Lecture: Multi-Agent Systems and Orchestration

Building a research harness

Boris Bolliet

NASA AI/ML Science & Technology Interest Group Lecture Series

16 November 2026

Contents

1 Building a research harness **2**
1.1 Model calls and state . . . 2
1.2 Structured outputs . . . 3
1.3 Execution and termination . . . 3
1.4 Tracing . . . 4
2 Roles and control flow **4**
2.1 Roles and context . . . 4
2.2 Control in LangGraph . . . 4
2.3 Propose–critique . . . 5
2.4 Self-debugging . . . 6
2.5 Planning and control . . . 6
3 Evaluation **7**
3.1 Test cases and checks . . . 7
3.2 Comparing configurations . . . 7
3.3 Datasets in Langfuse . . . 8
3.4 Experiments in Langfuse . . . 9
4 Existing research harnesses **9**
4.1 CMBAgent . . . 9
4.2 Denario . . . 10
4.3 OpenCode . . . 10
4.4 Skepthical . . . 11
4.5 ClarXivR . . . 11
5 The Pantheon+ experiment **11**
5.1 Reference fit . . . 11
5.2 Comparing organisations . . . 12
5.3 Reading a trace . . . 12
5.4 Checking reports . . . 13
6 Outlook **13**
References **15**



To fit a cosmological model to a catalogue, an agent needs data, somewhere to run code, and a way to check the result. Building the harness that provides these facilities gives us control over:

- **Traceability:** how results link to sources, code and checks;
- **Sovereignty:** where computation runs and which data leave our systems;
- **Customisation:** which domain tools and checks improve the task;
- **Portability:** how models can change without changing the analysis.

We use the Pantheon+ supernova catalogue as a teaching example: it combines data handling and numerical inference with a result we can check independently. The choice is also a nod to Bassett (2024), who used GPT-4's data-analysis agent to revisit the supernova evidence for cosmic acceleration, documenting both successful calculations and fabricated data.

Three notebooks run this task with a single LangGraph agent, CMBAgent and OpenCode with a reviewer. They record calls, time and cost; the CMBAgent and OpenCode runs also have Langfuse traces.

1 Building a research harness

Definition 1.1: Research harness

A **research harness** is the software that supplies a model with information, executes its requested tools, controls the workflow and records and checks the results.

The harness accesses models and services through application programming interfaces (APIs) and decides what information to send to each (Figure 1). For example, a tool can analyse proprietary data locally and return a summary for the model to use.

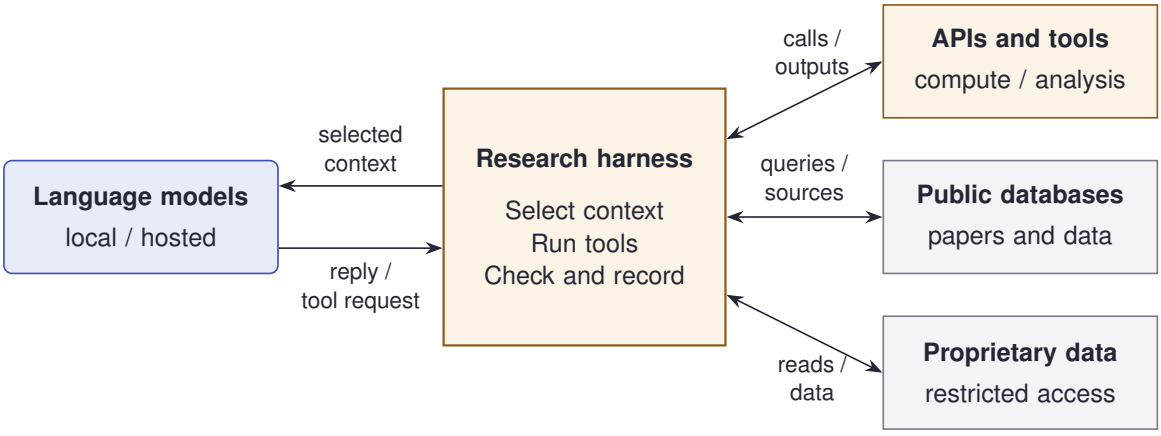

Figure 1: The harness mediates model calls and resource access. Each two-way arrow pairs an allowed operation with its result; data used by a tool need not enter a model request.

1.1 Model calls and state

An **agent** repeatedly calls a language model. At each call, the model receives a **context**: instructions, messages, tool descriptions and any supplied file contents. If it requests a tool, the harness runs that tool and includes the result in the next call.

LangGraph represents this workflow as a graph of Python functions. The functions read and update a shared **state**, which stores the task, messages, code and results. Each function constructs the context for its own model call. For example, a reviewer needs the requirements,

executed code and outputs, while a writer may only need the accepted results. Storing information in the state does not automatically include it in a call.

