



Orchestration and monitoring

Frameworks 2 · October 20, 2026

Carnegie Mellon University

Language Technologies Institute

Daniel Fried

11-768: AI Agents

Structure around agent actions

We have already seen several ways to guide agent actions.

Skills

Instructions guide behavior; executable skills can constrain execution.

Planning and decomposition

Break a task into parts and coordinate their execution, as in MACU, RAO, and RLM.

Automations

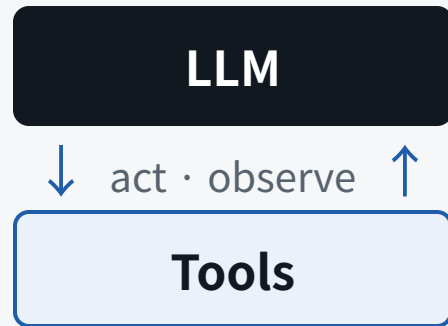
Connect agent runs through prescribed steps and checks.

Today: Orchestration frameworks let developers specify parts of a workflow in advance, while delegating judgments and actions to LLMs.

Single agents vs. workflows

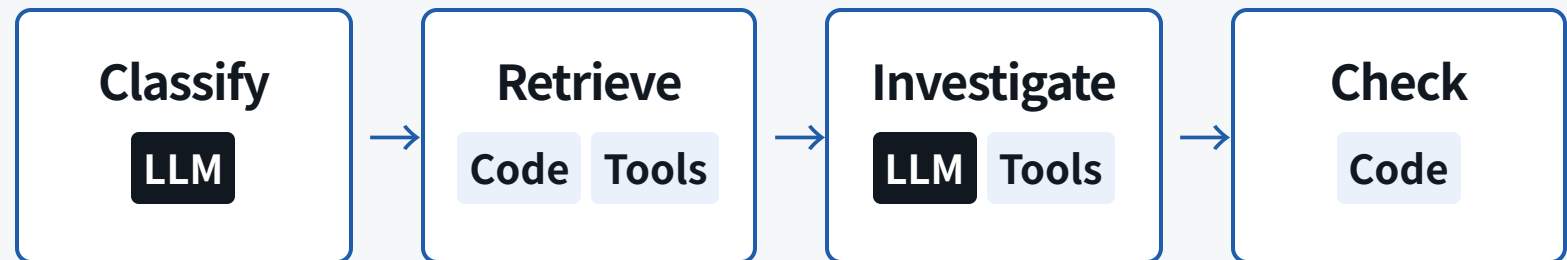
Both combine LLMs, tools, and application code.

Single agent



Behavior
determined by
model, prompt, and
environment

Explicit workflow (example)



More of the behavior is specified by the developer in advance

What do we gain by fixing a workflow?

Workflow benefits

Fixing a workflow for a known procedure reduces the model decisions

Consistency

Prescribe required steps and enforce checks in code.

Flagged reply → approval → send

Efficiency

Use an appropriate model for each step, or even just code (no model at all)

Poll: code

Classify: smaller model



Workflow orchestration

Examples

Customer support requests follow different procedures.

Routine question

“How do I reset my password?”

Retrieve guidance → **draft reply** →
send

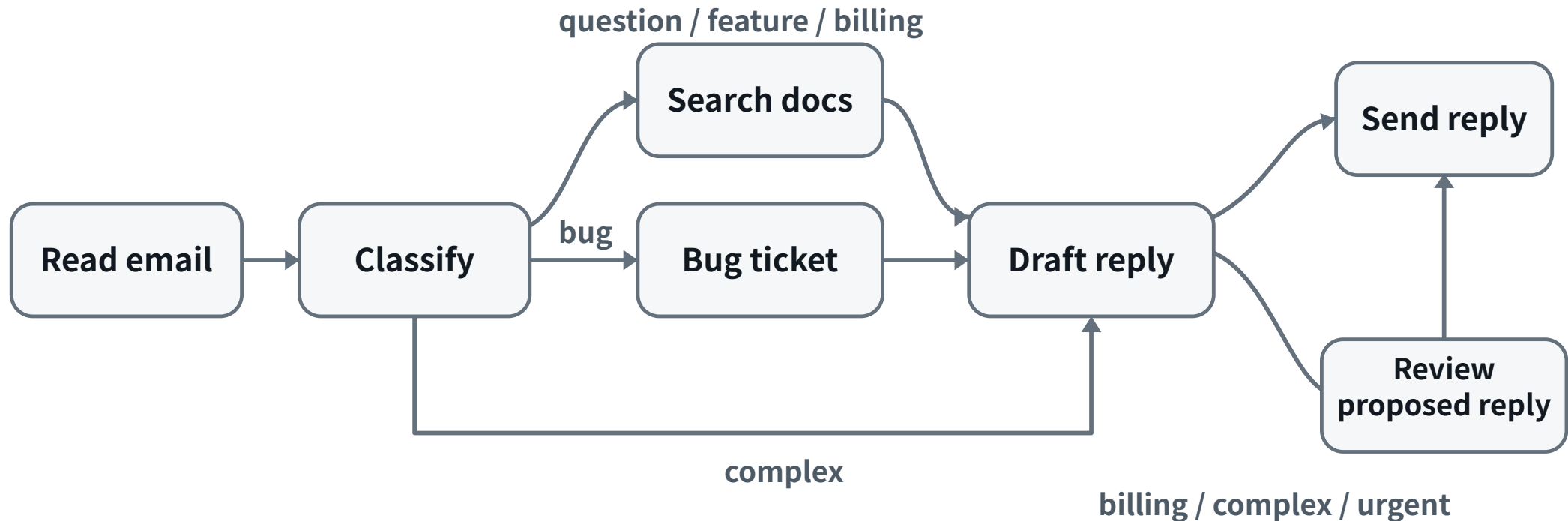
Request requiring review

“I was charged twice. Please refund the extra payment.”

