© 2026 Yiguedan Philippe Andy Yaro



Local AI Orchestrator: Code Checks the Judge

Yiguedan Philippe Andy Yaro

View GitHub Repository: github.com/andyyaro/local-ai-orchestrator

Let's connect! Email: yyaro@gmu.edu

LinkedIn: linkedin.com/in/andy-yaro



Local AI Orchestrator runs seven AI agents on my Mac through Ollama, a program that serves open language models locally. They refine the goal, plan, draft, critique, fix, score and polish, with no paid services and no API keys, the passwords cloud AI services need.

Its core rule: plain code, not a model, has the last word on what can be checked. In the main pipeline, a broken word limit, or code that fails its own test, scores 0 whatever the judge model said. I built version 1 from my build guide on 3–4 July 2026, then the version 2 upgrade with Claude Code, Anthropic's coding agent: 166 commits, tagged v2.0.0 on 5 July.

On 3 November 2026 this deck runs the tests, replays the launch film's recorded run through the real rules, and runs both pipelines against a scripted stand-in for Ollama. No language model ran here. I am Yiguedan Philippe Andy Yaro.

A. Why It Exists



Ask a language model once and you get one answer, with no way to know whether it is good or a weak first draft.

The trigger for version 2: a test asked for 120 words, the answer ignored the limit, and the test passed. Midway through the upgrade, a 50-word request came back at about 450 words, and the judge model still scored it 88, a pass.

The fix is a quality loop that runs on my own machine, where plain code checks whatever can be checked, and no model's score can overrule those checks.

1. What Went Wrong, and What Changed

What Went Wrong in Version 1

A test asked for 120 words. The output ignored the limit, and the test still passed.

A full run took 12 minutes 49 seconds; a measured baseline, with the smallest model in every role, took 8 minutes 45 seconds.

Midway through the upgrade on 4 July, a 50-word goal came back as an essay of about 450 words, scored 88 out of 100: “passed: true”.

What Version 2 Changed

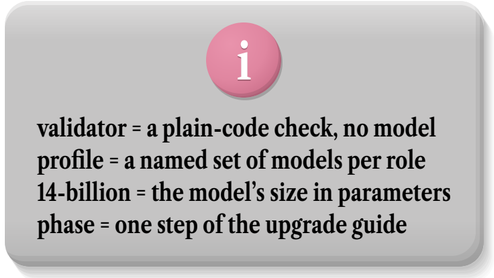
Plain-code validators read the user’s original goal and can force the score to 0.

A rule picks a fast, normal or deep path, so short tasks skip whole stages.

Timeouts by model size, one retry on a dropped connection, and a smaller model on a timeout.

Each model profile uses at most one 14-billion-parameter model family, so models are not swapped in and out of memory.

From docs/upgrade-guide/00-overview.md and the phase 0 and phase 6b audit reports, written by the implementation agent on 4 July 2026. The machine is a MacBook Pro with an M3 chip and 24 GB of memory.



2. Options Considered

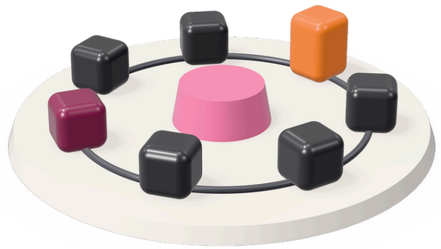
Option considered	What happened
A context window of a million tokens on the laptop	Rejected as impossible in 24 GB; retrieval memory instead
Retrying a timed-out call after a wait	Rejected; fall back to a smaller model instead
Several 12- to 14-billion-parameter models in one run, swapped in and out of memory	Replaced by one model family per profile
A paid cloud model as a fallback	Built behind several switches; never turned on
A test job on my own MacBook, started by hand, that runs real models	Designed in the guide; not added
Newer Qwen3 models, Apple's MLX framework, a judge from another model family	Not adopted: the default builder and judge are now the same model

From two AI-generated research reports in docs/research and the 13-phase upgrade guide in docs/upgrade-guide. There is no separate decision record. Version 1 had a judge from another family.



context window = how much text a model reads at once
token = a word or piece of a word
retrieval = finding past text to reuse
MLX = Apple's machine-learning library
Qwen3 = Alibaba's newer open models

