System Design Template

A structured design template to guide the planning, architecture, and evaluation of LLM-based systems, with a focus on prompting, trade-offs, data flow, and practical implementation.

Section 1: What & Why

1. Problem Statement & Functional Requirements

Define what the system is solving and what it must do. Start by identifying the users and stakeholders. Understand their goals and what tasks the system needs to support. Think about essential features and how your system can improve on existing solutions.

Questions to guide you:
- Who are the users?
- What tasks should the system perform?
- What core and advanced features are needed?

Output:
- A short problem statement
- A clear list of functional requirements (e.g., retrieve context, generate response, log outputs, low latency

2. Success Criteria

Define what "success" looks like for the system using **clear, measurable outcomes**. These criteria help guide development, testing, and iteration.

Questions to Guide You:
- How will we know if the system is doing its job well?
- What measurable goals should the system hit?
- Are we prioritizing quality, speed, user satisfaction, or cost?

Output:
A short list (3–5 bullet points) of **qualitative and quantitative** success indicators tied to your problem statement.

3. Assumptions

List all conditions you're treating as true during the current phase of system design. These help clarify what you're relying on — and what may need revisiting if circumstances change.

Questions to Guide You:

- Are we assuming access to specific LLMs (e.g., GPT-4, Claude, Mistral)?
- Is the data already cleaned, structured, and available?
- Are we assuming a particular user interaction pattern (e.g., chat interface, API)?
- Are we assuming a certain prompt format or length limit?
- Will internet access or external API calls be allowed?

Output:
A bullet list of current design assumptions, which should be revisited during testing or deployment.

Section 2: How

1. **System-Level Tasks (Engineer POV)**

   This section breaks down the system into concrete technical tasks and components, focusing on how the functional requirements translate into engineering work.

   Questions to Ask:

   - What are the sequential steps from user input to final output?
   - What modular tools or components will we build? (e.g., retriever, ranker, prompt builder)
   - Which APIs or models will each component integrate with?
   - What are the expected input and output formats for each tool?
   - Will the system follow a pipeline architecture, agent-based orchestration, or a hybrid?
   - How will components communicate or pass data among themselves

   Output:
   A clear outline of system tasks, tools, integrations, and their interactions from an engineering perspective.

2. **Data:**

   Detail the kind of data your system will process, how it's stored, and how it's queried. This ensures your design aligns with storage, retrieval, and performance constraints.

   Questions to Ask:

   - What data types will the system use? (e.g., text, PDFs, images, audio, structured tables)
   - What is the data source? (e.g., internal knowledge base, user uploads, third-party APIs)
   - Where will the data be stored? (e.g., vector databases, SQL, NoSQL, S3, file system)
   - Will we transform raw data into embeddings or preprocessed formats

   Output:
   A summary of:

   - Data types
   - Source(s)
   - Storage system(s)

3. **Prompting Techniques**

   Define how inputs will be structured for the LLM to optimize performance.

   Questions to Ask:

   - From the list of prompting techniques we studied in our course, which ones should be applied for this use case?

   Output:
   List of chosen prompting strategies and how/where they'll be applied.

4. **Validation: Evaluation Metrics**

   Outline how you'll measure the model's performance and reliability.

Questions to Ask:

- What metrics best reflect performance? (Accuracy, Reliability, Hallucinations)
- How will be evaluate each of this metrics
- Can we include human evaluations (A/B testing)?
- How will we detect and measure hallucinations?
- Do we need task-specific benchmarks on which we improve our performance?

Output:
Set of evaluation metrics (automated + human) to assess output quality and system effectiveness.

5. Trade-offs: Accuracy vs Hallucination vs Cost

Define how you'll balance competing priorities—output quality, truthfulness, and system cost.

Questions to Ask:

- Are we optimising for factual accuracy, creativity, or speed?
- What level of hallucination is acceptable for our use case?
- Can we use smaller/cheaper models without losing key quality?
- Where can caching or retrieval reduce cost without hurting accuracy?

Output:
A clear trade-off strategy tailored to your system's goals and constraints.

6. Experiment Setup

Plan structured experiments to validate key design choices and optimize performance.

Questions to Ask:

- What hypotheses are we testing (e.g., prompt style impact, model choice)?
- Can we A/B test different prompting strategies or model configs?
- How do we reduce token usage while still getting meaningful results?
- Are we testing short vs long prompts, or with/without context?

Output:
List of experiments, expected outcomes, and resource constraints (e.g., budget, tokens).

7. Other: Monitoring, Logging, Security & Ethics

Plan for safe, reliable, and accountable system behavior in production.

Questions to Guide You:

- How will we monitor output quality, latency, and token usage?
- What should we log (prompts, responses, errors)?
- Are we securely handling user data and avoiding PII exposure?
- Could the system produce biased or harmful outputs?

Output:
A basic strategy for observability, data privacy, and ethical safeguards.

Section 3: System Architecture Overview

1. System Context Diagram

A System Context Diagram provides a high-level visual representation of your system and its interactions with external entities. It sets the boundaries of your system, showing what's *inside* (your components) and what's *outside* (users, APIs, databases, services).

The goal is to answer:
"Who interacts with the system, and how?"

It typically includes:

- External actors: End users, admin users, other services, or APIs.
- System boundary: The box representing your system.
- Interfaces: How data flows in and out of the system — user requests, API calls, database queries, file uploads, etc.

Think of it as a "big picture" overview before diving into internal components or data flow. It's especially useful for aligning stakeholders and clarifying integration points early on.

Refer this video to understand it better: Context Diagrams Overview

2. Data Flow Diagram

A Data Flow Diagram (DFD) shows how data moves within the system — from input to output — across internal components. It focuses on how information is processed, not just who interacts with the system.

It typically includes:

- Processes/components: Tools that we learned in our class
- Data stores: Where data is read from or written to (e.g., databases, file systems).
- Data flows: Arrows that show how data moves from one component to another.
- External inputs/outputs: User queries, API calls, or files entering/leaving the system.

Think of it as the step-by-step execution map:
User input → Prompt generation → LLM inference → Response

This helps engineers identify:

- Where transformations occur
- When models are called
- What dependencies or latencies exist between components

It's essential for debugging, optimising pipelines, and planning infrastructure.

Refer to this video to understand it better: Data Flow Diagram Overview

3. UX/UI Design

This defines the user interface and how users interact with the system, including how inputs are entered and how outputs are displayed. It ensures the system is usable and meets user needs, balancing simplicity and flexibility. Basically the web design layout of your product.