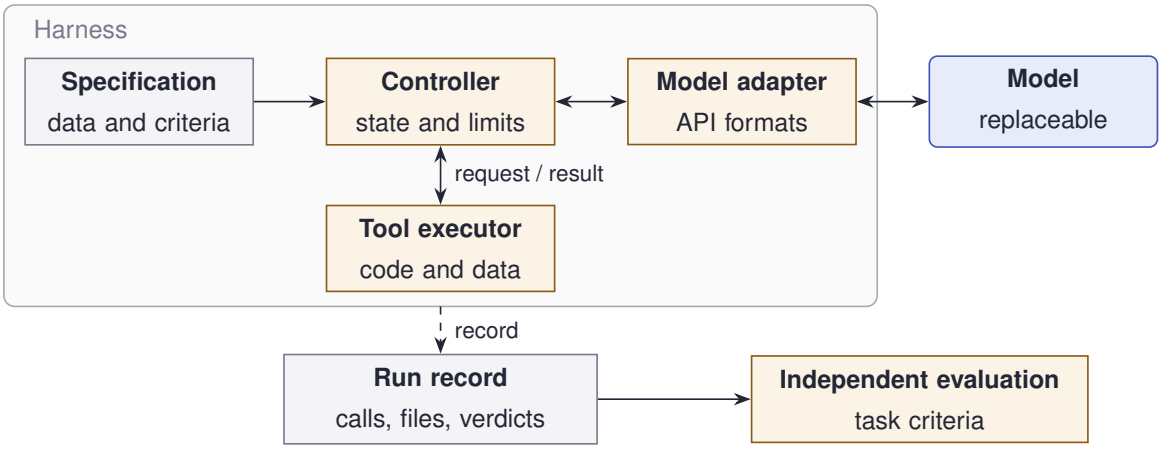

Figure 2: Information flow (solid arrows) and recording (dashed). The adapter separates the model API from the controller and tools. Independent evaluation checks the saved evidence against the task criteria.

An **adapter** translates a provider's messages, tool requests and usage records into the harness's format (Figure 2). We can then change the model without rewriting the scientific tools and checks, although each replacement needs evaluation.

1.2 Structured outputs

The harness often needs to read a model's answer as data. A **structured output** gives this answer a specified form, or schema. Pydantic defines schemas as Python classes and validates the returned values:

```python
from typing import Literal
from pydantic import BaseModel, Field, ValidationError

class Step(BaseModel):
    agent: Literal["engineer", "reviewer", "writer"]
    goal: str = Field(description="What the step produces")
    bullet_points: list[str] = Field(description="Concrete instructions")

class Review(BaseModel):
    approved: bool
    issues: list[str]                    # problems for the engineer to fix
```

The first schema rejects an unknown agent name. The second lets the program read `approved` and pass any `issues` back to the engineer. Validation checks the form of this answer; checking the reviewer's judgement requires a separate test.

1.3 Execution and termination

The **executor** runs the generated code and records what happened: the script, its output, exit status and saved files. Our notebook starts each script in a fresh Python process with a time limit. Variables from one script are therefore unavailable to the next unless they are passed explicitly or saved to a file.

If a script fails, the harness can return the error to the engineer for another attempt. We limit the number of attempts and the run's time and cost. If the attempts are exhausted, the report should record the failure rather than require a numerical answer.

A fresh process can still access the host's files and environment, so isolation also requires restrictions on those resources. The teaching executor removes API keys from its child environment. The versions of CMBAgent and OpenCode used here pass them through.

1.4 Tracing

Definition 1.2: Trace

A **trace** records the operations in a run and how they are nested. In Langfuse, an operation is a **span**, and a model call is a **generation** with inputs, outputs, timing and usage. A **session** groups related traces.

For example, one step may contain an engineer call, a script execution and a review. If the review requests a correction, the trace records a second attempt within the same step. We can then see how much time went into generating code, running it and revising it.

Langfuse's callback for LangChain, the library used here to call models, records graph and model calls:

```
from langfuse.langchain import CallbackHandler

handler = CallbackHandler()        # keys and server from LANGFUSE_* variables
result = graph.invoke(inputs, config={
    "callbacks": [handler],
    "metadata": {"langfuse_session_id": "pantheon-fit"}})
```

Here, `graph` is the compiled workflow and `inputs` is its initial state. Labelling calls by agent lets us add up usage for each role. Models measure text in **tokens**; the input and output token counts contribute to the cost.

To reconstruct the analysis, we also save the code, data versions, outputs and review decisions under the same run identifiers. Comparing the trace with the executor's records reveals missing events. The trace contains what the provider exposes, which may include a reasoning-token count without the reasoning text. Its prompts and outputs are sent to the configured Langfuse server, so that server must be included in the data-access policy.

2 Roles and control flow

2.1 Roles and context

A **multi-agent system** divides a task between agents with separate instructions, tools and contexts. They can use the same model: their roles differ because they receive different information and requests. The **orchestration** determines which agent runs next.

Managing context means selecting the requirements, sources and outputs relevant to each step. Passing the full history increases cost and can obscure evidence. When summarising earlier work, we should retain essential decisions, assumptions and links to the original outputs. Recording what each call received helps us diagnose failures.

2.2 Control in LangGraph

A fixed edge in LangGraph runs the next node unconditionally. A conditional edge chooses a node by reading the state. Consider the tool loop in Figure 3a. After a model call, a function checks whether the reply requests a tool. If so, it routes to the tool node; otherwise, the run finishes. The tool node always returns to the model.

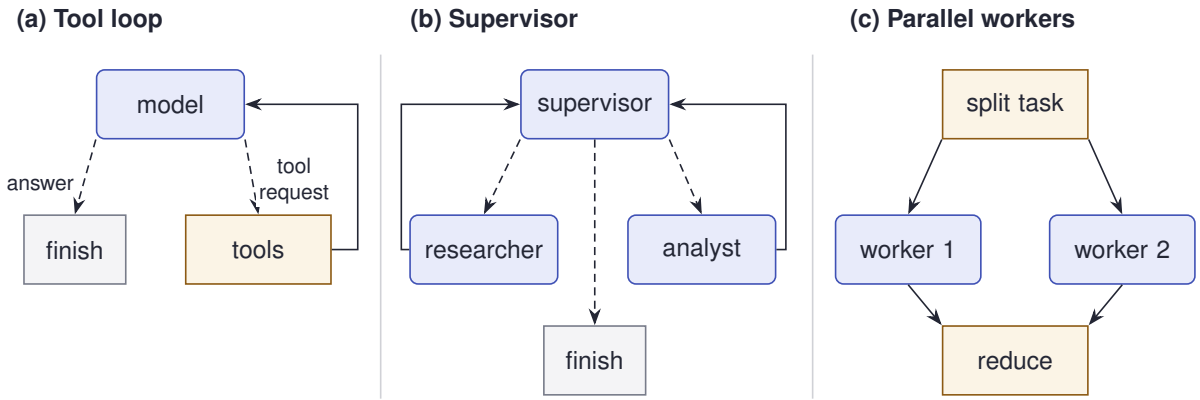

Figure 3: Three workflows. Rounded boxes call models. Solid arrows are fixed paths; dashed arrows are alternatives. (a) A tool result returns to the model. (b) A supervisor selects one specialist, which returns control. (c) All workers run before the reduce step; two are shown.