B. Architecture

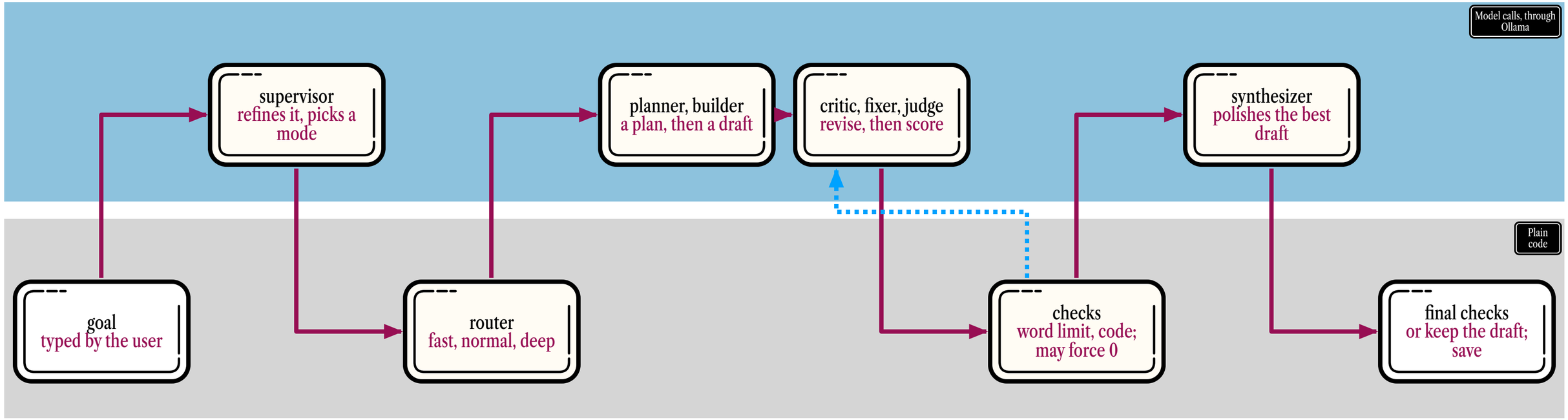


Seven roles, each one prompt sent to a local model: supervisor, planner, builder, critic, fixer, judge and synthesizer.

Around them, plain Python decides: which path to take, when to stop, whether code runs, and whether a score stands.

Two ways to run it share the main pipeline: a command line and a dashboard built with Streamlit, a Python web library. A second pipeline, built with LangGraph, a library for agent workflows, has fewer checks.

1. One Request, End to End



Low score, loops left and still improving: back to the critic, and plain code adds the exact failure to the critique the fixer reads. The fast path skips the planner and, by default, judges the builder's draft once, with no critic or fixer.

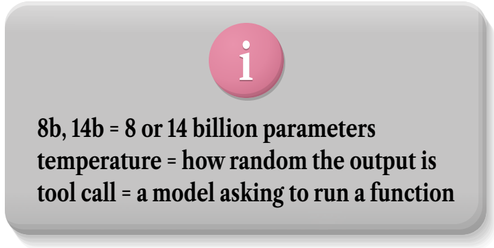
JSON = structured text
SQLite = a database in one file
mode = writing, coding, planning and so on

: Model call : Plain code : Next step : Back to the critic

2. Seven Roles, Seven Prompts

Role	Prompt lines	Model (default)	Returns
Supervisor	49	llama3.1:8b	A refined goal and a mode
Planner	31	llama3.1:8b	A numbered plan; skipped on the fast path
Builder	48	qwen2.5:14b	The first full draft
Critic	49	llama3.1:8b	What is weak or missing
Fixer	47	qwen2.5:14b	A revised draft
Judge	80	qwen2.5:14b	Four scores, hard failures and a reason, as JSON
Synthesizer	44	qwen2.5:14b	The polished final text

No agent calls tools: each is one prompt in and one text out, at temperature 0.7. The default profile is “serious”; “bootstrap” runs every role on llama3.2:3b, and coding and debugging goals move the 14-billion-parameter roles to qwen2.5-coder:14b.



C. Code Has the Last Word



Some limits can be checked exactly: how many words a draft has, or whether its code passes its own test.

For those, plain code checks the draft and, once the judge has scored it, can force the verdict to fail with a score of 0, however high the judge scored.

The word limit is read from the goal the user typed, not from the supervisor's rewrite, which once dropped it.

1. A Pass Turned Into a Fail

```
$ ../ci_venv/bin/python -I ../checks/override_demo.py media/launch-film/capture/run
goal typed:        Write a 50-word summary of why sleep matters.
supervisor's goal: Summarize the importance of sleep in 50 words. (8 words, mode writing)
route chosen by router.py: fast
limit found in the typed goal: (50, 'exact')
builder's draft: 216 words
judge's own verdict (dashboard): total_score 84 pass True
  [Validators] Hard fail: constraint_violation overrides Judge score (word count 216 is outside the 5-word tolerance for exact target 50)
after the validators: total_score 0 pass False hard_fails ['constraint_violation']
recorded run: scores [0, 0, 0] passed False | calls {'llama3.2:3b': 10}
```

1. The supervisor's 8-word goal goes down the fast path, by a plain word-count rule.
2. The builder wrote 216 words for a 50-word ask, and the judge still gave 84 and a pass.
3. The validator turns that into a fail with a score of 0, as the dashboard showed.

