Important LLM-Related Concepts

Prompt Engineering, RAG, Fine-Tuning, AI Agents, MCP, and more

Overview

We will cover the key concepts that surround modern LLM usage:

1. **Prompt Engineering** — How to talk to LLMs effectively
2. **RAG** (Retrieval-Augmented Generation) — Giving LLMs external knowledge
3. **Fine-Tuning** — Customizing an LLM for a specific task

4. **AI Agents** — LLMs that can take actions
5. **MCP** (Model Context Protocol) — A standard for connecting LLMs to tools
6. **Function Calling / Tool Use** — How LLMs invoke external functions
7. **Embeddings & Vector Databases** — The backbone of semantic search

Why Do We Need These Concepts?

LLMs are powerful, but they have **fundamental limitations**:

Limitation	Solution
Knowledge cutoff (training data is old)	**RAG**
Generic responses (not domain-specific)	**Fine-Tuning**
Can only generate text (no actions)	**AI Agents / Tool Use**
No access to your private data	**RAG / MCP**
Inconsistent output quality	**Prompt Engineering**

Part 1: Prompt Engineering

What is Prompt Engineering?

Prompt engineering = the art and science of crafting inputs to get the best outputs from an LLM.

Think of it like asking the right question to the right person in the right way.

Bad prompt: "Write code"
Good prompt: "Write a Python function that takes a list of integers and returns the top 3 largest values, sorted in descending order. Include docstrings and type hints."

Key Prompt Engineering Techniques

1. Zero-Shot Prompting

Just ask directly — no examples provided.

> "Classify the sentiment of this review: 'The battery life is amazing but the screen is too dim.'"

2. Few-Shot Prompting

Provide examples so the LLM learns the pattern.

> "Classify sentiment:
>
> 'Great product!' → Positive
>
> 'Terrible quality.' → Negative
>
> 'The battery is amazing but the screen is dim.' → ???"

3. Chain-of-Thought (CoT) Prompting

Ask the LLM to **think step by step**.

> "A store has 15 apples. 3 customers each buy 2 apples, and then a delivery of 10 apples arrives. How many apples are in the store? **Think step by step.**"

This significantly improves reasoning accuracy.

4. System Prompts

Set the LLM's role and behavior upfront.

> "You are an experienced Python tutor. Explain concepts simply. Always provide runnable code examples."

Prompt Engineering — Real-World Example

```
System: You are a senior code reviewer. Be concise and specific.

User: Review this code:

def calc(x):
    if x > 0:
        return x * 2
    else:
        return x * -1

Focus on: naming, edge cases, and documentation.                        .
```

Key takeaway: The more **context** and **constraints** you give, the better the response.

Part 2: RAG (Retrieval-Augmented Generation)

What is RAG?

RAG = Retrieval + Generation

Instead of relying solely on what the LLM learned during training, we **retrieve relevant documents first**, then let the LLM generate an answer based on them.

Without RAG:

User → LLM → Answer

(uses only training data)

With RAG:

User → **Search** → Retrieved Docs → LLM → Answer

(uses your actual data)

Why RAG?

Problem	How RAG Solves It
LLM doesn't know your company's data	Retrieve from your document store
LLM's knowledge is outdated	Retrieve latest documents
LLM "hallucinates" facts	Ground answers in real sources
You can't retrain the LLM (too expensive)	Just update the document store

RAG is the most popular way to add private/custom knowledge to an LLM.

How RAG Works — Step by Step

Indexing Phase (done once, offline)

1. **Collect documents** (PDFs, web pages, code, databases)
2. **Split** them into small chunks (e.g., 500 tokens each)
3. **Embed** each chunk → convert text to a vector (list of numbers)
4. **Store** vectors in a **vector database** (e.g., Pinecone, ChromaDB, pgvector)

Query Phase (done per question)

1. **Embed** the user's question → convert to a vector
2. **Search** the vector database for similar chunks
3. **Combine** the retrieved chunks with the question as context
4. **Send** to the LLM → generate the final answer

RAG — Simplified Code Example

```
# 1. Embed and store documents (indexing)
from sentence_transformers import SentenceTransformer
import chromadb

model = SentenceTransformer('all-MiniLM-L6-v2')
db = chromadb.Client()
collection = db.create_collection("my_docs")

docs = ["Flutter uses Dart language.", "MVVM separates UI from logic."]
embeddings = model.encode(docs)
collection.add(documents=docs, embeddings=embeddings, ids=["d1", "d2"])

# 2. Query (retrieval + generation)
query = "What language does Flutter use?"
query_embedding = model.encode([query])
results = collection.query(query_embeddings=query_embedding, n_results=1)
# results → "Flutter uses Dart language."
# Send this context + query to LLM for final answer
```

Part 3: Fine-Tuning

What is Fine-Tuning?

