CS 839 · FALL 2026 · LECTURE 3

Agent Architectures and Frameworks

Orchestration, state, control flow, and debugging

Fred Sala
University of Wisconsin–Madison

September 10, 2026

Logistics and Announcements

Office Hours: Tuesday at 3:30-5:00 pm.

- But let me know if that doesn't work for too many of you.

Team Signup: Link.

- For presentations/projects, but also for in-class mini-assignments.
- I will jointly give each group some API credits.

From the last lecture

Model: Gets to propose the next output from its current context.

Agent loop: Choose an action, observe a result, update state, and decide whether to continue.

Today: Study the loop!

- Our goal: understand what to do, implement it, debug it, revise it, etc..

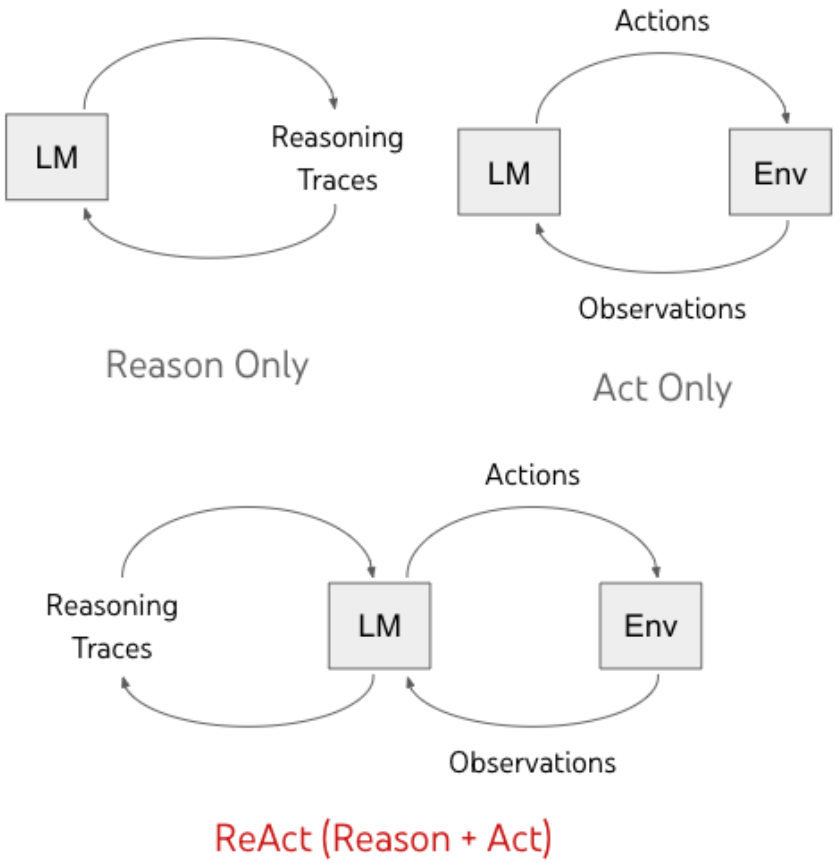
Key aspect of agent design: choosing which decisions to assign to the model and which to standard software.

From Last Time: ReAct

ReAct interleaves reasoning with actions and observations.

- Reason about the next information need
- Act (e.g., through a tool)
- Condition the next step on the observation

The environment can correct, surprise, or constrain the model.



