From AI to Generative AI

Conceptual Foundations and Modern Systems

Giorgos Filandrianos

July 13, 2026

From AI to Generative AI

Core distinction

AI, Machine Learning, Deep Learning, and Generative AI are not competing terms. They describe different levels in the evolution of intelligent systems.

Concept	Main idea	Typical role
AI	Intelligent behaviour	Broad research field
ML	Learning from data	Prediction and pattern discovery
DL	Neural representations	Complex perception and language tasks
GenAI	Content generation	Text, code, image, audio, and multimodal output

The Conceptual Shift

Earlier AI

- Symbolic reasoning
- Hand-crafted rules
- Expert systems
- Explicit knowledge representation

Modern AI

- Data-driven learning
- Distributed representations
- Foundation models
- Adaptation through prompting, tuning, and retrieval

Main point

The field moved from programming intelligence explicitly to learning reusable representations from large-scale data.

Why Foundation Models Matter

- Foundation models are trained on broad, heterogeneous data and can support many downstream tasks.
- Large Language Models made AI interaction natural by using language as the interface.
- The output is no longer only a class label or a score; it can be an explanation, summary, plan, dialogue, or executable code.
- This changed AI adoption: models became directly usable by non-specialists.

Key takeaway

Generative AI transformed AI from a mostly backend predictive technology into an interactive layer for knowledge work.

Capabilities and Open Problems

Capabilities

- Text and code generation
- Summarization and rewriting
- Translation and dialogue
- Multimodal generation
- Synthetic media

Open problems

- Hallucination
- Lack of grounding
- Bias and misuse
- Limited interpretability
- Weak robustness under distribution shift

Interpretation

The same generative ability that makes these models useful also creates reliability, safety, and evaluation challenges.

From Models to AI Systems

Modern perspective

The current frontier is not only about larger models, but about systems that combine models with external knowledge, tools, memory, and workflows.

- How can model outputs be grounded in reliable sources?
- How can models retrieve, use, and cite external knowledge?
- How can they reason over long documents, graphs, databases, and APIs?
- How can they act as components inside larger software systems?

Next topics: RAG, GraphRAG, long-context models, agents, tool use, and MCP.

Takeaway

Generative AI is best understood not as a separate technology, but as the latest stage in a broader transition:

from rules $\rightarrow$ to learned representations $\rightarrow$ to interactive AI systems

Prompting as Model Steering

Core idea

Prompting is the process of guiding a language model's behaviour at inference time, without changing its parameters.

- The prompt defines the task, constraints, expected format, and available information.
- It can strongly affect accuracy, reasoning quality, style, and robustness.
- Prompting is not only about asking a question; it is about designing the model's operating context.

Key point: prompting is the first layer of control before retrieval, tools, memory, or agents are introduced.

Common Prompting Strategies

Method	Idea	Use case
Zero-shot	No examples	General tasks
Few-shot	A few demonstrations	Format and pattern control
Chain-of-thought	Step-by-step reasoning	Complex reasoning tasks
Self-consistency	Multiple reasoning paths	More robust answers
Role prompting	Assigning a perspective	Style and expertise control

Interpretation

Prompting methods improve how the model uses its internal knowledge, but they do not solve missing, outdated, or inaccessible information.

From Prompting to Context Engineering

Prompt engineering

- Instructions
- Examples
- Output format
- Reasoning style
- Task decomposition

Beyond prompting

- External documents
- User-specific data
- APIs and tools
- Permissions
- Runtime retrieval

Main transition

Prompting controls how the model responds; context engineering controls what the model has access to when it responds.

Why Context Matters

Core idea

In many modern AI systems, the main bottleneck is not the model's reasoning ability, but whether the model receives the right context at the right time.

- Without context, an LLM may produce fluent but generic answers.
- With relevant context, the same model can generate answers grounded in specific documents, users, tasks, and constraints.
- Better context does not mean more context; it means more precise, relevant, and authorized context.