A **supervisor** instead asks a model to choose the next specialist (Figure 3b). It writes its choice into the state, and a conditional edge routes to the researcher, analyst or finish node. The routing function is ordinary code, but the choice it reads came from a model.

A fixed sequence of stages forms a **pipeline**. If work can be done independently, a **fan-out** sends it to concurrent workers and a **reduce** step combines their results (Figure 3c). LangGraph's `Send` supplies a separate input to each worker; a **reducer** specifies how their state updates combine, for example by appending results to a list.

2.3 Propose–critique

A **propose–critique loop** alternates a proposal with a review. The proposer uses the review to revise its output. Figure 4 shows this process for a research idea. Formatter calls convert the prose into structured objects. After N critiques, a final revision and formatting pass produces the returned idea.

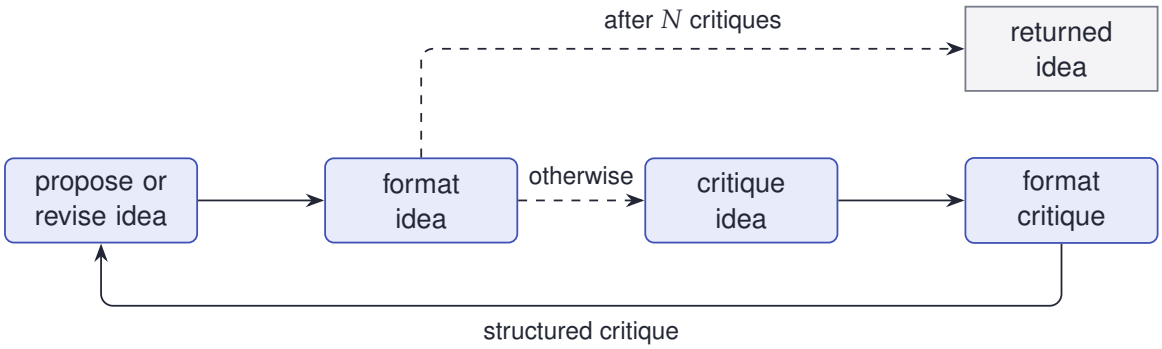

Figure 4: Proposal and critique. The controller branches after formatting the idea. Each critique feeds a revision; the final output is a formatted idea.

2.4 Self-debugging

Self-debugging uses execution results to guide revisions to code. An engineer writes a script, the harness runs it, and the outcome is checked. Figure 5 uses two evaluators: one checks whether the script ran, and the other whether it fulfilled the step's instructions. Either can request a correction, up to a shared attempt limit.

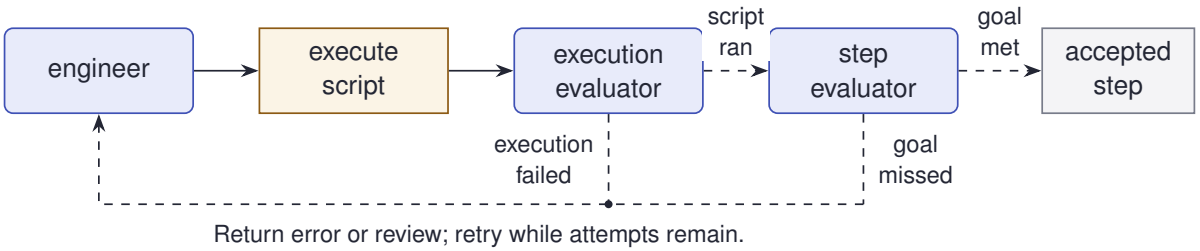

Figure 5: Self-debugging as used in CMBAgent. Both evaluators can request corrections from the engineer. Exhausted attempts leave the step unfulfilled.

2.5 Planning and control

Planning and control combines these loops to carry out a larger task (Figure 6). A planner and reviewer develop an ordered list of steps, each with a goal, agent and instructions. Planning ends with a revised plan that incorporates the final review.

A controller then dispatches the steps and records their outcomes. An engineer step can use the self-debugging loop; a researcher step can read results and write a report. A review may revise the remaining plan before the next step runs. Completed results stay in the run record.