Run on 3 November 2026 with override_demo.py, written for this deck. It reads the run recorded for the launch film (media/launch-film/capture/run) and passes the same goal, draft and judge verdict through the repo's own router and validators. No model ran; the original run used llama3.2:3b.

i

exact = within 10% of the number
hard fail = a failure no score can rescue

2. The Plain-Code Rules

```
$ ../ci_venv/bin/python -I ../checks/rules.py
route  fast   <- Summarize the importance of sleep in 50 words.
route  deep   <- Give a comprehensive comparison of three note-taking methods for stude...
route  deep   <- Write a Python function double(n) with a pytest test.
route  normal <- Plan a weekly study schedule for a part-time student who works evening...
limit  (50, 'exact')  <- Write a 50-word summary of why sleep matters.
limit  (100, 'exact') <- Explain photosynthesis in under 100 words.
limit  (3, 'exact')   <- Explain the 3 words every pilot learns first.
coding True   <- Fix the bug in this Python script.
coding True   <- Explain why human error causes most aviation accidents.
```

1

Word limits are read with patterns. “Under 100” becomes exactly 100, and a number of words in the topic becomes a limit.

2

Any of 23 words or phrases such as “error”, “bug” or “class” forces coding mode. A prose question about human error then runs in coding mode, where any answer without working Python scores 0.

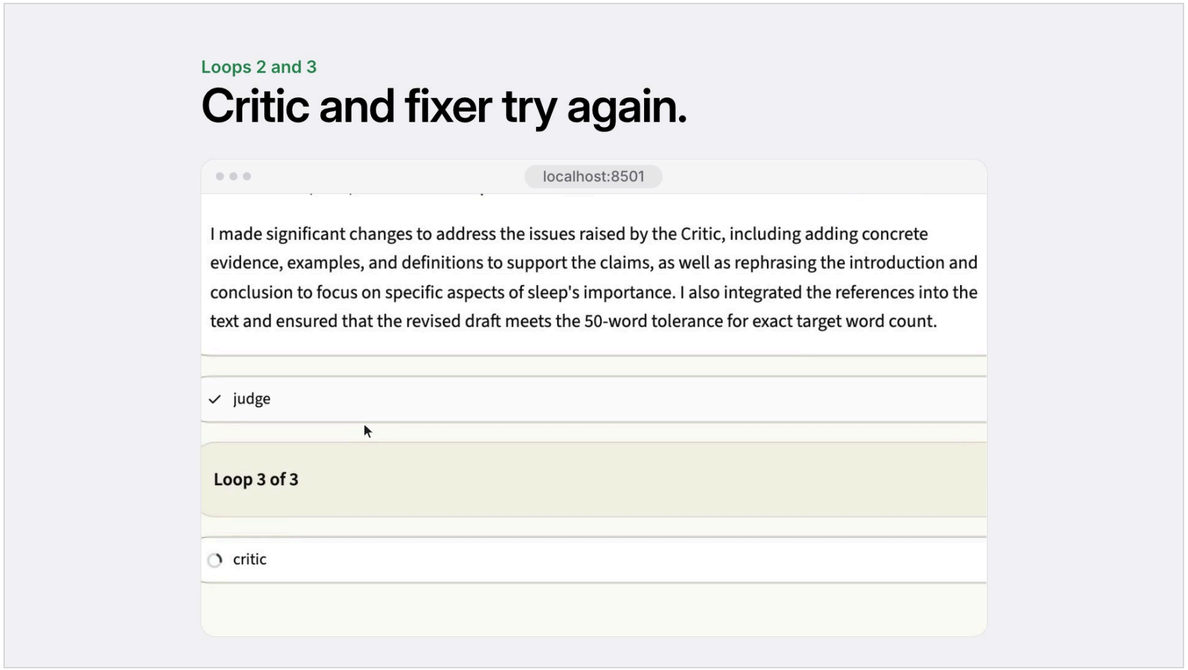
Fast: 25 words or fewer, no planner, no critic or fixer, 1 loop, passing score 60.

Normal: anything else, 2 loops, passing score 70.

