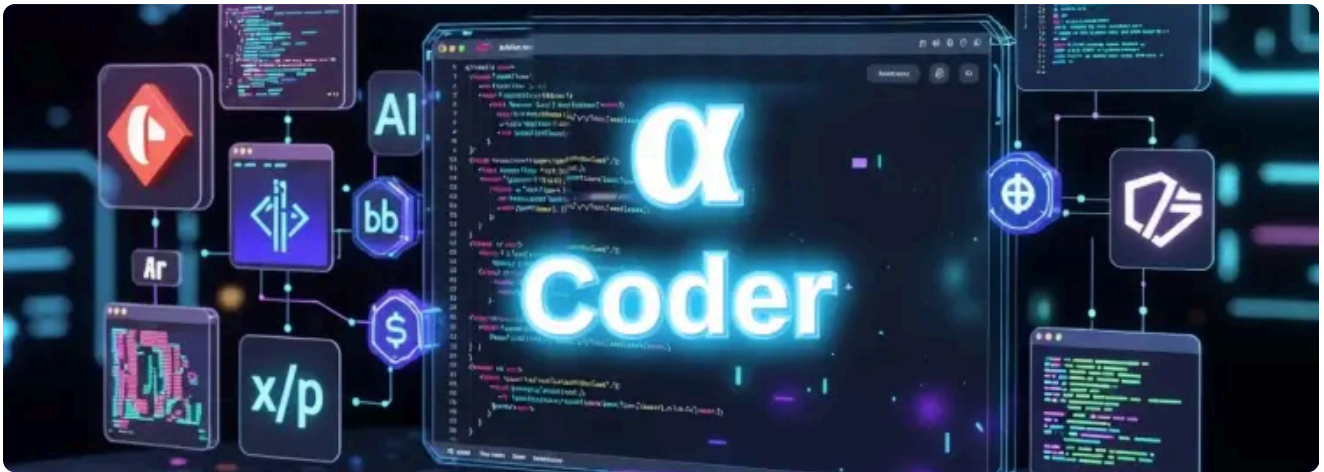




A council of models that cannot sign off its own work

Architecture and field notes from α -Coder

Taras Safonov, AlfaCan Defence Group

Published 27 October 2026 · Data: live stages, 11–27 October 2026.

Abstract

Single-model coding assistants frequently suffer from self-confirmation bias, contextual drift, and ungrounded hallucination. We present **α -Coder**, an asymmetric multi-agent architecture that replaces monolithic assistants with an eight-seat programmatic council where roles (orchestration, generation, adversarial breaking, validation) are strictly segregated across five distinct models from four labs. Unlike prompt-based guidelines, critical invariants — a green build and a trusted test run before hand-over, test runs whose exit code cannot be masked by the shell, and decisions accepted only from the operator's recorded provenance — are enforced programmatically by the execution harness. We report a 17-day field deployment on a production-grade Go publishing service (20,749 lines of implementation code, 16,802 lines of test code across 365 test functions): the events the harness recorded, the defects the council caught and missed, and the failure modes of our own enforcement rules. We report mechanisms and observed events, not comparative rates.

Contents

1. The problem with one model in a chat
 2. Architecture of the council
 3. Rules the program enforces
 4. Parallel agents
 5. The model gateway
 6. Field study: a publishing service
 7. What broke, and what we learned
 8. Limitations and threats to validity
 9. Artifact Availability and Evaluation Access
-

1 The problem with one model in a chat

A coding assistant built on one model plays every role at once. It reads the task, decides what the task means, writes the code, writes the tests for that code, runs them, reads the result and reports success. Each step is reasonable on its own. Together they form a closed loop in which the same reasoning that produced a mistake is the only reasoning asked to find it.

Three failures follow from that loop, and we saw all three before we built anything else.

Self-review. A model asked to check code it has just written tends to confirm its own intent rather than test it. The test it writes encodes the same assumption as the code, so both pass together and fail together.

Drift. Over a long task the working context fills with the model's own intermediate decisions. The original specification is still there, but it competes with dozens of later, more local choices. By the third hour the work is consistent with itself and only loosely consistent with what was asked.

