

The Ultimate AI Engineering Cheatsheet 2025













InferenceStack

Enterprise-grade AI Infrastructure

Created by Matt Vegas, Founder of InferenceStack www.inference-stack.com

The modern AI stack, explained: From prompts to production.

Table of Contents

1.  [Model Orchestration & LLM Infrastructure](#)
2.  [Embedding & Retrieval](#)
3.  [Inference Optimization & Deployment](#)
4.  [Evaluation & Observability](#)
5.  [Security, Compliance & Clinical Safety](#)
6.  [Reusable Engineering Patterns](#)
7.  [Fine-Tuning & Custom Models](#)
8.  [Prompt Engineering Patterns](#)
9.  [UI/UX Design for AI Applications](#)
10.  [Ultimate Tech Stack for 2025](#)

1. Model Orchestration & LLM Infrastructure

Building intelligent systems requires more than just prompting an LLM. This section focuses on orchestrating multi-step reasoning, handling memory, chaining tools, and structuring systems beyond single-turn chat.

Key Concepts

- Prompt Chaining Patterns

- Zero-shot: no examples, relies on model's generalization abilities
- Few-shot: embeds in-context examples for format/behavior priming
- RAG (Retrieval-Augmented Generation): combines search + LLM for grounded responses
- ReAct (Reasoning + Acting): interleaves reasoning steps with tool actions

• Memory Types

- BufferMemory : full chat transcript
- SummaryMemory : condensed history for long chains
- VectorMemory : semantic state using embeddings
- EntityMemory : tracks named entities across conversations

• Structured Output

- Using `response_format="json"` or `function_call` mode (OpenAI)
- Schema validation with Zod or Pydantic
- Reduces hallucinations and parsing issues

• Tool Use and Agents

- Tool routing: dynamically select between APIs, calculators, docs, search
- Agents: systems that plan, reason, and act using tool chains
- Autonomous systems: agents that make independent decisions

Code Example: LangGraph Agent Workflow

```
import { StateGraph } from "langgraph";
import { AIMessage, HumanMessage } from "langchain/schema";
import { ChatOpenAI } from "langchain/chat_models/openai";

// Create a model instance
const model = new ChatOpenAI({
  model: "gpt-4.5-turbo",
  temperature: 0
});

// Define agent states
const states = {
  plan: async (input, state) => {
    const result = await model.invoke([
      new HumanMessage(`Create a plan to solve: ${input.task}`)
    ]);
    return { plan: result.content, task: input.task };
  },
};
```

```

act: async (input, state) => {
  const result = await model.invoke([
    new HumanMessage(`Execute this plan: ${state.plan}\nTask: ${state.task}`)
  ]);
  return { ...state, action_result: result.content };
},

evaluate: async (input, state) => {
  const result = await model.invoke([
    new HumanMessage(`Evaluate if the task is complete:
    Task: ${state.task}
    Plan: ${state.plan}
    Result: ${state.action_result}

    Return ONLY "complete" or "incomplete"`)
  ]);

  return {
    ...state,
    evaluation: result.content.toLowerCase(),
    complete: result.content.toLowerCase().includes("complete")
  };
}
};

// Create the state graph
const graph = new StateGraph({
  channels: {
    task: { value: "" }
  }
});

// Add nodes
graph.addNode("plan", states.plan);
graph.addNode("act", states.act);
graph.addNode("evaluate", states.evaluate);

// Define edges
graph.addEdge("plan", "act");
graph.addEdge("act", "evaluate");
graph.addConditionalEdges(
  "evaluate",
  (state) => state.complete ? "end" : "plan"
);

// Compile
const runnable = graph.compile();

```

```
// Execute
const result = await runnable.invoke({
  task: "Research and summarize recent breakthroughs in fusion energy"
});
```

Use LangGraph when building complex workflows that need retry logic, dynamic routing, or step memory. Think of it like a programmable agent runtime.

Code Example: Function Calling for Structured Output

```
import OpenAI from "openai";

const openai = new OpenAI({
  apiKey: process.env.OPENAI_API_KEY,
});

const functions = [
  {
    name: "search_products",
    description: "Search for products in the catalog",
    parameters: {
      type: "object",
      properties: {
        query: {
          type: "string",
          description: "The search query",
        },
        category: {
          type: "string",
          enum: ["electronics", "clothing", "home", "beauty"],
          description: "Product category",
        },
        max_price: {
          type: "number",
          description: "Maximum price in USD",
        },
        sort_by: {
          type: "string",
          enum: ["relevance", "price_low", "price_high", "rating"],
          description: "How to sort results",
        },
      },
      required: ["query"],
    },
  },
];
```

```

async function handleUserMessage(userMessage) {
  const response = await openai.chat.completions.create({
    model: "gpt-4-turbo",
    messages: [
      { role: "system", content: "You are a helpful shopping assistant." },
      { role: "user", content: userMessage }
    ],
    tools: functions,
    tool_choice: "auto",
  });

  const message = response.choices[0].message;

  if (message.tool_calls) {
    const functionCall = message.tool_calls[0];
    const functionArgs = JSON.parse(functionCall.function.arguments);

    // Now you can use these structured parameters
    console.log(`Searching for: ${functionArgs.query}`);
    console.log(`Category: ${functionArgs.category || "all"}`);
    console.log(`Max price: ${functionArgs.max_price || "unlimited"}`);

    // Perform the actual search with these parameters
    const searchResults = await performProductSearch(functionArgs);
    return searchResults;
  }

  // Regular response
  return message.content;
}

// Usage
const results = await handleUserMessage("I need a new laptop under $1000");

```

When to Use Orchestration

- Multi-tool agent flow: LangGraph + LangChain tools
- Knowledge grounding (RAG): LangChain + Supabase + OpenAI
- Multi-step task breakdown: ReAct or Plan-Act-Eval agents
- Real-time UI interactions: Vercel AI SDK + stream utils
- Multi-agent systems: Simulate multiple actors coordinating in complex tasks

Engineering Tips

- Avoid nesting agents in agents — use graphs or state machines instead

- Use prompt compression (e.g., summarize, extract) to avoid token overflows
- Monitor LLM latency per step — some orchestration stacks add significant overhead
- Design explicit fallback paths for each orchestration step
- Use validation layers between steps to catch hallucinations early

Recommended Tools

- LangGraph – async, state-based agent design
- OpenAI Function Calling – predictable structured output
- LangChain Expression Language (LCEL) – declarative chaining with observability
- Vercel AI SDK – production-ready streaming for LLM chains
- crepe (by Modal) – lightweight, high-performance agent orchestration

2. Embedding & Retrieval

When grounding your LLM in factual context, semantic search and embedding strategies make or break the quality of your system. This section covers best practices for chunking, embedding, storing, and retrieving content.

Key Concepts

- **Chunking Strategy**
 - Chunk by semantic units (paragraphs, topics), not fixed tokens
 - Use overlapping windows (e.g., 200 tokens with 40 overlap) to preserve context
 - Headers and metadata should be repeated in relevant chunks
- **Embedding Models**
 - OpenAI: `text-embedding-3-small` (1,536 dimensions) for speed, `text-embedding-3-large` (3,072 dimensions) for fidelity
 - BGE-Small (BAAI) for multilingual open-weight use cases
 - Cohere Embed v3 for documents with longer structure
 - Voyage-2 for complex querying and multimodal tasks
- **Vector Stores**
 - `pgvector` + Supabase: fast, self-hosted, Postgres-native
 - Weaviate: schema-rich, scalable, and Python/JS native

- Pinecone: great for hybrid search + metadata filters at scale
- Qdrant: open-source, high-performance, flexible schema
- **Metadata Design**
 - Store fields like `doc_id`, `section`, `type`, `created_at`
 - Keep sensitive data out of vector storage — never embed raw PHI
 - Use filterable fields for dynamic query construction
- **Retrieval Techniques**
 - Basic vector similarity search (cosine, dot product, Euclidean)
 - Hybrid search: combine vector similarity with keyword/BM25
 - Re-ranking: use a second model to refine initial results
 - Query expansion: generate multiple queries to improve recall

Code Example: Chunk & Embed Pipeline

```
import { MarkdownTextSplitter } from "langchain/text_splitter";
import { OpenAIEmbeddings } from "langchain/embeddings/openai";
import { SupabaseVectorStore } from "langchain/vectorstores/supabase";
import { createClient } from "@supabase/supabase-js";

// Initialize services
const openAIEmbeddings = new OpenAIEmbeddings({
  model: "text-embedding-3-small",
  dimensions: 1536,
});

const supabase = createClient(
  process.env.SUPABASE_URL,
  process.env.SUPABASE_API_KEY
);

// Chunk documents with semantic awareness
async function processDocument(docText, metadata) {
  // Split by markdown headers and paragraphs
  const splitter = new MarkdownTextSplitter({
    chunkSize: 500,
    chunkOverlap: 50,
  });

  const chunks = await splitter.splitText(docText);

  // Add positional and document metadata to each chunk
```

```
const docsWithMetadata = chunks.map((chunk, i) => ({
  pageContent: chunk,
  metadata: {
    ...metadata,
    chunk_id: i,
    chunk_count: chunks.length,
    source_doc: metadata.title,
    created_at: new Date().toISOString(),
  }
}));

// Store in vector database
await SupabaseVectorStore.fromDocuments(
  docsWithMetadata,
  openAIEmbeddings,
  {
    client: supabase,
    tableName: "documents",
  }
);

return {
  chunkCount: chunks.length,
  docId: metadata.id,
};
}

// Query with metadata filtering
async function queryDocuments(query, filters = {}) {
  const vectorStore = await SupabaseVectorStore.fromExistingIndex(
    openAIEmbeddings,
    {
      client: supabase,
      tableName: "documents",
    }
  );

  // Perform similarity search with metadata filtering
  const results = await vectorStore.similaritySearch(
    query,
    5, // Top 5 results
    filters // E.g., { category: "technical", created_at: { $gt: "2023-01-01" } }
  );

  return results;
}

// Usage
const metadata = {
```



```

    id: "doc-123",
    title: "AI Engineering Handbook",
    author: "Jane Smith",
    category: "technical",
};

await processDocument(markdownText, metadata);

const results = await queryDocuments(
  "How do I implement hybrid search?",
  { category: "technical" }
);

