead of always reaching for the biggest one, and using different models for different steps, such as a cheap model for simple work and a strong model for the hard parts.

- **Data and retrieval design**

If you are using retrieval, this is where much of your system's quality is decided. Think through what data you need, how you ingest it, how you split it into chunks, which embedding model you use, which vector store you choose, whether you combine keyword and semantic search, whether you rerank results, and how you assemble the final context you hand to the model. Chunking and retrieval quality make or break a retrieval system, so treat these as real decisions, not defaults.

- **The surrounding stack**

The AI is one component inside a real product. Decide on the rest deliberately: the vector store, whether you use an orchestration framework or none, the backend and API, the frontend, the database, and where and how you deploy. Do not adopt a heavy framework you do not need just because a tutorial used it. Every tool in your stack should earn its place.

Design for reliability, cost, and evaluation

This is the section that most separates a top one percent candidate from everyone else, because almost no portfolio project thinks about it at all. Production thinking is what signals seniority.

- **Plan evaluation from the start**

This is the single biggest differentiator. Most projects have no way of knowing whether they actually work. Decide up front how you will measure success: build a small evaluation set of representative inputs with expected or acceptable outputs, define metrics that matter for your system, such as retrieval quality, answer correctness and faithfulness, or task success rate, and choose how you will measure them, whether by human review, by using a model as a judge, or by automated checks. Plan to evaluate offline while you build, and to watch real signals once it is live. Having an evaluation plan at all puts you ahead of the large majority of candidates.

- **Handle failure modes and reliability**

Language models fail in specific ways: they hallucinate, call the wrong tool, return malformed output, spike in latency, hit rate limits, or go down. Design for each. Ground answers with retrieval to reduce hallucination, validate and structure outputs so downstream code can trust them, add retries and fallbacks, put guardrails around inputs and outputs, and keep a human in the loop wherever the stakes are high. Naming your failure modes and your mitigations is exactly the kind of thinking interviewers probe for.

- **Manage cost and latency**

Token costs add up quickly, and slow responses ruin the experience. Design for both: cache where you can, choose right-sized models, limit how much context you send, stream responses so the interface feels fast, and batch work where it helps. You should be able to say roughly what your system costs per request and where its slowest parts are. That awareness reads as real engineering.

- **Address security and privacy**

Think about how data is handled, whether any of it is sensitive or personal, the risk of prompt injection, and whether you might leak private data into prompts or logs. This matters even more if your problem lives in a regulated industry like finance or healthcare, where getting this right is itself a strong signal to those companies.

Draw the architecture diagram

Turn the design into a single clear diagram, because it is the first thing anyone looks at. Show the user and entry point, the app and backend, the AI components and the pattern they use, including retrieval, model calls, and tools, the data stores such as your database and vector store, any external services, and the path a single request takes from start to finish. Keep it legible rather than fancy. Good tools for this include Excalidraw, draw.io, and Mermaid, or even a clean photo of a whiteboard. A diagram that lets someone understand your system in thirty seconds is worth more than a beautiful one they cannot follow.

Write the design document

Now write it up like a real engineering design document. This is the artifact that makes you look like an engineer companies want. Use a clear structure:

DESIGN DOC TEMPLATE (COPY THIS)

1. Problem statement and goals
2. Requirements (functional and non-functional)
3. Architecture overview (the diagram, plus a walkthrough)
4. Key decisions and tradeoffs
5. Evaluation plan
6. Risks, failure modes, and mitigations
7. Cost and scalability
8. Out of scope and future work

The key decisions and tradeoffs section is where you win. This is the exact conversation you will have in interviews, written down in advance. For every major choice, document it in the same simple format:

HOW TO DOCUMENT A DECISION (DO THIS FOR EVERY MAJOR CHOICE)

Decision. What you are deciding, for example: how to give the model access to the company's internal docs.

Options considered. The real alternatives, for example: fine-tuning, retrieval, or stuffing everything into the prompt.

Choice. What you picked.

Why. The reasoning, tied directly to your constraints.

Tradeoffs. What you gave up, and when you would revisit the decision.

A design document that clearly explains your decisions and tradeoffs is, by itself, one of the most impressive things you can put in front of a company. Most candidates cannot produce one. You will.

Validate the design before you build

Before you commit to building, de-risk the design with a few checks.

- ▶ Trace a real user scenario through the whole system, end to end, and confirm it actually solves the problem in your brief.
- ▶ Confirm it is buildable in one to two months by you and a small team. If it is not, cut scope now, not later.
- ▶ For your single riskiest assumption, such as whether retrieval will be good enough, consider a tiny throwaway experiment to test it. Do not start building the real product.
- ▶ Make sure you can explain every major decision and its alternative out loud. If you cannot, research until you can.

- ▶ If you can, sanity-check the design with a mentor or one of the people you connected with. A second set of eyes catches gaps early.

Common mistakes to avoid

- ▶ **Reaching for AI where something simpler is better.** Over-engineering with models everywhere is a junior signal. Use AI only where it earns its place.
- ▶ **Jumping to a framework or tool first.** Understand the problem and the design before you pick tools, not the other way around.
- ▶ **Skipping evaluation.** This is the biggest miss. If you cannot measure whether it works, you have not finished the design.
- ▶ **Ignoring cost, latency, and failure modes.** A design with no production thinking looks junior. Address them explicitly.
- ▶ **Designing something you cannot build in the timeframe.** Scope creep kills projects. Design for one to two months.
- ▶ **Copying a tutorial architecture without understanding it.** If you cannot justify every part, you cannot defend it in an interview.
- ▶ **Not writing it down.** The design doc and the decision log are the proof. Keep them.
- ▶ **Starting to build before the design is clear.** This phase is thinking. Finish the design, then build in the next phase.

What you should have before moving on

By the end of phase four, you should have:

- ▶ A crisp problem statement and a list of functional and non-functional requirements.
- ▶ A chosen core AI pattern and model, with the tradeoffs written down.
- ▶ A data and retrieval design, or an agent design, as applicable to your system.
- ▶ An evaluation plan that says how you will know it works.
- ▶ A plan for reliability, cost, and failure modes.
- ▶ A clear architecture diagram.
- ▶ A written design document, including a decision log of your key tradeoffs.
- ▶ The ability to explain every major decision out loud, without notes.

That design document and diagram are exactly what the next phase needs, and they are what let you split the work across a team. Do not move on until you can walk someone through your entire system and defend every decision in it.

→ **With your system designed and your design doc in hand, move on to Phase 5: Build (with your team).**