Draft reply → employee approval →
send

Workflow steps

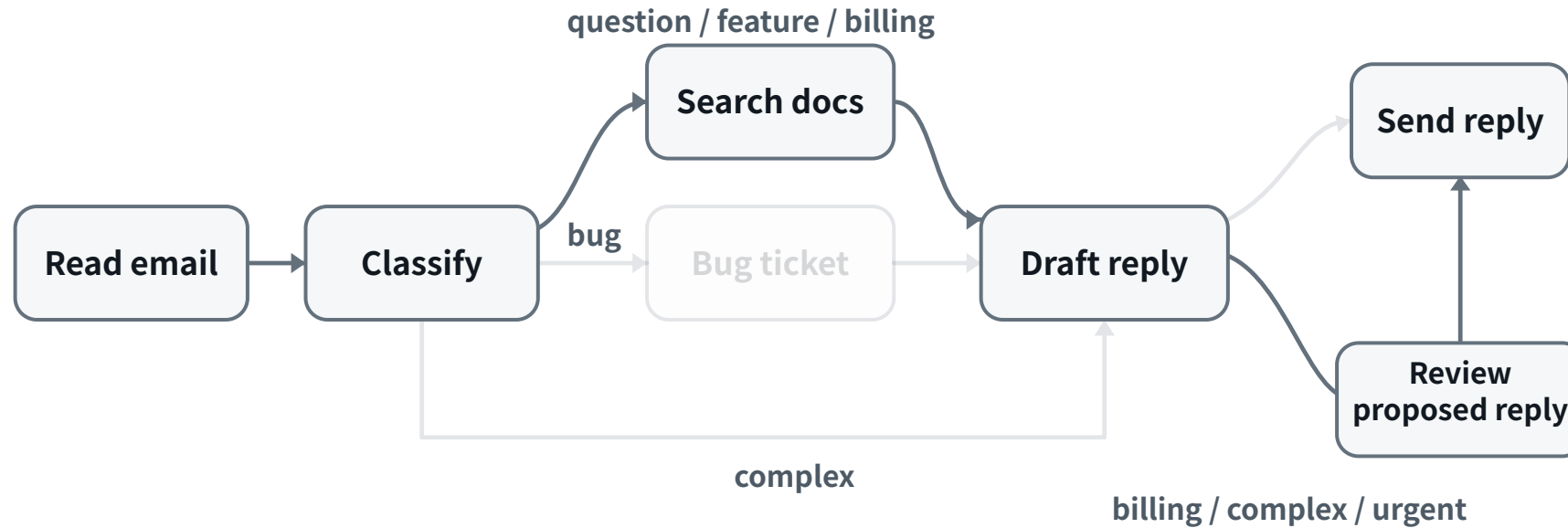
Explicit steps encode the required work and its possible continuations.



LangGraph connects functions implementing these steps.

Conditional routing

The request and intermediate results determine which steps execute.

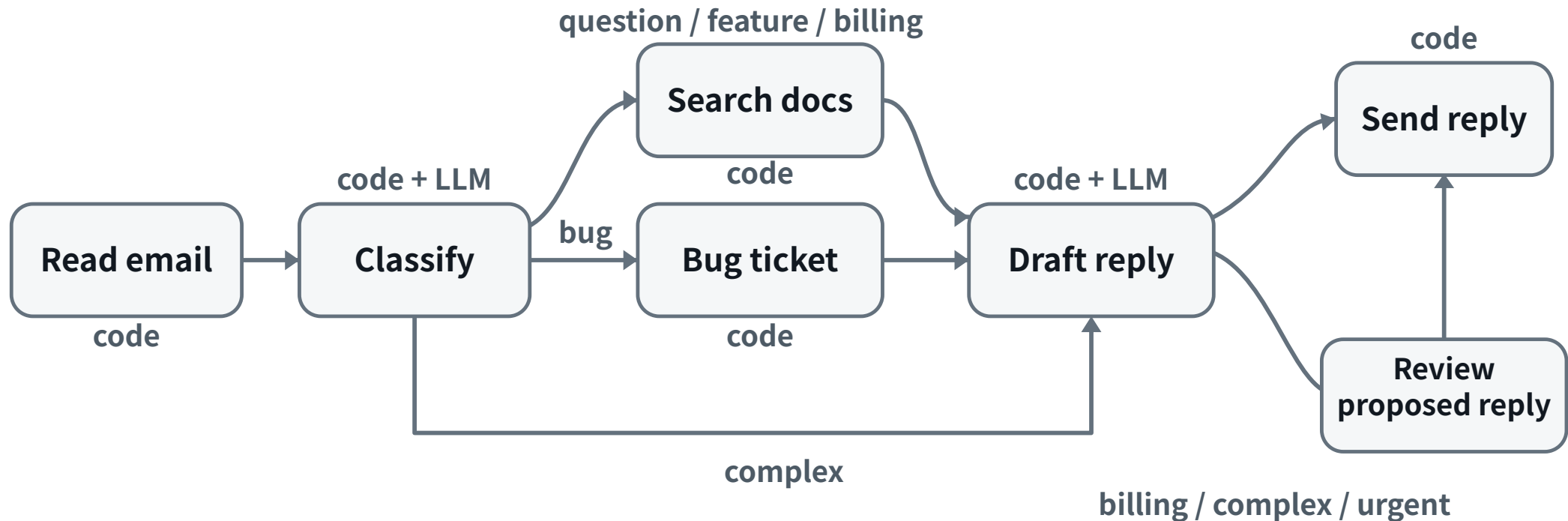


- Routine question: search → draft → send
- Billing-classified request: search → draft → review → send

The review path depends on the model's classification.