Example of a scientific ReAct trace

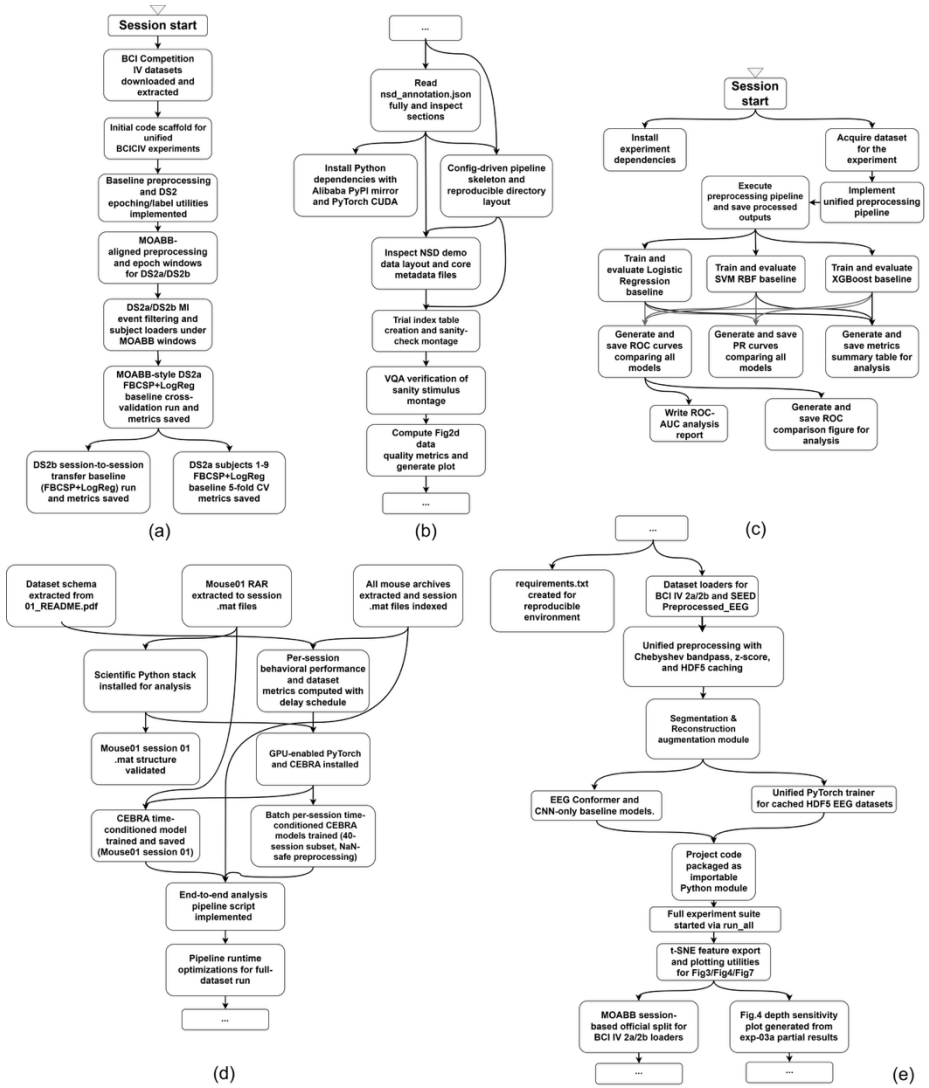
Question: Does paper X report improved stability over baseline Y?

Reason: I need the experimental section and the metric definition.

Act: Search the paper, then retrieve the relevant table and caption.

Observe: The table reports a different metric and a shorter evaluation horizon.

Revise: The claimed comparison lacks support. Search for a matched evaluation.



Li et al '26, "Graph of Trace"

Model call and agent system: key differences

	Model call	Agent system
Goal	Encoded in one prompt	Maintained across steps
State	Current context	Context plus workflow state
Actions	Generate output	Select and execute operations
Feedback	Usually none during generation	Observations affect later choices
Termination	End of generation	Success, stop rule, budget, or human decision

The minimal agent loop

Smallest version:

Goal: What counts as progress?

State: What is known now?

Action: What can the system do?

Observation: What happened?

Decision: Should the system continue, revise, or stop?

Takeaway: An agent implements a policy through repeated model calls plus external state and actions.

Stop rules

Success: A verifiable condition has been met.

Budget: Time, tokens, money, or tool calls are exhausted.

Uncertainty: Evidence is insufficient or contradictory.