```

Advanced Example: Hybrid Search with Re-ranking

```

from langchain.retrievers import ContextualCompressionRetriever
from langchain.retrievers.document_compressors import LLMChainExtractor
from langchain.chat_models import ChatOpenAI
from langchain.vectorstores import PGVector
from langchain.embeddings import OpenAIEmbeddings

# Create base retriever
embeddings = OpenAIEmbeddings(model="text-embedding-3-small")
vectordb = PGVector.from_existing_index(
    embedding=embeddings,
    connection_string="postgresql://user:pass@localhost:5432/vectordb"
)
base_retriever = vectordb.as_retriever(
    search_type="hybrid", # Combines vector search with BM25 keyword search
    search_kwargs={"k": 10} # Get 10 initial candidates
)

# Create re-ranking compressor
llm = ChatOpenAI(model="gpt-4o", temperature=0)
compressor = LLMChainExtractor.from_llm(llm)

# Create final retriever with re-ranking
retriever = ContextualCompressionRetriever(
    base_compressor=compressor,
    base_retriever=base_retriever
)

# Usage
query = "What are the security implications of embedding PHI?"
compressed_docs = retriever.get_relevant_documents(query)

```

When to Use Retrieval Augmentation

- Answering questions from internal documentation
- Aligning LLMs with proprietary knowledge
- Augmenting memory for multi-turn chat agents
- Regulatory-compliant auditing or evidence-backed explanations
- Customer support that requires specific product knowledge
- Academic or research applications requiring citation

Engineering Tips

- Avoid re-embedding unchanged data — cache your vector IDs
- Store chunk index + length in metadata for traceability
- If using `text-embedding-3-large`, batch inputs to avoid token limits
- Implement multi-stage retrieval for higher quality results:
 - i. Initial broad retrieval (high recall)
 - ii. Re-rank to improve precision
 - iii. Format context for optimal LLM consumption
- Log retrieval scores to detect poor matches and adjust thresholds

Recommended Tools

- OpenAI Embedding API – scalable, well-documented
- Supabase + pgvector – cost-effective and SQL-native
- LlamaIndex or LangChain Retriever – plug-and-play retrievers
- Weaviate – production-grade hybrid search with filtering and auth
- txtai – lightweight embedding database with built-in workflows
- ChromaDB – open-source, easy setup for quick prototyping

3. ⚡ Inference Optimization & Deployment

Once your chain or agent is working, it's time to ship it somewhere fast. This section focuses on speed, cost-efficiency, and stream-ready deployment of LLMs into production environments.

Key Concepts

- Latency Optimization

- Use streaming responses to reduce perceived delay
 - Warm models in memory if deploying on cold-start platforms
 - Implement client-side caching for repeated queries
 - Consider batching for high-volume systems
- **Cost Optimization**
 - Use the cheapest model that meets quality requirements
 - Limit prompt and completion tokens with thoughtful design
 - Cache common responses to reduce redundant inference
 - Implement token counting and budgeting systems
 - **Hosting Models**
 - Modal: dynamic GPU inference jobs with cold start isolation
 - Fireworks AI: OpenAI-compatible API, lower latency than native
 - Groq: ultra-fast open-weight models (LLaMA 3, Mixtral)
 - AWS SageMaker: enterprise-grade model hosting with autoscaling
 - **Streaming & UI Integration**
 - Vercel AI SDK or `ReadableStream`-based handlers for async LLM outputs
 - Combine with debounce and partial rendering in frontend
 - Use Server-Sent Events (SSE) or WebSockets for real-time updates
 - **Model Caching**
 - Avoid redundant inference by caching recent queries and results
 - Use Redis, Upstash, or a thin SQLite key-value layer
 - Implement LRU caching for high-volume applications

Code Example: Optimized Streaming with Caching

```
import { OpenAI } from "openai";
import { Redis } from "@upstash/redis";
import { streamText } from "ai";

// Initialize clients
const openai = new OpenAI({
  apiKey: process.env.OPENAI_API_KEY,
});
```

```
const redis = new Redis({
  url: process.env.UPSTASH_REDIS_URL,
  token: process.env.UPSTASH_REDIS_TOKEN,
});

// Create a cache key from request parameters
function createCacheKey(messages) {
  const lastMessage = messages[messages.length - 1].content;
  return `chat:${Buffer.from(lastMessage).toString('base64')}`;
}

// Streaming handler with cache
export async function POST(req) {
  const { messages } = await req.json();
  const cacheKey = createCacheKey(messages);

  // Check cache first
  const cachedResponse = await redis.get(cacheKey);

  if (cachedResponse) {
    // Return cached response as a stream to maintain UX consistency
    return streamText(cachedResponse);
  }

  // No cache hit, call the API
  const response = await openai.chat.completions.create({
    model: "gpt-4-turbo",
    messages,
    stream: true,
  });

  // Create a TransformStream to collect the full response while streaming
  const { readable, writable } = new TransformStream();
  const writer = writable.getWriter();

  let fullResponse = "";

  // Process the stream
  (async () => {
    for await (const chunk of response) {
      const content = chunk.choices[0]?.delta?.content || "";
      if (content) {
        fullResponse += content;
        writer.write(content);
      }
    }
  })();

  // Cache the full response (with TTL of 1 hour)
  await redis.set(cacheKey, fullResponse, { ex: 3600 });
}
```

```

        writer.close();
    })();

    return new Response(readable);
}

```

Code Example: Parallel Inference with Batching

```

import asyncio
from typing import List, Dict
from openai import AsyncOpenAI

client = AsyncOpenAI()

async def process_batch(batch: List[Dict]):
    """Process multiple prompts in parallel"""
    async def process_item(item):
        try:
            response = await client.chat.completions.create(
                model="gpt-4o",
                messages=[
                    {"role": "system", "content": item["system"]},
                    {"role": "user", "content": item["user"]}
                ],
                temperature=0.7,
                max_tokens=500
            )
            return {
                "id": item["id"],
                "result": response.choices[0].message.content,
                "error": None
            }
        except Exception as e:
            return {
                "id": item["id"],
                "result": None,
                "error": str(e)
            }

    # Run all requests concurrently
    tasks = [process_item(item) for item in batch]
    results = await asyncio.gather(*tasks)
    return results

# Example usage
async def main():
    batch = [

```

```

    {
        "id": "user_1",
        "system": "You are a helpful assistant.",
        "user": "What is machine learning?"
    },
    {
        "id": "user_2",
        "system": "You are a helpful assistant.",
        "user": "Explain neural networks."
    },
    # Add more items...
]

```

```

results = await process_batch(batch)
return results

```

```

# Run the batch processing
results = asyncio.run(main())

```

When to Optimize for Speed

- Agents with multiple tool calls or long inference chains
- Public-facing tools or chat UIs where user experience matters
- Evaluation loops where latency compounds quickly
- Production systems with high concurrency requirements
- Mobile applications where perceived performance is critical

Engineering Tips

- Deploy hot paths on edge (e.g., Vercel Edge Functions)
- Consider token streaming even if you're not rendering it
- Precompute static completions (e.g., tool lists, docs summaries)
- For batch inference, implement async processing with rate limiting
- Monitor token usage per request to detect prompt inefficiencies
- Use a token counting library to estimate costs before sending requests

Recommended Tools

- Vercel AI SDK – for seamless streaming in serverless
- FireworksAI – great GPT-compatible inference with multi-model support
- Modal – code-first, batch and async inference workflows
- Groq API – real-time performance for LLaMA/Mistral models

- Upstash Redis – serverless-friendly caching layer
- NVIDIA TensorRT-LLM – high-performance inference optimization for on-prem

4. Evaluation & Observability

Evaluating your LLM outputs isn't just about correctness—it's about reliability, trust, and long-term performance. This section covers the tools and techniques to measure quality, debug failures, and optimize the feedback loop.

