Retrieval, Tool Use, and Function Calling

Lecture 7 Notes

CDSS 94: Building Thoughtful AI Systems (Spring 2026)

Contents

1 From Models to Interfaces

1.1 Review: Model vs. Agent

Lecture 6 established the distinction between models (stateless, single-pass, passive) and agents (stateful, iterative, proactive). That lecture focused on the *internal* capabilities of agents: planning, memory, and reasoning. This lecture turns to the *external* interfaces that connect agents to the world.

1.2 LLM Limitations That Interfaces Solve

Despite their power, LLMs have four fundamental limitations that cannot be solved by making models larger or smarter:

Definition 1.1: The Four Limitations

(i) **Static knowledge:** Training data has a cutoff. The model cannot know what happened yesterday.
(ii) **No computation:** LLMs predict tokens, not execute code. Arithmetic is unreliable.
(iii) **No world interaction:** Cannot send emails, query databases, call APIs, or modify files.
(iv) **No grounding:** Outputs are unverifiable without external evidence or execution.

Each limitation is addressed by a different class of interface:

Interface	What It Provides	Solves
Retrieval	Access external knowledge (vector DBs, search engines, document stores)	Stale knowledge, hallucination
Tools	Extend capabilities (calculators, code interpreters, browsers, image generators)	Computation, verification
APIs / Functions	Integrate with software (REST APIs, databases, SaaS platforms, OS commands)	World interaction, automation

Remark 1.2

These three interface classes are not mutually exclusive. A production agent typically uses all three: retrieval for knowledge, tools for computation, and APIs for action. The art of agent design is orchestrating them within the control loop from Lecture 6.

2 Agents and Environments

2.1 From Prompting to Environment Interaction

Early systems operated as single-turn functions: prompt → completion. Agent systems operate as environment interactions:

$$\text{state} \rightarrow \text{action} \rightarrow \text{observation} \rightarrow \text{next state}$$

This matches the structure of reinforcement learning environments. Agents interact with external systems repeatedly, building up context and adapting their strategy based on feedback.

2.2 Agent Trajectories

Definition 2.1: Agent Trajectory

An agent produces a **trajectory** of interaction: a sequence of (state, action, observation) tuples.

Example: reasoning → tool call → observation → reasoning → answer

Errors can occur at any step. Evaluation must measure the *entire interaction trajectory*, not just the final output.

Remark 2.2: Evaluation Shift

Traditional LLM evaluation measures a single output: prompt → completion. Agent evaluation must assess entire trajectories: was the plan sensible? Were the right tools selected? Did the agent recover from errors? This is a fundamentally different evaluation paradigm.

2.3 The Environment Loop

Definition 2.3: Environment Loop

(a) Agent observes environment state.
(b) Agent selects an action (text generation or tool call).
(c) Environment executes the action.
(d) Environment returns an observation.
(e) Loop continues until task completion.

This is the same agent loop from Definition 2.1 of Lecture 6, but now viewed from the environment's perspective. The environment is an active participant—it executes actions and provides feedback.

2.4 Verifiable Environments

Some environments allow automatic verification of agent behavior:

Environment	Verification Mechanism
Coding	Run unit tests (pass/fail)
Math	Exact answer checking (binary reward)
Search	Grounding against source documents
APIs	Execution success or failure

Remark 2.4: Why Verifiability Matters

Human feedback is expensive and slow. Verifiable environments enable automated evaluation and, crucially, reinforcement learning with clear reward signals. This is why coding benchmarks (SWE-Bench) and math benchmarks (GSM8K, AIME) have become the primary testbeds for agent capability—they offer automatic reward signals that close the training loop.

2.5 The RL Perspective

Agent systems naturally map to the RL formulation:

Definition 2.5: Agent-as-MDP

- **State:** conversation context (prompt + history + observations so far).
- **Action:** tool call or text output.
- **Observation:** tool result or environment feedback.
- **Reward:** task success, correctness, efficiency.

> *Old paradigm: better models → better intelligence.*
> *New paradigm: better environments → better training signals.*
> *Environment interaction enables both learning and evaluation. The shift from "build a smarter model" to "build a richer environment" is one of the defining trends in agent development.*

3 Retrieval

3.1 Why Retrieval?

Parametric knowledge is frozen after training. The model's "memory" is its weights, which encode a snapshot of the world as of the training data cutoff. Everything after that is invisible.