Key shift: from asking *"Can the model answer?"* to asking *"Does the model have the right information to answer?"*

Context Engineering

Definition

Context engineering is the design of systems that deliver the right information, from the right sources, with the right permissions, into the model context at runtime.

What it involves

- Connected access to data sources
- Knowledge layers and entity linking
- Precision retrieval
- Runtime governance

Why it matters

- Reduces irrelevant context
- Improves grounding
- Preserves permissions
- Supports domain-specific reasoning

Main point: context is not just prompt text; it is infrastructure.

From Search to Retrieval-Augmented AI

Core idea

Modern AI systems do not only generate answers; they must first decide what information is relevant.

- Early search systems focused on exact keyword matching.
- Semantic search introduced meaning-aware retrieval through embeddings.
- RAG connected retrieval with language generation.
- Agentic RAG turns retrieval into a dynamic reasoning step.

Key shift: from finding documents to constructing grounded answers.

Classical Search: Strong but Shallow

Keyword retrieval

- Inverted indices
- TF-IDF
- BM25
- Exact term matching

Main limitation

- No real language understanding
- Weak handling of synonyms
- Sensitive to wording
- Limited understanding of user intent

Interpretation

Traditional search is efficient and precise, but it mostly answers: *"Where do these words appear?"*

Semantic and Hybrid Retrieval

- Semantic search represents text as dense vectors in an embedding space.
- Similar meanings are placed close together, even when exact words differ.
- This improves recall and captures user intent more effectively.
- In practice, strong systems often combine semantic retrieval with keyword-based retrieval.

Hybrid retrieval

BM25 provides lexical precision, while embeddings provide semantic recall. Together, they make retrieval more robust.

Why RAG?

LLMs can generate fluent answers, but their knowledge is limited by training data, cutoff dates, and lack of access to private or domain-specific documents.

1. The user asks a question.
2. The system retrieves relevant external documents.
3. Retrieved evidence is added to the model context.
4. The LLM generates a grounded answer.

Result: less hallucination, better domain adaptation, and easier access to updated knowledge.

What is Agentic RAG?

Definition

Agentic RAG extends standard RAG by giving the LLM control over the retrieval process. Instead of retrieving once, the system can plan, search, evaluate, and refine its own information-gathering steps.

- Decides whether retrieval is needed.
- Chooses where to search: documents, databases, APIs, web, or tools.
- Reformulates queries when the first results are not sufficient.
- Compares and validates evidence across multiple sources.
- Stops when enough information has been collected to answer reliably.

Key idea: retrieval becomes part of reasoning, not just a preprocessing step.

From Static RAG to Agentic RAG

Static RAG

- Fixed retrieval pipeline
- One retrieval step
- Predefined knowledge source
- Limited adaptation

Agentic RAG

- Retrieval as a tool
- Query rewriting
- Iterative search
- Source comparison
- Multi-step synthesis

Main point

Agentic RAG moves from a fixed pipeline to a decision-making system that can decide what to search, where to search, and when enough evidence has been collected.

Main Takeaway

The evolution of retrieval is not only about better search results.

It is about building systems that know
what to look for, where to look, and how to use what they find.

Connection to the next topics

This motivates RAG, GraphRAG, long-context models, tool use, agents, and MCP.

Prompting vs RAG

Aspect	Prompting	RAG
Main control	Instructions and examples	External evidence
Knowledge source	Model parameters	Retrieved documents
Best for	Task framing and output control	Grounded, domain-specific answers
Limitation	Cannot add missing knowledge	Depends on retrieval quality
Typical failure	Generic or unsupported answers	Irrelevant or incomplete context

Key distinction

Prompting improves how the model uses what it already knows; RAG gives the model access to information it may not know.