Permission: The next action requires human approval.

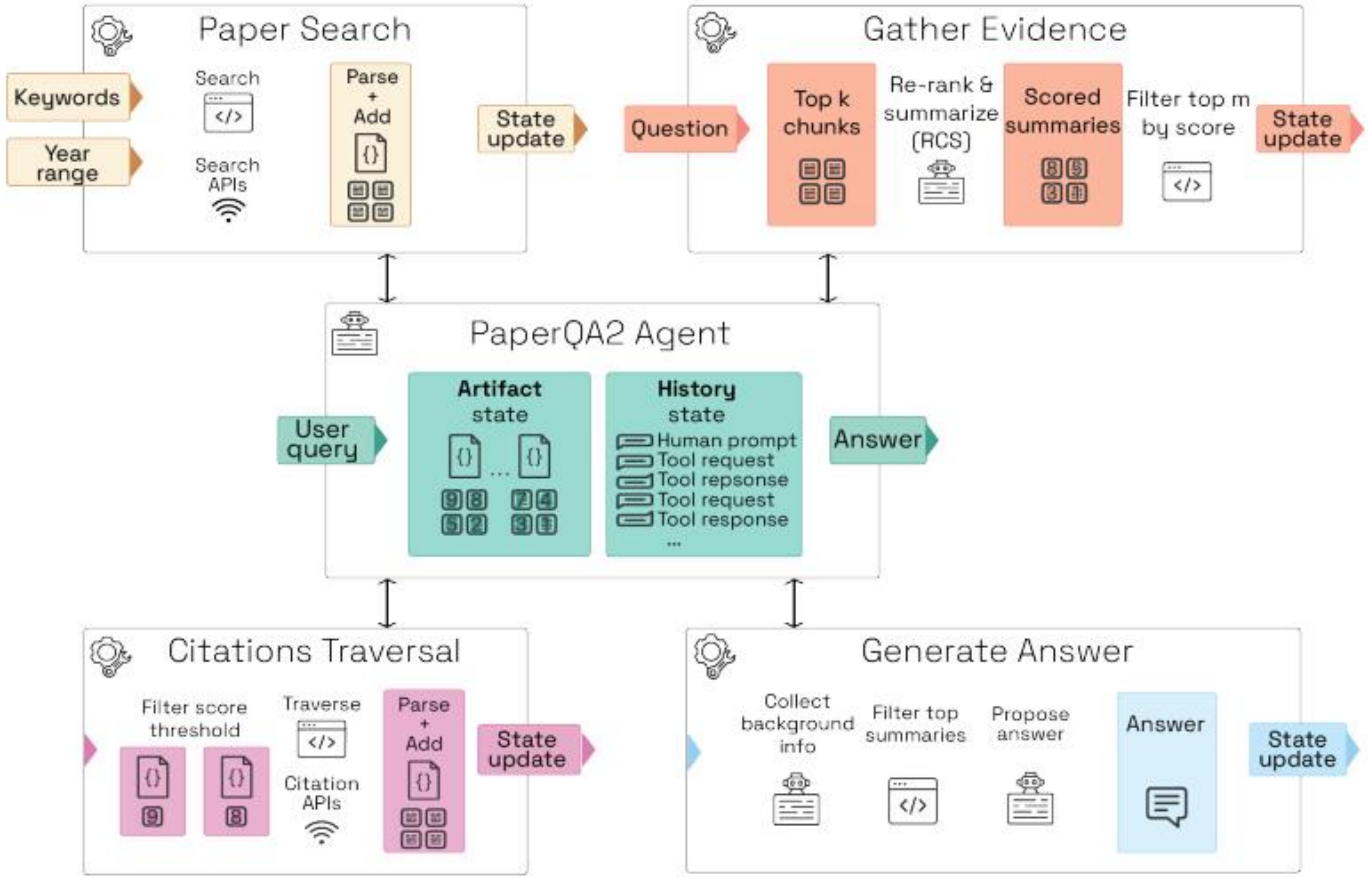
Without a stop rule, the agent will go on forever!