Key Concepts

- **Eval Types**
 - String Match: Compare against ground truth with basic similarity
 - Embedding Similarity: Vector distance to gold label
 - LLM-as-Judge: Use GPT/Claude to critique and score outputs
 - Human Feedback: Structured user evaluations with qualitative notes
- **Evaluation Metrics**
 - Relevance: Does the answer address the question?
 - Factuality: Are statements truthful and accurate?
 - Conciseness: Is the response appropriately brief?
 - Helpfulness: Does it provide useful information?
 - Toxicity: Does it contain harmful content?
- **Observability Metrics**
 - Response latency (p95, p99)
 - Token cost per call
 - Failure rates and empty completions
 - Toxicity, hallucination, and refusal rates
 - User satisfaction scores
- **Feedback Integration**
 - Collect thumbs-up/down or user selections
 - Store alongside input metadata for future fine-tuning
 - Use structured forms to collect true/false labels

- Implement A/B testing framework for prompt variations

Code Example: RAG Evaluation with Ragas

```

from datasets import Dataset
from ragas.evaluation import evaluate
from ragas.metrics import (
    answer_relevancy,
    faithfulness,
    context_precision,
    context_recall
)

# Create a dataset with questions, generated answers, and retrieved contexts
eval_dataset = Dataset.from_dict({
    "question": [
        "What is HIPAA?",
        "When was HIPAA enacted?",
        "What data is protected under HIPAA?"
    ],
    "answer": [
        "HIPAA is a US health privacy law that protects patient information.",
        "HIPAA was enacted in 1996 by the US Congress.",
        "Protected Health Information (PHI) includes identifiable health data."
    ],
    "contexts": [
        ["HIPAA stands for the Health Insurance Portability and Accountability Act, a US"],
        ["The Health Insurance Portability and Accountability Act (HIPAA) was enacted by"],
        ["Under HIPAA, protected health information (PHI) includes names, addresses, dat
    ]
})

# Run the evaluation with multiple metrics
results = evaluate(
    eval_dataset,
    metrics=[
        faithfulness,          # Are statements supported by the context?
        answer_relevancy,     # Is the answer relevant to the question?
        context_precision,    # How precisely was the context used?
        context_recall        # How much relevant context was incorporated?
    ]
)

print(f"Faithfulness score: {results['faithfulness']:.2f}")
print(f"Answer relevancy score: {results['answer_relevancy']:.2f}")
print(f"Context precision score: {results['context_precision']:.2f}")
print(f"Context recall score: {results['context_recall']:.2f}")

```


Code Example: LLM-as-Judge Evaluation

```
import OpenAI from "openai";

const openai = new OpenAI({
  apiKey: process.env.OPENAI_API_KEY,
});

async function evaluateResponse(question, answer, criteria) {
  const prompt = `
You are an expert evaluator of AI responses. Rate the following response to a user question.

Question: ${question}

AI Response: ${answer}

Please evaluate this response on the following criteria on a scale of 1-5, where:
1 = Poor, 2 = Fair, 3 = Good, 4 = Very Good, 5 = Excellent

For each criterion, provide:
1. A numerical score (1-5)
2. A brief explanation of the rating

${criteria.map(c => `- ${c}`).join('\n')}

Format your response as a JSON object with this structure:
{
  "criteria": {
    "criterion_name": {
      "score": number,
      "explanation": "string"
    },
    ...
  },
  "overall_score": number,
  "summary": "string"
}
`;

  const response = await openai.chat.completions.create({
    model: "gpt-4-turbo",
    messages: [
      { role: "system", content: "You are an expert evaluator of AI responses." },
      { role: "user", content: prompt }
    ],
    response_format: { type: "json_object" }
  });
}
```

```

    return JSON.parse(response.choices[0].message.content);
  }

  // Usage example
  const evaluation = await evaluateResponse(
    "How do embeddings work in RAG systems?",
    "Embeddings in RAG systems convert text into vectors, which allows for semantic similar
    [
      "Accuracy",
      "Completeness",
      "Clarity",
      "Conciseness",
      "Helpfulness"
    ]
  );

  console.log(evaluation);

```

Implementing a Feedback Collection System

```

// Database schema for feedback
type Feedback = {
  id: string;
  userId: string;
  sessionId: string;
  questionId: string;
  rating: "positive" | "negative" | "neutral";
  category?: "accuracy" | "clarity" | "helpfulness" | "other";
  comment?: string;
  modelVersion: string;
  promptTemplate: string;
  timestamp: Date;
};

// Frontend component for feedback collection
function FeedbackComponent({ questionId, responseId }) {
  const [rating, setRating] = useState(null);
  const [category, setCategory] = useState(null);
  const [comment, setComment] = useState("");

  const submitFeedback = async () => {
    await fetch("/api/feedback", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({
        questionId,
        responseId,

```

```

        rating,
        category,
        comment
    })
  });
};

return (
  <div className="feedback-container">
    <p>Was this response helpful?</p>
    <div className="feedback-buttons">
      <button onClick={() => setRating("positive")}>
        👍 Yes
      </button>
      <button onClick={() => setRating("negative")}>
        👎 No
      </button>
    </div>

    {rating === "negative" && (
      <>
        <p>What could be improved?</p>
        <select value={category} onChange={(e) => setCategory(e.target.value)}>
          <option value="">Select a category</option>
          <option value="accuracy">Accuracy</option>
          <option value="clarity">Clarity</option>
          <option value="helpfulness">Helpfulness</option>
          <option value="other">Other</option>
        </select>

        <textarea
          placeholder="Additional feedback (optional)"
          value={comment}
          onChange={(e) => setComment(e.target.value)}
        />
      </>
    )}

    <button onClick={submitFeedback} disabled={!rating}>
      Submit Feedback
    </button>
  </div>
);
}

```

When to Use Evaluation Pipelines

- Comparing different prompt templates or model vendors

- Validating structured output consistency
- Measuring hallucination or completeness of retrieval
- During fine-tuning / RAG QA system dev
- Continuous monitoring of production systems
- Comparing versions of your AI system before deployment

Engineering Tips

- Use synthetic test sets generated by the model itself for early iterations
- Validate chain consistency (e.g., tools called in correct order)
- Flag all incomplete, null, or fallback responses for inspection
- Store model version, temperature, and runtime with every eval record
- Implement dashboards to track quality metrics over time
- Create a red team process for adversarial testing

Recommended Tools

- Ragas – ground truth + context-based evaluation for RAG pipelines
- Promptfoo – eval CLI + dashboards for LLM completions
- TruLens – production LLM observability + tracing + user feedback
- LangSmith – LangChain-native evals, traces, and feedback capture
- DeepEval – regression testing for LLM applications
- Giskard – AI quality tests including fairness and robustness testing

5. Security, Compliance & Clinical Safety

For AI systems in regulated industries—especially healthcare—compliance isn't an afterthought. This section covers the techniques, frameworks, and patterns used to ensure your systems are audit-ready and privacy-safe.

Key Concepts

- **Privacy by Design**
 - Remove or mask PHI (Protected Health Information) before embedding or logging
 - Use data minimization in prompts and retrieval
 - Implement role-based access controls (RBAC)

- Apply least-privilege principles to all system components
- **Access Control**
 - Implement Role-Based Access Control (RBAC) for sensitive endpoints
 - Log every access to model inference, especially when dealing with patient data
 - Use fine-grained permissions for different AI capabilities
- **Data Handling**
 - Avoid storing raw prompts that include PHI or proprietary data
 - Use encrypted storage for any saved chat or context history
 - Implement proper data retention and deletion policies
- **Compliance Frameworks**
 - HIPAA (US healthcare): governs PHI transmission and access
 - GDPR (EU): enforces consent, data access, and right to be forgotten
 - SOC 2: audit framework for data security and operational integrity
 - FDA (for medical AI): regulatory pathway for medical devices
- **Prompt Injection Prevention**
 - Validate and sanitize user inputs before passing to LLMs
 - Use parameterized templates with clear boundaries
 - Implement guardrails to detect potential attacks
 - Monitor for unusual patterns in user requests

Code Example: PHI Redaction with SpaCy + Regex

```
import spacy
import re
from typing import List, Dict, Any

# Load the NER model
nlp = spacy.load("en_core_web_sm")

# Define PHI patterns to redact
PHI_PATTERNS = [
    # SSNs
    re.compile(r"\b\d{3}-\d{2}-\d{4}\b"),
    # Dates
    re.compile(r"\b\d{1,2}/\d{1,2}/\d{2,4}\b"),
    # Phone numbers
```

```

re.compile(r"\b(\d{3})\s*\d{3}-\d{4}\b"),
re.compile(r"\b\d{3}-\d{3}-\d{4}\b"),
# Medical record numbers
re.compile(r"\bMRN\s*[:=]?s*\d{5,10}\b", re.IGNORECASE),
# Hospital names
re.compile(r"\b[A-Z][a-z]+ Hospital\b"),
re.compile(r"\b[A-Z][a-z]+ Medical Center\b")
]

def redact_phi(text: str) -> Dict[str, Any]:
    """
    Redact PHI from text using both NER and regex patterns.
    Returns the redacted text and a count of redactions.
    """
    redacted = text
    redaction_count = 0

    # Use NER to find and redact named entities
    doc = nlp(text)
    for ent in doc.ents:
        if ent.label_ in ["PERSON", "ORG", "GPE", "DATE"]:
            redacted = redacted.replace(ent.text, f"[REDACTED:{ent.label_}]")
            redaction_count += 1

    # Use regex patterns for structured PHI
    for pattern in PHI_PATTERNS:
        matches = pattern.findall(redacted)
        redaction_count += len(matches)
        redacted = pattern.sub("[REDACTED:PHI]", redacted)

    return {
        "redacted_text": redacted,
        "redaction_count": redaction_count,
        "has_phi": redaction_count > 0
    }

# Usage example
def process_clinical_text(user_input: str):
    redaction_result = redact_phi(user_input)

    if redaction_result["has_phi"]:
        print(f"WARNING: {redaction_result['redaction_count']} instances of PHI were detected")

    # Now safe to use with LLM
    return redaction_result["redacted_text"]

# Example
text = "Patient John Smith (MRN: 12345678) visited Central Hospital on 04/15/2023. His phone number is 123-456-7890."
safe_text = process_clinical_text(text)

```

```
print(safe_text)
```

```
# Output: "Patient [REDACTED:PERSON] (MRN: [REDACTED:PHI]) visited [REDACTED:ORG] on [REDACTED:DATE]"
```