RAG vs Agentic RAG

Aspect	Standard RAG	Agentic RAG
Retrieval process	Fixed pipeline	Dynamic decision process
Query handling	Usually one query	Query rewriting and refinement
Evidence gathering	Single retrieval step	Iterative, multi-step retrieval
Tool use	Limited or predefined	Can use retrievers, APIs, tools, memory
Best for	Direct knowledge lookup	Complex research and synthesis tasks
Main risk	Poor retrieval leads to poor answers	More complexity and harder evaluation

Key distinction

Standard RAG retrieves context once; Agentic RAG treats retrieval as part of

Beyond RAG: Long Context and CAG

Core idea

RAG is not the only way to give an LLM access to external knowledge. Long context and Cache-Augmented Generation provide alternative ways to place information inside the model's inference process.

- **RAG** retrieves only selected pieces of information.
- **Long context** puts large amounts of information directly into the prompt.
- **Cache-Augmented Generation** reuses previously processed context through cached internal states.

Main question: should the model search for information, read everything, or reuse what it has already processed?

Long Context

Definition

Long-context models can process very large prompts, allowing many documents or long inputs to be placed directly into the context window.

- Instead of retrieving small passages, the system provides the full document collection or a large subset of it.
- This avoids retrieval errors, such as missing an important document.
- It is simple to implement: put the relevant material into the prompt and let the model read it.
- It works well for one-off analysis of long documents or small collections.

Key idea: long context replaces retrieval with direct reading.

Limitations of Long Context

- Large prompts increase cost, because API pricing usually scales with input tokens.
- They also increase latency, since the model must process more text.
- The model may suffer from the *lost-in-the-middle* effect, where information in the middle of a long context is harder to use reliably.
- Repeated questions over the same documents require reprocessing the same context again and again.

Main limitation

Long context is simple, but it can become expensive and inefficient when the same knowledge is queried repeatedly.

Cache-Augmented Generation

Definition

Cache-Augmented Generation stores the model's internal representation of a fixed knowledge base, so repeated queries do not require processing the same documents from scratch.

1. **Knowledge preparation:** documents are selected and formatted.
2. **Pre-computation:** the model processes the documents and creates a key-value cache.
3. **Inference:** new queries reuse the cache and append only the user question.

Key idea: the model reads the knowledge base once and reuses that computation later.

What is the KV Cache?

Simple explanation

When a transformer model reads text, it creates internal key-value representations that help it remember and attend to what it has already processed.

- These representations are part of the model's working memory during inference.
- Normally, they are recomputed every time the same long prompt is sent.
- CAG saves these representations after the documents are processed.
- Later questions can reuse the saved cache instead of rereading the full context.

Simple analogy: long context rereads the textbook every time; CAG keeps prepared notes from the first reading.

Long Context vs CAG

Aspect	Long Context	CAG
Main idea	Put everything in the prompt	Reuse precomputed context
Computation	Repeated every query	Done once, then reused
Best for	One-off document analysis	Repeated queries over stable data
Cost pattern	High cost every time	Higher initial cost, lower repeated cost
Data changes	Easy to update	Cache must be recomputed
Main limitation	Cost, latency, lost-in-the-middle	Requires stable knowledge and cache management

Takeaway

Long context is simple and flexible; CAG is efficient when the same stable knowledge base is queried many times.

Prompt Caching

Practical version of CAG

Prompt caching allows repeated requests with the same long prompt prefix to reuse previous computation.

- The system sends a long shared context once, such as documentation, policies, or product information.
- Later requests reuse the same prefix and only add the new user query.
- The provider manages the caching mechanism behind the scenes.
- This can reduce latency and cost for repeated queries over the same context.

Example: an HR assistant repeatedly answering questions over the same company policy documents.

When to Use Each Approach

Approach	Best fit
RAG	Large, changing knowledge bases where only some documents are relevant
Long context	One-off analysis where the relevant material can fit in the context window
CAG	Repeated queries over a stable knowledge base
Prompt caching	Repeated prompts with a shared long prefix

Main point

These methods are not mutually exclusive. Modern systems may combine retrieval, long context, caching, and tool access depending on the workload.

Why MCP Exists

Motivation

LLMs are powerful at reasoning over text, but real-world tasks often require access to external systems.

