CS 839 · FALL 2026 · LECTURE 4

Tools, Environments, and Agent–Computer Interfaces

Function calling, code execution, scientific software, and error recovery

Fred Sala
University of Wisconsin–Madison

Logistics and Announcements

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

- Let me know if that time does not work for many of you.

Team Signup: Use the posted link for presentations, projects, and in-class work.

- Each group will receive a shared allocation of API credits.

From the last lecture

Architecture: Assigns responsibilities to the model, controller, state, tools, and evaluators.

Control flow: Combines ordinary software structure with model-selected actions.

State: Carries explicit facts, artifacts, budgets, and permissions across steps.

Trace: Records enough information to reconstruct why each action ran.

Today we focus on the boundary between a proposed action and an executed action.

From last time: components of an agent system

Component	Responsibility	Relevant questions
Model	Propose actions or structured outputs	What tool description does the model see?
Controller	Choose what runs next	How does it validate and authorize a call?
State	Carry information across steps	What result and error data persist?
Tool	Act on an environment	What operation actually executes?
Environment	Return consequences and observations	What does the agent get to observe?
Trace	Record the execution	Can we replay and diagnose the call?

From last time: debugging by layer

Model: The agent selected an unsuitable tool.

Arguments: The selected tool received invalid or incomplete inputs.

Execution: The tool timed out, failed, or produced a partial side effect.

Interpretation: The model misunderstood a valid observation.

State: The controller stored the wrong result or dropped a constraint.

Our challenge: the same error can arise from many different failure points. Which one caused it?

Questions for today

1: What information defines a useful tool interface?

2: How does interface design affect agent behavior?

3: What should a code or laboratory environment expose?

4: How should the system report and recover from failures?

Outline

Tool Calls and Contracts

Function calling, schemas, results, and tool selection

Environments and Agent–Computer Interfaces

APIs, code execution, scientific software, and physical systems

Protocols, Recovery, and Safe Action

MCP, typed errors, retries, side effects, and permissions

Tool use & motivati0no

The model does not **directly execute code**. It produces a structured request that another component validates and runs.

Components of a tool call:

Definition: The application describes available operations and their input schemas.

Call: The model returns a tool name and structured arguments.

Execution: The controller checks the request and invokes the implementation.

Result: The application returns an observation to the model.

Early forms of tool calling capabilities

Toolformer trained a language model to insert API calls into generated text.

- The call has a recognizable name and arguments
- The returned answer becomes part of the context
- Later tokens condition on the observation

Your task is to add calls to a Question Answering API to a piece of text. The questions should help you get information required to complete the text. You can call the API by writing "[QA(question)]" where "question" is the question you want to ask. Here are some examples of API calls:

Input: Joe Biden was born in Scranton, Pennsylvania.

Output: Joe Biden was born in [QA("Where was Joe Biden born?")] Scranton, [QA("In which state is Scranton?")] Pennsylvania.

Input: Coca-Cola, or Coke, is a carbonated soft drink manufactured by the Coca-Cola Company.

Output: Coca-Cola, or [QA("What other name is Coca-Cola known by?")] Coke, is a carbonated soft drink manufactured by [QA("Who manufactures Coca-Cola?")] the Coca-Cola Company.

Input: x

Output:

Where tools are used

Execution location	Who runs it	Application responsibility	Example
Application	We do (or, our controller)	Validate, execute, and return the result	Database query or internal simulator
Model provider	Provider infrastructure	Enable the tool and consume its result	Hosted search or code execution
Remote service	External API or server	Authenticate, enforce policy, and handle failures	Materials database or cloud lab
Local environment	Sandbox on the user's machine	Limit files, processes, network, and time	Shell, editor, or local analysis

Anatomy of a tool definition

Name: Usually a short identifier

Description: When the tool applies, what it does, important limits.

Input schema: Types, required fields, allowed values, field descriptions.

Output requirements: Successful data, warnings, and error.

Execution policy: Permissions, budgets, timeout, and approval requirements.

Not too different from any API---models see an interface, may not have access to the full implementation.