Code Example: RBAC for AI Services

```
import { NextRequest, NextResponse } from 'next/server';
import { getAuth } from '@clerk/nextjs/server';
import { hasPermission } from '@lib/permissions';

// Define permission levels for different AI capabilities
enum AICapability {
  BASIC_CHAT = 'basic_chat',      // Simple conversational features
  DOCUMENT_ANALYSIS = 'doc_analysis', // Upload and analyze documents
  CODE_GENERATION = 'code_gen',   // Generate code
  CLINICAL_ADVICE = 'clinical',   // Medical/health related responses
  ADMIN_TOOLS = 'admin'          // System administration capabilities
}

// AI service role-capability mapping
const roleCapabilities = {
  'user': [AICapability.BASIC_CHAT],
  'premium_user': [AICapability.BASIC_CHAT, AICapability.DOCUMENT_ANALYSIS, AICapability.CLINICAL_ADVICE],
  'clinician': [AICapability.BASIC_CHAT, AICapability.DOCUMENT_ANALYSIS, AICapability.CLINICAL_ADVICE],
  'admin': [AICapability.BASIC_CHAT, AICapability.DOCUMENT_ANALYSIS, AICapability.CODE_GENERATION, AICapability.ADMIN_TOOLS]
};

// Middleware to check permissions before accessing AI endpoints
export async function POST(req: NextRequest) {
  // Get the user's auth status
  const { userId, sessionId } = getAuth(req);

  if (!userId) {
    return NextResponse.json(
      { error: 'Unauthorized' },
      { status: 401 }
    );
  }

  // Get the requested capability from the endpoint path or body
  const body = await req.json();
  const requestedCapability = body.capability || AICapability.BASIC_CHAT;

  try {
    // Check if user has permission for this capability
    const userRole = await getUserRole(userId);
    const authorized = hasPermission(userRole, requestedCapability, roleCapabilities);
  } catch (error) {
    // Handle error (e.g., user not found)
    return NextResponse.json(
      { error: 'User not found' },
      { status: 404 }
    );
  }

  if (!authorized) {
    return NextResponse.json(
      { error: 'Unauthorized' },
      { status: 401 }
    );
  }

  // If authorized, proceed with the request
  // ... (rest of the code) ...
}
```

```
    if (!authorized) {
      return NextResponse.json(
        { error: 'Permission denied for this AI capability' },
        { status: 403 }
      );
    }

    // Log the access attempt for audit purposes
    await logAIAccess({
      userId,
      sessionId,
      capability: requestedCapability,
      timestamp: new Date(),
      success: true
    });

    // Process the actual AI request
    const result = await processAIRequest(body, requestedCapability);

    return NextResponse.json({ result });
  } catch (error) {
    // Log failed attempts
    await logAIAccess({
      userId,
      sessionId,
      capability: requestedCapability,
      timestamp: new Date(),
      success: false,
      error: error.message
    });

    return NextResponse.json(
      { error: 'Error processing request' },
      { status: 500 }
    );
  }
}

// Helper functions
async function getUserRole(userId: string): Promise<string> {
  // Implementation to fetch user role from database
}

async function logAIAccess(accessData: any): Promise<void> {
  // Implementation to log access attempts to secure audit log
}

async function processAIRequest(body: any, capability: AICapability): Promise<any> {
```



```
// Implementation to route to the appropriate AI service  
}
```

When to Prioritize Compliance

- Any use case involving identifiable patient information
- Deployments within healthcare systems or digital therapeutics
- Enterprise clients in pharma, biotech, or insurance
- AI systems used in regulated industries (finance, legal, etc.)
- When processing personally identifiable information (PII)
- Customer-facing applications with data protection requirements

Engineering Tips

- Run threat modeling on your LLM endpoints (e.g., what can be extracted from logs?)
- Use FHIR-compliant output formatting for clinical AI tools
- Create internal documentation for model access, scopes, and behavior
- Implement circuit breakers for unexpected model behavior
- Separate storage of sensitive and non-sensitive data
- Use canary tokens to detect potential data leaks

Recommended Tools

- SpaCy + Regex – fast client-side redaction
- Presidio (Microsoft) – open-source PII anonymization service
- FHIR libraries – for formatting LLM output into structured clinical resources
- LangChain + Azure OpenAI – for HIPAA-compliant inference pipelines
- Nightfall AI – data loss prevention for LLM applications
- NextAuth.js or Clerk – authentication/authorization for AI applications

6. Reusable Engineering Patterns

There are a handful of engineering patterns that show up across nearly every production-grade AI system. This section distills those patterns into copy-pasteable formats and strategic recommendations.

Key Concepts

- **Prompt Templates**
 - Define prompt schemas as reusable functions with variables
 - Version them and test them like code (prompt engineering = config)
 - Use inheritance and composition for prompt families
- **Tool Wrapping**
 - Wrap calculators, search APIs, or other functions into tools
 - Use decorators or interfaces to standardize signature & metadata
 - Implement consistent error handling and retry logic
- **LLM Router Functions**
 - Classify user inputs into types (e.g., question , task , command)
 - Route to correct toolchain or prompt template
 - Implement fallback strategies for edge cases
- **Chain Builders**
 - Create modular blocks (e.g., retrieval → rewriter → summary)
 - Use LangChain Expression Language (LCEL) or direct async composition
 - Enable hot-swapping of components for A/B testing
- **Caching Patterns**
 - Multi-level caching (memory → Redis → persistent)
 - Semantic caching (similar questions get same answers)
 - TTL-based expiration for time-sensitive data

Code Example: Prompt Template System

```
// types.ts
export interface PromptTemplate {
  id: string;
  version: string;
  format(variables: Record<string, any>): string;
}

// prompt-templates.ts
class BasePromptTemplate implements PromptTemplate {
  constructor(
    public id: string,
    public version: string,
```

```

    private template: string,
    private defaultVariables: Record<string, any> = {}
  ) {}

  format(variables: Record<string, any> = {}): string {
    const mergedVars = { ...this.defaultVariables, ...variables };
    let result = this.template;

    // Replace all variables
    for (const [key, value] of Object.entries(mergedVars)) {
      result = result.replace(new RegExp(`{{\\s*${key}\\s*}}`, 'g'), String(value));
    }

    // Check for any unreplaced variables
    const unreplaced = result.match(/{{[^}]+}}/g);
    if (unreplaced) {
      throw new Error(`Missing variables: ${unreplaced.join(', ')} `);
    }

    return result;
  }
}

// Create specific prompt templates
export const qaTemplate = new BasePromptTemplate(
  'qa-general',
  '1.0.0',
  `You are a helpful assistant.

  Question: {{ question }}

  Please provide a clear, accurate, and concise answer.`
);

export const summaryTemplate = new BasePromptTemplate(
  'summary',
  '2.1.0',
  `Summarize the following text in {{ format }} format.
  Length: {{ length }}

  Text to summarize:
  {{ text }}

  Summary:`
);

// Usage
const prompt = qaTemplate.format({
  question: "How do vector databases work?"
});

```

```
});

// Template with default values
export const technicalExplanationTemplate = new BasePromptTemplate(
  'technical-explanation',
  '1.2.0',
  `Explain {{ concept }} in {{ detail_level }} technical detail.
  Include {{ include_code ? 'code examples' : 'no code examples' }}.` ,
  {
    detail_level: 'medium',
    include_code: true
  }
);

// You can override defaults
const prompt2 = technicalExplanationTemplate.format({
  concept: "BGE embeddings",
  detail_level: "high"
});
```

Code Example: Input Routing & Chain Execution

```
import OpenAI from "openai";
import { z } from "zod";

// Schema for classifier output
const ClassifierSchema = z.object({
  type: z.enum(["question", "command", "chat", "feedback", "unknown"]),
  topic: z.string().optional(),
  sentiment: z.enum(["positive", "negative", "neutral"]).optional(),
  confidence: z.number().min(0).max(1).optional()
});

type ClassifierOutput = z.infer<typeof ClassifierSchema>;

// Initialize OpenAI client
const openai = new OpenAI({
  apiKey: process.env.OPENAI_API_KEY,
});

// Input classifier function
async function classifyInput(input: string): Promise<ClassifierOutput> {
  const response = await openai.chat.completions.create({
    model: "gpt-4o",
    messages: [
      {
        role: "system",
```

content: `Classify the user input into one of these types:

- question: User is asking for information
- command: User wants you to perform a specific task
- chat: General conversation
- feedback: User is providing feedback
- unknown: Cannot determine intent

Return a JSON object with the classification.`

```

    },
    { role: "user", content: input }
  ],
  response_format: { type: "json_object" }
});

// Parse and validate the response
const rawOutput = JSON.parse(response.choices[0].message.content);
return ClassifierSchema.parse(rawOutput);
}

// Chain router
async function routeAndProcess(userInput: string) {
  try {
    // Step 1: Classify the input
    const classification = await classifyInput(userInput);

    // Step 2: Route to appropriate chain based on classification
    switch (classification.type) {
      case "question":
        return await handleQuestion(userInput);

      case "command":
        return await handleCommand(userInput);

      case "chat":
        return await handleChat(userInput);

      case "feedback":
        return await handleFeedback(userInput);

      default:
        return await handleFallback(userInput);
    }
  } catch (error) {
    console.error("Error in processing:", error);
    return {
      message: "I encountered an error processing your request. Please try again.",
      error: process.env.NODE_ENV === "development" ? error.message : undefined
    };
  }
}
```

```
}

// Handler implementations
async function handleQuestion(input: string) {
  // Implement RAG pipeline for questions
}

async function handleCommand(input: string) {
  // Implement tool calling or action chain
}

async function handleChat(input: string) {
  // Simple chat completion
}

async function handleFeedback(input: string) {
  // Store feedback and respond appropriately
}

async function handleFallback(input: string) {
  // Generic response for unclear inputs
}

// Usage
const response = await routeAndProcess("Can you help me analyze this dataset?");
```