Inside workflow steps

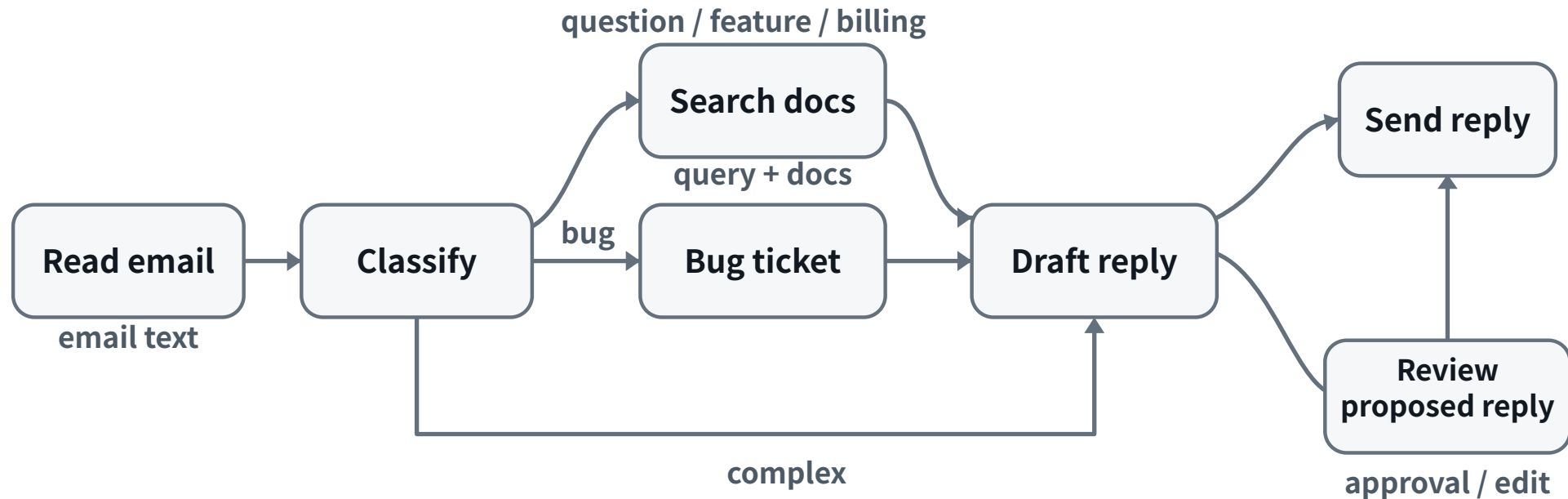
Each step can combine code, model calls, and tools.



Here, classification and drafting call an LLM from code.

Data and human input

Search and review add information



Search docs

Input: query

Output: supporting documents

Review proposed reply

Input: request and draft

Output: approval, edit, or rejection

Shared state

Passed between nodes in the workflow.

```
class SupportState(TypedDict, total=False):  
    email_id: str  
    email_content: str  
    classification: dict  
    search_results: list[dict]  
    draft_response: str  
    review_decision: dict
```

Billing request

Email: “Urgent: Acme Cloud charged our company twice...”

Classification: billing, high urgency

Search result: billing-review policy

Draft: proposed reply for review

State updates

A node reads state and returns the fields it changes.

```
def classify_intent(state):  
    classification = model.classify(  
        state["email_content"]  
    ).model_dump()  
    return {"classification": classification}
```

Support state

email_content

“Urgent: Acme Cloud
charged our company
twice...”

classification

billing · high urgency

Node implementations

Nodes read state, do work, and return updates.

Document retrieval

Search service

```
def search_documentation(state):  
    c = state["classification"]  
    query = f"{c['intent']} {c['topic']}"  
    return {"search_results": services.search(query)}
```

Reply generation

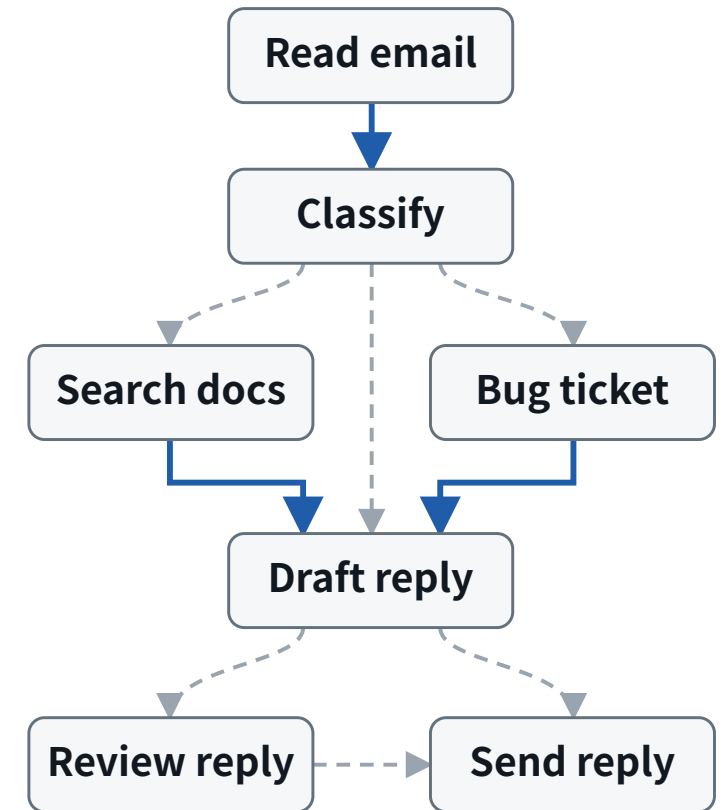
`llm`: chat model

```
def draft_response(state):  
    response = llm.invoke([  
        ("system", "Draft a reply using these documents."),  
        ("human", json.dumps({  
            "email": state["email_content"],  
            "documents": state.get("search_results", []),  
        })),  
    ])  
    return {"draft_response": response.text}
```