Selection as a function of descriptions

Interface element	Weak version	More useful version
Name	calculate	estimate_diffusion_coefficient
Description	Calculate a value	Estimate diffusion from a named trajectory and temperature
Input	query: string	trajectory_id, temperature_K, method
Output	A text answer	value, unit, uncertainty, artifact URI
Limits	None stated	Read-only, valid for equilibrated trajectories

Argument schemas

```
name: estimate_diffusion_coefficient
input_schema:
  type: object
  properties:
    trajectory_id: {type: string}
    temperature_K: {type: number, minimum: 1}
    method: {enum: [einstein, green_kubo]}
  required: [trajectory_id, temperature_K]
  additionalProperties: false
```

A schema makes syntax checkable before the environment receives the request.

Schema validity

Schema check: Is temperature_K a number and is the method allowed?

Domain check: Does the trajectory correspond to that temperature and method?

Existence check: Does trajectory_id identify an available artifact?

Policy check: May this run access the artifact and spend the requested compute?

Key to note: **valid JSON can still describe an invalid experiment**.

Looking at a result

What comes out of a completed tool call? Several things:

Status: Success, error, partial result, or pending action.

Data: A compact structured answer with units and uncertainty.

Artifacts: Stable references to files, logs, figures, and raw outputs.

Provenance: Tool version, parameters, environment, and source identifiers.

Warnings: Conditions that limit interpretation or require review.

Need to balance between (1) a concise result (helps the next model call) and (2) a complete artifact (supports later verification).

Tool granularity

Too broad: solve_research_problem hides many decisions and failure points.

Too narrow: append_one_character creates long trajectories and excessive overhead.

Useful unit: run_md_simulation performs one coherent operation with checkable inputs and outputs.

Make sure you pick good granularity/tool boundaries. Should correspond to a meaningful action that the controller can work with.

Scientific tool families

Family	Purpose	Representative operation	Important contract
Retrieval	Acquire external evidence	Search papers or query a database	Return source identifiers and passages
Computation	Run exact or specialized methods	Fit a model or run a simulator	Record code, parameters, and versions
Data	Read or transform artifacts	Load a table or normalize units	Preserve schema and lineage
Laboratory	Act on physical equipment	Prepare a plate or start a run	Require preflight checks and approval
Human	Resolve intent or authorize action	Ask for a decision	Show evidence and the proposed consequence

Tool errors

Very common! Here: the database rejected the query because the requested column did not exist.

- The tool executed and returned a useful failure signal
- The controller should preserve the raw error and the attempted arguments
- The model may repair the call if the failure is safe and recoverable

```
================================================== tool_calls ==================================================
---------------------------------------------------- tool ----------------------------------------------------
read_query

---------------------------------------------------- input ----------------------------------------------------
{"query": "SELECT DISTINCT NAICS_Code, Description FROM naics WHERE Description LIKE '%flour milling%' OR Description
LIKE '%wheat flour%' OR Description LIKE '%corn flour%'"}

================================================== read_query ==================================================
Error: ToolException('Error executing tool read_query: SQLite error: no such column: NAICS_Code')  Please fix your
mistakes.
```

BREAK & QUESTIONS

Which failures should the model see, and which should the controller handle?

We will pick up with environments and agent–computer interfaces.

Outline

Tool Calls and Contracts

Function calling, schemas, results, and tool selection

Environments and Agent–Computer Interfaces

APIs, code execution, scientific software, and physical systems

Protocols, Recovery, and Safe Action

MCP, typed errors, retries, side effects, and permissions

The environment

State: The part of the world relevant to the task.

Observation: What the interface reveals about that state.

Action: An operation the agent may request.

Transition: How the environment changes after an action.

Feedback: Evidence about success, failure, or progress.

Key idea: the interface determines which parts of the environment the model can perceive and change.

Interaction surfaces

Surface	Action representation	Observation	Typical strength
API	Function name plus typed arguments	Structured response	Compact, testable
CLI	Command and flags	Text output, files, exit code	Scientific software access
Code	Program in a sandbox	Values, logs, and artifacts	Flexible composition and exact computation
GUI	Clicks, typing, and visual actions	Screenshots or accessibility state	Works when no suitable API exists
Instrument	Hardware command or protocol	Measurements and device status	Direct physical experimentation

API and direct computer interaction

Dimension	Purpose-built API	GUI or computer control
Actions	Small set of named operations	Large set of low-level interactions
Observations	Structured fields	Pixels, layout, and application state
Stability	Usually versioned and documented	Sensitive to layout and timing changes
Auditability	Arguments and results are easy to log	Needs screenshots and action traces
Coverage	Limited to exposed capabilities	Can reach existing applications without an API

Agent–computer interfaces

An agent–computer interface adapts a computer environment to the capabilities and limitations of a language-model agent.