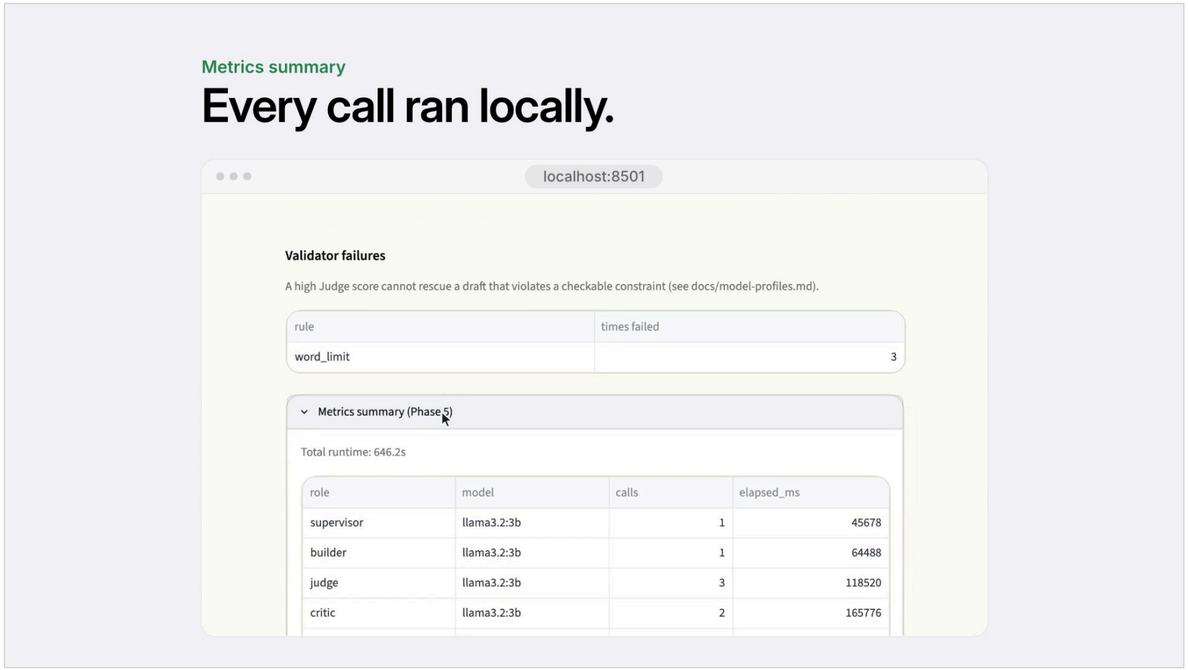
Deep: coding or debugging, or words like “comprehensive”, “step by step”; 4 loops, passing score 80.

Run on 3 November 2026 with rules.py, written for this deck, which calls the repo’s functions directly. The project’s own notes accept these false positives, rules firing when they should not, on purpose, to catch real coding goals.

3. The Film's Real Run



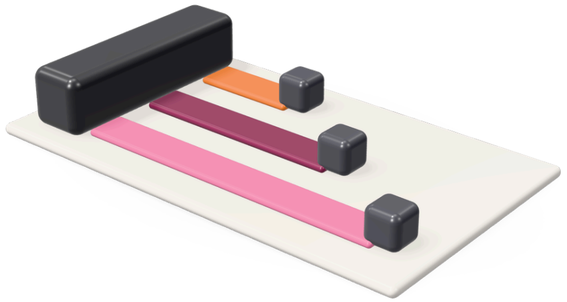
Loop 2. The fixer says its draft now meets the 50-word target. That draft has 360 words.



The result. Three loops, 216, 360 and 401 words, all forced to 0: "Failed". 10 calls, all to llama3.2:3b, in 646 seconds on a busy shared machine.

The synthesizer's polished version had 193 words and broke the limit too, so the run kept the builder's draft. A small model can miss the target; the run cannot report a pass while breaking a limit that code can check.

D. Paths and Resilience



The router sends each goal down a fast, normal or deep path, so a short task skips the planner and the critique loop.

When a local model stalls, the call falls back to a smaller model instead of waiting again. A dropped connection gets one retry.

Every run of the main pipeline writes its timings, retries and fallbacks to a summary file.

1. Three Paths, and What Happens When a Call Fails

Path	Chosen when	Runs	Loops	Passing score
Fast	25 words or fewer, no deep signal	No planner, no critique loop	1	60
Normal	Everything else	Full pipeline	2	70
Deep	Coding or debugging, or a deep keyword	Full pipeline	4	80

When a Call Fails

Timeout limits by model size: 120 s small, 300 s medium, 600 s large.

A timeout falls back once to llama3.2:3b; in the all-llama3.2:3b profile that is the same model, so a timeout simply ends the call.

A dropped connection is retried once after 2.5 s; other errors fail at once.

Where the Settings Are Overridden

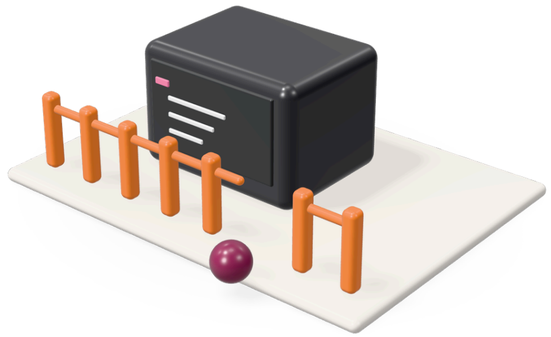
The dashboard always sends its slider values, 3 loops and a passing score of 70 unless changed, so its runs never use the per-path loops or scores.

The LangGraph command line does the same, with its own defaults.

So the film's fast-path run had 3 loops and a passing score of 70, not 1 and 60.

From config/models.yaml, orchestrator/router.py, orchestrator/resilience.py and the film's recorded run summary, at commit 934da18.

E. Coding Mode and Safety



In coding mode the pipeline runs the generated Python and its tests with pytest, Python's test runner. Code that fails is a hard failure, scored 0.