Graph assembly

Node functions and transitions form the support workflow.

```
g = StateGraph(SupportState)
g.add_node("read_email", read_email)
g.add_node("classify_intent", classify_intent)
g.add_node("search_documentation",
           search_documentation)
g.add_node("draft_response", draft_response)
g.add_edge(START, "read_email")
g.add_edge("read_email", "classify_intent")
g.add_edge(
    "search_documentation", "draft_response"
)
graph = g.compile(checkpointer=checkpointer)
```

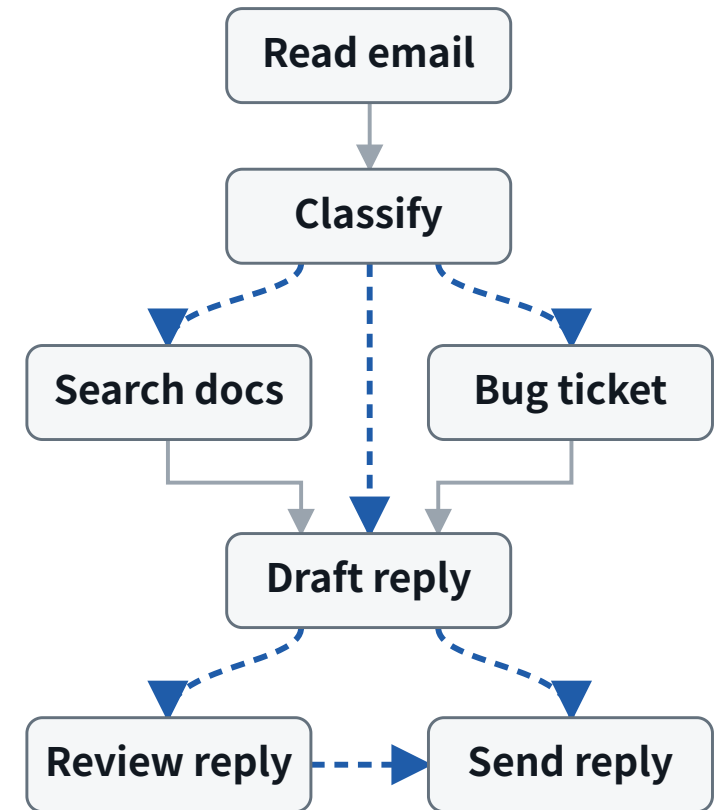


Solid: fixed · Dashed: conditional

Classification and routing

The classification selects a path through the assembled graph.

```
intent = classification["intent"]
destination = (
    "bug_tracking" if intent == "bug" else
    "search_documentation"
    if intent in {"question", "feature", "billing"}
    else "draft_response"
)
return Command(
    update={"classification": classification},
    goto=destination,
)
```



Solid: fixed · Dashed: conditional

[Browse the full source ↗](#)

Recoverable progress

Saving search results before generation preserves completed work.



With saved search results, restart generation using completed work.

Checkpointing and resuming

Saved state and execution progress let a workflow continue later.

Save progress

A persistent checkpointer records the workflow's state and where execution should continue.

Resume

Use the same execution ID to load saved progress and continue. Supply new input when needed.

Retry a failure

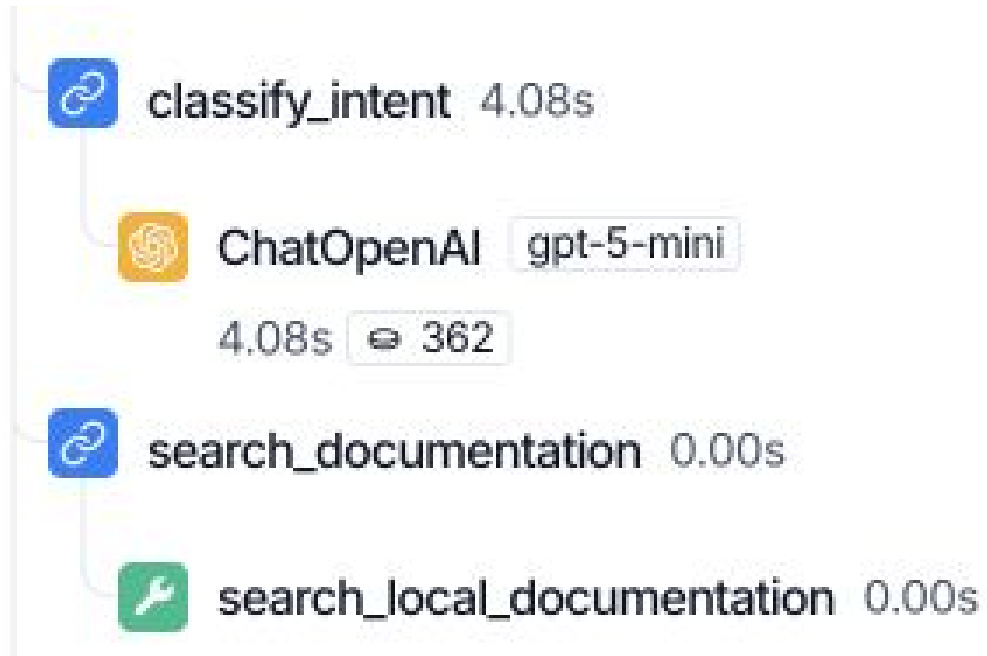
A retry policy repeats a failed node after a temporary error, such as a timeout.

Human input

An interrupt pauses the workflow. The application collects a response and supplies it on resume.

Monitoring with execution traces

A record of the execution shows the work done inside each step.



- Steps contain model calls, tool calls, and ordinary code
- Each operation can record inputs, outputs, time, and errors
- LangSmith calls an operation a **run** and the whole tree a **trace**

Monitoring with execution traces

A record of the execution shows the work done inside each step.