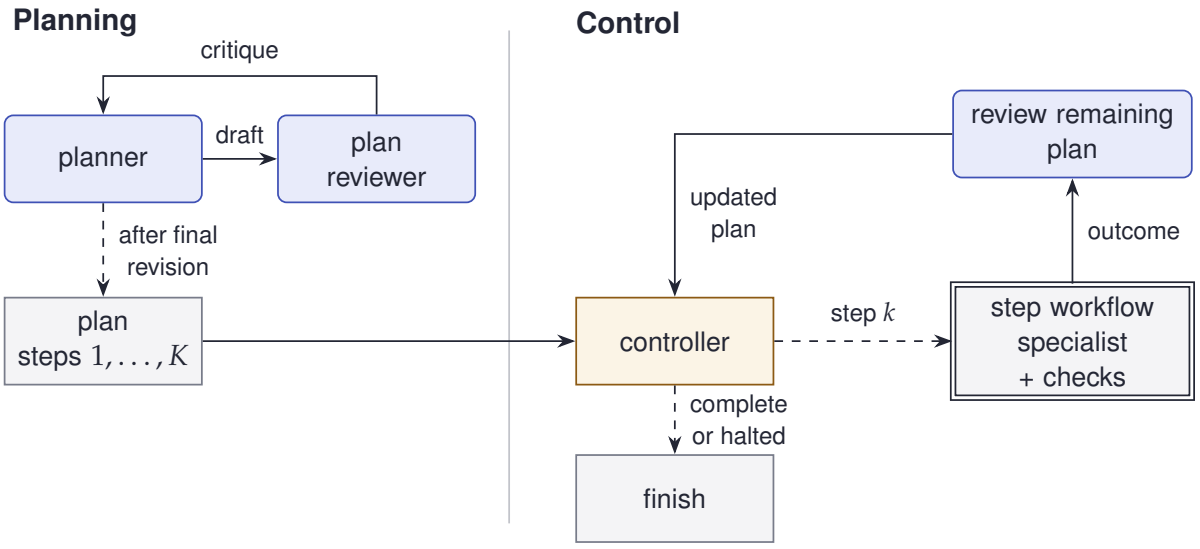

Figure 6: Planning produces the steps; control dispatches them and uses their outcomes to continue, replan or stop. The double-bordered box contains a specialist workflow with its checks and retries. Formatter calls are omitted.

3 Evaluation

3.1 Test cases and checks

To evaluate a harness, we need tasks with results that we can check. An independent implementation can provide a reference for a known task; simulated data and limiting cases can help with new research. We check both the result and how it was obtained. A correct number can accompany a wrong method or an inaccurate report.

Fault injection tests a checker with a deliberately wrong script or report, such as the covariance error in Table 1. The expected answer must be established independently of the reviewer being tested.

Test case	Expected harness behaviour
Reference fit	Return the specified analysis and a result within a predeclared tolerance.
Wrong covariance	Reject a script using diagonal errors when full covariance is required.
Missing input	Record an unresolved step; do not manufacture a numerical result.
Unsupported claim	Reject or flag a report value absent from the executed outputs.
Interrupted execution	Preserve the failure and completed artefacts; do not label the run successful.

Table 1: Proposed evaluation cases. The saved runs illustrate failures but do not implement this full test suite.

3.2 Comparing configurations

When comparing models, we keep the tasks, tools, checks and budgets fixed and repeat each run. Final evaluation uses cases held apart from prompt and checker development.

We measure failures, missing trace events, time and cost, counting failed attempts in the cost of obtaining a correct result. The experiments in Section 5 compare harness configurations using one model; they do not measure performance across models.

We can compare task performance against cost (Figure 7). A configuration is **dominated** if another costs no more and performs at least as well, with a strict improvement in one of the two. The remaining configurations form the **Pareto frontier**. Evaluation determines where a particular design lies; adding agents or using a larger model need not improve the trade-off.

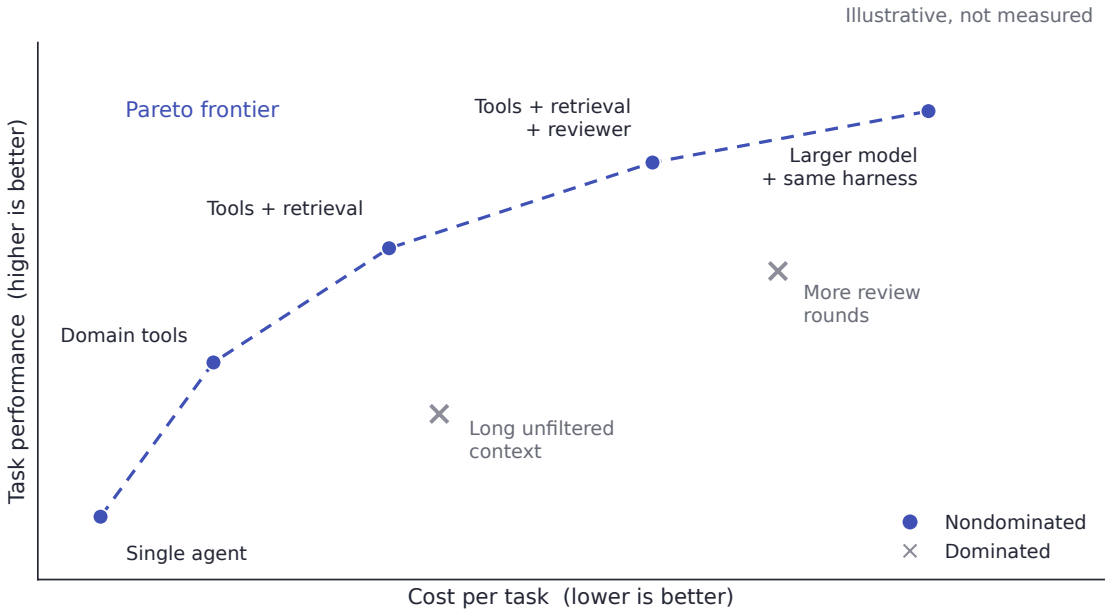

Figure 7: Hypothetical configurations, labelled by their distinguishing design choices. Positions are illustrative, not measured. Performance means success on the chosen task criteria. The dashed line joins nondominated choices as a guide to the eye.

3.3 Datasets in Langfuse

A Langfuse **dataset** stores evaluation cases: each item has an input, an optional expected output and metadata. In a Langfuse project, open `Datasets`, create `pantheon-checks` and add items manually or from a CSV file. For the reference fit of Section 5.1, the input contains the analysis requirements and data locations; the expected output contains `omega_m` $= 0.3316$ and `sigma` $= 0.0182$.

For Python access, install `langfuse` and set `LANGFUSE_PUBLIC_KEY`, `LANGFUSE_SECRET_KEY` and `LANGFUSE_BASE_URL` from the project's connection settings. The SDK also provides `create_dataset` and `create_dataset_item` for adding cases from code.