Actions: Expose concise operations that match common tasks.

Observations: Return relevant state without flooding the context.

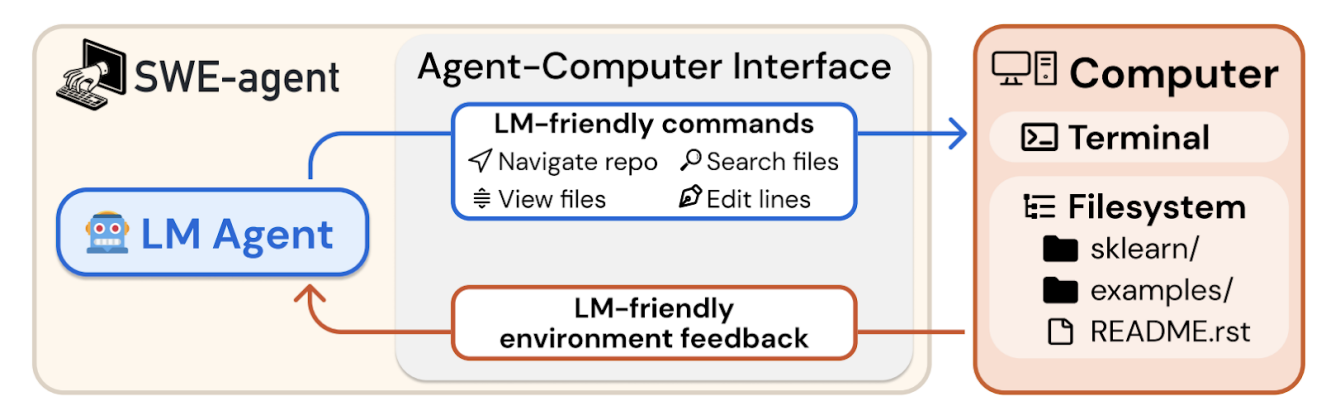
Feedback: Detect invalid actions early and explain how to recover.

Guardrails: Restrict scope before commands reach the environment.

SWE-agent and interface design

Example: **SWE-agent** gives the model commands for repo navigation, search, file viewing, and editing.

- Commands are a set of common software-engineering operations
- Feedback keeps the agent oriented in the repo
- Interface design impacts the trajectory (even if the language model stays fixed!)



Observation design

Relevance: Return the part of the state needed for the next decision.

Structure: Separate status, data, warnings, and artifact references.

Scale: Summarize long output while preserving the complete artifact.

Freshness: Include timestamps or version identifiers when state can change.

Security: Remove secrets and data outside the authorized scope.

Note: **more output does not always produce a better observation.**

Code execution environments

Isolation: Separate the run from the host and from other users.

Resources: Bound CPU, memory, storage, wall time, and process count.

Filesystem: Mount only approved inputs and dedicated output locations.

Network: Disable access by default or allow only required services.

Dependencies: Pin packages, system tools, and environment variables.

Record: Capture code, logs, exit status, and produced artifacts.

Reproducible code and simulation runs

Record	Why it matters	Example
Input snapshot	Separates the run from later data changes	Dataset hash or artifact ID
Code reference	Identifies the executed implementation	Commit and file path
Environment	Captures numerical and dependency differences	Container image and package lock
Parameters	Defines the scientific configuration	Temperature, timestep, and model
Randomness	Supports repeated stochastic runs	Seed and sampling settings
Outputs	Preserves evidence beyond a summary	Logs, trajectory, metrics, and figures

Wrappers around scientific software

Why do we use wrapper?

- Can expose stable operation to agent while keeping domain software, file formats, and execution details behind the interface.

Before execution: Check units, parameter ranges, required files, and estimated cost.

During execution: Track status, logs, checkpoints, and cancellation.

