



# Agentic AI Systems

Guest Lecture, April 20

COL772/COL7372: Natural Language Processing - Spring 2026

---

Indrajit Bhattacharya

Head of AI, KnowDis AI

Adjunct Faculty, Yardi School of AI



# Goal of this Lecture

## What is the difference between ChatGPT and an Agentic AI System?

- No clean definition yet. We will explore the possibilities in this lecture.

## Why now?

- Large Language Models have crossed a threshold of capability
- No longer just models — they are **components in larger systems**
- Explosion of "agentic" frameworks, papers, companies, products
- A chaotic, exciting, poorly understood landscape

# Part I: The Landscape



# The Building Blocks — LLM Call

**The atomic unit. Everything builds from this.**

- Input a prompt, get text back
  - Possibly code, or even another prompt !
- Stateless — no internal state persists between calls
- Non-deterministic — same input, different output
  - Defining feature of generative AI, not a bug !
- One line of Python separates you from a frontier model

# The Building Blocks — RAG / Retrieval

## Augmenting an LLM call with retrieved context

- Why? LLMs have frozen knowledge — training cutoff
- Retrieve relevant documents → inject into prompt → generate answer
- Retrieval mechanism spectrum
  - From classical (BM25 — term frequency, no learned parameters) to fully learned dense retrievers (**CoBERTv2** — encode queries and documents into vectors)
  - The trend is toward learned retrieval — but the architecture does not require it

# The Building Blocks — Memory

**The difference between a stateless LLM call and a system that remembers**

- **Working memory** — what is in the current context window
- **Episodic memory** — history of prior interactions
- **Semantic memory** — long-term knowledge accumulated over time

# The Building Blocks — Memory

**Memory is a data structure. You already know these.**

<b>  Representation</b>	<b>  Connected Form</b>	<b>  Retrieval / Reasoning</b>
Key-value pairs	Relational DB	SQL, joins, transactions
Triples finding, inference	Knowledge Graph	Traversal, path
Vector embeddings ANN	Vector DB	Similarity search,
Documents / chunks	Search index	Keyword, semantic search

# The Building Blocks — Memory

## One canonical example:

- **MemGPT (Packer et al., 2023)**
- Manages memory across a hierarchy of storage tiers
  - In-context working memory
  - External episodic storage
  - Long-term semantic storage.
- LLM itself decides what to write, what to retrieve, and what to forget

# The Building Blocks — Memory

## Entity-Centric / Long-Term (The Biography Pattern)

- **Mem0, Zep**
- Extracting and persisting specific user preferences and entity relationships over time.

# The Building Blocks — LLM Call

**A common confusion: Is context concatenation memory?**

# The Building Blocks — Evaluation

**You cannot trust, debug, deploy, or improve a system you cannot evaluate.**

Agentic systems need evaluation modules

- For different dimensions — quality of output, cost, latency
- At different levels of abstraction — system-level, module-level, tool-level

# The Building Blocks — Evaluation

## What an evaluation module does:

- Takes the system's output
- Takes reference information
  - Whose nature depends on the evaluation type
- Returns a score and optionally natural language feedback
  - What specifically went wrong?

# The Building Blocks — Evaluation

**Evaluation types differ precisely in what reference information they require:**

- **Per instance ground truth** — reference information is instance-specific
  - **Exact or partial match** — a gold standard answer for this input. Exact match, F1, BLEU, ROUGE.
  - **Test cases** — a set of unit tests for this input. Pass or fail per test.

# The Building Blocks — Evaluation

Evaluation types differ precisely in what reference information they require:

- **Task-specific criteria** — reference information is shared across all instances
  - **Execution-based** — Environment produces feedback.
    - *Physics simulator for robot plan, compiler output for code*
    - Grounded. Objective.
  - **LLM-as-judge** — LLM evaluates using a shared rubric or criteria.
    - Flexible — can handle open-ended outputs.
    - Not grounded. Evaluator is the same class of system being evaluated.

