Reasoning Agents: Control Loops, Planning & Memory

Lecture 6 Notes

CDSS 94: Building Thoughtful AI Systems (Spring 2026)

Contents

1 From Models to Agents

1.1 The Bitter Lesson

We begin with the philosophical foundation of the entire agents module. Rich Sutton's 2019 essay, *The Bitter Lesson*, articulates a pattern that has held across seventy years of AI research:

> *"The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin."*

The core claim is that hand-engineering human knowledge into AI systems is ultimately less effective than letting systems **search** and **learn**. The scaling lesson for training is well understood: more data + more compute beats clever inductive biases. The same lesson now applies at *inference time*: give the model more computation (more steps, more reasoning) rather than engineering smarter prompts. This is the philosophical foundation of reasoning agents.

Remark 1.1: The Bitter Lesson for Agents

The inference-time restatement: *let the model think longer, not just be smarter by design.* The shift from static models to reasoning agents is the inference-time analogue of the shift from hand-crafted features to learned representations.

1.2 Model vs. Agent

The fundamental distinction between a model and an agent lies in statefulness, iteration, and autonomy.

Property	**Model**	**Agent**
State	Stateless across calls	Stateful (context & history)
Computation	Single forward pass: Prompt → Completion	Iterative: acts, observes, (re-)plans
Initiative	Passive: awaits instructions	Proactive: selects its own next action
Output	One response per query	Multi-step: traces, tool calls, final answer

Remark 1.2

A model is a function. An agent is a program that calls that function in a loop. The power comes not from the function itself, but from the structure of the loop.

1.3 The Agent Abstraction

Definition 1.3: Agent — Russell & Norvig

An agent perceives its environment through **sensors** and acts on it through **actuators**. For LLM agents:

- **Sensors:** the context window (conversation history, tool outputs, observations).
- **Actuators:** token generation, tool calls, code execution.

An agent is (1) a *model + control loop* operating in (2) an *environment*. The control loop defines how the model's outputs feed back into its inputs over time.

This definition is deliberately broad. A chatbot that maintains conversation history is a (simple) agent. A research assistant that searches the web, reads papers, and synthesizes findings is a (complex) agent. What varies is the sophistication of the loop.

2 Control Loops

2.1 The Agent Loop

The agent loop is the fundamental unit of agentic behavior. Every agent, regardless of architecture, executes some variant of the following cycle:

Definition 2.1: The Agent Loop

(a) **Observe:** Read the current state—context window, environment feedback, tool results (sensors).
(b) **Think:** Reason about what to do next—chain of thought, planning, scratchpad.
(c) **Act:** Generate an action—text output, tool call, code execution (actuators).
(d) **Update:** Incorporate the result of the action into context.
(e) **Terminate:** Stop when the task is complete or a stopping condition is met.

The agent loop wraps the LLM:

$$\text{Planning} \rightarrow \text{Memory} \rightarrow \text{Action} \rightarrow \text{Observation} \rightarrow \text{repeat}$$

Each iteration adds new information to the context, allowing the agent to refine its approach based on real feedback rather than speculation.

2.2 ReAct: Reason + Act

ReAct (Yao et al., 2022) demonstrated a key insight: LLMs can operate as agents via prompting alone, without retraining. The method interleaves reasoning traces and actions within the same context loop.

Prior work separated reasoning from action. Chain-of-thought methods (reason-only) produced reasoning traces but could not interact with the environment. Action-only methods (e.g., SayCan, WebGPT) could call tools but lacked explicit deliberation. ReAct combines both: the model generates a **Thought**, then an **Action**, then observes the result, and repeats.

Example 2.2: ReAct on HotpotQA

Question: Aside from the Apple Remote, what other device can control the program Apple Remote was originally designed to interact with?