3.4 Experiments in Langfuse

An **experiment** runs the harness on these inputs and scores its outputs. Here, `run_harness` is our own adapter, which executes an analysis and returns a dictionary with the fitted `omega_m` and `sigma`. The harness receives only the input; the evaluator receives the reference values.

```
from langfuse import Evaluation, get_client

langfuse = get_client()
dataset = langfuse.get_dataset("pantheon-checks")

def task(*, item, **kwargs):
    return run_harness(item.input)

def fit_match(*, output, expected_output, **kwargs):
    try:
        passed = all(abs(output[k] - expected_output[k]) <= 0.005
                     for k in ("omega_m", "sigma"))
    except (KeyError, TypeError):
        passed = False
    return Evaluation(name="fit_match", value=int(passed))

result = dataset.run_experiment(
    name="reference-fit-v1", task=task,
    evaluators=[fit_match], max_concurrency=1)
langfuse.flush()
```

This evaluator checks numerical agreement; method checks and the failure cases in Table 1 need their own evaluators. The runner traces each case. Keeping the callback of Section 1.4 inside the harness also records its individual model calls and usage.

For comparisons, retrieve the same dataset version with the `version` timestamp argument to `get_dataset`, and record the model, code and evaluator versions in the experiment's `metadata`. In `Experiments`, select runs to compare scores, cost and latency, then open individual traces to investigate failures. The dataset and experiment documentation gives the full SDK interfaces.

4 Existing research harnesses

4.1 CMBAgent

CMBAgent[1] was first introduced by Laverick et al. (2024) for cosmological parameter analysis. Xu et al. (2025) extended it to planning and control without a human in the loop, and to quantitative tasks beyond cosmology.

CMBAgent subsequently won a first-place award in Phase 1 of the NeurIPS 2025 FAIR Universe Weak Lensing Uncertainty Challenge. Borrett et al. (2026) describe the winning workflow, which combined autonomous exploration with human intervention.

The current implementation uses LangGraph for the planning and control flow of Figure 6. Its controller passes saved files and summaries between steps. Alongside the model-based checks in the self-debugging loop, ordinary code checks script syntax and expected output files. A claim auditor checks reported numbers against the run's files and can request revision.

[1] https://github.com/CMBAgents/cmbagent

4.2 Denario

Denario[2] runs full research cycles: generating ideas, searching the literature, developing methods, analysing data and writing papers. Its analysis module uses CMBAgent. The modules can also run separately, with intermediate files available for inspection and revision. Villaescusa-Navarro et al. (2026) describe the architecture and present scientific applications with evaluations by domain experts.

At fleet scale, many research cycles can explore a common problem or dataset. Their ideas, methods, results, evaluations and papers form a **knowledge graph**. Its links record successive iterations of a project and similarities between different lines of inquiry. Figure 8 shows an ecology example; the interactive timeline lets us follow its growth and inspect individual nodes. The video Fleet Knowledge Graph and Parallel Science gives an overview of Denario deployed at fleet scale.

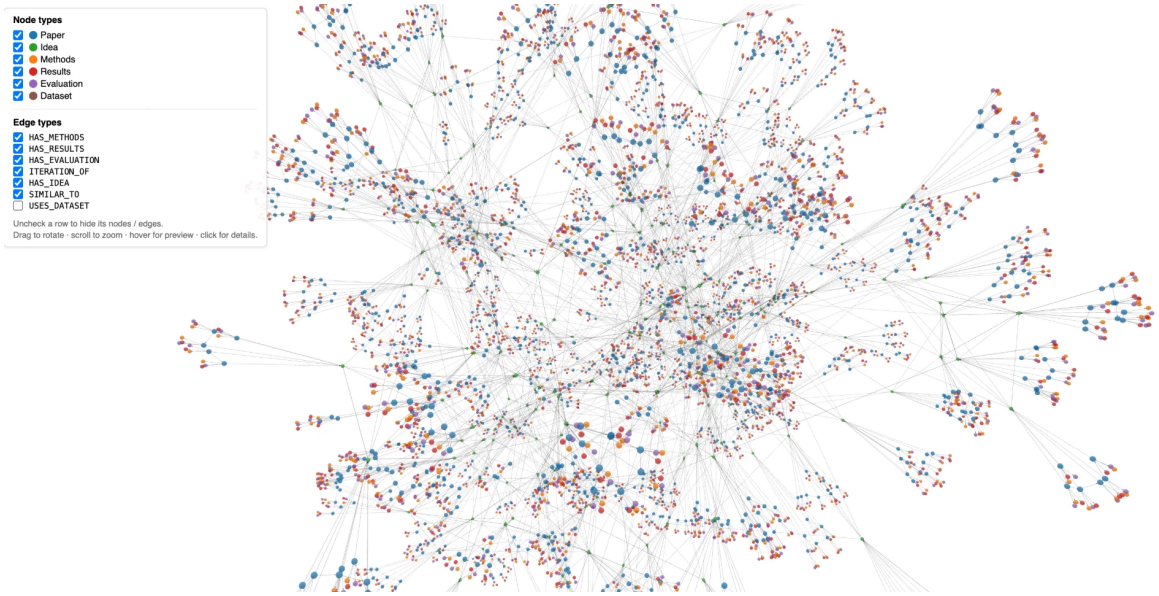

Figure 8: An ecology knowledge graph built from a fleet of Denario research cycles: 5,871 nodes in this snapshot of 26 June 2026. Colours distinguish papers (blue), ideas (green), methods (orange), results (red), evaluations (purple) and datasets (brown). Edges connect these records within projects, across iterations and between similar ideas.

4.3 OpenCode

OpenCode is an open-source coding harness. Its primary agent can read and edit files, run commands and delegate work to subagents. Each subagent starts with the delegation prompt and returns its result. The model chooses these actions through tool calls. Session exports record calls, tools, tokens and costs, including the subagents' work.