Invention. When the specification is silent — a rate limit, a field name, a size cap — a model fills the gap with a plausible value and moves on. The value is then load-bearing: other code depends on it, tests assert it, and nobody remembers that it was never decided.

None of these is a capability problem that a larger model removes. They are structural: they come from one agent holding every role. Our hypothesis was that separating the roles across different models, and moving the most important rules out of prompts and into the program, would remove the loop rather than ask the model to resist it.

2 Architecture of the council

2.1 Seats

The council has eight seats.

- **Orchestrator** runs the process: it reads the task, keeps the plan, hands stages to the other seats and reports to the human operator.
- **Architect** writes the design documents for a stage: data model, interfaces, decisions and open questions.
- **Code** implements the stage and owns the build and the tests.
- **Breaker** attacks the result as an adversary would: it tries to prove that something is broken, insecure or unnecessary. It does not fix anything.
- **Skeptic** reads the Breaker's findings and tries to refute them against the code. Only findings that survive go further.
- **Reviewer** turns the confirmed findings into a fix plan, or declares the stage clean.
- **Validator** signs off a stage only with evidence for every acceptance criterion of the specification.
- **Curator** reads every stage while it runs and raises notes where the work drifts from the specification or from earlier decisions.

The seats that attack, review and validate Code's work run on models from other labs than Code. The model behind any seat is a route in our model gateway, so it can be changed without changing the program — we changed three seats during the period this paper covers.

8 seats, 5 models, 4 labs

The seats that attack, review and validate Code's work run on models from other labs than Code. Three seats changed models during the period.

Model gateway routes, 11–27 Oct 2026

2.2 The meeting

Before any code, the seats meet. Each answers the stage's questions independently in the first round, without seeing the others. Later rounds discuss only the points where answers differ. The meeting ends with decisions per item, dissenting opinions recorded, and a list of what tests must prove. A human approves the resulting plan; nothing runs before that approval.

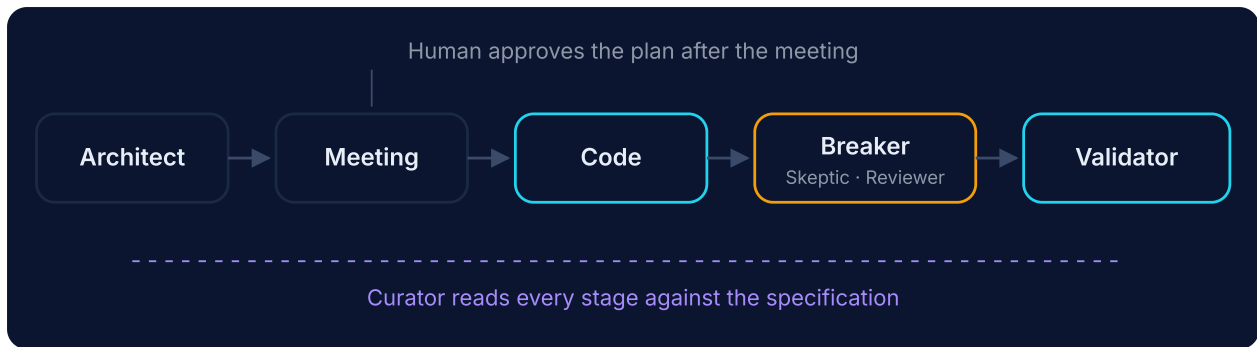


Figure 1. One stage of work. Hand-over is blocked until the build is green and the tests pass; the Validator signs only with evidence.

The independent first round matters more than the discussion. When models see each other's answers first, they converge on the first confident one. When they answer blind, disagreements surface, and disagreements are where the specification is ambiguous.

2.3 The human

The operator is not a reviewer of last resort but a source of decisions. The program treats the operator's recorded decisions — in the task specification or in the correspondence log — as the only authority for anything the specification does not state. A model may not write "per the operator's decision" unless such a record exists (§3.3).

3 Rules the program enforces

A rule written in a prompt is a request. A rule written in the program is a fact. We moved four rules out of prompts.