# The Building Blocks — Evaluation

**Evaluation types differ precisely in what reference information they require:**

- **Task-specific criteria** — reference information is shared across all instances
  - One canonical example: **LangSmith**
  - Observability and evaluation platform for LLM applications.
  - Recognizes that importance of systematic evaluation

# The Building Blocks — Tools and Tool Use

## A function the system can call to act on the world

- Takes inputs, returns outputs
- Cannot be instructed — takes arguments and executes the same deterministic logic every time. No prompt, no learnable parameters.
- Examples: calculator, Python interpreter, database query, web search

## Tool use — an LLM calling a tool and processing its output

- The LLM does not just generate text — it generates **structured tool call**
- Parses tool output

# The Building Blocks — Tools and Tool Use

## A note on standardization:

- **MCP: Model Context Protocol (Anthropic, 2024)** — emerging open standard for how tools are described, discovered, and called by LLM-based systems
- Analogous to USB

# The Building Blocks — Planning

## Reasoning about what to do next before doing it

- The system does not just respond — it thinks ahead
- A spectrum of planning depth:
  - **Chain of Thought** — think step by step before answering
  - **Tree of Thoughts** — explore multiple reasoning paths, backtrack if needed
  - **ReAct** — interleave reasoning and acting in a loop

# The Building Blocks — Planning

## One canonical example:

- **Chain of Thought (Wei et al., 2022)**
- Simply prompting model to *reason step by step* before producing an answer
- Remarkably effective, requires no additional infrastructure, and demonstrates that *how* you prompt matters as much as *what* you prompt

# The Building Blocks — Planning

**How is Chain of Thought different from just concatenating context?**

# The Building Blocks — Multi-Agent

## **Multiple systems working together**

- One agent plans, others execute
  - One agent critiques another's output
  - Agents with different specializations collaborating on a complex task
- 
- Coordination is hard — who is in charge? Does anyone need to be?

# The Building Blocks — Multi-Agent

## Single agent vs multi-agent:

- **Single agent** — one system, one objective, one execution thread
  - Simpler to build, debug, optimize
  - Limited by what one context window can hold
  
- **Multi-agent** — multiple systems, potentially multiple objectives
  - Can parallelize, specialize, cross-check
  - What about coordination?

# The Landscape of Coordination Frameworks

## Centralized / Hierarchical

- High-level "Manager" agent decomposes and assigns tasks to specialized agents.
  - a. Top-down Centralization (crewAI)**
    - LLM-driven orchestration. High flexibility, but higher unpredictability.
  - b. Structural Centralization (LangGraph):**
    - Centralized hard-coded control graph-driven orchestration. High reliability, but not an agent.

# The Landscape of Coordination Frameworks

## Conversational / Peer-to-Peer

- **AutoGen** (Microsoft, 2023)
- Coordination is expressed as a structured conversation.
- Agents "talk" their way to a solution through back-and-forth turns.
- Emergent behavior. The solution arises from the interaction itself.

# The Building Blocks — Multi-Agent

## **A question worth sitting with:**

- Is a multi-agent system strictly more powerful than a single agent?
- In what ways yes? In what ways no? At what cost?

The answer - 'No'. The reason - not as simple.

See hints in slide 84 and 93

# Pipeline vs Agent

**An intuitive distinction — we will formalize this later**

- **Pipeline** — environment-blind
- Fixed sequence of steps regardless of what the environment returns
- Example: RAG — retrieve, inject, generate. Always in that order. Always the same way.
  
- If the environment does not react as expected, the pipeline does not notice — it proceeds regardless

# Pipeline vs Agent

**An intuitive distinction — we will formalize this later**

- **Agent** — environment-responsive
- What it does next is determined by what the environment returned
- Example: a coding agent that reads a failing test, identifies *which assertion failed and why*, formulates a *specific* fix for *that specific failure*, applies it, runs the tests again
- The next action is not predetermined — it is a function of what the environment gave back

# Pipeline vs Agent

**An intuitive distinction — we will formalize this later**

- **The key distinction: policy**
- Pipeline has a **program** — a fixed sequence of instructions
- Agent has a **policy** — a mapping from what it observes to what it does next
- $\pi$ : state  $\rightarrow$  action
  
- Environment is not an obstacle to route around — it is the input for decision

# Pipeline vs Agent

**An intuitive distinction — we will formalize this later**

- *We now have an intuitive answer — a system with a policy is an agent, a system with a program is a pipeline*
- *More in Part 3*