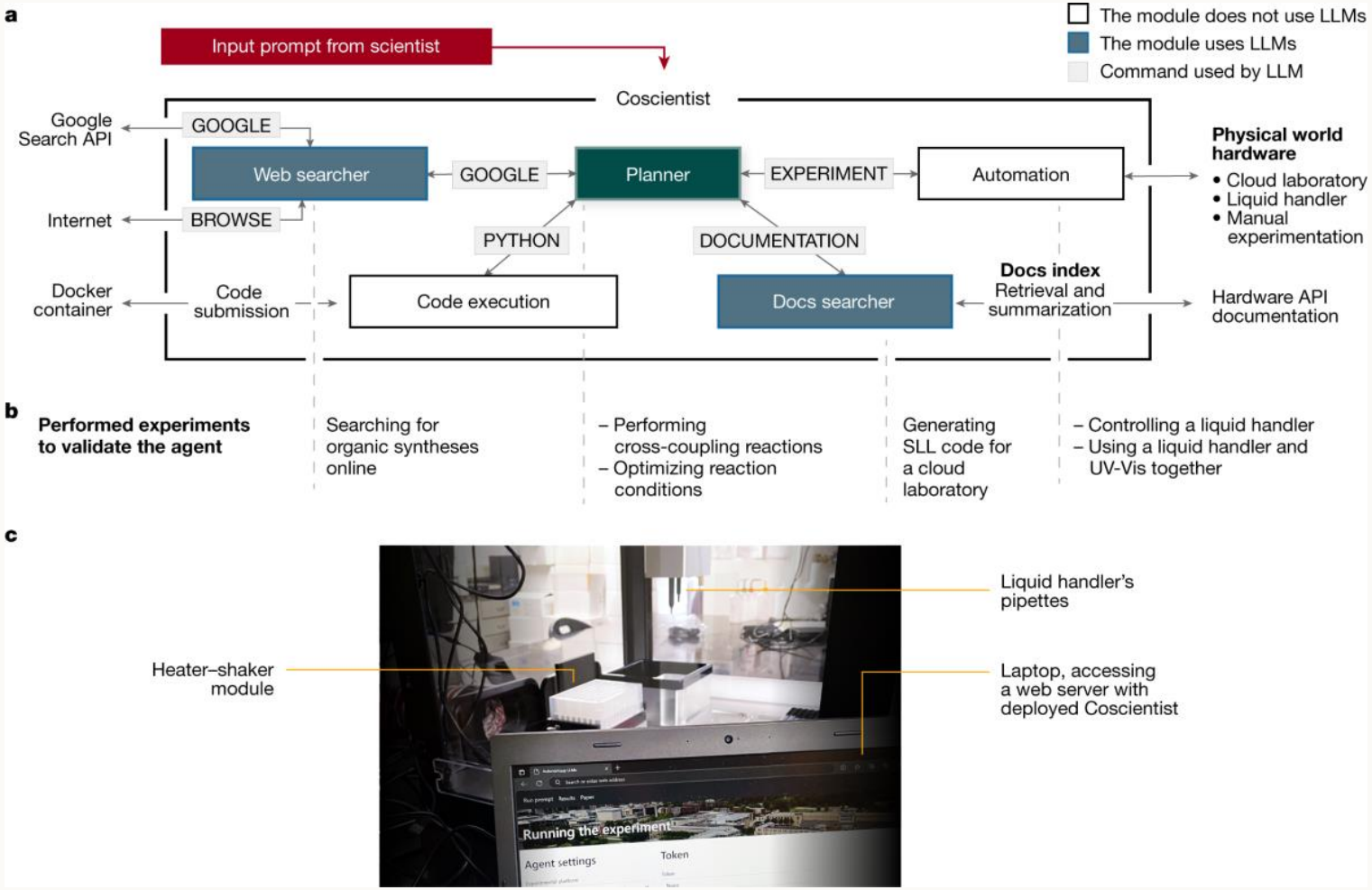
After execution: Parse outputs, validate invariants, and retain raw artifacts.

Warning: the wrapper becomes part of the scientific approach and requires its own tests.

Coscientist: tools across digital and physical environments

Coscientist connected a planner to web search, documentation retrieval, code execution, and laboratory automation.