The code runs on my machine with a list of forbidden text patterns and a time limit. It is not a sandbox, a sealed-off space that cannot reach my files, network or secrets, and the code says so.

A separate coding-agent library, code other programs call, edits files until tests pass. It has its own guards, and gaps.

1. Same Broken Code, Two Verdicts

```
$ ../checks/stub.sh start
stub listening on 127.0.0.1:11434 (fixed answers, no model)
$ ../ci_venv/bin/python run.py --goal "Write a Python function double(n) that returns n times 2, \
    with a pytest test." 2>&1 | grep -E "Path selected|Stop reason|Final score|Passed|Code checks"
  Stop reason    : hard_fail: ['broken_code']
  Final score    : [                                                  ] 0/100
  Passed         : NO ✗
  Path selected  : deep
  Code checks    : 4 run, 4 failed
$ ../ci_venv/bin/python run_langgraph.py --goal "Write a Python function double(n) that returns n \
    times 2, with a pytest test." 2>&1 | grep -E "Stop reason|Final score|Passed|Code checks"
  Stop reason    : passed (score 78 >= threshold 70)
  Final score    : [                                                  ] 78/100
  Passed         : YES ✓
$ ../checks/stub.sh stop
stub stopped
```

1 The main pipeline runs the code four times, sees its test fail each time, and reports a fail with a score of 0.

2 The LangGraph pipeline never runs the code, and its command line uses a passing score of 70, not the deep path's 80: the judge's 78 stands and the same broken code "passes".

Run on 3 November 2026. The stand-in, stub_ollama.py, was written for this deck: it is not a model. Every builder and fixer answer is the same code, whose test fails because it returns n + 2, and the judge always answers 84, which the judge code caps to 78 under the coding scoring weights. The v2 final report in the repo says the LangGraph pipeline is kept in step with the main one; it is not.

i

pytest = Python's test runner
LangGraph = a library for agent workflows

2. A Block List, Not a Sandbox

```
$ ../ci_venv/bin/python -I ../checks/blocklist.py
BLOCKED  os.system("ls")
allowed  from os import system; system("ls")
BLOCKED  subprocess.run("ls", shell=True)
allowed  subprocess.run(["ls"])
BLOCKED  os.remove("notes.txt")
allowed  os.unlink("notes.txt")
BLOCKED  requests.get(url)
allowed  import http.client
child sees DEMO_KEY: not-a-real-secret
child cwd: /home/claude/run_lao_ev/repo
```

1. The block list matches text, so the same action written another way passes: an import of system, a command as a list, unlink instead of remove, another web library.

2. The code runs as my user and sees every environment variable, so any key loaded from a .env file of secret settings. It starts in whatever folder I launched from, normally the repository.

Run on 3 November 2026 with blocklist.py, written for this deck. The block-list lines are only matched, never run; the one snippet that runs prints a fake key set just for the test.

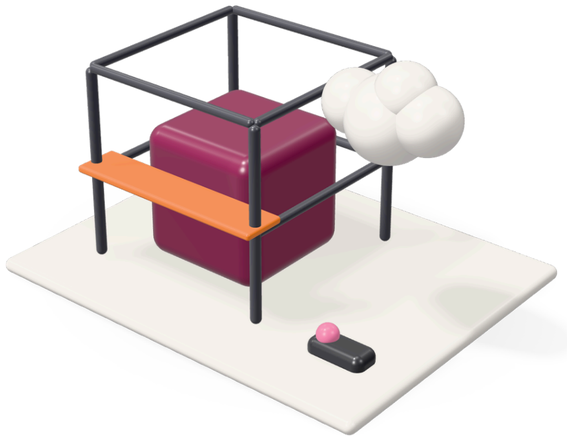
The Coding-Agent Library

Limits: paths kept in the target folder, the same block list, at most 40 changed lines, 5 tries, 60 s of tests.

Gaps: the “never edit your own repository” guard checks only the exact top folder, and it may write a conftest.py, which pytest runs on the very next test, or overwrite an existing git hook.

environment variable = a setting every program inherits
conftest.py = a file pytest loads before tests
git hook = a script git runs on events

F. Built, Switched Off



Three features exist in code and tests but are off by default: retrieval memory, web research and a cloud fallback.

Each is scaffolding, partly built: it can be turned on, but as far as the repo records, none has run against real data with real models.

One cannot work as wired, and one does more than its settings say.

1. Three Features, Off by Default

Retrieval Memory

Stores past text in SQLite with an embedding, a list of numbers for its meaning.

Search mixes a keyword score and a meaning score, half each, and returns 5 chunks.

Nothing in the pipeline fills it; only one evaluation scenario does.

The keyword half finds nothing for goals with ordinary punctuation.

Web Research

A separate command; needs a setting and a flag to go online.

Reads robots.txt first, the site's rules for automated tools; 15 s and 2 MB limits; marks fetched text as untrusted.