Fine-tuning = taking a pre-trained LLM and training it further on **your specific data** so it becomes specialized.

Pre-trained LLM

- General knowledge
- Generic style
- Broad capabilities

Fine-tuned LLM

- Domain-specific knowledge
- Your preferred style/format
- Specialized behavior

Analogy: A medical school graduate (pre-trained) doing a residency in cardiology (fine-tuning).

Fine-Tuning vs RAG — When to Use Which?

Criteria	RAG	Fine-Tuning
You need up-to-date info	✅ Best choice	❌ Needs retraining
You need a specific style/format	❌ Limited	✅ Best choice
Cost	Low (just a vector DB)	High (GPU training)
Setup complexity	Moderate	High
Data changes frequently	✅ Easy to update	❌ Must retrain
You need domain expertise	Good (with good docs)	Best

Part 4: AI Agents

What is an AI Agent?

An **AI Agent** = an LLM that can **plan, decide, and take actions** — not just generate text.

Regular LLM: "Here's how to check the weather in Python..."
AI Agent: *Actually checks the weather API and tells you the result.*

The Agent Loop

```
User Request → LLM thinks → Decides action → Executes tool
    ↑                                              ↓
    └──────── Observes result ← Gets result ←──────┘
```

The LLM keeps looping until the task is complete.

AI Agent — Example Flow

User: "What's the weather in Cincinnati and should I bring an umbrella?"

```
Step 1: LLM decides → call weather_api("Cincinnati")
Step 2: Tool returns → { "temp": 45, "condition": "rain", "chance": 80% }
Step 3: LLM decides → no more tools needed, generate response
Step 4: LLM responds → "It's 45°F with 80% chance of rain.
                         Definitely bring an umbrella!"
```

Key insight: The LLM is the "brain" that decides **which tools to call** and **when to stop**.

Agentic Frameworks

Several frameworks exist to build AI agents:

Framework	Description
LangChain	Popular Python framework for LLM apps and agents
LangGraph	Graph-based agent workflows (by LangChain team)
CrewAI	Multi-agent collaboration framework
AutoGen (Microsoft)	Multi-agent conversation framework
Claude Code (Anthropic)	CLI agent for coding tasks

These frameworks provide the **orchestration layer** — the loop that lets the LLM call tools, observe results, and decide next steps.

Part 5: Function Calling / Tool Use

What is Function Calling?

Function calling (or **Tool Use**) = the LLM's ability to output structured requests to call external functions.

The LLM **does not execute code** — it produces a JSON object describing what function to call and with what arguments. Your application then executes it.

```
User: "What's 15% tip on a $84.50 bill?"

LLM output (not text, but structured):
{
  "function": "calculate_tip",
  "arguments": { "bill_amount": 84.50, "tip_percent": 15 }
}

Your app: executes calculate_tip(84.50, 15) → returns $12.68

LLM final response: "A 15% tip on $84.50 would be $12.68."
```

Function Calling — How It Works

Step 1: Define available tools

```
{
  "tools": [
    {
      "name": "get_stock_price",
      "description": "Get current stock price by ticker symbol",
      "parameters": {
        "type": "object",
        "properties": {
          "ticker": { "type": "string", "description": "e.g., AAPL" }
        }
      }
    }
  ]
}
```

Step 2: LLM decides to use a tool (or not)

Step 3: Your code executes the function

Step 4: Send result back to LLM for final response

Part 6: MCP (Model Context Protocol)

What is MCP?

MCP = Model Context Protocol

An **open standard** (by Anthropic) that provides a **universal way** for LLMs to connect to external data sources and tools.

Analogy: MCP is like **USB for AI** — a standard plug that lets any LLM connect to any tool.

Before MCP:
Each AI app builds custom integrations for every tool (N × M problem)

With MCP:
Tools implement MCP once, all AI apps can use them (N + M problem)

MCP Architecture

```
┌──────────────────┐     ┌──────────────────┐     ┌──────────────────┐
│    MCP Host      │     │   MCP Client     │     │   MCP Server     │
│ (Claude Desktop, │────>│ (protocol        │────>│ (provides        │
│  IDE, etc.)      │     │  handler)        │     │  tools/data)     │
└──────────────────┘     └──────────────────┘     └──────────────────┘
                                                           │
                                                           ▼
                                                   ┌───────────────┐
                                                   │   External    │
                                                   │   Resources   │
                                                   │   (DB, API,   │
                                                   │    Files...)  │
                                                   └───────────────┘
```

- **Host:** The AI application (e.g., Claude Desktop, VS Code)
- **Client:** Manages the connection (1:1 with a server)
- **Server:** Exposes tools, resources, and prompts via MCP

MCP — What Can a Server Provide?

