LLM Application Architecture

---

1. Introduction

LLM Application Architecture defines **how Large Language Models are integrated into real software systems**. Unlike simple chatbots, production-grade LLM applications require:

- Clear separation of concerns
- Scalable design patterns
- Controlled data access
- Reliable conversation flows
- Business-aligned goals

From a software engineering perspective, LLMs are **stateless reasoning engines** that must be wrapped in architecture patterns to become **useful, safe, and scalable applications**.

---

2. Core Components of an LLM Application

2.1 High-Level Architecture Components

Component	Responsibility
Client (UI / API)	User interaction
Orchestrator	Controls flow and logic
Prompt Layer	Templates and constraints
Context Manager	Maintains conversation state

Component	Responsibility
LLM API	Language model inference
Knowledge Source	Documents, databases
Tooling / Actions	APIs, functions
Observability	Logging, metrics

2.2 Logical Flow

User → UI → Orchestrator → Context Manager → Prompt Builder → LLM

→ Tools / Retrieval (optional) → Response → User

---

3. Key LLM Design Patterns

---

4. Pattern 1: Retrieval-Augmented Generation (RAG)

4.1 What Is RAG?

Retrieval-Augmented Generation (RAG) combines:

- LLM reasoning capabilities
- External knowledge retrieval

Instead of relying solely on model training data, RAG injects **relevant documents at runtime**.

4.2 RAG Architecture

User Query
↓
Query Embedding
↓
Vector Search (Knowledge Base)
↓
Relevant Documents
↓
Prompt Injection
↓
LLM Response

4.3 When to Use RAG

Scenario	**Why RAG**
Internal documentation	Private data
Regulatory content	Accuracy
Frequently changing data	Freshness
Large knowledge bases	Scalability

4.4 RAG Example Prompt

System:

You are an internal documentation assistant.

Context:

Use only the provided documents.

Documents:

[Extracted policy text]

Task:

Answer the user's question.

4.5 RAG Benefits and Limitations

Benefits	Limitations
Reduces hallucinations	Retrieval quality matters
Uses private data	Token overhead
No retraining required	Latency increase

---

5. Pattern 2: Agent-Based Systems

5.1 What Is an Agent?

An **LLM agent** is an autonomous component that:

- Has a goal
- Can plan steps
- Uses tools
- Evaluates results

5.2 Agent Architecture

Goal
↓
Planner
↓
LLM Reasoning
↓
Tool Invocation
↓
Observation
↓
Iteration

5.3 Types of Agents

Agent Type	Purpose
Single-agent	Simple workflows
Multi-agent	Complex coordination
Tool-using agent	API interaction
Validator agent	Output checking

5.4 Example: DevOps Agent

Goal: Investigate deployment failure

Steps:

1. Analyze logs
2. Identify error
3. Suggest fix
4. Validate solution

5.5 Pros and Cons of Agents

Pros	Cons
Autonomous reasoning	Harder to debug
Flexible workflows	Higher cost
Multi-step tasks	Risk of loops

---

6. Pattern 3: Prompt-Chain / Workflow Pattern

6.1 Description

Break a complex task into **deterministic steps**.

6.2 Example Chain

Step 1: Extract entities

Step 2: Validate data

Step 3: Generate response

6.3 Use Case

- Data processing pipelines
- Document analysis
- Compliance checks

---

7. Pattern 4: Tool-Calling Architecture

7.1 Description

LLM acts as a **controller**, invoking backend services.

7.2 Example

LLM → getCustomerData()

LLM → calculateRisk()

LLM → generateSummary()

---

8. Common Architectural Mistakes

- Treating LLM as a database
- No retrieval grounding
- Overusing agents
- No conversation goals
- Ignoring cost and latency

---

9. Best Practices Summary

- Choose patterns based on business goals
- Keep LLM stateless
- Externalize memory and knowledge
- Use RAG for factual accuracy
- Add agents only when needed

---

10. Conclusion

LLM Application Architecture is the foundation for building **scalable, safe, and business-aligned AI systems**. By applying proven design patterns such as **Retrieval-Augmented Generation and Agent-based systems**, and by defining **clear goals and conversation flows**, organizations can move from experimental chatbots to **production-ready AI applications**.

Well-designed architectures ensure:

- Consistency
- Maintainability
- Accuracy