Thought 1: I need to search Apple Remote and find the program it was originally designed to interact with.
Act 1: `Search[Apple Remote]`
Obs 1: The Apple Remote is a remote control ... originally designed to control the Front Row media center program ...
Thought 2: Apple Remote was designed to control Front Row. I need to search Front Row and find what other device can control it.
Act 2: `Search[Front Row]` → not found → `Search[Front Row (software)]`
Obs 3: Front Row is a discontinued media center software...
Thought 4: Front Row (software) is controlled by an Apple Remote or the keyboard function keys. So the answer is keyboard function keys.
Act 4: `Finish[keyboard function keys]` ✓

Remark 2.3

ReAct's significance is architectural: it showed that the agent loop can be implemented entirely through the prompt format. No special training, no separate planner module—just interleaved Thought/Action/Observation strings in the context window.

2.3 Agent Architectures

In practice, real systems organize agent loops into different architectures depending on the system structure and the environment.

System Structure:

(i) **Single-agent:** one LLM in a loop.
(ii) **Multi-agent:** orchestrator + subagents, each with specialized roles.
(iii) **Hierarchical:** a planner decomposes; executors act on sub-tasks.
(iv) **Pipeline:** a fixed sequence of LLM calls.

Environment:

(i) **Closed/simulated:** math, chess, games—full observability, clear reward.
(ii) **Grounded/text:** QA, document tasks—retrievable context, verifiable answers.
(iii) **Open/real-world:** web, code, computer use—partial observability, complex state.
(iv) **Multi-modal:** vision + action (robotics, GUI agents).

3 Planning

3.1 Why Planning Matters

Single-pass generation fails on complex multi-step tasks because the model cannot anticipate all sub-problems upfront. Good planning is what separates a chatbot from an agent capable of long-horizon work.

Definition 3.1: Planning

Planning is *decomposing a problem into intermediate steps, deciding which step to take, and updating the plan based on observations.*
The planner module interacts bidirectionally with the LLM: the LLM proposes a high-level plan, decomposes it into sub-tasks, and an executor LLM module carries out each sub-task. Crucially, the plan is *revisable*—new observations can trigger replanning.

3.2 A Framework for Planning Methods

Planning methods can be characterized along two axes:

Axis 1—When does planning happen?

- **Upfront:** plan then execute (generate a full plan before acting).
- **Interleaved:** plan while acting (revise the plan between steps).
- **Post-hoc:** revise after observing (reflect on failures and replan).

Axis 2—What is the search space?

- **Token-level:** next word (finest granularity).
- **Step-level:** reasoning step (e.g., one line of a proof).
- **Action/tool-level:** external call (coarsest granularity).

Crossing these axes yields a taxonomy of planning methods:

	Upfront	**Interleaved**	**Post-Hoc**
Token-Level	Chain-of-Thought		Best-of-N
Step-Level		Tree-of-Thought	Reflexion
Action-Level	SayCan	ReAct	

Remark 3.2

The axes also indicate what needs to be trained vs. what is purely inference-time: methods that replan benefit from learned tool-use, while pure search does not require any additional training.

3.3 Chain-of-Thought

Chain-of-Thought (CoT) prompting (Wei et al., 2022) demonstrated that large models prompted with step-by-step reasoning examples significantly outperform direct answer prompting.

Definition 3.3: Chain-of-Thought Prompting

Given a question q, instead of directly producing an answer a, the model generates a sequence of intermediate reasoning steps $s_1, s_2, \ldots, s_n$ before producing a. Two variants:

(a) **Few-shot CoT:** include examples with explicit reasoning traces in the prompt.
(b) **Zero-shot CoT:** append "Let's think step by step." Surprisingly effective even without examples.

Why it works: Intermediate reasoning steps are written into tokens, which the model can attend to when generating the next step. The scratchpad externalizes working memory.

Key limitation: CoT is linear. Steps are generated left-to-right with no branching or backtracking. Once the model commits to a reasoning path, there is no recovery.

Example 3.4: CoT vs. Direct Prompting on Arithmetic

Problem: "The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?"