3.1 Green before hand-over

Code cannot hand over a stage until, after its last change of code, a build and a test run have both succeeded. The check is made by the program when the model calls the hand-over tool; a refused hand-over returns to the model with the reason.

3.2 Trusted runs

Not every test run counts. A run whose exit code belongs to something other than the test runner — `make test | tail -40`, `go test ./... || true` — is recorded as untrusted, because the exit code is `tail`'s or `true`'s, not the tests'. A run that changes files, or ends on a signal, is untrusted too. Models write such commands constantly and in good faith; the program does not argue with them, it simply does not count the run.

42 of 46 test runs did not count

Step 2: piped into `tail` (24), `grep` (10) or `head` (3), or followed by a command that set the exit code itself.

Test-run log, step 2, Oct 2026

```
# Hand-over gate
$ make test | tail -40      x untrusted: exit code is tail's
$ go test ./... || true    x untrusted: exit code is true's
x hand-over refused no trusted test run after the last change of code
$ make test                ✓ exit 0, trusted
✓ hand-over accepted
```

Figure 2. How the gate reads test runs. An illustration built from the commands in §3.2, not a verbatim log.

3.3 Decisions only from the record

When the model attributes something to the operator, the program looks for the decision in the task's specification or correspondence. A signature without a record does not stand. The same rule governs approval of the plan: the operator approves with a fixed phrase, optionally followed by a reason from a closed list of whole phrases. Anything else is treated as an amendment and the program asks again (§7.3 explains why the list is closed).

18 refusals for a signature without a record

In step 1 the program refused 18 hand-overs that attributed something to the operator without a recorded decision — the most common reason among its 36 refusals.

Stage ledgers, step 1, 11–19 Oct 2026

3.4 Notes need proof

The Curator's notes are tracked in a ledger with statuses. A stage closes a note by answering it in its report — "done, where" or "disagree, why". A "done" is accepted only with proof: a change of code after the note was issued and a trusted green run after that change. A note about a missing document is closed by creating exactly that document. A change to the plan, the journal or the specification is never proof that code was fixed. Notes that stay open travel to the next stage and, in the end, to the Validator.

4 Parallel agents

A seat can split its work across agents — short-lived workers with a narrow task and a clean context.

- **Read agents** (up to four per group) answer questions about the codebase: how a component is built, where a value comes from, what a test covers. They read; they cannot write.
- **Write agents** (up to four for Code, three for the Architect) each implement an independent part in a separate working copy. The program merges the parts back; a conflict stops the merge rather than being resolved silently.

Agents do not replace the seats' judgment. They replace the seats' reading: a stage that would spend most of its context opening files gets short reports instead, and keeps its context for decisions. The operator sees the agents that are working right now and what each is doing.

5 The model gateway

Every request from every seat and agent passes through our gateway. It routes a role to a model, enforces per-role budgets, and keeps the council working when a provider does not.

- **Budgets.** The smallest valid output budget a client sends wins; the gateway may lower it but never raise it. The ceiling is set by what fits inside the turn deadline, not by what the provider allows.
- **Spare models.** The seats on the busiest providers have a spare model from another lab. On a provider failure, a timeout, or when all of the provider's seats have been busy for more than 15 seconds, the request goes to the spare — before the first byte of the answer, never in the middle of a stream.
- **Circuit breaker.** After repeated provider failures the gateway stops calling that provider for a minute. Its own queue — seats busy with long turns — is never counted as a provider failure.

6 Field study: a publishing service

The codebase is a Go service that publishes content to social platforms on behalf of organisations: connections through each platform's OAuth, scheduled publishing, platform webhooks, per-platform limits and accounting of paid API calls. It was built by the council from an empty repository, with one human operator, between 11 and 27 October 2026. Every number below comes from the program's own logs — stage ledgers, test runs, the Curator's ledger and agent manifests — and from git.

6.1 Scale

	End of step 1 (19 Oct)	Now (27 Oct)
Go code, lines (without tests)	15,378	20,749
Test code, lines	12,156	16,802
Test functions	276	365
Packages	—	32
Database migrations	20	23

Note on test density: the test-to-code ratio is **0.81** (16,802 lines of test code to 20,749 lines of implementation). We report it without a baseline: we have not compared it with human-written or single-model projects of similar scope.