- A model may need to read files, query databases, call APIs, or use enterprise tools.
- Without a standard interface, every tool integration must be built separately.
- This makes AI systems harder to scale, maintain, secure, and govern.
- MCP addresses this by defining a common way for AI agents to connect to external tools and data.

Key idea: MCP helps move AI from isolated text generation to tool-using, context-aware systems.

From Context Delivery to Tool Access

Transition

So far, we have focused on how to provide knowledge to the model: through retrieval, long context, caching, or prompt caching.

- These methods help the model access and use relevant information.
- But many real-world tasks require more than reading context.
- An AI system may need to query a database, call an API, read files, update a ticket, or interact with enterprise software.
- This requires a structured way for agents to connect to external tools and systems.

This is where MCP becomes important: it standardizes how AI agents access external capabilities.

What is MCP?

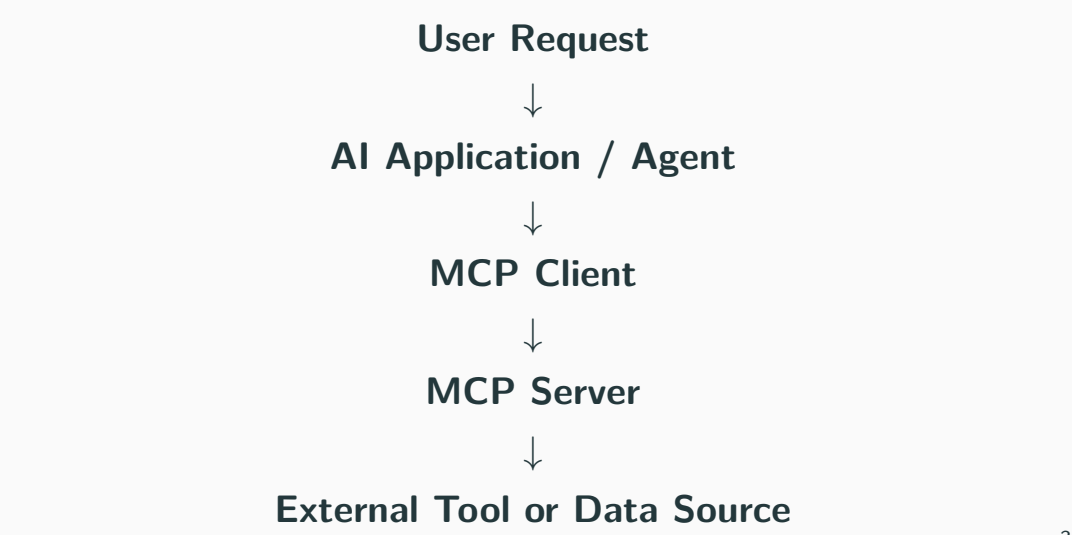
Definition

Model Context Protocol is a standard protocol that allows AI applications to access external tools, data sources, and services through structured MCP servers.

- It acts as a communication layer between the AI agent and the outside world.
- MCP servers expose specific capabilities as tools.
- Each tool has a clear name, description, input schema, and output format.
- The agent can inspect these tools and decide which one to use for a given task.

Simple analogy: MCP is like a universal adapter that lets AI agents safely plug into many different systems.

The Basic MCP Architecture



MCP Components

Core architecture

MCP separates the AI application from the external systems it needs to access. This separation makes tool use more modular, reusable, and easier to govern.

Component	Role
MCP host	The application that contains the AI assistant, e.g. chat app or IDE assistant
MCP client	The part of the host that communicates using the MCP protocol
MCP server	Exposes tools connected to files, databases, APIs, code, or services
External system	The actual resource being accessed, such as a database, API, or local file system

Main point: the host does not need a custom integration for every external system; it connects through MCP servers.

MCP Tool-Calling Flow

1. The user asks a question in an MCP host, such as a chat app or IDE assistant.
2. The host discovers available tools from one or more MCP servers.
3. The question and available tool descriptions are sent to the LLM.
4. The LLM decides whether a tool is needed and which tool should be called.
5. The MCP client calls the appropriate MCP server.
6. The server executes the operation against a database, API, file system, or service.
7. The tool result is returned to the LLM, which produces the final answer.