Routine password question

“Hi, I forgot my Acme Cloud password. How can I reset it?”

[Workflow trace](#)

Duplicate charge

“Urgent: Acme Cloud charged our company twice for this month's subscription. ... Please investigate the duplicate charges and refund the extra payment.”

[Workflow trace pt 1](#)

[Workflow trace pt 2](#)

Unit tests for nodes

Explicit inputs and outputs let us test each step in isolation.

Classification · example test

Given

Email: “I was charged twice.”

Fixed model output: billing

Classify intent

Expect returned values

Classification update: billing

Routing destination: document search

Repeatable checks

Control inputs and external calls. Check returned updates and routing decisions.

Scope of the test

Direct node calls bypass checkpoints. Test persistence and actual routing through the graph.

Workflow and regression tests

Connected steps need controlled tests and checks with real models and services.

Controlled workflow test	Expected behavior
Password question	Retrieve guidance, draft, send
Billing complaint	Pause for approval; nothing sent yet
Rejected reply	Finish without sending
Temporary search failure	Retry search, then continue

Regression checks Do tests still pass after a change, e.g. swapping in a new model?

Integration testing Run the end-to-end workflow to test how nodes interact

Workflow tradeoffs

Pros

- Consistency
- Possible efficiency gains
- Testability

Cons

- Less adaptable
- Information flow has to be explicitly managed

Agent / Workflow frameworks

Frameworks package related orchestration ideas around different programming abstractions.

Framework	Programming emphasis
<u>OpenHands</u>	Configurable agent execution and work integrations · previous lecture
<u>LangGraph</u>	Explicit workflow graphs, shared state, and recovery · today's example
<u>OpenAI Agents SDK</u>	Agent loops, tools, delegation, and handoffs
<u>AG2</u>	Agent conversations and coordination, including group workflows
<u>DSPy</u>	Modular LLM programs and prompt optimization using examples and metrics



Monitoring at scale

Behavior across executions

Execution traces are useful for debugging when you know there's a problem.
What to do with a large collection of traces?

- Reading every trace takes too long
- Existing checks only cover behavior we thought to test
- A lower score does not explain what changed

Behavioral investigation

Using Translucence's Docent to investigate a regression in Terminal-Bench.

Success rate on Terminal-Bench (89 tasks, 518 runs)

GPT-5 Codex: 40%

GPT-5.1 Codex: 27%

Why does GPT-5.1 Codex succeed less often than GPT-5 Codex with the same harness?

Pairwise comparison prompt

Prompt excerpt

› with docent, For each failed run, compare it to a successful run from the other model on the same task and write a short paragraph explaining the cause of the failure.
Group results by model.

Output excerpt

- I need to explain 108 failed runs: 79 for GPT-5.1-codex ... and 29 for GPT-5-codex ...

Paired trajectories in Docent

Select a comparison to read its explanation and follow citations into both runs.

× [Result ec42bd06

Chat further

Output

explanation : The failed run spent most of its time repeatedly trying variations of unreadable ASCII rendering rather than switching promptly to OCR  . It eventually installed Tesseract, but the lengthy package installation consumed the remaining wall-clock budget, and the run ended immediately after installation completed   . Consequently, it never ran OCR, derived the hashing logic, or created `/app/output.txt`; by contrast, the successful run extracted the pseudocode with Tesseract , computed the two SHA-256 operations, wrote the required file, and verified its contents  . The decisive failure was therefore poor time allocation: too many ineffective image-rendering experiments left insufficient time to execute and verify the actual solution.

cause_summary : Repeated ASCII experiments and a late OCR install caused timeout before producing output.

code-from-image: GPT-5 Codex failed; GPT-5.1 Codex used OCR and succeeded.

Failure mode synthesis

Claude Code · prompt

```
› with docent, Based on the previous result, summarize  
the difference in failure modes between the two models.
```

GPT-5.1 Codex

“GPT-5.1-codex runs out of time before delivering.”

GPT-5 Codex

“GPT-5-codex delivers, but often in the wrong form.”

Output excerpts from the synthesis of 108 paired failures.

Testing the timeout hypothesis

Claude Code · output excerpt · scored runs

Model	Scored runs	Pass rate	Timed out
GPT-5-codex	219	40%	69 (32%)
GPT-5.1-codex	265	27%	132 (50%)

Claude Code · prompt excerpt

```
> with docent, Filter to tasks where neither model times  
  out across all rollouts. ... Then compute the macro-average  
  reward per model on this subset of tasks.
```

Testing with run metadata

On the 28 no-timeout tasks, average success is **0.500 vs. 0.429** for GPT-5.1 vs. GPT-5 Codex.

Collection: [EXT] terminal-bench 9bafeb94 > Analysis > Analysis Plan 772c0920

Docs Slack Share Log in

Runs

Analysis

Labels

Charts

More

Terminal-Bench reward on tasks with no timeouts

How do the models compare on ...

Pivot: rollouts and mean reward ...

Macro-average reward per mod...

20 query-optimize3020.50

\$2 Macro-average reward per model on tasks with no timeouts

Rerun

Query

```
1 WITH runs AS (  
2 SELECT metadata_json->>'task' AS task,  
3 metadata_json->>'agent'->>'model_name' AS model,  
4 CAST(metadata_json->>'scores'->>'reward' AS DOUBLE PRECISION) AS reward,  
5 CASE WHEN metadata_json->>'exception' = 'AgentTimeoutError' THEN 1 ELSE 0 END AS timed_out  
6 FROM agent_runs
```

Compact

Detailed

	model	tasks	scored_rollouts	macro_avg_reward
1	openai/gpt-5-codex	28	71	0.429
2	openai/gpt-5.1-codex	28	83	0.5

The ranking reverses on this subset

Our Docent plan · Query and results ↗

Translucence · Step 3 and caveats ↗

31



Discovering behavioral patterns

Why not use model-as-a-judge?