6.2 Step 1: the full review cycle

The first step built the core: organisations and access, connections, publishing, scheduling and operations. It closed with the complete cycle. The Breaker reported 17 findings — none critical, 5 high, 8 medium, 4 low — plus one hypothesis. The Skeptic confirmed all 17 against the code, refuted none, and lowered the severity of one. The Reviewer turned them into a 15-item fix plan; the operator added five mandatory items of his own. Two control rounds of the Breaker then went from 5 medium and 4 low findings to zero. The Validator signed off only after running the full test suite itself — 23 packages, exit code 0, no pipes — and attaching evidence to each of the 11 acceptance criteria. Its first attempt to sign off had been refused by the program: a verdict of PASS is not accepted unless the Validator ran the tests in its own stage.

17 findings, 0 refuted

The Skeptic confirmed all 17 of the Breaker's findings against the code. The Validator signed off after its own run: 23 packages, exit code 0, evidence for each of 11 acceptance criteria.

Review-cycle reports, step 1, 19 Oct 2026

6.3 Step 2: stages with timings

Stage	Seat	Duration	Code lines added	Test lines added	Test functions	Hand-over
Core of step 2: OAuth login, webhooks, streaming files, paid-call accounting, settings loader	Code	1 h 56 min	2,992	2,615	46	first attempt
Telegram connector, plus the core's tail	Code	4 h 12 min, including operator pauses	1,510	1,388	29	first attempt
Telegram: the operator's three decisions	Code	48 min	827	663	14	first attempt
Bluesky: verification of the platform's documentation	Code	44 min	0	0	0	second attempt
Core gaps for all remaining platforms	Architect	58 min	0	0	0	with one open note

Line counts come from the stage's working snapshots and exclude SQL migrations. Before the step, the council met on the architecture: four rounds, after which the operator took 13 decisions in one sitting.

6.4 What the program refused

Across both steps the program refused a hand-over 37 times. In step 1, 36 refusals: 18 for attributing something to the operator without a recorded decision, 8 for open Curator notes, 6 for the absence of a green run, 2 for claiming documentation without a verbatim quote, one for a Breaker report that did not end with its findings line, and one for a Validator verdict without its own test run. In step 2, after we turned the Curator's notes and unrecorded signatures from blocks into remarks for the Validator, there was one refusal: the stage had decided its own exit code with `code=$?` after the tests.

The most frequent single event in the logs is not a refusal but a run that does not count. In step 2 the models ran tests 46 times; 42 of those runs were untrusted — piped into `tail` (24), `grep` (10) or `head` (3), or followed by a command that set the exit code itself. The models were not hiding anything: piping long output is a habit. It is also exactly the habit that turns a red build green, which is why the program, not the model, decides what counts.

6.5 The Curator, before and after

In step 1 the Curator issued 73 notes in one task; the stage refused hand-over seven times in under an hour while the count went 22, 20, 13, 11, 11, 12, 11; the operator switched the Curator off. That loop is §7.1. After the redesign, in step 2, the Curator ran 41 times across the stages, issued 9 notes — all of the kind "missed", none repeated, none without proof — and 8 were closed. Among them: a body-size limit invented in code without asking (closed by asking the operator), a stage handed over with no test files (closed by tests), Telegram limits without a

quote from the platform's documentation (closed by quotes), and routes missing from the API description (closed by adding 39 paths and 50 operations).

Curator: 73 notes, then 9

Step 1: 73 notes in one task, then the operator switched it off. Step 2, after the redesign: 41 runs, 9 notes, none repeated, 8 closed.

Curator ledger, steps 1–2, 11–27 Oct 2026

6.6 Defects found without being asked

Three times in step 2 a stage found and fixed a defect outside its task, proved it with a test and said so in its report. While building the core, Code found that one SQL parameter was used both as a UUID and as a timestamp, which made creating any connection fail with a server error. While building the Telegram connector, it found that an empty UUID was written instead of NULL, violating a foreign key and failing every incoming channel event. In the next stage it found the connector storing a sign-in token in the field meant for the platform's access token.