Example

For the question "How many customers do I have?", the agent may call a database tool exposed by an MCP server and then use the result to answer in natural language.

What Does an MCP Server Provide?

MCP server

An MCP server exposes a set of capabilities that an AI agent can use in a structured and controlled way.

Component	Meaning
Tool name	The action available to the agent, e.g. `search_files`
Description	Explains what the tool does in natural language
Input schema	Defines exactly what arguments the tool expects
Output format	Defines what result is returned to the agent
Permissions	Controls what the agent is allowed to access or execute

Example tools: `read_file`, `search_documents`, `query_database`, `fetch_url`, `create_ticket`.

Simple Example: Research Assistant

User request

"Find the most relevant internal documents about project X and summarize the current status."

1. The agent identifies that external information is needed.
2. It calls an MCP tool such as `search_documents`.
3. The MCP server searches the authorized document repository.
4. The server returns relevant documents or passages.
5. The LLM summarizes the evidence and generates an answer.

Important: the model is not guessing from memory; it is using external context retrieved through MCP.

Example: Enterprise Assistant

User request

"Prepare me for tomorrow's meeting with this client."

- A calendar MCP server identifies the meeting and the client.
- A CRM MCP server retrieves deal history and renewal information.
- A support-ticket MCP server checks recent unresolved issues.
- A document MCP server retrieves relevant internal notes.
- Permission rules prevent the agent from accessing restricted information.

Result: the agent produces a meeting brief grounded in real, authorized, and task-specific context.

Why MCP is Useful

For AI agents

- Discover available tools
- Call tools in a structured way
- Retrieve external context
- Act beyond text generation

For organizations

- Reuse integrations
- Enforce permissions
- Standardize access
- Improve auditability and control

Key takeaway

MCP makes tool use more systematic, reusable, and governable.

MCP and Context Engineering

Connection

Context engineering is about giving the model the right information at the right time. MCP helps deliver that context from external systems.

- RAG retrieves relevant knowledge for the model.
- Agentic RAG lets the agent decide what information to search for.
- MCP provides the interface through which the agent can access tools and data sources.
- Together, they support context-aware and action-capable AI systems.

Main point: MCP is not the reasoning engine; it is the infrastructure that lets reasoning agents interact with external context.

MCP vs Direct Tool Use

Practical question

If an AI agent can already use tools directly, why do we need MCP?

- Some tasks can be solved with simple, direct tool use.
- For example, reading a local file or searching text may not require a complex protocol.
- MCP becomes more useful when the task involves structured tools, authentication, permissions, or external systems.
- The value of MCP depends on whether its abstraction is justified by the complexity of the task.

Main idea: MCP is not always necessary, but it becomes important when tool use needs structure, control, and governance.

When MCP Clearly Helps

Key intuition

MCP is most useful when there is a gap between the raw operation and the result the agent actually needs.

- Some web pages require JavaScript rendering before their content is readable.
- Some systems require authentication, token refresh, or user-specific access.
- Some enterprise tools require audit trails and permission checks.
- MCP servers can hide this complexity behind structured tools.

Example: instead of manually handling a complex web page, an MCP server can expose a simple tool such as `fetch_url` that returns clean readable content.

MCP: The Right Abstraction

MCP may be unnecessary when

- The task is simple and local.
- The operation is already well-defined.
- Extra tool schemas add unnecessary context overhead.
- The agent does not need permissions or external services.

MCP is valuable when

- The task involves external systems.
- Authentication must be managed.
- Access control matters.
- Outputs need to be normalized.
- The system must be auditable and governable.

Takeaway

MCP should be used when the abstraction, security, and governance benefits are worth the additional context and system complexity.

From Models to Compound AI Systems

Core idea

A language model alone is powerful, but it is limited by what it has learned during training and by the context it receives at inference time.