An MCP server can expose three types of capabilities:

1. Tools

Functions the LLM can call (e.g., `search_database`, `send_email`)

2. Resources

Data the LLM can read (e.g., file contents, database records)

3. Prompts

Pre-defined prompt templates (e.g., "summarize this code")

Example MCP servers: filesystem access, GitHub, Slack, databases, web search, etc.

MCP Server Example (Python)

```python
# A simple MCP server that provides a "greet" tool
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("GreetingServer")

@mcp.tool()
def greet(name: str) -> str:
    """Greet a user by name."""
    return f"Hello, {name}! Welcome to the MCP world."

@mcp.tool()
def add(a: int, b: int) -> int:
    """Add two numbers together."""
    return a + b

if __name__ == "__main__":
    mcp.run()
```

Once this server runs, any MCP-compatible host (like Claude Desktop) can discover and use `greet` and `add` as tools.

MCP vs Function Calling — What's the Difference?

Aspect	Function Calling	MCP
Scope	Built into one LLM provider's API	Universal open standard
Who defines tools?	The app developer	The MCP server
Discovery	App tells LLM what tools exist	LLM discovers tools from servers
Reusability	Custom per app	Write once, use everywhere
Transport	API request/response	stdio or HTTP (SSE)

MCP builds on top of function calling — it standardizes how tools are discovered, described, and invoked across different AI applications.

Part 7: Embeddings & Vector Databases

What are Embeddings?

Embedding = converting text (or images, audio) into a **list of numbers** (a vector) that captures its **meaning**.

```
"king"   → [0.21, 0.83, -0.45, 0.12, ...]   (768 numbers)
"queen"  → [0.19, 0.81, -0.43, 0.15, ...]   (similar!)
"apple"  → [0.72, -0.31, 0.55, -0.08, ...]  (very different)
```

Key property: Similar meanings → similar vectors → close in vector space.

This enables **semantic search** — finding documents by meaning, not just keyword matching.

Vector Database

A **vector database** stores embeddings and enables fast similarity search.

```
Traditional DB:  SELECT * FROM docs WHERE text LIKE '%Flutter%'
                 → keyword match only ❌

Vector DB:       Find docs closest to embed("mobile app framework")
                 → returns Flutter docs, React Native docs, etc. ✅
```

Popular vector databases:

Database	Type	Notes
ChromaDB	Open source, embedded	Great for learning and prototyping
Pinecone	Cloud-hosted	Managed service, easy to scale
pgvector	PostgreSQL extension	Add vectors to your existing DB
FAISS	Library (Meta)	Fast, in-memory, for large-scale search

Part 8: Putting It All Together

How These Concepts Connect

```
User Question
     │
     ▼
┌──────────────────────────────┐
│  Prompt Engineering          │  ← How the question is framed
│  (system prompt, few-shot)   │
└──────────────────────────────┘
           │
           ▼
┌──────────────────────────────┐
│  RAG (optional)              │  ← Retrieve relevant documents
│  Embeddings + Vector DB      │
└──────────────────────────────┘
           │
           ▼
┌──────────────────────────────┐
│  LLM (possibly fine-tuned)   │  ← Generate response                          .
└──────────────────────────────┘
           │
           ▼
┌──────────────────────────────┐
│  Agent / Tool Use / MCP      │  ← Take actions if needed
└──────────────────────────────┘
           │
           ▼
     Final Answer
```

A Real-World Example: AI-Powered Customer Support

Scenario: A company builds an AI chatbot for customer support.

1. **Prompt Engineering:** System prompt defines tone, policies, escalation rules
2. **RAG:** Retrieves relevant support articles and past tickets from a vector DB

3. **Fine-Tuning:** Model trained on 10,000 past support conversations for company-specific style
4. **Tool Use / MCP:** Can look up order status, process refunds, create tickets via MCP servers
5. **Agent Loop:** Chatbot autonomously handles multi-step requests (check order → find issue → process refund → confirm)

All of these concepts work **together** in modern AI systems.

Summary

Concept	One-Liner
Prompt Engineering	Craft better inputs → get better outputs
RAG	Give the LLM your own data at query time
Fine-Tuning	Train the LLM further on your specific data
AI Agents	LLMs that can plan and take actions
Function Calling	LLM outputs structured tool requests
MCP	Universal standard to connect LLMs to tools
Embeddings	Convert text to numbers that capture meaning
Vector DB	Store and search embeddings by similarity

What to Explore Next

- **Hands-on:** Build a simple RAG system with ChromaDB + OpenAI/Claude API
- **Hands-on:** Create an MCP server and connect it to Claude Desktop
- **Hands-on:** Experiment with prompt engineering techniques on real tasks

- **Read:** Anthropic MCP Documentation
- **Read:** OpenAI Function Calling Guide
- **Read:** LangChain Documentation

The best way to understand these concepts is to build with them!