Standard prompting $\rightarrow$ "The answer is 27." $\times$
Chain-of-thought prompting $\rightarrow$ "The cafeteria had 23 apples originally. They used 20 to make lunch. So they had $23 - 20 = 3$. They bought 6 more apples, so they have $3 + 6 = 9$. The answer is 9." $\checkmark$

3.4 Tree-of-Thought

Tree-of-Thought (ToT) (Yao et al., 2023) addresses CoT's linearity by framing problem-solving as a tree search.

Definition 3.5: Tree-of-Thought

At each reasoning step:

(i) **Generate** multiple candidate next thoughts.
(ii) **Evaluate** each branch (via model self-evaluation or task-specific feedback).
(iii) **Prune** poor paths and expand promising ones.

Search strategies include breadth-first (explore width before depth), depth-first (commit until stuck), and beam search (maintain top-k candidates).

ToT shows significant improvements on tasks that require exploration—problems where the correct approach is not obvious from the start, and trying multiple paths is essential.

Remark 3.6

CoT is a special case of ToT with branching factor 1. ToT trades inference cost (more LLM calls) for the ability to explore and recover from dead ends. The compute–quality tradeoff is controlled by the branching factor and search depth.

4 Memory & Context

4.1 The Memory Problem

An LLM's context window is finite and stateless across API calls. For long-horizon tasks, agents need to remember: what has been done, what was observed, and what is still pending.

Memory in agents mirrors cognitive science: **working memory** (what is active now) vs. **long-term memory** (retrievable on demand). The design of memory is an architectural decision that constrains what tasks are even possible.

Remark 4.1

Most production failures in agentic systems are memory failures: the agent forgets what it tried, enters loops, or contradicts itself. Memory management is not a secondary concern—it is often the binding constraint on agent capability.

4.2 Four Types of Agent Memory

Definition 4.2: Agent Memory Types

Short-Term Memory:

- **In-context:** everything currently in the context window.
- **Working memory:** a structured scratchpad within context (e.g., a to-do list or plan the agent maintains).
- Fast reads and writes, but bounded—fills up and truncates. Disappears when the session ends.

Long-Term Memory:

- **External retrieval:** vector database, key-value store.
- **Episodic:** logs of past actions and outcomes.
- **Semantic:** compressed knowledge summaries.
- Persistent across sessions, but requires explicit read/write operations.

The mapping to cognitive science is instructive:

Cognitive Analogue	Agent Implementation
Sensory memory (visual, auditory, haptic)	Input embeddings
Short-term / Working memory	In-context learning
Long-term: Episodic (life events)	Episodic logs, structured traces
Long-term: Semantic (facts, concepts)	Vector DB, knowledge summaries
Long-term: Procedural (skills)	Model weights (implicit)

4.3 Retrieval Augmented Generation (RAG)

RAG (Lewis et al., 2020) addresses a fundamental limitation: instead of compressing all knowledge into model weights, retrieve relevant documents at query time.

Definition 4.3: RAG

Given a query q, a retriever fetches the top-k relevant documents from an external knowledge base. These documents are concatenated with the query and passed to the LLM for generation.
Advantages:
(a) Knowledge is updatable without retraining.
(b) Outputs are grounded in verifiable sources.
(c) Scales beyond the context window.
Limitations: Retrieval quality is the bottleneck—chunking strategy, embedding model, and index size all matter. If the retriever misses a relevant document, the generator cannot recover.

4.4 Memory Design Tradeoffs

Strategy	Mechanism	Tradeoff
Buffer	Keep all turns verbatim	Lossless but fills up fast
Summary	Compress old turns with a model call	Cheap but lossy; nuance may be lost
Vector	Embed observations, retrieve by similarity	Scalable but retrieval can miss context
Episodic	Structured logs of (action, obs, outcome)	Rich and auditable but expensive to retrieve

Remark 4.4

Most production agents combine strategies: short-term buffer + long-term vector retrieval + structured task state. The choice of memory architecture is as consequential as the choice of model.

5 Test-Time Reasoning

5.1 Two Axes of Capability Scaling

So far, model capability has scaled primarily through training: larger models, more data, better post-training. Test-time reasoning introduces a second axis: spending more computation at inference time.