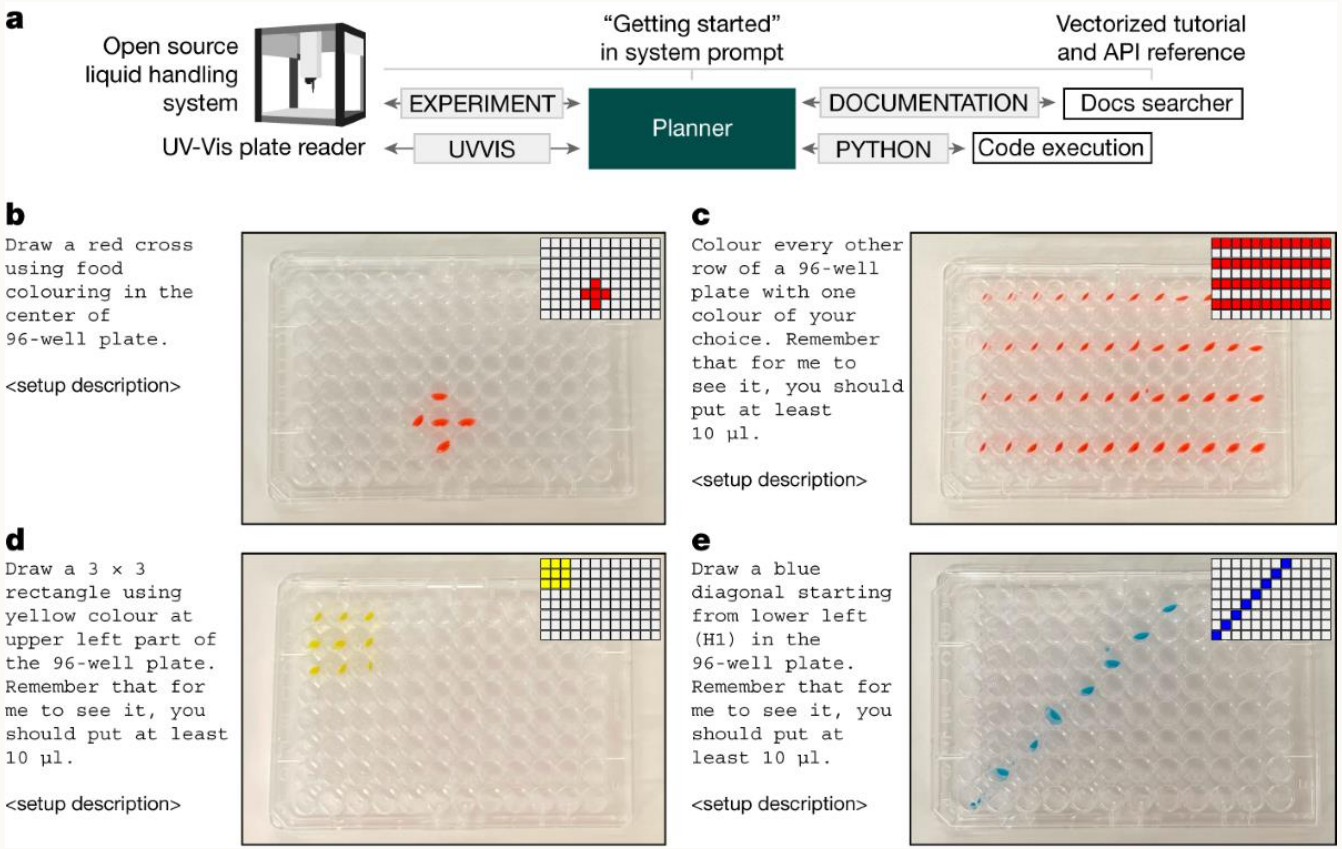
- Each module: a different action and observation space
- Documentation search helps the planner learn hardware interfaces
- Laboratory execution uses a costly physical environment



Lab environment: more than text

The physical system produces device state, images, measurements, warnings, etc.

- "Preflight checks" confirm materials, calibration, and available hardware
- Device APIs should expose status and explicit error conditions
- The trace should connect commands to measurements and physical samples



Digital and physical actions

Read: Search, inspect, and measure without changing the target system.

Compute: Consume resources while keeping external state unchanged.

Write: Modify files, databases, schedules, or shared artifacts.

Operate: Change a physical process, sample, or instrument.

Warning: as cost and irreversibility increase, interface needs stronger validation, observation, and approval requirements.

BREAK & QUESTIONS

What would you record before an agent can operate a real instrument?