Checks that each citation points to a fetched source sharing at least 40% of the claim's key words.

The only search is a mock, a fake for testing, with sample addresses. Its notes say it makes no network call; switched on, it fetches them.

Cloud Fallback

Wired for the synthesizer only (the settings also allow the judge), behind seven checks: a flag, three settings, a privacy check, a budget and a typed "y".

One gate checks the main provider setting, not the cloud's own, so turning it on breaks every local call.

The code that would call the cloud is an empty placeholder, and the dashboard never asks for it.

From memory/, research/, orchestrator/cloud_policy.py and orchestrator/adapters.py at commit 934da18. The research fetch is plain in research/run_research.py; an earlier review run, with both switches on, sent requests to those sample addresses.

G. Tests and Evaluation



227 unit tests run in about 8 seconds with every model call replaced by a stand-in, so they check the code, not the models.

An end-to-end evaluation suite has 11 scenarios; 5 of them need a real model and could not run here.

Continuous integration (CI) runs the linter, a style and error checker, the tests and a dashboard import on every push.

1. The Tests, Run

```
$ ../ci_venv/bin/python -m pytest -q -p no:cacheprovider 2>&1 | tail -1
227 passed, 1 warning in 8.13s
$ grep -rhE --include="*.py" "^\s*(async )?def test_" tests | wc -l
213
$ ../ci_venv/bin/ruff --version && ../ci_venv/bin/ruff check --no-cache .
ruff 0.6.9
All checks passed!
$ ../ci_venv/bin/python -c "import app.streamlit_app" 2>/dev/null && echo "dashboard import: ok"
dashboard import: ok
$ ../lock_venv/bin/python -m pytest -q -p no:cacheprovider 2>&1 | grep -E "No module \
    named|Interrupted"
E   ModuleNotFoundError: No module named 'bs4'
!!!!!!!!!!!!!!!!!!!! Interrupted: 1 error during collection !!!!!!!!!!!!!!!!!!!!
$ grep -ciE "^(beautifulsoup4|sqlite-vec)" requirements-lock.txt
0
```

Run on 3 November 2026 on a clean copy at commit 934da18, Python 3.12, in two environments: one from requirements.txt as CI does, one from requirements-lock.txt.

i

linter = flags likely mistakes and style
lock file = a list of exact package versions
CI = checks that run on every push

1. All 227 tests pass, as CI installs the packages.
2. They come from 212 test functions; two run with several inputs. grep finds 213, but one is sample code in a string. My notes said 215.
3. The linter passes at the exact version CI installs, 0.6.9. The latest, 0.17.3, finds 66 issues, mostly style.
4. Installed from the lock file of exact versions, one test file cannot load, and pytest stops...
5. ...because the lock file predates two packages added on 4 July.

2. The Evaluation Suite

Scenario	Needs a model	What “pass” means
Word limit	yes	Pass also covers “broke the limit and reported failure”
Judge returns JSON	yes	Parseable JSON with the five expected keys
Simple coding task	yes	At least one code check ran and none failed
Timeout fallback	yes	A forced 1-second timeout led to a fallback
Local only	yes	No cloud call recorded; cloud cannot fire anyway
Retrieval	an embedding model	Finds the indexed text; skipped today
Five others	no	Mock cloud, privacy, budget, citations, dashboard import

Recorded results



The full run on 4 July: 9 passed, 1 failed, 1 skipped. A later report claims 10 passed, but only one scenario was run again. Today, offline: the 5 that need a model failed with no Ollama; 1 was skipped.

What the Unit Tests Cover

64% of the lines in the main packages.

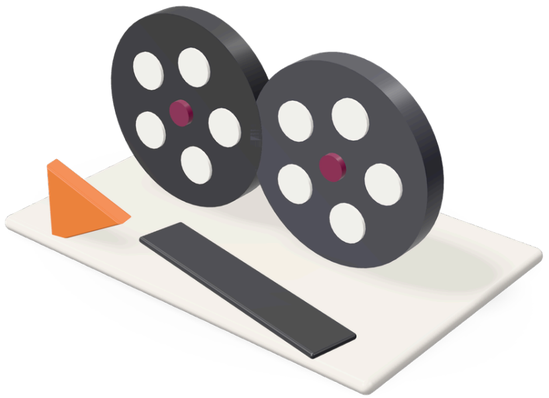
0% of the LangGraph pipeline and of the dashboard, apart from the import check.

No test or evaluation measures output quality; each scenario is pass or fail.

scenario = one end-to-end case
coverage = share of lines tests run
mock = a fake stand-in for testing

Coverage measured in this review on 3 November 2026; the repo’s CI does not measure it.

H. History and the Film



Built in two days: 132 commits on 3 July 2026 and 34 on 4 July, with the v2.0.0 tag at 00:31 on 5 July.

I built version 1 from my build guide, then the upgrade with Claude Code as the implementation agent, the AI tool that wrote code from my instructions: the guide says so, and one session ran three phases on its own.