Remark 5.1

The key idea: better answers often come from *thinking longer*, not just from *bigger models*. Agents can think → evaluate → revise → repeat, converting inference compute into quality.

5.2 Self-Consistency (Best-of-N)

Self-consistency (Wang et al., 2023) applies a simple insight: instead of generating one reasoning chain, generate many and take a majority vote on the final answer.

Definition 5.2: Self-Consistency

Given a question, sample N independent reasoning paths. Extract the final answer from each path. Return the answer that appears most frequently (majority vote).
The method works because different reasoning paths may make different errors, but the correct answer is more likely to be consistently reached across samples.

Remark 5.3

The tradeoff is direct: quality increases with N, but cost increases linearly ($N\times$ inference). Connecting to Lecture 3: the effective KL cost of Best-of-N grows logarithmically with N.

5.3 Process vs. Outcome Reward Models

When selecting the best candidate from multiple generations, we need a scoring mechanism. Two approaches exist:

Definition 5.4: ORM vs. PRM

Outcome Reward Model (ORM): Scores the final answer only. Fast and coarse—a wrong answer with correct intermediate steps gets the same score as a wrong answer with incorrect steps.
Process Reward Model (PRM): Scores each intermediate reasoning step. Finer signal but harder to train—requires step-level correctness labels.
PRMs enable **step-level beam search**: expand and prune at each step, not just at the end. OpenAI's PRM800K provides ∼800K human-labeled step correctness labels on GSM8K math problems.

Remark 5.5

In the compression framework from Lecture 3: a good reasoning step reduces the remaining Kolmogorov complexity of the problem. PRMs learn to measure this reduction.

5.4 o1 and R1: Test-Time Scaling in Practice

OpenAI's o1 and DeepSeek's R1 are the most prominent demonstrations of test-time scaling:

OpenAI o1: Trained with RL to produce extended internal chain-of-thought before responding. Inference compute scales with problem difficulty—the model learns *when* to think longer.

DeepSeek R1: Open-weight model replicating o1-style reasoning via GRPO (from Lecture 3). Trained with only binary correctness rewards, no process-level supervision.

Theorem 5.6: Test-Time Scaling — Informal

Performance on AIME, MATH, and coding benchmarks improves roughly **log-linearly** with inference compute budget. Moreover, performance scales with both train-time and test-time compute, establishing two independent axes for improving capability.

Remark 5.7: Emergent Behaviors

Models trained with test-time RL exhibit behaviors that were never explicitly trained:

(i) **Self-verification:** checking their own work before committing.
(ii) **Backtracking:** revisiting assumptions when an approach fails.
(iii) **"Aha moments":** spontaneously reconsidering mid-trace when an error is detected.
(iv) **Progressive lengthening:** reasoning chains grow longer as training proceeds.

The model learns when to think longer—it is not just more tokens, it is more *useful* tokens.

6 Case Study: Math Agents

6.1 Why Math?

Mathematical reasoning is a clean testbed for agents: ground truth is verifiable, difficulty is controllable, and intermediate steps have structure. The agent loop maps naturally: generate a candidate solution → verify each step → identify first error → revise from that point. If an agent cannot reliably solve grade-school math, harder tasks are out of reach.

6.2 GSM8K and the Benchmark Ladder

Definition 6.1: GSM8K

GSM8K (Cobbe et al., 2021) contains 8,500 grade-school math word problems, each requiring 2–8 chained arithmetic steps. Small enough to be tractable, hard enough to require reasoning.

The progression on GSM8K tracks the history of post-training methods:

Method	Approx. Accuracy
GPT-3 direct prompting	∼35%
Chain-of-Thought	∼55%
Supervised Fine-Tuning	∼70%
RL with Verified Rewards	90%+

The benchmark ladder—GSM8K → MATH → AIME 2024—tracks the capability frontier over four years. The verification shortcut is key: math agents can check answers by running a calculator or symbolic solver, providing an automatic and unambiguous binary reward signal.