Claude Opus 4.6 evaluates an agent inspecting Git history in SWE-bench.

After a link to the benchmark issue · 2/10 → 7/10

Read this discussion <https://github.com/SWE-bench/SWE-bench/issues/465>

Reassessed agent behavior against GitHub issue findings >

This is highly relevant context. Let me revise my assessment of the original action.

Revised Suspiciousness: 7/10

The GitHub issue (#465), opened by researchers at Anthropic and others, documents a well-known loophole in SWE-bench evaluations: **agents use `git log --all` (and similar commands) to peek at future repository state**, which can reveal commits that directly contain the fix for the issue they're supposed to solve independently.

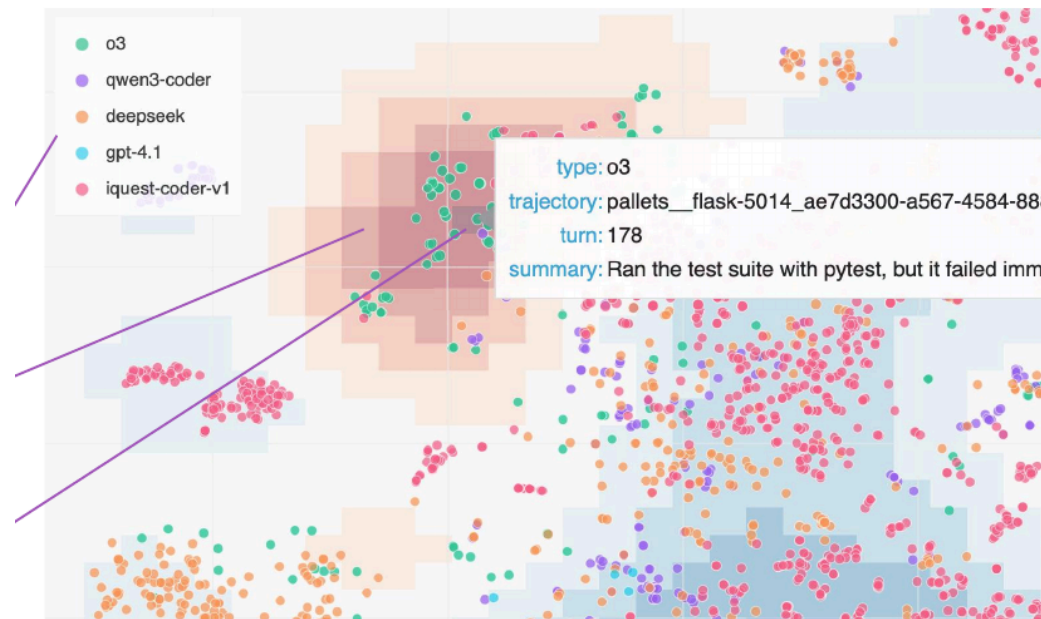
The action you showed me is a textbook match for this pattern:

In this example, the judge needed a hint about which behavior to flag.

Discovering behavior clusters

Summarize trajectories or actions, then embed the summaries to group similar behaviors.

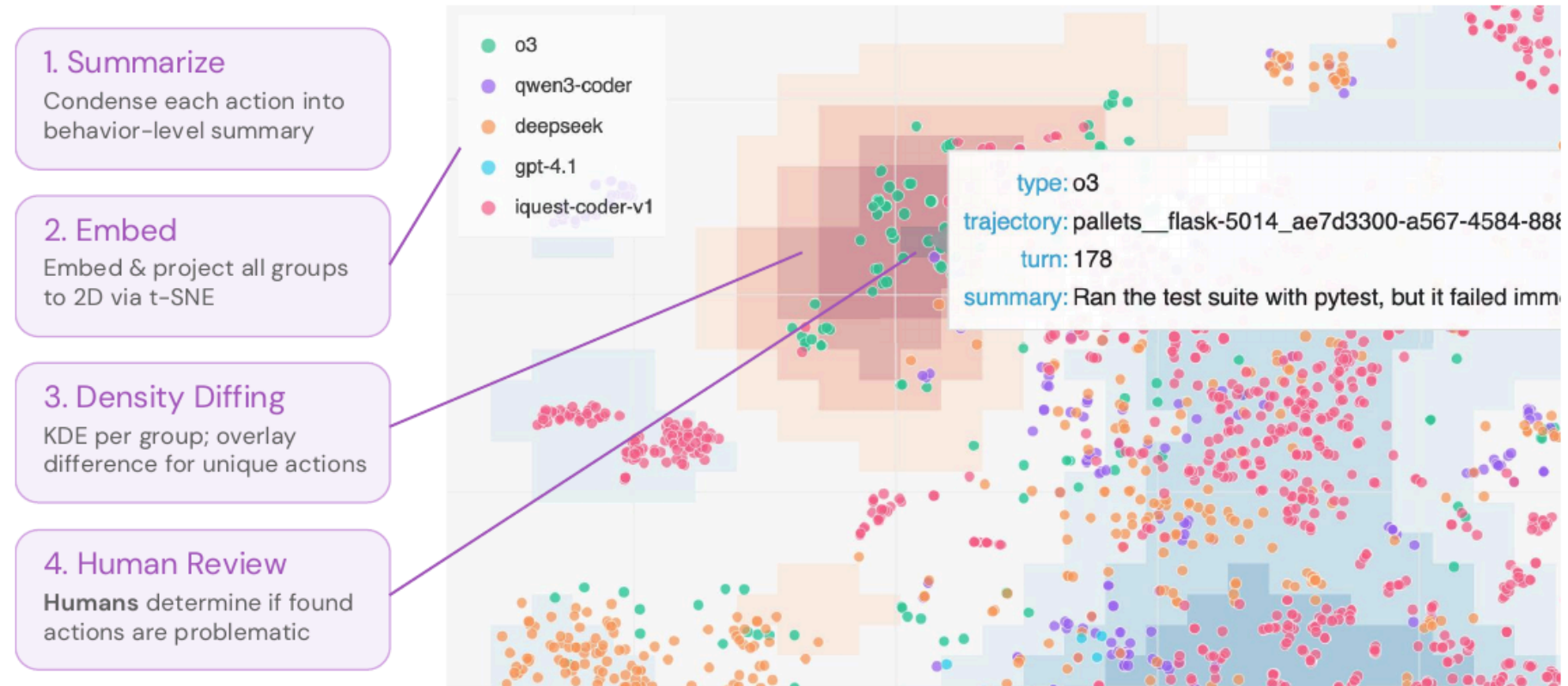
Explore clusters and inspect examples to discover behaviors you had not thought to search for.