- A standalone model cannot access private databases, live APIs, user-specific information, or internal tools.
- Many real-world tasks require external data and actions.
- Compound AI systems combine models with other components: retrieval, databases, tools, verifiers, planners, and APIs.
- The model becomes one component inside a larger system.

Main shift: from isolated models to modular AI systems.

Example: Why the Model Alone is Not Enough

User request

"How many vacation days do I have left?"

Standalone LLM

- Does not know the user.
- Has no access to HR data.
- May produce a generic or incorrect answer.

Compound AI system

- Generates a database query.
- Retrieves the user's vacation balance.
- Produces a grounded natural-language answer.

Key point: correctness often depends on system design, not only model capability.

Control Logic: Programmatic vs Agentic

Control logic

Control logic defines the path a system follows to answer a query or complete a task.

Programmatic control

- Fixed workflow.
- Designed by developers.
- Efficient for narrow tasks.
- Example: always search the vacation database.

Agentic control

- LLM decides the next step.
- Plans, acts, observes, and adapts.
- Useful for complex or open-ended tasks.
- Example: choose between database, web search, calculator, or memory.

Main distinction: in agentic systems, the model participates in deciding the workflow.

What is an AI Agent?

Definition

An AI agent is a system where a language model can reason about a task, choose actions, use tools, observe results, and iterate toward a goal.

- It does not simply generate one answer immediately.
- It can break a problem into smaller steps.
- It can decide when external help is needed.
- It can call tools such as search, databases, calculators, APIs, or other models.
- It can revise its plan based on observations.

Simple view: an agent is an LLM placed inside a loop of reasoning, action, observation, and refinement.

The ReAct Pattern

ReAct

ReAct combines **reasoning** and **acting**. The agent thinks through the task, performs an action, observes the result, and then decides what to do next.

Reason → Act → Observe → Revise

- **Reason:** understand the task and create a plan.
- **Act:** call a tool or retrieve information.
- **Observe:** inspect the result or error.
- **Revise:** continue, change strategy, or produce the final answer.

From Tool-Using Agents to Multi-Agent Systems

Core idea

As AI agents become more capable, a single agent may not be enough to solve every task. Some problems require collaboration between multiple specialized agents.

- A travel-planning system may involve a flight agent, hotel agent, calendar agent, and itinerary agent.
- A software engineering workflow may involve coding, testing, documentation, and review agents.
- Each agent may have its own tools, memory, policies, and internal logic.
- The challenge is how these agents can communicate without custom integrations every time.

Main shift: from agents using tools to agents collaborating with other agents.

What is Agent-to-Agent Protocol?

Definition

Agent-to-Agent Protocol is a communication protocol that allows AI agents to discover, authenticate, and communicate with other agents through a shared interface.

- It provides a standard way for one agent to request help from another agent.
- The agents do not need to expose their internal reasoning, memory, or proprietary implementation.
- Each agent can remain an independent system while still collaborating with others.
- This is useful when complex tasks require multiple specialized agents.

Simple analogy: A2A is like a common language that lets different agents work together.

How A2A Works

1. A **user** gives a goal or request.
2. A **client agent** receives the request and decides that another agent is needed.
3. A **remote agent** publishes an **agent card** describing its identity, capabilities, endpoint, and authentication requirements.
4. The client agent authenticates and sends a task to the remote agent.
5. The remote agent processes the task and may request more information.
6. The remote agent returns a result, message, or artifact.

Key concept

The agent card tells other agents: who I am, what I can do, how to contact me, and how to authenticate.

MCP vs A2A

Aspect	MCP	A2A
Main purpose	Connect agents to tools and data	Connect agents to other agents
Interaction	Agent-to-tool	Agent-to-agent
Typical target	Files, APIs, databases, services	Specialized autonomous agents
Interface	Tool descriptions and schemas	Agent cards and task communication
Best for	External capabilities and context access	Multi-agent collaboration

Takeaway

MCP helps an agent use external systems. A2A helps agents collaborate with other agents. Together, they support richer agentic ecosystems.