Remark 6.2: Contamination Caveat

Many GSM8K questions appear in training data. GSM1K (novel questions with the same difficulty distribution) shows lower but more honest accuracy, suggesting that some reported results are inflated by memorization.

6.3 DeepSeek R1: RL for Reasoning at Scale

R1 uses Group Relative Policy Optimization (GRPO, from Lecture 3) on math and code tasks with verified rewards. The training setup uses no process-level supervision—only binary correctness of the final answer.

Despite this minimal signal, the model exhibits several emergent behaviors:

(i) Long, structured reasoning traces—without explicit instruction to produce them.
(ii) "Aha moments"—the model spontaneously learns to reconsider its approach when it detects an error mid-trace.
(iii) Self-verification—checking its own work before committing.
(iv) Progressive lengthening—reasoning chains get longer as training proceeds.

R1 is an open-weight model matching or exceeding o1 on AIME 2024, representing a significant open-science contribution.

6.4 The Math Agent Control Loop

Definition 6.3: Math Agent Pipeline

1. **Parse:** Understand the problem statement and identify the unknown.
2. **Plan:** Decompose into sub-goals (set up equations, isolate variable, etc.).
3. **Attempt:** Generate a candidate solution with explicit intermediate steps.
4. **Verify:** Check each step with a symbolic checker, Python executor, or model self-critique.
5. **Revise:** If a step fails, backtrack to that exact step and regenerate from there.

The key ingredient is **verifiable intermediate steps**. Without these, the loop is operating blind.

7 Case Study: Coding Agents

7.1 Coding as an Environment

Code is among the best environments for agents: outputs are executable, correctness is verifiable, and the feedback loop is clear (run → error → fix). The control loop maps naturally to software engineering:

$$\text{write code} \rightarrow \text{execute} \rightarrow \text{observe stdout/stderr} \rightarrow \text{debug} \rightarrow \text{repeat}$$

Modern coding agents operate on real repositories: navigating files, reading existing code, running tests, and opening PRs. This requires integrating all three agent components: memory (codebase context), planning (task decomposition), and tool use (shell, linter, test runner).

7.2 SWE-Bench

Definition 7.1: SWE-Bench

- **Task:** Given a real GitHub issue and its repository, generate a patch that resolves the issue and passes the test suite.
- **Dataset:** 2,294 tasks from 12 popular Python repositories (Django, Flask, scikit-learn, etc.).
- **Evaluation:** Apply the generated patch, run the official test suite—pass/fail is the only metric.
- **Why hard:** Requires reading existing code, understanding intent, localizing the bug, and writing a correct targeted fix.

The progression on SWE-Bench reflects the rapid advance of coding agents:

System	Approx. Score
GPT-4 (naive prompting)	∼1–2%
SWE-agent	∼13%
SOTA agents (2025)	50%+

7.3 Key Systems

The typical coding agent pipeline follows a consistent pattern: localize relevant files and functions → plan required changes → generate patch → execute test suite → read errors and revise → submit patch or open PR.

System	Description
Devin (Cognition)	First prominent demo on SWE-bench; full-featured IDE agent
Claude Code (Anthropic)	Shell-native agentic coding; reads files, runs commands, iterates
SWE-agent	Open-source scaffold for SWE-bench evaluation
OpenHands	Open platform for software development agents

Remark 7.2: Lessons from Devin and Claude Code

Large-scale RL can improve coding agent intelligence and incentivize longer thinking, but introduces tradeoffs: overthinking / reasoning in loops, excessive self-verification, high turn counts, and sequential tool calls that could have been parallelized. From Anthropic's experience with Claude Code: the hardest problems are not generation quality—they are *context management* and *task tracking* in long sessions. The most common failure is the agent forgetting what it already tried.

8 Failure Modes of Reasoning Agents

8.1 How Agents Fail

Agent failures fall into two broad categories:

Planning Failures:

(a) **Poor task decomposition:** sub-goals don't add up to the goal.
(b) **Underspecified plans:** too abstract to execute.
(c) **Failure to replan:** not updating when observations contradict the plan.
(d) **Infinite retry loops:** repeating the same failed action.