OpenCode was used for the independent reproduction and reanalysis of Atacama Cosmology Telescope Data Release 6 (ACT DR6).[3] The workflow ran locally, using Qwen and DeepSeek models served on a single NVIDIA RTX 6000 GPU.

[2] https://github.com/Denario-AI/Denario
[3] ACT DR6 replication: Licong Xu.

4.4 Skepthical

Skepthical is a harness for reviewing scientific papers. Four modules examine general issues, mathematics, numerical claims and citations in parallel, then combine their findings into a review. The checks use tools suited to each claim: computer algebra for equations, executed code for numerical consistency, and retrieved papers for claims supported by citations.

Zubeldia et al. (2026) evaluate twelve reviewers on 30 astrophysics and machine-learning papers with 357 deliberately inserted errors. Unmodified copies provide a separate check on false positives. This applies the fault-injection approach of Section 3 to the reviewers themselves.

On the astrophysics corpus, enforced citation retrieval gives Skepthical the highest citation-error detection rate among the tested systems. This advantage is not established on machine learning, where its result is statistically indistinguishable from the best single-model reviewer.

4.5 ClarXivR

CLArXivR (Contextual Language model Assisted arXiv Retrieval; work led by G. Farren) recommends new papers using a reader's interests, saved library and relevance feedback. Its reports summarise selected papers and explain how they relate to specific work in that library.

The pipeline retrieves arXiv listings from a local metadata mirror and ranks titles and abstracts against the library. A further relevance check narrows the selection before full papers and figures are fetched. Models then summarise the findings and compare them with the matched library papers. Each stage saves its results and records model costs.

A separate evaluation suite replays fixed sets of papers through the pipeline. It checks whether summaries are supported by their sources, cover the main findings and preserve numerical values. Comparisons are checked for justified connections and supporting citations. Deliberately corrupted passages test the graders themselves. Prompt changes are evaluated on papers held apart from development, with prompt versions, runs and scores retained.

5 The Pantheon+ experiment

5.1 Reference fit

Type Ia supernovae let us measure the relation between distance and redshift. We fit their standardised magnitudes using a spatially flat universe containing matter and a cosmological constant (flat ΛCDM). The parameter Ω_{m} is the present matter density divided by the critical density.

The file `Pantheon+SH0ES.dat` contains light curves, including repeated observations of some supernovae. Its `zHD` column gives redshifts corrected to the cosmic microwave background rest frame and for peculiar velocities; `zHEL` gives heliocentric redshifts. We select $z_{\mathrm{HD}} > 0.01$ to reduce the influence of peculiar velocities. This leaves 1590 light curves of 1473 supernovae, with standardised magnitudes in `m_b_corr`.

For speed of light c and present expansion rate H_0, the comoving distance is

$$D_{\mathrm{C}}(z) = \frac{c}{H_0}\int_0^z \frac{\mathrm{d}z'}{E(z')}, \qquad E(z) = \sqrt{\Omega_{\mathrm{m}}(1+z)^3 + 1 - \Omega_{\mathrm{m}}},$$

where $E(z)$ is the expansion rate relative to H_0. The luminosity distance of light curve i is $d_{L,i} = (1+z_{\mathrm{HEL},i})D_{\mathrm{C}}(z_{\mathrm{HD},i})$, giving the model apparent magnitude

$$m_i = 5\log_{10}\left(\frac{d_{L,i}}{\mathrm{Mpc}}\right) + 25 + M.$$

We fix $H_0 = 70\,\mathrm{km\,s^{-1}\,Mpc^{-1}}$. Its effect is absorbed by the fitted absolute magnitude M, leaving Ω_m unchanged if H_0 changes.

Let Δ_i be the observed magnitude minus $5\log_{10}(d_{L,i}/\mathrm{Mpc}) + 25$, $\mathbf{1}$ a vector of ones, and C the covariance block for the selected rows. We minimise

$$\chi^2(\Omega_\mathrm{m}, M) = (\boldsymbol{\Delta} - M\mathbf{1})^\mathsf{T}\,\mathsf{C}^{-1}(\boldsymbol{\Delta} - M\mathbf{1}).$$

For each value of Ω_m, we minimise analytically over M. The resulting curve is the profile χ^2. Under the Gaussian-likelihood approximation, values within $\Delta\chi^2 = 1$ of its minimum give an approximate 68% confidence interval. We quote half its width as the uncertainty.

Using the full statistical and systematic covariance from `Pantheon+SH0ES_STAT+SYS.cov`, the independent calculation gives

$$\Omega_\mathrm{m} = 0.3316 \pm 0.0182, \qquad \chi^2_\mathrm{min} = 1402.9.$$

These values provide the reference for grading; they are not supplied to the agents.

5.2 Comparing organisations

We give each **organisation**, or arrangement of agents, the same data files and a fully specified analysis: the selection, model, covariance and fitting procedure above. We test how reliably each implements it. A numerical match requires both the estimate and its uncertainty to lie within 0.005 of the reference; method checks are separate.

All roles use `gemini-3.1-flash-lite`. The single agent uses the API's default temperature and **thinking level**, a setting that controls reasoning before the reply. In CMBAgent, the engineer uses high thinking. The OpenCode prompt explicitly requests a reviewer subagent; these runs do not test whether the model would choose to delegate. Table 2 compares the three organisations.

Organisation	Runs	Matches	Calls	Time (s)	Cost (US$)
LangGraph single agent	10	8	3–10	7.3–25.8	0.003–0.022
CMBAgent	2	2	30–31	558–588	0.426–0.440
OpenCode, build and reviewer	2	2	17–23	28–50	0.025–0.045

Table 2: Runs saved in the three notebooks on 16 November 2026; calls, time and cost are ranges. Matches use the numerical criterion above. Costs use list prices for the single agent, Langfuse estimates for CMBAgent, and OpenCode's catalogue including cache discounts. OpenCode times cover sessions; the other times cover recorded runs.