In November came a logo and a 70-second launch film of one real run.

1. Two Days of Building, Then Two Later Visits

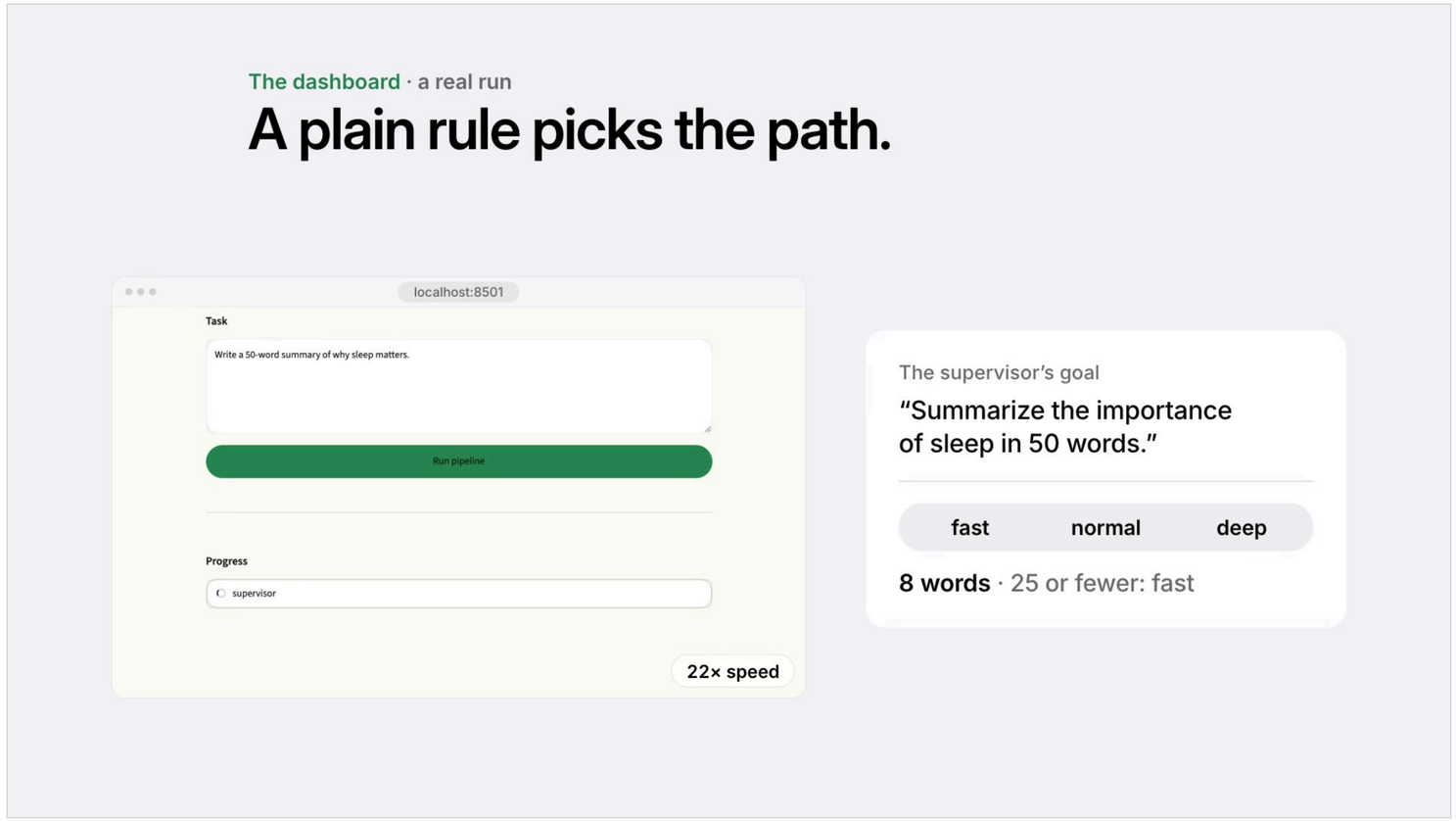
```
$ git log --date=format:%Y-%m-%d --format=%ad | sort | uniq -c
    132 2026-07-03
     34 2026-07-04
      3 2026-08-19
      1 2026-11-01
      1 2026-11-02
$ git for-each-ref refs/tags --format='%(refname:short)  %(creatordate:format:%Y-%m-%d %H:%M)'
film-v1  2026-11-02 14:56
v1.0.0  2026-07-04 10:56
v2.0.0  2026-07-05 00:31
$ git log --format=%s | grep -cE '\(#[0-9]+\)$|^Merge pull request #'
20
$ git log --format=%B | grep -c "Co-Authored-By: Claude"
1
$ grep -n "Claude Code as your implementation agent" docs/upgrade-guide/00-overview.md | cut -c1-96
26:You're using Claude Code as your implementation agent, which means it's tempting to say "impl
```

1. 166 commits over two days.
2. Version 1.0.0 on the morning of 4 July; 2.0.0 just after midnight, after 13 upgrade phases.
3. 20 pull requests, all merged on 4 July.
4. The upgrade guide names Claude Code as the implementation agent.