Reasoning Failures:

(a) **Hallucinated tool calls:** calling tools that don't exist.
(b) **Fabricated observations:** making up tool outputs.
(c) **Premature convergence:** stopping before task completion.
(d) **Overconfident commitment:** not backtracking when wrong.

8.2 Error Compounding

Agentic tasks are multi-step by definition, and errors accumulate across steps. This is perhaps the most fundamental challenge in building reliable agents.

Theorem 8.1: Error Compounding

If the per-step success rate is p and the task requires n steps, the task-level success probability is:

$$P(\text{task success}) = p^n$$

At $p = 0.9$ (90% per step):

- 10 steps: $0.9^{10} \approx 35\%$
- 20 steps: $0.9^{20} \approx 12\%$

Even high per-step reliability yields low task-level reliability over long horizons.

Remark 8.2

Open problem: How do we build agents that can reliably succeed over long horizons without human correction? The GAIA benchmark—tasks requiring real multi-step tool use—reveals that frontier models which ace standard language tasks often fail on sustained multi-step reasoning.

8.3 Evaluation Challenges

Single-turn evals do not capture multi-step correctness, partial credit, or efficiency. What makes agentic evals fundamentally different is the need to measure plan quality, tool use accuracy, recovery from errors, and final outcome—all simultaneously.

Benchmark	What It Measures
GAIA	Sustained multi-step tool use (better proxy for real-world agent capability)
METR RE-Bench	ML research engineering tasks (can agents do real technical work?)
SWE-Bench	Real software engineering on GitHub repositories

Remark 8.3

Current gap: we do not have good automated metrics for whether the agent took a sensible path, independent of the final answer. A correct final answer reached through a lucky guess is fundamentally different from one reached through sound reasoning—but most benchmarks cannot distinguish the two.

9 Summary: The Inference-Time Frame

Lecture 3 established the compression frame: every post-training technique is a different way to compress human values into a neural network. Lecture 6 extends this to inference time: every agent technique is a different way to *spend computation at test time* to improve outputs.

Component	**What It Provides**
Control Loop	Iterative refinement—observe, think, act, update, terminate
Planning	Structured decomposition—break hard problems into tractable sub-tasks
Memory	Persistent state—remember what was tried, observed, and what remains
Test-Time Reasoning	Scaled inference—more compute → better answers (log-linearly)
Verification	Grounded feedback—binary reward signals close the loop

The central lesson parallels the Bitter Lesson: general methods that leverage more inference computation outperform clever engineering of agent behavior. Give the model a loop, tools, and the ability to check its own work—then let it think.

> *"Let the model think longer, not just be smarter by design."*

The limits of agentic reasoning are the limits of how effectively we can convert inference computation into task progress. Error compounding is the tax. Memory management is the constraint. Verification is the leverage. And that is the work.

9 References

1. Sutton, R. (2019). *The Bitter Lesson.* incompleteideas.net
2. Weng, L. (2023). *LLM Powered Autonomous Agents.* lilianweng.github.io
3. Yao, S. et al. (2022). ReAct: Synergizing Reasoning and Acting in Language Models. *arXiv:2210.11610*
4. Wei, J. et al. (2022). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *arXiv:2201.11903*
5. Yao, S. et al. (2023). Tree of Thoughts: Deliberate Problem Solving with Large Language Models. *arXiv:2305.10601*
6. Lewis, P. et al. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *arXiv:2005.11401*
7. Wang, X. et al. (2023). Self-Consistency Improves Chain of Thought Reasoning in Language Models. *arXiv:2203.11171*
8. Lightman, H. et al. (2023). Let's Verify Step by Step. *arXiv:2305.20050*
9. Cobbe, K. et al. (2021). Training Verifiers to Solve Math Word Problems. *arXiv:2110.14168*
10. DeepSeek (2025). DeepSeek R1 Technical Report. *arXiv:2501.12948*
11. Jimenez, C. et al. (2023). SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *swebench.com*