When to Use These Patterns

- You're scaling to new use cases or customer types
- You need to test multiple prompt strategies side-by-side
- You want separation of concerns in your LLM codebase
- You're building a multi-agent or multi-component system
- You need to maintain and version AI components over time

Engineering Tips

- Treat prompt templates like config files (store in `prompts/` with versions)
- Wrap tools using a common `run(input: string): Promise<string>` signature
- Log the full chain (inputs, outputs, time) for each user action
- Implement robust error handling at each step of your pipeline
- Use validation libraries (Zod, Pydantic) for input/output schemas
- Create a standard input/output format for all your tools and chains

Recommended Tools

- LangChain Expression Language (LCEL) – declarative chains
- TypeScript + tRPC – for safely routing input + mapping tools
- PromptLayer or LangSmith – for prompt versioning & logging
- zod – schema validation of tool output
- Instructor – structured output validation for LLM responses
- Continuations – typed chains for LLM workflows

7. 🧠 Fine-Tuning & Custom Models

When off-the-shelf models don't quite meet your needs, fine-tuning can create tailored solutions. This section covers strategies for dataset creation, model adaptation, and deployment of custom models.

Key Concepts

- **Fine-Tuning Types**
 - Supervised fine-tuning (SFT): train on examples of desired outputs
 - RLHF (Reinforcement Learning from Human Feedback): align with human preferences
 - LoRA (Low-Rank Adaptation): efficient parameter-efficient tuning
 - Instruction tuning: improve response to specific instruction formats
- **Dataset Creation**
 - Clean, diverse, representative examples
 - Balanced classes/formats/domains
 - Clear quality criteria and validation process
 - Synthetic data generation for augmentation
- **Training Infrastructure**
 - Cloud GPUs (AWS, GCP, Azure)
 - Specialized training platforms (Weights & Biases, Hugging Face)
 - Local setup with consumer GPUs for smaller models
 - Distributed training for large models
- **Evaluation Metrics**
 - Task-specific metrics (ROUGE, BLEU, F1, etc.)

- Custom evaluation suites
- Comparative evaluation against baselines
- Human evaluation for subjective quality

Code Example: OpenAI Fine-tuning

```
import openai
import json
import pandas as pd
from sklearn.model_selection import train_test_split

# Set up OpenAI client
client = openai.OpenAI(api_key="your-api-key")

# Load and prepare your dataset
df = pd.read_csv("customer_support_conversations.csv")

# Process conversations into the required format
def format_conversation(row):
    return {
        "messages": [
            {"role": "system", "content": "You are a helpful customer support assistant."},
            {"role": "user", "content": row["customer_query"]},
            {"role": "assistant", "content": row["agent_response"]}
        ]
    }

# Convert dataframe to JSONL format
formatted_data = [format_conversation(row) for _, row in df.iterrows()]

# Split into training and validation sets
train_data, val_data = train_test_split(formatted_data, test_size=0.2)

# Write to JSONL files
with open("train_data.jsonl", "w") as f:
    for item in train_data:
        f.write(json.dumps(item) + "\n")

with open("val_data.jsonl", "w") as f:
    for item in val_data:
        f.write(json.dumps(item) + "\n")

# Create training file
train_file = client.files.create(
    file=open("train_data.jsonl", "rb"),
    purpose="fine-tune"
)
```



```

# Create validation file
val_file = client.files.create(
    file=open("val_data.jsonl", "rb"),
    purpose="fine-tune"
)

# Create fine-tuning job
job = client.fine_tuning.jobs.create(
    model="gpt-3.5-turbo",
    training_file=train_file.id,
    validation_file=val_file.id,
    hyperparameters={
        "n_epochs": 3,
        "batch_size": 8,
        "learning_rate_multiplier": 1.0
    }
)

print(f"Fine-tuning job created: {job.id}")

# Monitor job status
job_status = client.fine_tuning.jobs.retrieve(job.id)
print(f"Status: {job_status.status}")

# List metrics when job is complete
if job_status.status == "succeeded":
    results = client.fine_tuning.jobs.list_metrics(job.id)
    print(results)

```

Code Example: LoRA Fine-tuning with Hugging Face

```

from datasets import load_dataset
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    TrainingArguments,
    Trainer,
    DataCollatorForLanguageModeling
)
from peft import LoraConfig, get_peft_model, prepare_model_for_kbit_training
import torch

# Load base model
model_name = "meta-llama/Llama-3-8b"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(

```

```

    model_name,
    torch_dtype=torch.float16,
    device_map="auto",
    load_in_8bit=True
)

# Prepare model for LoRA fine-tuning
model = prepare_model_for_kbit_training(model)

# Define LoRA configuration
lora_config = LoraConfig(
    r=16,                                # Rank
    lora_alpha=32,                        # Alpha parameter for LoRA scaling
    lora_dropout=0.05,                    # Dropout probability for LoRA layers
    bias="none",                          # Bias type
    task_type="CAUSAL_LM",                # Task type
    target_modules=[                      # Which modules to apply LoRA to
        "q_proj", "k_proj", "v_proj", "o_proj",
        "gate_proj", "up_proj", "down_proj"
    ]
)

# Apply LoRA to model
model = get_peft_model(model, lora_config)

# Load and preprocess dataset
dataset = load_dataset("json", data_files="financial_qa.json")

def format_instruction(example):
    return {
        "text": f"""<s>[INST] You are a financial advisor. Answer the following question

Question: {example['question']}
[/INST]

{example['answer']}</s>"""
    }

# Tokenize dataset
tokenized_dataset = dataset.map(
    lambda examples: tokenizer(
        [format_instruction(ex) for ex in examples],
        truncation=True,
        max_length=512
    ),
    batched=True,
    remove_columns=dataset["train"].column_names
)

```

```
# Set up training arguments
training_args = TrainingArguments(
    output_dir="./financial-advisor-llama",
    per_device_train_batch_size=4,
    gradient_accumulation_steps=4,
    learning_rate=2e-4,
    num_train_epochs=3,
    weight_decay=0.01,
    save_steps=500,
    logging_steps=100,
    fp16=True,
    report_to="wandb"
)

# Create Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_dataset["train"],
    data_collator=DataCollatorForLanguageModeling(tokenizer, mlm=False)
)

# Train model
trainer.train()

# Save model and tokenizer
model.save_pretrained("./financial-advisor-llama-final")
tokenizer.save_pretrained("./financial-advisor-llama-final")
```

When to Use Fine-Tuning

- Domain-specific terminology or knowledge bases
- Consistent style, tone, or formatting requirements
- Custom reasoning patterns or domain-specific instructions
- Performance critical applications where every % improvement matters
- Reducing prompt length by baking instructions into the model
- Building specialized models for vertical industries

Engineering Tips

- Start with prompt engineering and RAG before jumping to fine-tuning
- Use synthetic data generation to expand limited datasets
- Monitor for overfitting, especially with small datasets
- Implement robust evaluation before and after fine-tuning

- Consider model distillation for deployment efficiency
- Test the model's ability to follow instructions outside the training data

Recommended Tools

- OpenAI Fine-tuning API – easy to use, production ready
- Hugging Face PEFT library – parameter-efficient fine-tuning tools
- RLHF Framework – align models with human feedback
- Transformers Trainer – simplified training loop for HF models
- TRL (Transformer Reinforcement Learning) – RLHF and PPO implementation
- Weights & Biases – experiment tracking and model versioning

8. ✨ Prompt Engineering Patterns

Effective prompt design is both art and science. This section covers battle-tested patterns, templates, and techniques for getting the most out of LLMs through prompt engineering.

Key Concepts

- **Core Techniques**
 - Few-shot learning: providing examples in the prompt
 - Chain-of-thought: guiding step-by-step reasoning
 - Role prompting: assigning specific personas to the model
 - Multi-persona approach: creating internal dialogue between experts
 - Format restriction: constraining response format
- **Prompt Components**
 - System message: overall instructions and behavior guidelines
 - Examples: demonstrations of desired output format/style
 - Context: relevant background information
 - Task specification: clear instructions for desired output
 - Format templates: explicit structure for responses
- **Advanced Patterns**
 - Self-critique: asking the model to evaluate its own responses
 - Tree-of-thought: exploring multiple reasoning paths

- Constitutional AI: providing ethical guidelines
- ReAct: reasoning and acting with tool use
- Automatic prompt optimization: using LLMs to improve prompts

Code Example: Advanced Prompting Patterns

```
// Multi-persona expert debate pattern
const expertDebatePrompt = `
You will analyze this problem from the perspective of multiple experts.

PROBLEM: ${problem}

Expert #1: Financial Analyst
Consider market dynamics, financial implications, ROI, and economic factors.

Expert #2: Technical Architect
Consider technical feasibility, scalability, maintenance, and integration challenges.

Expert #3: User Experience Designer
Consider usability, accessibility, user needs, and potential adoption barriers.

Expert #4: Risk Manager
Consider regulatory compliance, security concerns, and potential failure modes.

For each expert:
1. Analyze the problem from their unique perspective
2. Provide key insights (minimum 3)
3. Recommend specific actions

Then, synthesize these perspectives into a balanced final recommendation.
`;

// Chain-of-thought with self-critique pattern
const cotWithCritiquePrompt = `
PROBLEM: ${problem}

Please solve this step by step:

Step 1: Understand the problem and identify key variables
Step 2: Plan your approach to solving the problem
Step 3: Execute the solution, showing all work
Step 4: Verify your answer

After completing these steps, critique your own solution:
- What assumptions did you make?
- What are potential weaknesses in your approach?
- How confident are you in your answer (low/medium/high)?
```

– How could you improve this solution?