- Even if it doesn't really have anything to do ☺

Evidence-synthesis agent

Goal: Answer a scientific question with traceable evidence.

1. Form search queries
2. Retrieve candidate papers
3. Extract passages and metadata
4. Compare claims and evidence
5. Draft an answer with citations
6. Verify that citations support the claims



Control/permissions for scientific agents

Allowed: Read papers, run sandboxed code, and analyze provided data.

Approval required: Spend money, access credentials, change shared artifacts, or operate equipment.

Forbidden: Fabricate evidence, conceal provenance, or bypass safety constraints.

Capabilities and permissions are separate choices!

Takeaways

- **Instruction following** turns natural language into a practical interface
- **Reasoning methods** spend more inference compute on a problem, but do not create more evidence (because they can't interact!)
- Tools provide external information, exact computation, and more
- An agent repeatedly chooses actions using state, observations, and a stopping rule

Key design question: What should remain inside the model call, and what should become an explicit component in our system?

Questions for today

1. What components do we put around an agent model call?
2. How should control flow and state be represented?
3. What does an agent framework actually give us?
4. How do we localize failures in a scientific agent?

Outline

Architecture of an Agent System

Components, responsibilities, workflows, and agents

Control Flow and State

Routing, parallel work, loops, checkpoints, and recovery

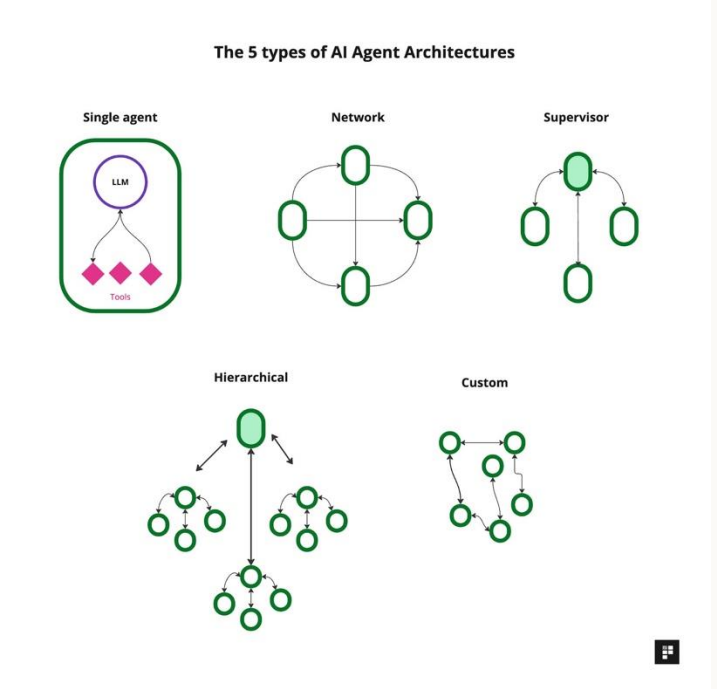
Frameworks and Debugging

Framework choices, execution traces, and failure analysis

What is an agent architecture?

An architecture is an explicit arrangement of components that turns model calls into a running system. Defines:

- **Where decisions happen**
- **What state is carried forward**
- **Which actions may execute**
- **How the run stops or recovers**



Holmes ‘25

The minimal loop, concrete version

1. **Read:** Goal, current state, available actions, constraints, etc.
2. **Decide:** Ask the model for the next step.
3. **Act:** Validate and execute the selected operation.
4. **Record:** Store the observation, artifacts, metadata about it.
5. **Stop or continue:** Apply success, budget, uncertainty, and permission rules.

Next, let’s look at the components.