Example: Travel Planning with A2A

User goal

"Plan a 4-day trip to Rome next month within a fixed budget."

1. The **client agent** receives the user's travel goal.
2. It discovers specialized remote agents through their **agent cards**.
3. A **flight agent** searches for available flights.
4. A **hotel agent** finds accommodation options.
5. An **activities agent** suggests museums, restaurants, and local experiences.
6. The client agent combines the results into one coherent itinerary.

Key idea: each agent keeps its own internal logic, tools, and data, but they collaborate through a shared communication protocol.

When Do We Need Agents?

Programmatic systems work well when

- The task is narrow.
- The workflow is predictable.
- The same steps should always be followed.
- Efficiency and reliability matter most.

Agents are useful when

- The task is complex.
- The path is not known in advance.
- Multiple tools may be needed.
- The system must plan, adapt, and recover from errors.

Takeaway

Agents are most valuable when it is too difficult to predefine every possible path the system may need to follow.

Why Agent Memory Matters

Core idea

Memory is one of the main differences between a simple chatbot and an AI agent.

- A chatbot mainly responds using the current conversation.
- An agent may need to remember facts, procedures, preferences, past decisions, and previous mistakes.
- Memory allows the agent to behave consistently across tasks and sessions.
- Without memory, the system may repeat the same errors and lose important context over time.

Main point: memory turns isolated responses into persistent, context-aware behaviour.

Four Types of Agent Memory

Memory type	Role in an agent
Working memory	What the agent can see and use right now
Semantic memory	Stable knowledge, facts, rules, and documentation
Procedural memory	Skills and instructions for how to perform tasks
Episodic memory	Past experiences, interactions, decisions, and lessons learned

Interpretation

These memory types correspond to different needs: current context, factual knowledge, task execution, and learning from experience.

Working and Semantic Memory

Working memory

- The current context window.
- Includes the conversation, system instructions, files, and retrieved context.
- Fast and immediately available.
- Limited and usually temporary.

Semantic memory

- Long-term factual knowledge.
- Includes documentation, rules, project conventions, and domain facts.
- Can be implemented with files, vector stores, or knowledge graphs.
- Helps the agent avoid repeating the same knowledge gaps.

Simple analogy

Working memory is what the agent has on its desk right now; semantic memory is the reference library it can rely on.

Procedural and Episodic Memory

Procedural memory

- Stores how to do tasks.
- Encodes skills, workflows, templates, and step-by-step procedures.
- Example: how to run a code review or prepare a presentation.
- Often loaded only when needed to save context.

Episodic memory

- Stores what happened before.
- Includes previous interactions, decisions, failures, and useful lessons.
- Usually distilled rather than stored as full transcripts.
- Enables the agent to improve across sessions.

Key distinction

Procedural memory helps the agent know *how to act*; episodic memory helps it remember *what happened before*.

Memory Requirements Depend on the Agent

Agent type	Memory needs
Simple routing bot	Mostly working memory
Password reset agent	Working memory + procedural memory
Customer support agent	Working, semantic, and procedural memory
Coding agent	Working, semantic, procedural, and episodic memory
Research agent	Long-term semantic and episodic memory for synthesis across tasks

Takeaway

More complex agents need richer memory architectures, but memory must be managed carefully: what to store, what to retrieve, and what to forget.

Adapting Foundation Models

Core idea

Foundation models are general-purpose models trained on broad data, but many applications require specialization for a specific task, domain, or style.

- A single LLM can support many tasks, but it may not be optimal for every specialized use case.
- Adaptation methods help steer the model toward a desired behaviour.
- The main options include fine-tuning, prompt engineering, and prompt tuning.
- These methods differ in cost, data requirements, flexibility, and interpretability.

Main question: should we change the model, change the prompt, or learn a task-specific prompt representation?