`;

// ReAct pattern for tool use

const reactToolUsePrompt = `

You will solve this problem by thinking step by step and using tools when necessary.

PROBLEM: \${problem}

For each step:

1. THINK: Analyze the current state and reason about what to do next
2. ACTION: Select one of these tools if needed:
 - Calculator: For mathematical operations
 - Database: To query for relevant information
 - Search: To find external information
3. OBSERVATION: Review results from tools
4. DECIDE: Determine if more steps are needed or if you can provide the final answer

Begin by understanding the problem and breaking it down.

`;

// Tree of thought reasoning

const treeOfThoughtPrompt = `

PROBLEM: \${problem}

Explore three different approaches to this problem. For each approach:

APPROACH 1:

- Initial idea: [describe approach]
- Reasoning path: [explore this line of thinking]
- Potential outcome: [where this approach leads]
- Confidence: [low/medium/high]

APPROACH 2:

- Initial idea: [describe different approach]
- Reasoning path: [explore this line of thinking]
- Potential outcome: [where this approach leads]
- Confidence: [low/medium/high]

APPROACH 3:

- Initial idea: [describe different approach]
- Reasoning path: [explore this line of thinking]
- Potential outcome: [where this approach leads]
- Confidence: [low/medium/high]

FINAL RECOMMENDATION:

Based on these explorations, which approach is most promising and why?

`;

Structured Format Prompt Examples

```
// JSON output formatting
```

```
const jsonOutputPrompt = `
```

```
Generate a JSON object representing a user profile with the following information:
```

```
User: ${userName}
```

```
Details: ${userDetails}
```

```
The JSON should include:
```

- basic_info (name, age, location)
- preferences (array of interests)
- subscription_status (active/inactive)
- account_metrics (object with login_count and last_active)

```
Response must be valid JSON without explanations or markdown.
```

```
`;
```

```
// Table formatting
```

```
const tableFormatPrompt = `
```

```
Create a comparison table of the top 3 cloud providers with the following structure:
```

```
Subject: ${comparisonTopic}
```

```
Format the response as a markdown table with these columns:
```

- Provider
- Main Strengths
- Limitations
- Pricing Model
- Best Use Cases

```
Keep each cell concise (max 15 words) for readability.
```

```
`;
```

```
// Step-by-step guide
```

```
const procedurePrompt = `
```

```
Create a step-by-step guide for: ${procedure}
```

```
Format requirements:
```

1. Start with a brief overview (2-3 sentences)
2. List all required materials/prerequisites
3. Provide numbered steps (7-10 steps maximum)
4. For each step include:
 - Clear action instruction
 - Why this step matters (1 sentence)
 - Common mistake to avoid (if applicable)
5. End with a troubleshooting section for 2-3 common issues

```
`;
```

```
// Expert recommendation
const expertRecommendationPrompt = `
As an expert in ${domain}, provide your recommendation for ${situation}.

Format your response as follows:
1. Executive Summary (3 sentences maximum)
2. Background Analysis (paragraph)
3. Recommendations (3-5 bullet points)
4. Implementation Considerations (paragraph)
5. Expected Outcomes (paragraph)

Use technical terminology appropriate for professionals in ${domain}.
`;
```

When to Optimize Prompts

- Inconsistent or hallucinated outputs
- Tasks requiring structured, predictable formats
- Complex reasoning problems requiring step-by-step thinking
- Systems with strict accuracy or format requirements
- When balancing multiple aspects of a complex task
- Before resorting to fine-tuning (prompt engineering is faster)

Engineering Tips

- Test prompts systematically with various inputs
- Use prompt templates with clear variable placeholders
- Start simple and add complexity incrementally
- Maintain a prompt library with versioned, tested prompts
- Document the reasoning behind prompt designs
- Use A/B testing to compare prompt effectiveness

Recommended Patterns by Task

- **Classification:** Few-shot learning with diverse examples
- **Creative Writing:** Role-based prompting with style examples
- **Reasoning:** Chain-of-thought or tree-of-thought
- **Decision Making:** Multi-persona debate
- **Tool Use:** ReAct pattern with clear tool definitions
- **Code Generation:** Test-driven development approach

- **Data Analysis:** Step-by-step reasoning with verification

9. UI/UX Design for AI Applications

Crafting interfaces that effectively harness AI capabilities requires specialized patterns. This section focuses on design patterns, component templates, and interaction models for AI-native applications.

Key Concepts

- **Chat Interface Patterns**
 - Progressive disclosure (revealing AI capabilities gradually)
 - Persistent context and history management
 - Clearly signaled AI vs. human content
 - Expectations management during generation/thinking
- **User Input Design**
 - Natural language input with appropriate constraints
 - Hybrid interfaces (combining NL with structured controls)
 - Guided input suggestions and templates
 - Error recovery and reformulation assistance
- **Response Presentation**
 - Streaming tokens for perceived performance
 - Structured vs. conversational output modes
 - Citations and source attribution
 - Confidence indicators for generated content
- **Feedback Mechanisms**
 - Explicit feedback collection (voting, rating)
 - Implicit feedback tracking (time spent, interaction patterns)
 - Contextual feedback requests at appropriate moments
 - Clear paths for error reporting and improvement suggestions

Code Example: AI Chat Interface with Advanced UX Patterns

```

import React, { useState, useRef, useEffect } from 'react';
import { useChat } from 'ai/react';
import { Send, ThumbsUp, ThumbsDown, RefreshCw, Copy } from 'lucide-react';
import Markdown from 'react-markdown';
import { cn } from '@lib/utils';

// Custom hook for tracking typing state
function useTypingIndicator(timeout = 1000) {
  const [isTyping, setIsTyping] = useState(false);
  const timer = useRef<NodeJS.Timeout | null>(null);

  const handleTyping = () => {
    setIsTyping(true);

    if (timer.current) {
      clearTimeout(timer.current);
    }

    timer.current = setTimeout(() => {
      setIsTyping(false);
    }, timeout);
  };

  useEffect(() => {
    return () => {
      if (timer.current) {
        clearTimeout(timer.current);
      }
    };
  }, []);

  return { isTyping, handleTyping };
}

// Suggested prompts component
function SuggestedPrompts({ onSelect }: { onSelect: (prompt: string) => void }) {
  const suggestions = [
    "Explain embedding techniques for RAG systems",
    "How can I implement tool calling with OpenAI?",
    "What's the best way to handle memory in a chatbot?",
    "Generate a prompt template for classification"
  ];

  return (
    <div className="flex flex-wrap gap-2 mb-4">
      {suggestions.map((suggestion, i) => (
        <button
          key={i}
          onClick={() => onSelect(suggestion)}
        />
      ))}
    </div>
  );
}

```

```

        className="px-3 py-1.5 text-sm bg-gray-100 hover:bg-gray-200 rounded-full text-
    >
        {suggestion}
    </button>
  )})
</div>
);
}

// Message component with feedback
function Message({ role, content, id, onFeedback }) {
  const [feedback, setFeedback] = useState<'positive' | 'negative' | null>(null);
  const [copied, setCopied] = useState(false);

  const copyToClipboard = () => {
    navigator.clipboard.writeText(content);
    setCopied(true);
    setTimeout(() => setCopied(false), 2000);
  };

  const handleFeedback = (type: 'positive' | 'negative') => {
    setFeedback(type);
    onFeedback(id, type);
  };

  return (
    <div className={cn(
      "py-4 px-6 rounded-lg mb-4",
      role === 'user' ? "bg-blue-50" : "bg-white border border-gray-200"
    )}>
      <div className="flex justify-between mb-2">
        <span className="text-sm font-medium text-gray-500">
          {role === 'user' ? 'You' : 'AI Assistant'}
        </span>

        {role === 'assistant' && (
          <div className="flex gap-2">
            <button
              onClick={copyToClipboard}
              className="text-gray-400 hover:text-gray-600"
              aria-label="Copy to clipboard"
            >
              <Copy size={16} />
              {copied && <span className="text-xs ml-1">Copied!</span>}
            </button>
          </div>
        )}
      </div>
    </div>
  );
}

```

```

<div className="prose prose-sm max-w-none">
  <Markdown>{content}</Markdown>
</div>

{role === 'assistant' && (
  <div className="flex gap-2 mt-4 justify-end">
    <span className="text-xs text-gray-500 mr-2">Was this response helpful?</span>
    <button
      onClick={() => handleFeedback('positive')}
      className={cn(
        "p-1 rounded-full",
        feedback === 'positive' ? "bg-green-100 text-green-600" : "text-gray-400 hc
      )}
      aria-label="Thumbs up"
    >
      <ThumbsUp size={16} />
    </button>
    <button
      onClick={() => handleFeedback('negative')}
      className={cn(
        "p-1 rounded-full",
        feedback === 'negative' ? "bg-red-100 text-red-600" : "text-gray-400 hover:
      )}
      aria-label="Thumbs down"
    >
      <ThumbsDown size={16} />
    </button>
  </div>
)}
</div>
);
}

// Main chat interface component
export default function AIChatInterface() {
  const { messages, input, handleInputChange, handleSubmit, isLoading, error, reload } =
    api: '/api/chat',
    onResponse: (response) => {
      // Track successful responses for analytics
      if (response.ok) {
        logAnalyticsEvent('chat_response', {
          messageCount: messages.length
        });
      }
    }
  };

  const { isTyping, handleTyping } = useTypingIndicator();
  const messagesEndRef = useRef<HTMLDivElement>(null);