Retrieval allows dynamic, external knowledge access at query time. Instead of retraining (expensive, slow), we inject relevant documents into the context window at inference time. The model reasons over fresh evidence.

Remark 3.1

This is the fundamental insight behind RAG and all retrieval-augmented systems: *separate what the model knows from what it can access.* Knowledge in weights is static and expensive to update. Knowledge in an index is dynamic and cheap to update.

3.2 Retrieval-Augmented Generation (RAG)

Definition 3.2: RAG — Lewis et al., 2020

The RAG pipeline operates in five steps:

1. User submits a query.
2. Query is embedded into a vector.
3. Vector similarity search retrieves top-k relevant documents from an index.
4. Retrieved documents are inserted into the prompt as context.
5. LLM generates an answer conditioned on query + retrieved context.

Why RAG works:

- Knowledge is updatable without retraining (just update the index).
- Outputs can cite sources (verifiable, auditable).
- Scales beyond context window (index can hold millions of documents).
- Reduces hallucination (model reasons over evidence, not memory).

3.3 The RAG Pipeline: Component Deep Dive

Each component of the RAG pipeline involves critical design decisions:

Component	Details
Chunking	Fixed-size (512 tokens) vs. semantic (paragraph/section) vs. recursive. Overlap at boundaries preserves context. **Chunk size is the single most impactful hyperparameter.**
Embedding	Dense (OpenAI text-embedding-3, Cohere embed-v3, BGE) vs. sparse (BM25). Hybrid often outperforms either alone. MTEB leaderboard tracks quality.
Indexing	Vector DB (Pinecone, Weaviate, Chroma, FAISS). ANN search at scale. Metadata filtering for structured constraints.
Retrieval	Top-k similarity, MMR for diversity, cross-encoder re-ranking for precision. Typical $k = 3$–10.
Generation	Pack chunks into prompt with query. Instruct: "Answer using ONLY the provided context. Cite sources." Context window is finite.

Remark 3.3: Retrieval Design Decisions

Retrieval quality is usually the main bottleneck in RAG systems—not generation quality. Most failures trace back to the retriever missing relevant documents or returning irrelevant ones, not to the LLM misinterpreting good context.

3.4 Retrieval Failure Modes

Retrieval is fundamentally a selection problem. Three failure modes dominate:

Definition 3.4: Retrieval Failure Taxonomy

(i) **Recall failure:** Relevant documents are missed. The answer exists in the corpus but the retriever did not find it. Caused by embedding limitations, poor chunking, or query-document mismatch.

(ii) **Precision failure:** Irrelevant context is retrieved. Documents are topically related but do not contain the answer. Wastes context window and can mislead the model.

(iii) **Utilization failure:** The model ignores retrieved evidence. The right document is in context, but the model relies on parametric memory instead. The "lost in the middle" phenomenon: models attend more to the beginning and end of context, ignoring documents placed in the middle.

3.5 Static vs. Agentic Retrieval

	Static Retrieval	**Agentic Retrieval**
Flow	query → retrieve → generate	think → retrieve → think → retrieve
Retrieval	Once, before generation	During reasoning, as needed
Adaptivity	Cannot adapt if initial retrieval misses	Can refine queries based on partial results
Best for	Simple factual QA	Multi-hop reasoning, complex tasks

3.6 Multi-Hop Retrieval

Some questions cannot be answered in a single retrieval step.

Example 3.5: Multi-Hop Retrieval

Question: "What is the GDP of the country where the inventor of the telephone was born?"

Hop 1: Who invented the telephone? → Alexander Graham Bell
Hop 2: Where was Bell born? → Edinburgh, Scotland (UK)
Hop 3: What is the GDP of the UK? → ∼$3.1 trillion

Each hop depends on the previous result. Single retrieval cannot bridge all three.

Benchmarks: HotpotQA (2 hops), MuSiQue (2–4 hops), 2WikiMultiHopQA.
Error compounding applies: 90% accuracy per hop × 3 hops = $0.9^3 \approx 73\%$ end-to-end.

3.7 Query Rewriting

User queries are often poor search queries. LLMs can rewrite them to improve retrieval:

Definition 3.6: Query Rewriting Strategies

(i) **Query decomposition:** Break a complex question into simpler sub-queries.
"Compare economic policies of the last three presidents" → 3 separate queries, one per president.
(ii) **Multi-query retrieval:** Generate multiple rephrasings and retrieve for each. Increases recall by capturing different phrasings of the same intent.
(iii) **Step-back prompting:** Ask a more general question first, use that context for the specific one.
"What is the solubility of X at 25°C?" → first ask "What factors affect solubility?"