Fine-tuning vs Prompt Engineering vs Prompt Tuning

Method	How it works	Main trade-off
Fine-tuning	Updates model parameters using labeled examples	Powerful but data- and compute-intensive
Prompt engineering	Uses human-written instructions and examples at inference time	Simple and interpretable, but can be brittle
Prompt tuning	Learns soft prompts, often as embeddings, while keeping the model frozen	Efficient, but less interpretable

Key distinction

Prompt engineering uses human-readable prompts. Prompt tuning learns optimized soft prompts that guide the model but are not directly interpretable by humans.

Three Ways to Specialize an LLM

Fine-tuning

Change the model parameters

Prompt engineering

Keep the model fixed, add human-written instructions

Prompt tuning

Keep the model fixed, learn task-specific soft prompts

Takeaway

All three approaches aim to specialize a foundation model, but they operate at different levels: parameters, natural-language prompts, or learned prompt embeddings.

Technology Stack: Core LLM Development

Topic	Typical tools and libraries
LLM APIs	OpenAI API, Anthropic API, Google Gemini API, Mistral API
Open-source models	Hugging Face Transformers, vLLM, Ollama, llama.cpp
Prompting	LangChain, LlamaIndex, Guidance, DSPy
Fine-tuning	Hugging Face PEFT, LoRA, QLoRA, Axolotl, TRL
Evaluation	lm-evaluation-harness, RAGAS, DeepEval, TruLens

Main idea

The practical LLM stack combines model access, prompting, adaptation, and evaluation tools.

Technology Stack: RAG, Context, and Retrieval

Topic	Typical tools and libraries
Vector databases	FAISS, Chroma, Milvus, Weaviate, Pinecone, Qdrant
Embeddings	OpenAI Embeddings, Sentence-Transformers, Cohere Embed, Voyage AI
RAG frameworks	LangChain, LlamaIndex, Haystack, Semantic Kernel
Hybrid retrieval	BM25, Elasticsearch, OpenSearch, Vespa
Reranking	Cohere Rerank, bge-reranker, ColBERT
GraphRAG	Neo4j, NetworkX, LlamaIndex GraphRAG, Microsoft GraphRAG
Long context	Gemini, Claude, GPT long-context models
Prompt caching / CAG	Provider-level prompt caching, KV-cache reuse, vLLM prefix caching

Technology Stack: Agents, Tools, MCP, and A2A

Topic	Typical tools and libraries
Agent frameworks	LangGraph, CrewAI, AutoGen, Semantic Kernel, OpenAI Agents SDK
Tool use	Function calling, tool schemas, API connectors, custom Python tools
MCP	MCP SDKs, MCP servers, filesystem servers, database servers, API servers
A2A	Agent cards, agent-to-agent communication, JSON-RPC style messaging
Memory	Vector stores, knowledge graphs, local files, conversation stores, Redis
Observability	LangSmith, Arize Phoenix, Weights & Biases, OpenTelemetry
Deployment	FastAPI, Docker, Kubernetes, serverless functions

What We Covered

- The evolution from AI, Machine Learning, and Deep Learning to Generative AI.
- Foundation models and Large Language Models as reusable general-purpose models.
- Prompting methods, including zero-shot, few-shot, chain-of-thought, and prompt chaining.
- Model adaptation through fine-tuning, prompt engineering, and prompt tuning.
- Context engineering as the process of providing the right information at runtime.
- Retrieval-based methods such as RAG, hybrid retrieval, reranking, and Agentic RAG.
- Long context, KV caching, Cache-Augmented Generation, and prompt caching.
- AI agents, memory, MCP, and A2A as building blocks of modern agentic systems.

Final Summary

Main message

Modern AI systems are moving from standalone models to integrated systems that combine models, context, retrieval, tools, memory, and governance.

- **Prompting** controls how the model behaves.
- **RAG and context engineering** control what information the model can use.
- **Long context and caching** improve how large or repeated context is handled.
- **Agents** add planning, tool use, iteration, and autonomy.
- **MCP** connects agents to tools and external systems.
- **A2A** enables collaboration between multiple agents.

Key takeaway: the future of AI is not only about larger models, but about better systems around models.

Thank you!

Questions?