```

```
const [showSuggestions, setShowSuggestions] = useState(true);

// Feedback handling
const handleFeedback = (messageId, feedbackType) => {
  // Send feedback to backend
  fetch('/api/feedback', {
    method: 'POST',
    body: JSON.stringify({
      messageId,
      feedbackType,
      timestamp: new Date().toISOString()
    })
  });

  // Analytics
  logAnalyticsEvent('message_feedback', {
    messageId,
    feedbackType
  });
};

// Example analytics function
const logAnalyticsEvent = (eventName, data) => {
  console.log(`Analytics event: ${eventName}`, data);
  // Implement your analytics logic here
};

// Auto-scroll to bottom on new messages
useEffect(() => {
  messagesEndRef.current?.scrollIntoView({ behavior: 'smooth' });
}, [messages]);

// Hide suggestions after first message
useEffect(() => {
  if (messages.length > 0) {
    setShowSuggestions(false);
  }
}, [messages]);

// Handle form submission
const onSubmit = (e) => {
  e.preventDefault();

  // Track submission for analytics
  logAnalyticsEvent('message_sent', {
    inputLength: input.length
  });

  handleSubmit(e);
};
```

```
};
```

```
// Handle suggested prompt selection
```

```
const handleSuggestionSelect = (suggestion) => {
  handleInputChange({ target: { value: suggestion } });
  setShowSuggestions(false);
```

```
// Track suggestion usage
```

```
logAnalyticsEvent('suggestion_selected', {
  suggestion
});
```

```
};
```

```
return (
```

```
<div className="flex flex-col h-full max-w-3xl mx-auto p-4">
```

```
  { /* Header */ }
```

```
<header className="border-b pb-4 mb-4">
```

```
  <h1 className="text-xl font-semibold">AI Engineering Assistant</h1>
```

```
  <p className="text-sm text-gray-500">
```

```
    Ask about LLMs, RAG, agent design, and deployment best practices
```

```
</p>
```

```
</header>
```

```
{ /* Messages container */ }
```

```
<div className="flex-1 overflow-y-auto mb-4 space-y-4">
```

```
  {messages.length === 0 ? (
```

```
    <div className="text-center py-8">
```

```
      <h2 className="text-lg font-medium mb-2">How can I help you today?</h2>
```

```
      <p className="text-gray-500 mb-6">
```

```
        I can explain concepts, review code, help troubleshoot, or suggest best pra
```

```
</p>
```

```
      {showSuggestions && (
```

```
        <SuggestedPrompts onSelect={handleSuggestionSelect} />
```

```
      )}
```

```
</div>
```

```
) : (
```

```
  messages.map((message) => (
```

```
    <Message
```

```
      key={message.id}
```

```
      role={message.role}
```

```
      content={message.content}
```

```
      id={message.id}
```

```
      onFeedback={handleFeedback}
```

```
    />
```

```
  ))
```

```
)}>
```

```
{ /* Loading indicator */ }
```

```

    {isLoading && (
      <div className="flex items-center text-sm text-gray-500 italic">
        <RefreshCw size={16} className="animate-spin mr-2" />
        AI is typing...
      </div>
    )}

    {/* Error message */}
    {error && (
      <div className="p-4 bg-red-50 border border-red-200 rounded-lg text-red-600 tex
        <p>Error: {error.message}</p>
        <button
          onClick={reload}
          className="mt-2 text-xs bg-red-100 hover:bg-red-200 px-2 py-1 rounded"
        >
          Try again
        </button>
      </div>
    )}

    <div ref={messagesEndRef} />
  </div>

  {/* Input form */}
  <form onSubmit={onSubmit} className="relative">
    <textarea
      value={input}
      onChange={(e) => {
        handleInputChange(e);
        handleTyping();
      }}
      placeholder="Ask a question about AI engineering..."
      className="w-full p-4 pr-12 border border-gray-300 rounded-lg focus:ring-2 focus:ring-blue-500"
      rows={3}
      onKeyDown={(e) => {
        // Submit on Enter (without Shift)
        if (e.key === 'Enter' && !e.shiftKey) {
          e.preventDefault();
          onSubmit(e);
        }
      }}
    />

    <button
      type="submit"
      disabled={isLoading || input.trim() === ''}
      className={cn(
        "absolute right-3 bottom-3 p-2 rounded-full",
        (isLoading || input.trim() === ''

```

```

      ? "bg-gray-100 text-gray-400"
      : "bg-blue-600 text-white hover:bg-blue-700"
    )}
  >
    <Send size={18} />
  </button>

  {isTyping && (
    <p className="absolute -bottom-6 left-0 text-xs text-gray-500">
      Press Enter to send, Shift+Enter for new line
    </p>
  )}
</form>
</div>
);
}

```

Code Example: Structured AI Interface for Document Analysis

```

import React, { useState } from 'react';
import { useDropzone } from 'react-dropzone';
import { UploadCloud, FileText, Settings, ChevronDown, Loader } from 'lucide-react';

type AnalysisOptions = {
  extractEntities: boolean;
  summarize: boolean;
  detectTopic: boolean;
  suggestActions: boolean;
  findSimilarDocs: boolean;
};

export default function DocumentAnalyzer() {
  const [files, setFiles] = useState<File[]>([]);
  const [isAnalyzing, setIsAnalyzing] = useState(false);
  const [results, setResults] = useState<any>(null);
  const [showOptions, setShowOptions] = useState(false);
  const [options, setOptions] = useState<AnalysisOptions>({
    extractEntities: true,
    summarize: true,
    detectTopic: true,
    suggestActions: false,
    findSimilarDocs: false
  });

  // Handle file uploads
  const { getRootProps, getInputProps } = useDropzone({
    accept: {

```



```

    'application/pdf': ['.pdf'],
    'text/plain': ['.txt'],
    'application/vnd.openxmlformats-officedocument.wordprocessingml.document': ['.docx']
  },
  maxFiles: 1,
  onDrop: acceptedFiles => {
    setFiles(acceptedFiles);
    setResults(null);
  }
});

// Start analysis process
const analyzeDocument = async () => {
  if (files.length === 0) return;

  setIsAnalyzing(true);

  try {
    const formData = new FormData();
    formData.append('file', files[0]);
    formData.append('options', JSON.stringify(options));

    const response = await fetch('/api/analyze-document', {
      method: 'POST',
      body: formData
    });

    if (!response.ok) {
      throw new Error('Analysis failed');
    }

    const data = await response.json();
    setResults(data);
  } catch (error) {
    console.error('Error analyzing document:', error);
    // Handle error state
  } finally {
    setIsAnalyzing(false);
  }
};

// Toggle analysis options
const handleOptionChange = (optionName: keyof AnalysisOptions) => {
  setOptions(prev => ({
    ...prev,
    [optionName]: !prev[optionName]
  }));
};

```

```

return (
  <div className="max-w-4xl mx-auto p-6">
    <h1 className="text-2xl font-bold mb-6">AI Document Analyzer</h1>

    {/* File upload area */}
    <div
      {...getRootProps()}
      className={`border-2 border-dashed rounded-lg p-8 text-center cursor-pointer transition
        ${files.length > 0 ? 'border-green-300 bg-green-50' : 'border-gray-300 hover:bc
      >
      <input {...getInputProps()} />

      {files.length > 0 ? (
        <div className="flex flex-col items-center">
          <FileText size={40} className="text-green-500 mb-3" />
          <p className="font-medium">{files[0].name}</p>
          <p className="text-sm text-gray-500 mt-1">
            {(files[0].size / 1024 / 1024).toFixed(2)} MB
          </p>
          <button
            onClick={(e) => {
              e.stopPropagation();
              setFiles([]);
            }}
            className="mt-3 text-sm text-red-500 hover:text-red-700"
          >
            Remove
          </button>
        </div>
      ) : (
        <div className="flex flex-col items-center">
          <UploadCloud size={40} className="text-gray-400 mb-3" />
          <p className="font-medium">Drag and drop a document, or click to select</p>
          <p className="text-sm text-gray-500 mt-1">
            Supports PDF, TXT, and DOCX (Max 10MB)
          </p>
        </div>
      )}
    </div>

    {/* Analysis options */}
    <div className="mt-6">
      <button
        onClick={() => setShowOptions(!showOptions)}
        className="flex items-center text-sm font-medium text-gray-600 hover:text-gray-
      >
      <Settings size={16} className="mr-2" />
      Analysis Options
      <ChevronDown size={16} className={`ml-1 transition-transform ${showOptions ? 't