My notes say “Solo”: every commit has one human author, with Claude Code as the implementation agent for the upgrade phases; only the film commit carries a co-author credit line. Run on 3 November 2026 at commit 934da18. Times are US Eastern.

tag = a named point in history
pull request = a change proposed, then merged

2. The 70-Second Film



A frame from the film: the dashboard footage is a frame-by-frame recording of one real run, sped up where the models were working.

How It Was Made

69.9 seconds, 1920 × 1080, 14.2 MB.

A web page animated with GSAP, a JavaScript animation library, and made into video with HyperFrames, an open-source tool that records web pages.

Narrated by Kokoro, an open text-to-speech model, run locally.

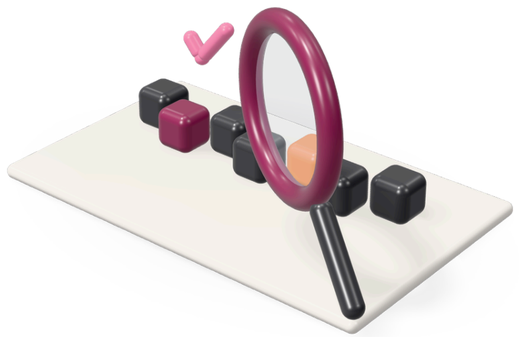
Made with Claude Code from my brief; the commit credits Claude as co-author.

Where to Watch

andyyaro.github.io/local-ai-orchestrator/film.html

The video loaded from this site on 3 November 2026.

I. Proof and Limits



Checked on 3 November 2026: the tests in two environments, the film's real run replayed through the rules, and both pipelines run against a scripted stand-in.

The core idea holds: plain code overrides the judge in the main pipeline. Some features and claims go further than the code.

The last slides set my claims against the repo, list fixes, and say what the project shows.

1. My Claims Against the Repo

My notes said	The repo shows	So I now say
Jul 3 to 5, 2026; Solo	Built 3–4 July, tagged 5 July; one human author, with Claude Code as the implementation agent for the upgrade	Solo, with Claude Code as the implementation agent
169 commits	171 now; 166 up to v2.0.0	171 commits, 166 to v2.0.0
~20 merged PRs in 12 phases	20 pull requests: 13 phases (0 to 12), five follow-up fixes and two documents	20 merged pull requests, 13 phases
215 tests	227 tests from 212 test functions, all passing	227 passing tests
~11,000 lines	About 10,700 lines of project Python (7,600 source, 3,100 tests), plus 1,900 of film scripts	About 10,700 lines of Python
Validation layer that overrides the models	Word limits, a required code block and code results override the judge; other validators are never called	Plain-code checks override the judge's score
Retrieval memory; citation-checked research	Built and off; memory never filled; research has only a mock search	Retrieval and research scaffolding, off by default
A sandboxed coding-agent loop	A block list and time limits, not a sandbox; a library with no command	A guarded coding-agent library, not sandboxed

From my career-hub notes, checked against the repo on 3 November 2026. PR = pull request.

2. What to Fix Next

1. **Give LangGraph the same checks**
   Run the generated code and its tests there too, use the per-path settings and save runs, or retire it.

2. **Make the runner a real sandbox**
   Run code in an isolated container with no network, no secrets and its own folder; stop the README calling it sandboxed.

3. **Fix the coding agent's guards**
   Check every folder under the repo, and refuse conftest.py and git hooks.

4. **Make research match its notes**
   Have the mock search return text without fetching, or change the note.

5. **Untangle the cloud switch**
   Make the gate check the cloud's own provider setting, not the main one, or remove the scaffolding.

6. **Sharpen the plain-code rules**
   Read "under N words" as a maximum, and stop "error" alone forcing coding mode.

7. **Refresh the lock file and README**
   Add the two missing packages, fix the screenshots and Python version, add a LICENSE file.

8. **Clean up what is public**
   Use the GitHub no-reply email, remove home-folder paths and private film notes, and keep session links out of commits.

Roughly in order of risk: wrong passes first, then running untrusted code, then features that say more than they do, then housekeeping. Fix 6 also stops wrong passes: "under 100 words" lets 110 through.

3. What It Shows, and What It Does Not

It Shows

A local multi-agent loop where plain code, not the judge model, decides checkable pass or fail.

Engineering for one laptop's limits: paths, timeouts, fallbacks, metrics.

227 passing tests and a launch film built on one honest, failed run.

It Does Not Show

That the loop improves quality: no evaluation measures it.

Safe code execution: the runner is not a sandbox, a sealed-off space.

Working memory, research or cloud features with real data.

That the second pipeline is as strict as the first.

The core rule works and is tested. The next step is to measure whether the loop makes answers better, and to run generated code somewhere it cannot reach my files or keys.