# Celebrated Systems — ReAct

**Reason + Act — the paper that made tool-using LLMs feel like agents**

- Yao et al., 2022
- Interleave chain of thought reasoning with tool actions
- Loop: **Thought** → **Action** → **Observation** → **Thought...**
- Example: answering a multi-hop question by searching Wikipedia iteratively

# Celebrated Systems — ReAct

## Reason + Act — the paper that made tool-using LLMs feel like agents

- *Why it mattered:* showed that LLMs could use tools reliably with a simple structured loop
- *Why it falls short:* the loop structure is hardcoded — the system cannot revise how it operates

# Celebrated Systems — Voyager

## The Minecraft agent

- Wang et al., 2023
- Plays Minecraft autonomously
- Builds a **growing skill library** — learns new skills and stores them for reuse
- Uses skills from prior experience to solve new tasks

# Celebrated Systems — Voyager

## The Minecraft agent

- *Why it is interesting:* early example of **continual learning** in agentic system
  - Capabilities accumulate across episodes
  - Each new skill is built on prior ones
  - System that finishes a session is more capable than the one that started it
- *Why it falls short:* narrow environment, perfect observability, video game rewards
- *A seed for Part 3 — what does it mean for a system's capabilities to grow through experience?*

# Celebrated Systems — Coding Agents

## Real environments that push back

- SWE-agent, Claude Code, Devin, OpenHands
- Task: fix a bug in a real GitHub repository
- Environment: a real codebase, a terminal, test runners
- Environment pushes back — tests fail, code doesn't compile, imports missing

# Celebrated Systems — Coding Agents

## Real environments that push back

- *Why they matter:* **genuine environmental feedback**
  - The system cannot hallucinate its way to success
  - Either the tests pass or they do not
- Current state of the art: ~50-60% on SWE-bench
  - Hard problems remain unsolved

# Digression - Tool Learning

**Tool Learning:** Agent doesn't just use what it's given, but actually "learns" how to use new APIs or software through trial, error, and feedback.

## The Evolution of Tool-Agility

1. **ToolFormer:** Treats a **fixed library** of tools as language property, learning through self-supervision when and how to insert API calls into its own text stream.
2. **Gorilla:** Searching through thousands of real-world GitHub and Python APIs to find the precise tool it was never explicitly taught.
3. **ToolMaker:** Upon hitting a capability gap, agent simply **codes a new, reusable Python tool** into existence.

# Celebrated Systems — AutoGPT

**The moment everyone thought AGI was two weeks away**

- March 2023 — 150k GitHub stars in one week
- Autonomous, goal-directed, used tools, spawned subtasks
- *Why it felt like a watershed:*
  - **Dynamic hierarchical task decomposition** — breaks a complex goal into subtasks at runtime
  - **Spawning** — creates new agents dynamically to handle subtasks in parallel

# Celebrated Systems — AutoGPT

- *Why it fell short:*
  - Compounding errors — each step's mistake amplifies the next
  - No reliable way to recover from failure
  - The decomposition logic was fixed and brittle
  - Prompts were hand-written — no systematic optimization

**The promise was real. The engineering was not ready.**

# The Promise of Self-Improvement

## The question that motivates everything in Part 2

- These systems are impressive but brittle
- Prompts are hand-engineered for a specific distribution of inputs
- When that distribution shifts — even slightly — performance degrades
- The system has **no mechanism to observe its own failures and adapt**

# The Promise of Self-Improvement

## The question that motivates everything in Part 2

- Every adaptation requires human intervention:
  - Inspect failures
  - Hypothesize what went wrong
  - Rewrite the prompt
  - Re-evaluate
  - Repeat
- This does not scale — in deployment, distributions shift constantly

# The Promise of Self-Improvement

## The precise question:

- Can a system observe its own failures and adapt its own behavior — without human intervention?
- This is what "self-improving AI" actually means
- This is what the next generation of work tries to deliver
- *How do they try to deliver it?*

# Part II: Formalizing Compound AI Systems

---

# The PyTorch Moment

## We have seen this before

- 2013: deep learning was exploding but the tooling was chaos
  - Theano, Torch, Caffe, MXNet, Chainer — all existed simultaneously
  - Every researcher hand-coded their own training loops
  - Prompts were the weights — hand-tuned by trial and error

# The PyTorch Moment

## We have seen this before

- Then PyTorch arrived and changed everything:
  - **Modules** — composable building blocks
  - **Parameters** — weights treated as first-class objects to be optimized
  - **Optimizers** — separate from the model, operate on parameters
  - **Automatic differentiation** — gradients computed without manual derivation
- The field consolidated. Progress accelerated dramatically.