OpenCode with a reviewer cost US$0.025–0.045 per run, against US$0.003–0.022 for the single agent and US$0.426–0.440 for CMBAgent. Most of OpenCode's input benefited from cache discounts, while CMBAgent's engineer spent heavily on reasoning (Section 5.3). The sample sizes and reasoning settings differ, so these results compare the tested configurations rather than establish a general ranking.

5.3 Reading a trace

The CMBAgent notebook records two runs in Langfuse, with three attempts allowed per step. We set the adaptive reviewer and replanner to the same model as the other roles. Langfuse records 30 and 31 generations, matching the local call counts. The runs take 558 and 588 s, respectively.

In run 1, four engineer calls produce 255.7 thousand of the run's 270.8 thousand output tokens. Of the engineer's output, 251.7 thousand tokens are reasoning, billed even though their text was not returned (Figure 9). Counting calls alone would miss this cost.

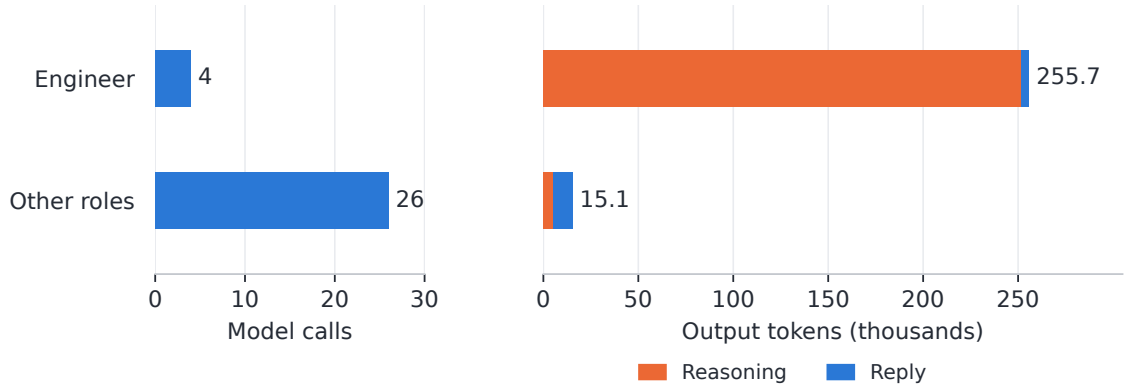

Figure 9: CMBAgent run 1: calls and output tokens, grouped by role from the notebook's callback record. Reasoning tokens are included in output. The four engineer calls account for 94% of output tokens; the remaining 26 calls account for 6%.

Both runs make four engineer calls, including two retries. In run 1, both retries belong to the first step; in run 2, each of the two engineer steps needs a second attempt. The call sequence and step outcomes show where the extra work occurs. To test a lower thinking setting, we would measure both cost and failures over repeated runs.

OpenCode's Langfuse plugin records 23 and 17 generations, matching its session exports. Its trace costs also agree with the exported costs. The saved single-agent runs have local usage records but no Langfuse traces.

5.4 Checking reports

The notebooks grade different outputs: the single agent's final answer, OpenCode's `report.md`, and CMBAgent's last `RESULT` line in a successful script log. The two unmatched single-agent runs have no parseable final result. A CMBAgent match checks the script's numbers, not the researcher's report.

The tolerance also matters. CMBAgent run 2 returns $\Omega_m = 0.32857 \pm 0.01844$, which passes the 0.005 criterion without reproducing the reference minimum as closely as run 1. The numerical grade does not check the redshift selection, covariance or wording of the method. Those claims need separate checks against the executed code and outputs.

6 Outlook

Research harnesses can now sustain work over days and produce substantial scientific results. An independent reanalysis of Atacama Cosmology Telescope Data Release 6 (ACT DR6)[4] reports reproducing released power spectra from sky maps, testing systematics and carrying the analysis through to cosmological inference. In mathematics, OpenAI reports a solution to the Navier–Stokes Millennium Prize problem, obtained through several days of coordinated agent work and accompanied by a formal proof in Lean. These results suggest substantial scope for discovery beyond routine analysis.

Such work needs infrastructure for storing, checking and comparing its outputs. Parallel Science provides a repository for papers and reviews. Linking these to the tasks, code, traces

[4] ACT DR6 replication: Licong Xu.

and evaluations would make past trials available for improving future runs. Denario v2, currently work in progress, aims to use this accumulated experience to adapt the harness itself: its workflows, tools and checks. Each proposed change can then be evaluated on further tasks.

The growing volume of research also creates a problem for its readers. Tools such as CLArXivR, currently designed for human users, help find and process papers and prepare summaries.

The ultimate design problem is an **impact-aware** research system: one that chooses questions for their expected contribution to scientific knowledge. This calls for a **science of science**, which studies how research develops and how discoveries spread (Fortunato et al., 2018). The animation of the astronomy citation network[5] shows how papers connect as the literature grows. These connections help us study scientific influence; choosing valuable research also requires judging novelty, reliability and what other work a result makes possible.

[5] Animation by Licong Xu.

References

Bruce A. Bassett. Integrals and integrity: Generative AI tries to learn cosmology. *MIT Case Studies in Social and Ethical Responsibilities of Computing*, February 2024.

Dan Scolnic, Dillon Brout, Anthony Carr, Adam G. Riess, Tamara M. Davis, et al. The Pantheon+ analysis: The full data set and light-curve release. *The Astrophysical Journal*, 938:113, 2022. arXiv:2112.03863.

LangChain. LangGraph documentation: Graph API (state, reducers, conditional edges, `Send`). https://docs.langchain.com/oss/python/langgraph/graph-api. accessed 15 November 2026.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023. arXiv:2210.03629.

Pydantic. Pydantic documentation: Models and validation. https://docs.pydantic.dev/latest/concepts/models/. accessed 15 November 2026.