Components of an agent system

Component	Responsibility	Typical contents
Model	Propose actions or structured outputs	Prompt, context, tool schema
Controller	Choose what runs next	Branches, loops, budgets, approvals
State	Carry information across steps	Messages, variables, artifacts, status
Tools	Act on an environment	Search, code, databases, instruments
Evaluator	Judge progress or validity	Tests, constraints, evidence checks
Trace	Record the execution	Inputs, outputs, timing, errors, cost

What types of control must we handle?

Before a call: Assemble context, select tools, handle permissions.

After a call: Validate result, dispatch actions, capture observations.

Between calls: Route, checkpoint, retry, pause, end.

Across the run: Track budget, progress, metadata, user intent.

Surprisingly involved given simplicity of agent loop!

Workflow or agent? Some key differences:

	Workflow	Agent
Path	Fixed/predetermined in code	Chosen dynamically
Best fit	Stable process with known steps	Open-ended task with uncertain steps
Testing	Each branch can be enumerated	Behavior has to be sampled and evaluated; cannot be easily enumerated
Risk	Brittle! Bad things happen when assumptions change	Drift, cost, and surprising/unpredictable actions
Science application example	Fixed ETL and analysis pipeline	Adaptive evidence search

Heuristic: use the least dynamic control that works

Known sequence: Use the sequence.

Known alternatives: Use a router and explicit branches.

Known quality test: Use an evaluator and add a revision loop.

Unknown next step: Let the model select among constrained actions.

Agentic control is most useful when the path depends on information discovered during the run; this is hard to pre-specify.

Running example: evidence to simulation

Stage	Software responsibility	Model responsibility
Question	Require material, property, and conditions	Clarify missing scientific intent
Evidence	Search approved sources and retain provenance	Choose queries and assess relevance
Extraction	Validate a structured schema	Map passages into fields
Simulation	Run versioned code in a sandbox	Propose parameters and interpret results
Report	Check citations, units, and artifacts	Synthesize claims and limitations

BREAK & QUESTIONS

What part of the architecture should be deterministic?

We will pick up with control flow and state.

Outline

Architecture of an Agent System

Components, responsibilities, workflows, and agents

Control Flow and State

Routing, parallel work, loops, checkpoints, and recovery

Frameworks and Debugging

Framework choices, execution traces, and failure analysis

Control flow primitives

Sequence: Run A, then B, then C.

Branch: Choose one path from explicit conditions.

Parallel: Run independent work concurrently, then merge.

Loop: Repeat until a check passes or a bound is reached.

Interrupt: Pause for an external decision, then resume.

Agent systems combine these familiar control structures (that you see in code) with probabilistic decisions.

Next, let's talk about some common patterns to be aware of for agentic design.