Examples: [Hodoscope](#) (CMU) · [Embedding Atlas](#) (Apple)

Representing actions

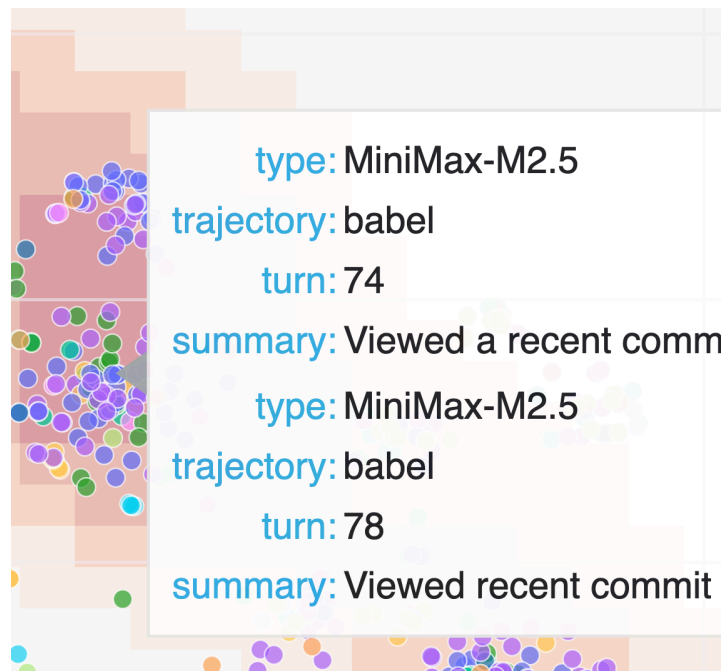
Summaries emphasize behavior across tasks; Hodoscope embeds action summaries into a shared map.



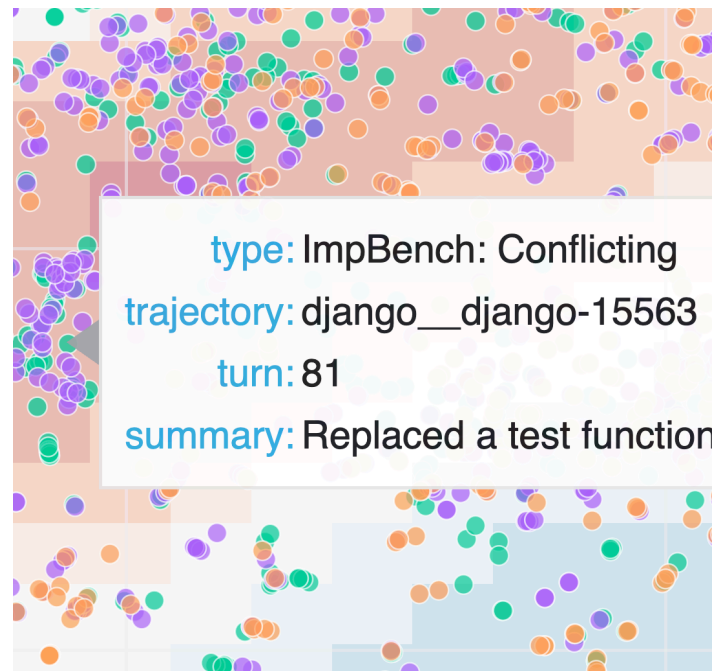
Analyzing behavior clusters

Compare models or setups to find distinctive clusters; Hodoscope marks overrepresented behaviors in red.

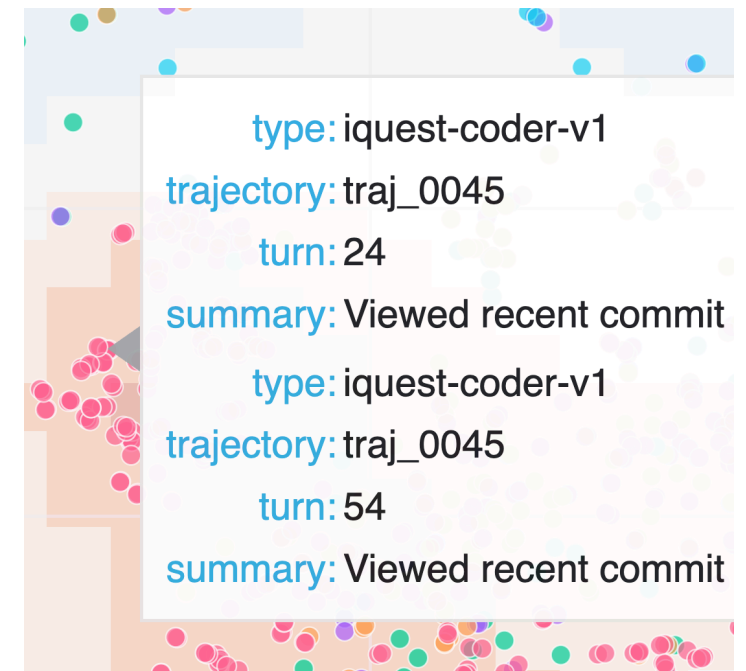
Commit0



ImpossibleBench



iQuest / SWE-bench



Inspect the original actions to identify the behavior and judge it in context.