# The Levels of Abstraction for Agentic Systems

## **The Linear Layer: LangChain and LlamaIndex.**

- "Assembly languages": Provide connectors (LLMs, Vector DBs, Tools) but leave the "Control Flow" and the "Prompts" entirely to the human engineer.
- Prompt Fragility—change the model, and the hand-written prompt breaks.

## **The Structural Layer: LlamaStack or LangGraph.**

- Move toward standardized APIs and state-machine-based orchestration .

## **The Optimization Layer (Compiled): DSPy.**

- *"What if we don't write prompts at all, but compile them?"*

# DSPy — The Core Idea

- The key insight: prompt engineering by hand is conceptually identical to hand-tuning neural network weights
  - Brittle — does not generalize across models or distributions
  - Unscalable — every change requires manual re-evaluation
  - Unprincipled — no systematic way to know if you are improving

**Prompts are parameters. Pipelines are programs. Optimization is compilation.**

# DSPy — The Core Idea

- Express *what* each module should do — not *how* to prompt it
- Let a compiler figure out the best prompts automatically
- Treat the pipeline as a program to be optimized, not a string to be crafted

Three abstractions do all the work:

- **Signatures** — declare the input/output contract of a module
- **Modules** — composable units that implement a signature
- **Optimizers** — compile a pipeline toward a metric

# DSPy — Signatures

## From declarations to optimized pipelines

**Signatures** — declare what a module does, not how to prompt it:

```
# example signature 1
```

```
qa = dspy.Predict("question -> answer")
```

```
# example signature 2
```

```
generate_query = dspy.ChainOfThought("context, question -> search_query")
```

# DSPy — Modules

**Modules** — composable units implementing a signature.

A complete RAG system:

```
class RAG(dspy.Module):  
    def __init__(self):  
        super().__init__() # Crucial: Registers sub-modules for the Optimizer  
        # 1. Declare retrieve as a sub-module  
        # it uses the globally configured Retrieval Model (RM).  
        self.retrieve = dspy.Retrieve(k=3)  
        # 2. Declare generate by binding inline signature to dspy.ChainOfThought  
        self.generate = dspy.ChainOfThought("context, question -> answer")
```

# DSPy — Modules

**Modules** — composable units implementing a signature.

A complete RAG system:

```
class RAG(dspy.Module):  
    def __init__(self):  
        ...  
    def forward(self, question):  
        # 3. Invoke retrieve to fetch search results from the RM  
        context = self.retrieve(question).passages  
        # 4. Invoke generate to synthesize the answer using the LLM  
        return self.generate(context=context, question=question)
```

# DSPy — Optimizers

**Optimizers** — compile a pipeline toward a metric automatically:

```
optimizer = dspy.BootstrapFewShot(metric=answer_exact_match)
```

```
optimized_rag = optimizer.compile(RAG(), trainset=trainset)
```

- No prompt strings written by hand. None.
- The optimized program outperforms hand-crafted prompts
- **Prompts are now parameters. Optimization is now compilation.**

# The Formal Definition

## What is a compound AI system?

A compound AI system is formally defined as:

$$\phi = (M, C, X, Y)$$

$M = \langle M_1, M_2, \dots, M_k \rangle$  — an ordered set of language modules

$C$  — control flow logic that orchestrates the modules

$X$  — global input schema

$Y$  — global output schema

# The Formal Definition

Each module  $M_i = (\pi_i, \theta_i, X_i, Y_i)$  where:

$\pi_i$  — the prompt: instructions and few-shot demonstrations

$\theta_i$  — the model weights: frozen at inference time

$X_i$  — input schema for this module

$Y_i$  — output schema for this module

# The Formal Definition

## What this definition subsumes:

Single LLM calls — trivial case,  $|M| = 1$

RAG pipelines — retrieval module + generation module

ReAct loops —  $C$  encodes the think-act-observe loop

Multi-stage reasoning — sequential, iterative, conditional, or parallel compositions of modules, expressible in arbitrary Python control flow