3.8 Dynamic Prompt Retrieval

Instead of retrieving documents, retrieve *prompts* or *demonstrations*:

Standard RAG	query → retrieve docs → insert in prompt → generate
Dynamic Prompt	query → retrieve similar examples → use as few-shot demos → generate

Remark 3.7

This works because few-shot performance depends heavily on which examples you pick. Retrieving task-similar examples beats random selection. It enables dynamic few-shot prompting without manual curation. A coding assistant retrieves similar solved problems as examples; a customer support agent retrieves similar resolved tickets as templates.

3.9 Case Study: WebGPT

Example 3.8: WebGPT — Nakano et al., 2021

WebGPT fine-tuned GPT-3 to answer questions using a text-based web browser. The agent loop:

1. Receive question + browser state summary.
2. Issue commands: `Search[...]`, `Click[...]`, `Quote[...]`, `Scroll`.
3. Observe result, decide next action.
4. Collect references while browsing.
5. Synthesize answer with citations.

Training: imitation learning on human demonstrations, then RL/rejection sampling against a reward model.

Result: WebGPT answers were preferred over human demonstrators 56% of the time. This was an early demonstration that retrieval could be treated as an environment for agent training.

3.10 Open Questions in Retrieval

Remark 3.9: When to Retrieve

Retrieval is not free. It adds latency (network + embedding + search), cost (API calls, vector DB queries), noise (irrelevant results can confuse the model), and consumes context window (retrieved docs displace reasoning tokens).

The optimal policy: retrieve when expected information gain $>$ cost of retrieval. Models must learn to calibrate when they know enough vs. when they need external evidence. This is an active area of research—Toolformer (Section 4) represents one approach.

4 Tools and Function Calling

4.1 What Are Tools?

Definition 4.1: Tools

Tools extend model capability beyond text generation. They provide capabilities the model **cannot replicate through text generation alone**:

- **Calculator:** reliable arithmetic (no hallucinated math).
- **Python interpreter:** execute code, process data, generate plots.
- **Web browser:** search and read live web content.
- **File system:** read, write, edit files.
- **Image generator:** DALL-E, Stable Diffusion.
- **Database client:** SQL queries against real databases.
- **Shell / terminal:** run system commands.

4.2 The Tool Execution Loop

The tool execution loop follows a four-step protocol:

Definition 4.2: Function Calling Protocol

1. **Define:** Tools are declared in the API request with name, description, and input schema.
   `tools: [{ name: "get_weather", description: "...", input_schema: {...} }]`
2. **Invoke:** The model returns a `tool_use` block (instead of text).
   `{ type: "tool_use", name: "get_weather", input: { city: "SF", units: "F" } }`
3. **Execute:** Your code executes the function and returns the result.
   `{ type: "tool_result", content: "72F, sunny, wind 5mph" }`
4. **Generate:** The model generates a final response using the tool output.
   "It's a beautiful day in SF—72°F and sunny with light winds."

Remark 4.3

The model never executes the tool itself. It generates a *structured request* (JSON with function name and arguments), external code executes it, and the result is fed back into the model's context. This separation is critical for safety and control.

4.3 Why Function Calling Matters

Proposition 4.4: Function Calling Makes LLMs Programmable

Without function calling: LLMs generate text. Useful for writing, not for *doing*.
With function calling: LLMs generate *actions*. They become the "brain" of software systems.
Three properties make this safe and reliable:

(i) **Structured outputs:** JSON schemas enforce valid argument types. Constrained decoding prevents malformed outputs. Schema validation catches errors before execution.
(ii) **Safe integration:** Whitelisted functions limit what the model can do. Permission models (e.g., Claude Code) give users explicit control.
(iii) **Sandboxed execution:** Tool execution is isolated, preventing unintended side effects.

4.4 MCP: Model Context Protocol

Definition 4.5: MCP — Model Context Protocol

MCP (Anthropic, open standard) is the standardization layer for tool use across models and platforms.
Problem: Every AI platform defines tools differently. Integrations are fragile, one-off, non-portable.
MCP defines:

- A **protocol** for how tools describe themselves (name, schema, capabilities).
- A **transport layer** for how models invoke tools (JSON-RPC over stdio/SSE).
- A **discovery mechanism** for how models find available tools.

Analogy: USB for AI tools. Write a tool once, it works with any MCP-compatible model.