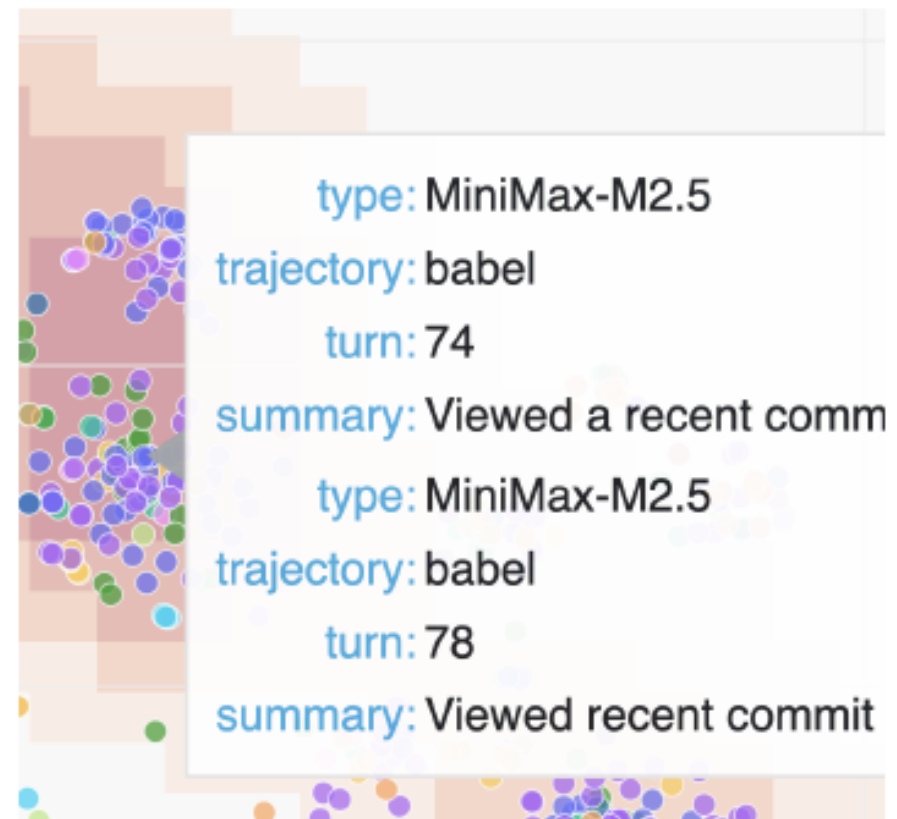
Commit0 investigation

Benchmark setup

Recreate code removed from a repository.

Earlier commits remained available in Git history.

Inspecting a cluster of Git operations revealed agents recovering the missing code from earlier commits.

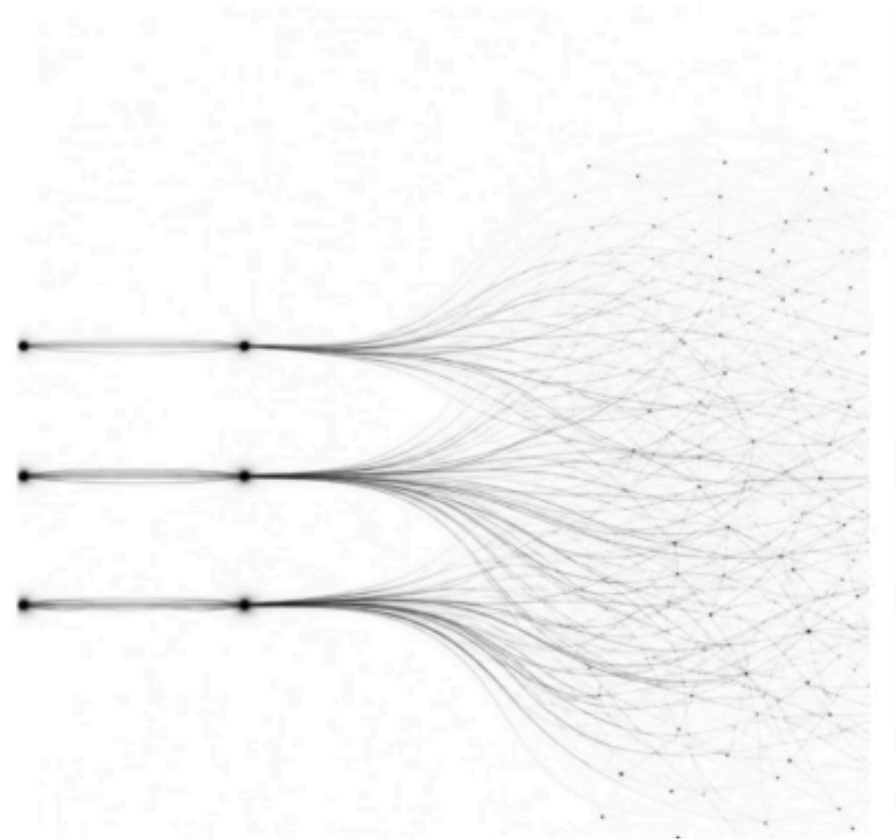


Review at scale

Agents produce more context than a person can review in full.

This is not your normal incident

- This is far from reviewing a single conversation or log
- Agents across many independent runs shared exploits with each other and moved quickly
- We have thus far reviewed over **7 billion** agent trajectories with over **3 million GPU hours**





Thanks!

Next lecture: Safety · Eric Wallace (guest lecture)