Pattern 1: sequential workflow

SCIENTIFIC EXAMPLE

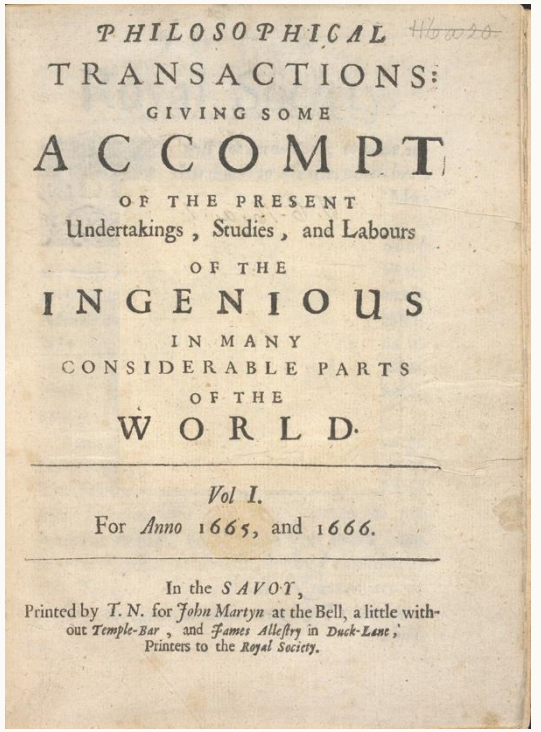
retrieve papers

extract structured results

normalize units

compare conditions

write a cited report


PHILOSOPHICAL
TRANSACTIONS:
GIVING SOME
ACCOMPT
OF THE PRESENT
Undertakings, Studies, and Labours
OF THE
INGENIOUS
IN MANY
CONSIDERABLE PARTS
OF THE
WORLD.
Vol I.
For Anno 1665, and 1666.
In the SAVOY,
Printed by T. N. for John Martyn at the Bell, a little without Temple-Bar, and James Allestry in Duck-Lane,
Printers to the Royal Society.

- Every stage has a defined input and output
- Failures can be attributed to a stage
- The model may be used inside a stage without controlling the whole path

Pattern 2: routing

Input: A scientific request plus available data.

Router: Classify the request using rules, a model, or both.

Branches: Literature search, data analysis, simulation, or clarification.

Merge: Normalize branch outputs to a shared result schema.

Router: key part of an agentic system. Also pretty challenging to make work!

- Good to integrate confidence levels into it

Pattern 3: parallelization

WHEN WORK IS INDEPENDENT

Search several databases at once, run independent analyses, or ask multiple reviewers to check different failure modes.

Fan out: Create bounded subtasks with explicit inputs.

Fan in: Deduplicate, rank, vote, or aggregate/synthesize results.

Watch out for: Shared budgets, rate limits, and inconsistent outputs.

Pattern 4: evaluator and optimizer

Generate: Produce a candidate output.

Evaluate: Apply an explicit rubric, test, constraint, etc.

Revise: Return targeted feedback and create a new candidate.

Terminate: Accept, reject, escalate, stop.

Goal of the evaluator: discriminate between candidates.

Checkpointing and durable execution

Checkpoint: persist the state after a meaningful step.

Resume: continue from the last successful point after failure or review.

Replay: re-run later steps from a known prior state.

Fork: change one state value and compare an alternative path.

Critical to include: turns one fragile call chain into something recoverable.

Retries can be subtle

Failure	Bad retry	Safer way to do it
Timeout	Assume nothing happened and repeat	Check status or use an idempotency key
Rate limit	Retry immediately in a tight loop	Back off within a bounded budget
Invalid output	Send the same request unchanged	Repair against a schema or escalate
Partial side effect	Repeat the entire step	Reconcile observed state before acting
Model uncertainty	Sample forever	Stop, request evidence, or ask a human

Human approval: another control-flow primitive

Pause before: Irreversible actions, spending, credential use, or instrument control.

Show: Proposed action, supporting evidence, expected effect, and alternatives.

Accept input: Approve, reject, edit, or request more evidence.

Resume: Persist the decision and continue from a checkpoint.

Good rule of thumb: make approval part of the architecture

What about the state?

Now that we've talked about control flow, let's see what can put in state. Some examples:

Conversation: Requests, responses, tool calls, observations.

Structured variables: Goal, plan, branch, counters, status.

Artifacts: Queries, datasets, code, figures, reports.

Evidence: Passages, citations, measurements, provenance.

Controls: Budget, permissions, checkpoints, approvals.

A state schema for our scientific example

question: str

constraints: {material, property, temperature}

queries: list[str]

sources: list[{id, url, passage}]

measurements: list[{value, unit, condition, source_id}]

simulation: {code_ref, parameters, result_ref}

budget: {tokens_left, tool_calls_left}

status: literal[searching, extracting, simulating, verifying, done]

A schema like this can be valuable!

- Lets the controller validate updates
- Lets us inspect the run without reading every message.