6.7 Stopping to ask

The specification contains one sentence that shaped the work more than any other: *"What the task does not state as a number or a rule — do not invent it, ask me."* The logs contain the questions it produced: the maximum size of a webhook body (asked mid-stage, answered in a minute and a half); how a Telegram channel is identified when the Bot API has no method to list a bot's channels; whether to apply the documented "about 30 messages per second". The most consequential came on Bluesky: its OAuth requires a DPoP key on every request and a public client-metadata document, and the core stored neither. Code stopped, named the exact lines where the core fell short and offered four options. The operator chose a separate Architect stage to find the core's gaps for every remaining platform at once.

6.8 Agents

Agents ran seven times: six groups of read agents and one group of write agents, 22 agents in total. Fifteen returned reports. Four failed — the first group ever run in production, on the provider defect described in §7.2. Three were stopped by the operator, who narrowed the stage while they were reading. The single write group split the Architect's document into two parts written in parallel in three and a half minutes; the program could not merge them automatically, because fewer than 80% of their lines matched what the stage expected, and the Architect assembled them itself. Write agents for code have not yet run on this project.

7 What broke, and what we learned

7.1 The reviewer that kept saying the same thing

The Curator kept raising the same note in slightly different words, and stages looped on it. The model was not the problem: we fed it only the number and title of its earlier notes — no status, no answer — and it could not see deleted lines, so a fix that removed code looked like no change. Giving it the full ledger and real diffs fixed the loop; duplicate detection then created the opposite bug ("migration 0019 not applied" and "migration 0020 not applied" merged into one). Similarity on headlines is not enough: numbers, identifiers and paths must tell notes apart, and anything only *probably* a repeat goes to the human once.

7.2 The thinking model that rejected its own agents

The first time agents ran in production, every agent died on its second request: the provider demanded its own reasoning back. The difference from the main loop was one line — the main loop always appends environment details after tool results, so the model sees a new turn; the agent loop did not. We had a test for exactly this request shape, and it had locked in the broken shape because it ran against mocks. Every release now runs the agents live on every model the seats use.

7.3 "I approve the plan, but..."

We accepted the approval phrase followed by an explanation. Five review rounds of an attacker model kept finding phrasings that smuggled an amendment through — "because ok with Redis instead of Postgres". Any open vocabulary can be assembled into an amendment. The version that held: a closed list of whole phrases; anything else is an amendment.

7.4 Our proof rule had a ceiling

Our first rule for closing a Curator note accepted any edit after the note plus a green run. An edit to the plan counted. The second version counted only code — but "named in the note" let a mention of the plan back in. The version that held is the one in §3.4, and it took five rounds of adversarial review. Even now the rule has a ceiling we accepted deliberately: one change of code and one trusted run can close several notes at once. Tying proof to the exact place of each note is the next step.

7.5 A rule's refusal is not the operator's

When the program refused a command because the operator's own deny-list matched it, the model was told "the user denied this operation" — wording inherited from the editor we built on. The model reported that the operator had refused, which was false, and avoided the command for the rest of the stage. A refusal by rule now says which rule and what to do.

8 Limitations and threats to validity

- **No benchmark.** We report a field study on one production codebase, not a benchmark. We have not measured the council against single-model assistants on a shared corpus; claims here are about mechanisms and observed events, not rates.
- **One operator.** All stages had the same human operator, who also designed the process. A less involved operator would approve faster and catch less.
- **Small sample.** Seventeen days and one project. Several of the failures in §7 appeared once, and write agents for code have not run yet.
- **Model churn.** Three seats changed models during the period. We do not attribute outcomes to specific models.
- **Self-reporting.** The authors built the system they evaluate. Every number here comes from logs we can share on request.

Artifact Availability and Evaluation Access

α -Coder is deployed as an active web environment and VS Code extension. Researchers, developers, and system architects interested in stress-testing this multi-agent council architecture can access the platform at ai-studio.alfa-can.com/alfa-coder (10 free testing hours available). We actively encourage blunt feedback regarding architectural edge cases, failure modes, and operational friction.