LangChain. LangChain documentation: Structured output. https://docs.langchain.com/oss/python/langchain/structured-output. accessed 15 November 2026.

Langfuse. Langfuse documentation: Observations, traces and sessions. https://langfuse.com/docs/observability/data-model. accessed 15 November 2026.

Langfuse. Langfuse documentation: LangChain and LangGraph integration. https://langfuse.com/integrations/frameworks/langchain. accessed 15 November 2026.

Langfuse. Langfuse documentation: Self-hosting. https://langfuse.com/self-hosting. accessed 15 November 2026.

Langfuse. Langfuse documentation: Datasets. https://langfuse.com/docs/evaluation/experiments/datasets. accessed 16 November 2026.

Langfuse. Langfuse documentation: Experiments via SDK. https://langfuse.com/docs/evaluation/experiments/experiments-via-sdk. accessed 16 November 2026.

Langfuse. Langfuse documentation: Compare experiments. https://langfuse.com/docs/evaluation/experiments/compare-experiments. accessed 16 November 2026.

Erik Schluntz and Barry Zhang. Building effective agents. Anthropic Engineering, https://www.anthropic.com/engineering/building-effective-agents, 2024. 19 December 2024.

Dillon Brout, Dan Scolnic, Brodie Popovic, Adam G. Riess, Joe Zuntz, et al. The Pantheon+ analysis: Cosmological constraints. *The Astrophysical Journal*, 938:110, 2022. arXiv:2202.04077.

Pantheon+ team. Pantheon+ data release: distances and covariance matrices, `README`. GitHub, https://github.com/PantheonPlusSH0ES/DataRelease/blob/main/Pantheon%2B_Data/4_DISTANCES_AND_COVAR/README. accessed 15 November 2026.

Andrew Laverick, Kristen Surrao, Inigo Zubeldia, Boris Bolliet, Miles Cranmer, Antony Lewis, Blake Sherwin, and Julien Lesgourgues. Multi-agent system for cosmological parameter analysis. arXiv:2412.00431, 2024.

Licong Xu, Milind Sarkar, Anto I. Lonappan, et al. Open source planning & control system with language agents for autonomous scientific discovery. ICML 2025 Workshop on Machine Learning for Astrophysics, arXiv:2507.07257, 2025.

CMBAgent developers. CMBAgent: planning and control in LangGraph. GitHub, https://github.com/CMBAgents/cmbagent. branch `main`; recorded runs used commit `397b775`; accessed 16 November 2026.

Biwei Dai, Po-Wen Chang, Wahid Bhimji, et al. FAIR Universe weak lensing ML uncertainty challenge: Handling uncertainties and distribution shifts for precision cosmology. arXiv:2604.14451, 2026.

Thomas Borrett, Licong Xu, Andy Nilipour, Boris Bolliet, Sebastien Pierre, Erwan Allys, Celia Lecat, Biwei Dai, Po-Wen Chang, and Wahid Bhimji. Competing with AI scientists: Agent-driven approach to astrophysics research. arXiv:2604.09621, 2026.

Shivam Rawat and Lucie Flek. Plausible but wrong: A case study on agentic failures in astrophysical workflows. arXiv:2604.25345, 2026.

Francisco Villaescusa-Navarro, Boris Bolliet, Pablo Villanueva-Domingo, Adrian E. Bayer, et al. Denario project: Deep knowledge artificial intelligence agents for scientific discovery. *PRX Intelligence*, 1:021001, 2026. arXiv:2510.26887.

Denario developers. Denario: a multi-agent research assistant. GitHub, https://github.com/Denario-AI/Denario. accessed 15 November 2026.

OpenCode developers. OpenCode: the open source coding agent. GitHub, https://github.com/anomalyco/opencode. version 1.19.5; accessed 15 November 2026.

Íñigo Zubeldia, Boris Bolliet, Francisco Villaescusa-Navarro, Pablo Villanueva-Domingo, and Miles Cranmer. Auditing AI peer reviewers: a benchmark on real scientific papers. https://arxiv.org/abs/2611.17052v1, 2026. Accepted at NeurIPS 2026.

ACT DR6 Independent Reanalysis Group and Licong Xu. Independent reanalysis of Atacama Cosmology Telescope Data Release 6: Daytime and nighttime power spectra, map-level null tests, beam systematics, and differentiable cosmological inference. Parallel ArXiv, PX:2611.00001, 2026. https://papers.parallelscience.org/abs/2611.00001.

OpenAI. On the Navier–Stokes Millennium Prize Problem. https://openai.com/index/navier-stokes-solution/, 2026. 8 September 2026.

Parallel Science. Parallel science. https://parallelscience.org/. accessed 16 November 2026.

CLArXivR. CLArXivR: paper discovery and processing. https://www.clarxivr.org/. accessed 16 November 2026.

Santo Fortunato, Carl T. Bergstrom, Katy Börner, James A. Evans, Dirk Helbing, Staša Milojević, Alexander M. Petersen, Filippo Radicchi, Roberta Sinatra, Brian Uzzi, Alessandro Vespignani, Ludo Waltman, Dashun Wang, and Albert-László Barabási. Science of science. *Science*, 359(6379):eaao0185, 2018.

Index

adapter, 3
agent, 2

context, 2

dataset, 8
dominated, 7

executor, 3
experiment, 9

fan-out, 5
Fault injection, 7

generation, 4

impact-aware, 14

knowledge graph, 10

multi-agent system, 4

orchestration, 4
organisation, 12

Pareto frontier, 7
pipeline, 5
Planning and control, 6
propose–critique loop, 5

reduce, 5
reducer, 5
research harness, 2

science of science, 14
Self-debugging, 6
session, 4
span, 4
state, 2
structured output, 3
supervisor, 5

thinking level, 12
tokens, 4
trace, 4