```

```
</button>
```

```
{showOptions && (  
  <div className="mt-3 p-4 bg-gray-50 rounded-lg grid grid-cols-2 gap-3">  
    <label className="flex items-center space-x-2">  
      <input  
        type="checkbox"  
        checked={options.summarize}  
        onChange={() => handleOptionChange('summarize')}  
        className="rounded"  
      />  
      <span>Generate Summary</span>  
    </label>  
  
    <label className="flex items-center space-x-2">  
      <input  
        type="checkbox"  
        checked={options.extractEntities}  
        onChange={() => handleOptionChange('extractEntities')}  
        className="rounded"  
      />  
      <span>Extract Key Entities</span>  
    </label>  
  
    <label className="flex items-center space-x-2">  
      <input  
        type="checkbox"  
        checked={options.detectTopic}  
        onChange={() => handleOptionChange('detectTopic')}  
        className="rounded"  
      />  
      <span>Detect Topics</span>  
    </label>  
  
    <label className="flex items-center space-x-2">  
      <input  
        type="checkbox"  
        checked={options.suggestActions}  
        onChange={() => handleOptionChange('suggestActions')}  
        className="rounded"  
      />  
      <span>Suggest Actions</span>  
    </label>  
  
    <label className="flex items-center space-x-2">  
      <input  
        type="checkbox"  
        checked={options.findSimilarDocs}  
        onChange={() => handleOptionChange('findSimilarDocs')}>
```

```

        className="rounded"
      />
      <span>Find Similar Documents</span>
    </label>
  </div>
})
</div>

{/* Analysis button */}
<div className="mt-6">
  <button
    onClick={analyzeDocument}
    disabled={files.length === 0 || isAnalyzing}
    className={`w-full py-3 px-4 rounded-lg font-medium text-white
      ${files.length === 0 ? 'bg-gray-300 cursor-not-allowed' : 'bg-blue-600 hover:
    >
    {isAnalyzing ? (
      <span className="flex items-center justify-center">
        <Loader size={20} className="animate-spin mr-2" />
        Analyzing Document...
      </span>
    ) : (
      'Analyze Document'
    )}
  </button>

  {files.length > 0 && (
    <p className="text-xs text-center mt-2 text-gray-500">
      Analysis typically takes 15-30 seconds depending on document length
    </p>
  )}
</div>

{/* Results display */}
{results && (
  <div className="mt-8 border rounded-lg overflow-hidden">
    <div className="bg-gray-50 p-4 border-b">
      <h2 className="font-semibold">Analysis Results</h2>
    </div>

    <div className="p-6 space-y-6">
      {options.summarize && results.summary && (
        <div>
          <h3 className="text-lg font-medium mb-2">Document Summary</h3>
          <div className="p-4 bg-white border rounded-md">
            <p>{results.summary}</p>
          </div>
        </div>
      )}
    </div>
  )}

```

```

{options.extractEntities && results.entities && (
  <div>
    <h3 className="text-lg font-medium mb-2">Key Entities</h3>
    <div className="grid grid-cols-2 gap-4">
      {Object.entries(results.entities).map(([category, items]) => (
        <div key={category} className="p-4 bg-white border rounded-md">
          <h4 className="font-medium mb-2 capitalize">{category}</h4>
          <ul className="space-y-1">
            {Array.isArray(items) && items.map((item, i) => (
              <li key={i} className="text-sm">{item}</li>
            ))}
          </ul>
        </div>
      ))}
    </div>
  </div>
)}

{options.detectTopic && results.topics && (
  <div>
    <h3 className="text-lg font-medium mb-2">Topics</h3>
    <div className="flex flex-wrap gap-2">
      {results.topics.map((topic, i) => (
        <span key={i} className="px-3 py-1 bg-blue-100 text-blue-800 rounded-
          {topic}
        </span>
      ))}
    </div>
  </div>
)}

{options.suggestActions && results.actions && (
  <div>
    <h3 className="text-lg font-medium mb-2">Suggested Actions</h3>
    <div className="p-4 bg-white border rounded-md">
      <ul className="space-y-2">
        {results.actions.map((action, i) => (
          <li key={i} className="flex items-start">
            <span className="inline-flex items-center justify-center w-6 h-6
              {i + 1}
            </span>
            <span>{action}</span>
          </li>
        ))}
      </ul>
    </div>
  </div>
)}

```

```

        </div>
    </div>
    )}
</div>
);
}

```

When to Invest in Specialized AI UX

- Complex AI capabilities requiring clear user guidance
- Systems with multiple AI capabilities or tools
- Conversational interfaces with multi-turn interactions
- When accuracy perception matters as much as actual accuracy
- Applications targeting non-technical users
- When user feedback is critical to system improvement

Engineering Tips

- Design for expectation management (loading states, confidence indicators)
- Use progressive disclosure to introduce complex AI capabilities
- Implement intelligent defaults while providing configuration options
- Provide clear feedback paths for incorrect or unhelpful responses
- Design for "conversation repair" when AI fails to understand
- Test with real users early and often to uncover mental model mismatches

Recommended UI Patterns

- **For Chat Interfaces:** Stream tokens, add thinking indicators, show reference sources
- **For Document Analysis:** Step-by-step progress indicators, confidence scores, highlighted extractions
- **For Coding Assistants:** Code diff visualization, explanation sidebars, test failure highlighting
- **For Content Generation:** Templates, structured inputs, version comparison
- **For Search Interfaces:** Hybrid keyword/semantic controls, context highlighting, source attribution

10. Ultimate Tech Stack for 2025

This stack reflects battle-tested libraries, platforms, and services used by top AI engineers shipping real-world products in 2025.

Language & Frameworks

- **TypeScript** – preferred for modern LLM apps with strong typing and DX
- **Next.js (App Router)** – SSR, Edge Functions, and stream-ready frontend
- **Python (FastAPI / Pydantic)** – fast backends and structured tool output
- **LangChain & LangGraph** – orchestration, memory, and agent flow
- **Vercel AI SDK** – optimized React hooks for AI UIs

Model Providers

- **OpenAI GPT-4o** – balanced performance, vision capabilities, and JSON/function calling
- **Anthropic Claude 3.5 Sonnet** – strong context management + guardrails
- **Google Gemini 1.5 Pro** – 1M token context + doc reasoning
- **Mistral Large 2** – open model with competitive performance and JSON capabilities
- **FireworksAI** – OpenAI-compatible hosted inference, great latency
- **Groq API** – ultra-fast inference for open-weight models (LLaMA 3, Mixtral)

Embedding & Vector DB

- **text-embedding-3-small** (OpenAI) – fast, high-quality general-purpose
- **pgvector + Supabase** – SQL-native, simple, cheap, scalable
- **Weaviate** – hybrid search, schema-first
- **Qdrant** – high-performance, cloud or self-hosted
- **LlamaIndex** – advanced routing + multi-index management

Retrieval & RAG Stack

- **LangChain Retriever / LCEL** – plug-and-play RAG pipelines
- **Semantic chunkers** (recursive, markdown-aware)
- **Ragas** – ground-truth-based eval of retrieval output
- **Multi-query retrieval** – for improved recall
- **Hybrid search** – combining vector + keyword search

Observability & Evaluation

- **LangSmith** – tracing + live feedback + dataset comparison
- **Promptfoo** – CLI + dashboards for model eval
- **DeepEval** – regression testing for production deployments
- **TruLens** – enterprise observability and scoring for LLM apps

- **Weights & Biases** – experiment tracking for fine-tuning

UI & Streaming

- **Vercel AI SDK** – stream-ready, token-by-token updates
- **ShadCN/UI + Tailwind** – clean components with rapid dev cycle
- **Reflex** – Python-first reactive UI framework
- **Resend** – email capture and user activation flows

Hosting & Deployment

- **Vercel** – frontend, edge inferencing, streaming-ready
- **Modal** – GPU-based async functions for tool+model integration
- **Render / Railway** – managed infra for non-serverless use cases
- **AWS Lambda + ECS** – enterprise deployments with autoscaling
- **Hugging Face Spaces** – quick demos and prototypes

Security & Compliance

- **Panther** – AI-specialized security monitoring
- **Presidio** – PII detection and anonymization
- **Langfuse** – LLM observability with compliance features
- **OrbStack** – secure local development environment

Development Tools

- **Cursor** – AI-enabled code editor optimized for LLM development
- **LM Studio** – local model testing and prompt development
- **GitHub Copilot** – AI pair programming assistant
- **VSCode AI Extension** – integrated AI development environment

Successful AI systems leverage this comprehensive stack to build robust, responsive applications with integrated security, evaluation, and human-centered designs.

Additional Resources

Learning Resources

- [LangChain Cookbook](#) - Practical code examples
- [LlamaIndex Documentation](#) - Advanced retrieval patterns
- [Prompt Engineering Guide](#) - Templates and techniques
- [Full Stack LLM Bootcamp](#) - Comprehensive video course

Community & Discussion

- [HuggingFace Forums](#) - Model discussions and help
- [LangChain Discord](#) - Community support for LangChain
- [AI Engineers Discord](#) - Practical engineering discussions

Papers & Publications

- [ReAct: Synergizing Reasoning and Acting in LLMs](#)
- [Retrieval-Augmented Generation: A Survey](#)
- [LoRA: Low-Rank Adaptation of Large Language Models](#)
- [LLM Powered Autonomous Agents](#)

Blogs & Newsletters

- [Sebastian Raschka's Blog](#) - Deep dives on LLM techniques
- [The Batch](#) - Andrew Ng's AI newsletter
- [Latent Space](#) - AI engineering insights

About This Cheatsheet

This AI Engineering Cheatsheet was created to provide practical guidance for building production-grade AI systems using Large Language Models. It focuses on real-world engineering patterns rather than theoretical machine learning concepts.

If you found this valuable, please consider:

- Sharing with your network on LinkedIn or Twitter (tag me @Matt Vegas)
- Following for more AI engineering resources, follow at www.inference-stack.com
- Providing feedback on what topics you'd like to see expanded, send to matt.vegas@inference-stack.com

Version: 2.0 (April 2025) Copyright: 2025, InferenceStack License: MIT