Three lifetimes of state

Lifetime	Purpose	Example	Failure possibilities
Step	Complete one operation	Tool arguments and returned rows	Hidden scratch data leaks forward
Run	Coordinate the current task	Plan, budget, evidence, artifacts	A retry loses progress
Across runs	Preserve durable knowledge	User preferences or validated records	Stale or private data contaminates work

BREAK & QUESTIONS

What state would you want before resuming a failed run?

We will pick up with frameworks and debugging.

Outline

Architecture of an Agent System

Components, responsibilities, workflows, and agents

Control Flow and State

Routing, parallel work, loops, checkpoints, and recovery

Frameworks and Debugging

Framework choices, execution traces, and failure analysis

What does a framework provide?

Handles everything we've discussed:

Abstractions: Agents, tasks, tools, messages, nodes, edges.

Runtime: Handles dispatch, concurrency, streaming, retries, cancellation.

State: Schemas, persistence, checkpoints, memory, artifacts.

Control: Routing, loops, interrupts, and multi-agent patterns.

Observability: Traces, logs, metrics, replay, etc.

A framework packages engineering choices. But… it does not choose the right architecture for your task.

Three common framework styles

Style	Main abstraction	Strength	Tradeoff
Graph or workflow	State plus nodes and edges	Explicit control and recoverability	More architecture to specify
Message or actor	Agents exchanging typed messages	Natural concurrency and distribution	Protocols can be hard to inspect
Role and task	Agents, goals, tasks, and processes	Fast multi-agent prototyping	Roles can hide weak task design

Some popular frameworks

Framework	Primary abstraction	State and control	Useful when
LangGraph	Graph or functional workflow	Shared typed state, checkpoints, interrupts	If we want **explicit orchestration** and durable runs
AutoGen	Event-driven agents and messages	Runtime routes asynchronous messages	We want **distributed** or **multi-agent communication**
CrewAI	Crews, agents, tasks, and flows	Processes or event-driven flows with state	We want **role-based collaboration** plus structured flows
Plain code	Functions, queues, and data structures	Whatever you implement	Loop is small or requirements are unusual

Choosing a framework

How do we pick a framework?

Start with: The state schema, tool contracts, stop rules, and evaluation plan.

Then decide: Do we need persistence, concurrency, human interrupts, deployment, or multi-agent messaging?

Prototype: One representative path in plain code or the simplest framework layer.

Adopt: What makes failures easier to understand and recover from.

Framework choice should follow execution requirements!

- No need to get fancy unnecessarily here.

Trace: the execution record

Model event: Prompt or context reference, response, latency, tokens.

Tool event: Arguments, result reference, status, duration.

State event: Fields read, fields changed, and checkpoint identifier.

Control event: Branch selected, retry, interrupt, stop reason, and budget.

Evidence event: Source identifier and link from claim to supporting artifact.

A useful trace makes the run reconstructible!

Debugging by failure layer

Layer	Question to ask	Typical symptom	First check
Model	Was the requested decision well specified?	Plausible but wrong output	Prompt, examples, model settings
Tool	Did the environment do what we think?	Error or misleading observation	Arguments, permissions, raw result
Control	Did the correct step run next?	Wrong branch or runaway loop	Router decision and stop rule
State	Was the right information carried forward?	Constraint disappears	Schema update and checkpoint diff
Evidence	Do claims trace to valid support?	Unsupported conclusion	Passage, units, and provenance

Takeaways

- Architecture makes model decisions, software decisions, state, actions, and stopping explicit
- Use deterministic control flow for known structure and agentic control only where observations determine the path
- Represent control-relevant information in state and checkpoint long runs
- Frameworks provide runtimes and abstractions. They do not replace task design
- Debug the earliest failure by inspecting model, tool, control, state, and evidence events

Core design question: Can we reconstruct why this action ran from the state and trace?

Next lecture

LECTURE 4

Tools, Environments, and Agent–Computer Interfaces

Tool schemas, execution environments, protocols, and safe action

How should tools expose capabilities?
What makes an observation trustworthy?
Where should permissions be enforced?