Remark 4.6

The MCP ecosystem already includes connectors for Google Drive, Slack, GitHub, Jira, Salesforce, Figma, and 100+ services. Standardization reduces the $N \times M$ integration problem (N models $\times$ M tools) to an $N + M$ problem.

4.5 Toolformer: Self-Supervised Tool Learning

Example 4.7: Toolformer — Schick et al., 2023

Question: Can a model learn *when* to call tools, entirely self-supervised?
Approach: Toolformer augments a language model with the ability to insert API calls into its own text. The model learns which tool calls reduce perplexity on future tokens—if calling a calculator helps the model predict the next token better, the model learns to call the calculator.
Significance: No human-labeled tool-use examples needed. The model discovers tool-use policies from the language modeling objective alone. This connects tool use back to the compression framework from Lecture 3: a tool call is useful precisely when it reduces the remaining uncertainty (perplexity) of the output.

4.6 Gorilla: LLM Connected with Massive APIs

Example 4.8: Gorilla — Patil et al., Berkeley, NeurIPS 2024

Problem: LLMs hallucinate APIs—they generate plausible-looking but nonexistent or incorrect function invocations.
Gorilla's innovations:

- Fine-tuned LLaMA on APIBench (1,645 APIs: HuggingFace, TorchHub, TensorFlow Hub).
- **Retriever-Aware Training (RAT):** trained with and without API docs in context, so the model works in both regimes.
- **AST tree matching** for evaluation: measures structural validity of API calls, not string matching.

Results: Surpasses GPT-4 on API accuracy. Adapts to API version changes via retrieval. The Berkeley Function Calling Leaderboard (BFCL) is now a standard benchmark for this capability.

4.7 Program of Thought

Definition 4.9: Program of Thought — Chen et al., 2022

Some agents reason through code instead of natural language.

Chain-of-Thought: "Total cost is 5 items at \$3 each, so $5 \times 3 = 15$."
Program-of-Thought: `items=5; price=3; total=items*price; print(total)` $\rightarrow$ `15`

Advantages:

- Code is executable and verifiable (run it, check the answer).
- Code is precise (no arithmetic ambiguity).
- Code supports loops, conditionals, data structures.
- Output is deterministic (same code $\rightarrow$ same result).

This is why every frontier model provider offers code execution as a tool.

Remark 4.10

Program-of-Thought connects to the verification theme from Lecture 6: code execution provides a free, automatic reward signal. The agent doesn't need a learned verifier—it can just run the code.

5 Tool Selection and Reward

5.1 Tool Selection as a Decision Problem

When agents have access to many tools, selecting the right one becomes a non-trivial decision problem.

Definition 5.1: Tool Selection

Given a set of available tools $\mathcal{T} = \{t_1, \ldots, t_n\}$ (e.g., `web_search`, `code_execute`, `file_read`, `db_query`, `image_gen`, `email_send`, ...), the agent must select which tool to call (or whether to generate text instead) at each step.

Challenges:

- More tools → larger action space → harder selection.
- Tool descriptions consume context window.
- Similar tools create ambiguity (`search` vs. `browse` vs. `fetch`).

RL formulation: State = conversation context. Action = tool call or text generation. Reward = task success. This is the agent-environment loop from Lecture 6.

5.2 Reward Signals for Tool-Using Agents

What reward do we give an agent for using tools well? Multiple signals are available, each capturing a different aspect of quality:

Reward Type	Description
Outcome reward	Binary: did the final answer match ground truth? Simple but sparse.
Process reward	Did the agent use the minimum tools needed? Penalizes unnecessary calls.
Citation reward	Did the agent cite sources? Are citations faithful to retrieved content?
Execution reward	Did the tool call execute without errors? Valid arguments, correct schema?

Remark 5.2: Credit Assignment

The fundamental challenge is **long-horizon credit assignment**. An early tool decision 10 steps ago may have caused the final failure. Sparse outcome rewards give no signal about *which* tool call was the problem. This connects directly to the Process vs. Outcome Reward Model distinction from Lecture 6: PRMs provide finer-grained signal but are harder to train.

6 Agent Architecture Patterns

6.1 The Planner-Executor Pattern

Definition 6.1: Planner-Executor

- **Planner:** Decomposes task into sub-goals. Determines ordering and dependencies. May use a more capable (expensive) model.
- **Executor:** Performs individual actions from the plan. Handles tool calls, retrieval, code execution. Can use a faster, cheaper model.