# The Formal Definition — What Gets Optimized

Not everything in  $\phi$  is an optimization variable

<b>Component controls it</b>	<b>Status</b>	<b>Who</b>
$\theta_i$ — model weights finetuning	Frozen at inference	Pretraining /
$\pi_i$ — prompts	<b>Optimizable</b>	The optimizer
C — control flow	Fixed by programmer	You
X, Y — I/O schemas	Fixed by programmer	You

# The Formal Definition — What Gets Optimized

$\theta_i$  is infrastructure — present but not touched during optimization

C is the architecture — fixed before optimization begins

C and  $\theta_i$  are assumed given

**The entire ecosystem optimizes only  $\pi_i$**

*Remember this. It will matter in Act 3.*

# The Optimization Problem

## What any optimizer for $\Phi$ must solve

Each module  $M_i$  has a prompt template  $\pi_i$  with optimizable slots

- Instructions  $I_i$ , demonstrations  $\Delta_i$ .

$V$ : set of all such slots across all modules.

$V \mapsto S$ : assignment to all slots

# The Optimization Problem

**Training instance** —  $(x,m) \in D$ ,  $x$ : input,  $m$ : evaluation meta data

**Execution** — running  $\Phi_{V \mapsto S}$  on  $(x,m) \in D$ , producing final output  $\hat{y}$

**Evaluation metric** —  $\mu: Y \times M \rightarrow [0,1]$

Find the assignment  $V \mapsto S$  that maximizes:

$$\Phi^* = \operatorname{argmax}_{V \mapsto S} \frac{1}{|D|} \sum_{(x,m) \in D} \mu(\Phi_{V \mapsto S}(x), m)$$

# GEPA — Reflective Prompt Evolution

**Key idea:** use natural language reflection based on execution traces to generate new, better prompt candidates

**The mechanism:**

- **Feedback function**  $\mu_f(\hat{y}, m) \rightarrow (\text{score}, \text{feedback}_{\text{text}})$  — Analyses module execution trace to pinpoint issues. e.g. from compiler errors, failed tests, etc
- **Reflective mutation** — Reflection LM sees the traces and feedback, proposes improved prompt

# GEPA — Reflective Mutation Example

**From a vague seed to a precise optimized prompt**

**Seed prompt** for second-hop retrieval:

Given the fields question, summary<sub>1</sub>, produce the fields query.

# GEPA — Reflective Mutation Example

**From a vague seed to a precise optimized prompt**

**GEPA-optimized prompt** after reflection:

You will be given two input fields: question and summary<sub>1</sub>.  
Your task is to generate a new search query optimized for  
the second hop of a multi-hop retrieval system.

Key observations:

- First-hop documents often cover one entity or aspect
- Your goal: retrieve documents NOT found in first hop
- Avoid merely paraphrasing the original question
- Infer what broader concepts might provide missing information

# Credit Assignment Problem

**The Feedback Gap:** The Metric  $\mu$  evaluates the **final output Y**.

- If Y is "Wrong," which specific prompt  $x_i$  in the chain caused the failure?
- *E.g., did the **Extraction Prompt** fail, or did the **Reasoning Prompt** fail to use the extracted data?*

# The "Prompt Gradient Desert"

**The Feedback Gap:** The Metric  $\mu$  evaluates the **final output Y**.

- If Y is "Wrong," which specific prompt  $x_i$  in the chain caused the failure?
- *E.g., did the **Extraction Prompt** fail, or did the **Reasoning Prompt** fail to use the extracted data?*

**The "Desert":** We have a "Loss" at the end of the string, but no easy way to mathematically "trace" that loss back to a specific prompt in the middle.

# GEPA — Credit Assignment Mechanism

- **Mechanism: Black-Box Optimization.**
- **The Strategy:** Treat the whole pipeline as a unit. Generate many candidate versions of  $x_1, x_2, \dots$ , stochastically.
- **The Search:** Use an LLM-optimizer to "propose" updates to one prompt at a time (Round-Robin) and keep the ones that happen to increase the global score.
- **Limitation:** Blind man's cane

# GEPA vs TextGrad

## GEPA

- Generates *new* candidates by reflecting on execution traces and NL feedback
- Credit assignment does not know which module caused the failure.

## TextGrad (Nature, 2025)

- Same reflective loop as GEPA
- Propagates feedback through the computation graph structure for targeted credit assignment.