We will pick up with protocols, recovery, and safe action.

Outline

Tool Calls and Contracts

Function calling, schemas, results, and tool selection

Environments and Agent–Computer Interfaces

APIs, code execution, scientific software, and physical systems

Protocols, Recovery, and Safe Action

MCP, typed errors, retries, side effects, and permissions

Why do we need tool protocols?

Discovery: A client can learn which capabilities a server exposes.

Interoperability: Different hosts can connect to the same tool provider.

Lifecycle: The system can negotiate capabilities and manage sessions.

Operations: Calls, results, progress, cancellation, and errors share a common format.

Protocols standardize connections.

- Warning: tools still require careful design.

MCP architecture

Component	Responsibility	Security aspects
Host	Coordinates the model, user interaction, and connected clients	Controls permissions, consent, and context aggregation
Client	Maintains one stateful connection to a server	Isolates one server connection from others
Server	Exposes focused resources, prompts, and tools	Receives only the context needed for its operation

MCP primitives and capabilities

Tools: Operations a model may request through the connected server.

Resources: Data or content that the host can place into context.

Prompts: Reusable message templates and workflows exposed to users.

Capabilities: Features declared during initialization and respected during the session.

Not covered: host is responsible for orchestration, authorization, and the context shown to each server.

Protocol boundaries and trust

Server isolation: Each client maintains a separate connection to one server.

Context minimization: Servers receive the data required for their focused function.

Capability negotiation: Both sides declare the protocol features they support.

User control: The host presents sensitive actions and obtains authorization.

Another warning: a protocol does not make an untrusted tool safe.

Errors as typed observations

Transport: The client could not reach the server.

Protocol: The message or capability violated the connection contract.

Execution: The tool implementation failed or timed out.

Domain: The request violated scientific constraints or missing prerequisites.

Partial result: Some work completed and may have changed state.

The error type determines whether the next action should retry, repair, etc.

Recovery policies

Failure	Controller response	Possible model role
Invalid arguments	Reject before execution and report field errors	Repair the structured call
Transient service error	Retry with bounded backoff	Usually none
Missing evidence	Change retrieval strategy or ask for clarification	Propose a new query
Permission required	Pause and request approval	Explain the proposed action
Partial side effect	Inspect external state before any retry	Interpret the reconciled observation
Scientific constraint failure	Stop or return to planning	Revise assumptions or parameters

Side effects and idempotence

Idempotency key: Let the service recognize repeated requests for the same logical action.

Precondition: Execute only if the target state still matches an expected version.

Dry run: Return the proposed change without applying it.

Transaction: Commit a group of changes together or leave the state unchanged.

Compensation: Define a separate action that repairs a completed side effect.

Permission tiers for scientific tools

Tier	Examples	Default policy	Evidence before action
Read	Search papers or inspect a dataset	Allow within approved sources	Query, source, and access scope
Compute	Run analysis or simulation	Allow within a fixed budget and sandbox	Inputs, code reference, and estimated cost
Write	Modify shared data or submit a job	Require stronger validation or approval	Diff, target, and rollback plan
Physical	Operate equipment or consume samples	Require explicit approval and preflight checks	Protocol, hazards, materials, and device state

Worked example: a simulation tool call

Request: estimate_diffusion(trajectory_id, temperature_K, method, seed)

Validate: Artifact exists, units match, method applies, and compute budget remains.

Execute: Run versioned code in a container with network access disabled.

Observe: Return value, units, uncertainty, warnings, and artifact references.

Record: Store the call, environment fingerprint, logs, and result in the trace.

Every step provides a place to catch a different class of error.

Takeaways

- Tool use separates a model's structured request from the code or device that executes it
- Useful interfaces specify action semantics, typed arguments, results, errors, and permissions
- Agent–computer interfaces shape both the available actions and the observations returned to the model
- Scientific code and laboratory tools require reproducible environments and complete artifact records
- Protocols such as MCP standardize connections, while the host still controls context and authorization
- Recovery depends on the failure type and the side effects that may already have occurred

Core design question: What contract must hold before a proposed action may affect the environment?

Next lecture

LECTURE 5

Planning and Task Decomposition

ReAct, search, reflection, verification, and long-horizon tasks

How should an agent represent a plan?
When should it revise or search over alternatives?
How can verification guide long trajectories?