This separation mirrors the hierarchical agent architecture from Lecture 6, now made concrete with tool use. The planner reasons at the level of sub-tasks; the executor operates at the level of tool calls.

6.2 Subagents and Orchestrator Architecture

Complex tasks can be divided across specialized subagents.

Example 6.2: Research Agent Architecture

Orchestrator (Opus/GPT-4): decomposes task, delegates to subagents, aggregates results.

- **Search Agent** (Sonnet): web search, find sources.
- **Reader Agent** (Sonnet): read pages, extract facts.
- **Analyst Agent** (Opus): synthesize, compare, reason.
- **Writer Agent** (Sonnet): draft final report.

Remark 6.3: Design Tradeoffs

(i) **Specialization** enables cheaper, faster subagents for routine subtasks.
(ii) **Communication overhead:** context must be passed explicitly between agents.
(iii) **Error cascading:** failure in one subagent can compound across the system—the error compounding problem from Lecture 6 now applies across agents, not just across steps.

7 Evaluating Tool-Using Agents

Agent evaluation differs fundamentally from model evaluation. Single-output metrics are insufficient.

7.1 What Must Be Measured

Definition 7.1: Agent Evaluation Dimensions

(i) **Final outcome:** Did the agent complete the task?
(ii) **Trajectory quality:** Did it take a sensible path?
(iii) **Tool correctness:** Right tools, right arguments?
(iv) **Efficiency:** Steps, tokens, API calls consumed.
(v) **Error recovery:** Did it detect and fix mistakes?

7.2 Key Benchmarks

Benchmark	What It Measures
GAIA	Multi-step real web interaction—sustained tool use over many hops
SWE-Bench	Real GitHub issues: code patches verified by test suites
BFCL	Function calling accuracy (Berkeley Function Calling Leaderboard)
METR RE-Bench	ML research engineering tasks—can agents do real technical work?
Post-train Bench	Post-training benchmark

Remark 7.2

The gap between model benchmarks and agent benchmarks is revealing. A model that achieves 90% on single-turn QA may achieve only 30% on multi-step tool-use tasks. The bottleneck is not knowledge or reasoning in isolation—it is the ability to *orchestrate* knowledge, reasoning, and tool use over extended trajectories.

8 Summary: The Interface Frame

Lecture 6 established the internal structure of agents: control loops, planning, and memory. Lecture 7 completes the picture with the *external interfaces* that connect agents to the world.

Component	What It Provides
Retrieval (RAG)	Dynamic knowledge access—query-time evidence instead of static weights
Tools	Extended capabilities—computation, verification, media generation
Function Calling	Structured action—LLMs generate JSON requests, code executes them
MCP	Standardization—write a tool once, works with any model
Environments	Training signal—verifiable feedback loops for RL

The central insight of this lecture is that the interfaces matter as much as the model. A mediocre model with excellent tools, retrieval, and environment feedback can outperform a superior model operating in isolation. This echoes the Bitter Lesson one more time: *general methods that leverage computation*—now including environment computation—*are ultimately the most effective.*

> *Without function calling, LLMs generate text. With function calling, LLMs generate actions. The shift from text to action is the shift from model to agent.*

The limits of tool-using agents are set by three factors: the quality of available tools, the agent's ability to select and sequence them, and the credit assignment problem across long tool-use trajectories. Retrieval quality is the bottleneck in knowledge tasks. Tool selection is the bottleneck in action tasks. And evaluation of the full trajectory remains an open problem.

8 References

1. Lewis, P. et al. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *arXiv:2005.11401*
2. Nakano, R. et al. (2021). WebGPT: Browser-assisted question-answering with human feedback. *arXiv:2112.09332*
3. Schick, T. et al. (2023). Toolformer: Language Models Can Teach Themselves to Use Tools. *arXiv:2302.04761*
4. Chen, W. et al. (2022). Program of Thoughts Prompting. *arXiv:2211.12588*
5. Patil, S. et al. (2024). Gorilla: Large Language Model Connected with Massive APIs. *NeurIPS 2024. arXiv:2305.15334*
6. Anthropic (2024). Model Context Protocol (MCP). `https://modelcontextprotocol.io`
7. Yao, S. et al. (2022). ReAct: Synergizing Reasoning and Acting in Language Models. *arXiv:2210.11610*
8. Mialon, G. et al. (2023). GAIA: A Benchmark for General AI Assistants. *arXiv:2311.12983*
9. Jimenez, C. et al. (2023). SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *swebench.com*