

# Counterexample-Guided Reflective Evolution

A GEPA/Operon Certificate Experiment

Preprint – Feedback Welcome

Bogdan Banu  
bogdan@banu.be

April 23, 2026

## Abstract

Reflective-evolution optimizers such as GEPA evolve text components (prompts, code, configs) by scoring candidates with a user-supplied evaluator and feeding the results, plus free-form feedback, to a reflection language model that proposes mutations. The evaluator traditionally returns a scalar reward. We test an alternative in which the evaluator returns a binary pass/fail verdict paired with structured obligation text drawn from a formal theorem verifier (“Operon certificates”). The conjecture is that, on tasks with enumerable failure modes, binary-plus-obligations converges in fewer mutations than a scalar reward of comparable information density.

We run three arms on a synthetic windowed-stability task (`behavioral_stability_windowed`): *cert-binary* (the treatment), *scalar-reward* (a graded reward that peaks when the theorem holds), and *scalar-with-evidence* (an active control carrying the same per-window violation text as the cert arm but without the theorem framing). Ten seeds per arm, 50-iteration budget, Gemma 4 as the reflection LM. **[Primary result: cert-binary vs strongest baseline mutations-to-convergence ratio, Mann-Whitney U p-value, rank-biserial effect size, Wilson 95% CIs.]**

**[Ablation on the obligation formatter]** isolates whether the signal comes from binarization or from obligation content. The work is an engineering test of a theoretical claim (Theorem 3 of the companion integration memo) rather than a new algorithm; the deliverable is an honest measurement, and negative results are reported as such.

## 1 Introduction

Reflective-evolution optimizers decouple three concerns: a *candidate* (the text being optimized), an *evaluator* (a function that scores a candidate on a data instance and optionally emits feedback), and a *reflection model* (a language model that reads the evaluator’s output and proposes a mutated candidate). GEPA [2] is a representative instance: it evolves candidates through LM-authored mutations guided by reflective feedback, Pareto-front candidate selection, and optional merge operators.

The evaluator’s contract with the rest of the system is deceptively simple. It returns a numeric score and a blob of feedback text. In practice, the information density of that feedback is the optimizer’s dominant lever. A scalar reward is small-content, small-precision, and domain-agnostic. A structured verifier output — specifically, a pass/fail verdict plus a list of unmet obligations — is large-content, high-precision, and bound to a formal predicate.

The companion integration memo (external-frameworks, v0.36.3 of the Operon repository [5]) states this as a conjecture: for theorems with enumerable failure modes, a certificate-based evaluator

that returns (*holds*, *obligations*) should converge in fewer mutations than a scalar-reward evaluator of comparable information density. The memo does not ship evidence. This paper does.

Our contribution is engineering, not theory:

1. An *OperonCertificateAdapter* that wraps any Operon theorem verifier as a GEPA-compatible evaluator (`operon_ai/convergence/gepa_adapter.py`).
2. Two baseline adapters: *ScalarRewardAdapter* (continuous reward, no structured feedback) and *ScalarWithEvidenceAdapter* (continuous reward plus text-matched per-window violation evidence, but without theorem framing) — the latter is the active control that separates binarization from obligation-text content.
3. A synthetic, single-knob task (`policy_throttle`) that isolates the evaluator-to-optimizer pipeline from retrieval noise, API flakiness, and agent-complexity variance.
4. Ten-seed runs per arm, with convergence determined by the *same* predicate — the theorem holds on the full validation set — across arms.
5. An honest decision gate: positive, null, and negative outcomes each lead to a specific follow-up action, declared before the experiment was run.

Section 2 situates the work against prior reflective and verifier-guided optimization. Section 3 details the three arms, the synthetic task, and the statistical protocol. Section 4 reports measurements. Section 5 interprets them. Section 6 enumerates what we did not do and why.

## 2 Related Work

**Reflective-evolution prompt optimization.** GEPA [2] evolves text components via reflection-driven mutation over a Pareto front. Earlier work on prompt-optimization for LLMs (OPRO [11], APE [12]) uses scalar reward or rank-based comparison as the primary evaluator signal; feedback is either absent or free-form output from a grading LLM. Our paper tests whether replacing that scalar channel with a constrained, formal-predicate verifier changes mutation efficiency in the reflective branch.

**Verifier-guided LLM generation.** Outcome verification has been used to filter, rerank, or constrain LLM outputs in program synthesis, math, and theorem proving. The signal there is typically used *outside* the optimization loop — as a final gate or as part of best-of- $N$  selection. Our work differs in using verifier output as the optimizer’s *inner-loop* score, driving which candidates the reflection model sees and how.

**Operon certificates.** The Operon project [5] exposes a registry of theorems with associated verify functions (e.g. `behavioral_stability_windowed`, `behavioral_quality`) over structural and behavioral predicates drawn from biological agent designs. A certificate records the theorem name, parameters, a conclusion, and a source trace; `verify()` re-derives the predicate from parameters and returns (`holds`, `evidence`). The evidence dictionary is what this paper’s treatment arm feeds to GEPA’s reflection LM as obligation text.

**DSPy and GEPA integration.** DSPy [10] provides a compile abstraction for LLM programs that can host GEPA-evolved components via the `DSPyFullProgramAdapter`. We do not use DSPy here — our candidate is a single text component rather than a program — but the same certificate

mechanism could bind to DSPy compile artifacts in a follow-up experiment (Theorem 2 of the companion memo).

**Agent2Agent (A2A) protocol.** The A2A specification [1] is a JSON-RPC-2.0 interoperability layer that defines `Message` objects with polymorphic `Part` payloads. The companion memo proposes a canonical `DataPart` schema for Operon certificates. The paper-6 experiment does not exchange certificates over A2A wire, but every candidate score reported here is in principle serializable as such a `Part`.

## 3 Methods

### 3.1 Task

The target predicate is `behavioral_stability_windowed` [5], defined in `operon_ai/core/certificate.py:261:`

$$\text{holds}(\mathbf{x}, \tau) \iff \max_i x_i \leq \tau,$$

where  $\mathbf{x}$  is a vector of per-window means and  $\tau$  is a stability threshold. The obligation set is enumerable: every violating window  $(i, x_i)$  is a concrete obligation.

The task is intentionally synthetic. The harness (`eval/convergence/synthetic_signal_harness.py`) parses a single numeric knob, `policy_throttle`  $\in [0, 1]$ , from the candidate’s prompt text. It then draws  $W \cdot O$  observations from  $\mathcal{N}(\text{throttle}, \sigma^2)$  —  $W$  non-overlapping windows of  $O$  observations each — and returns the per-window means as the theorem’s `signal_values`. The default calibration is  $\tau = 0.5$ ,  $\sigma = 0.08$ ,  $W = 4$ ,  $O = 8$ , target throttle  $\approx 0.35$ . The seed prompt uses `policy_throttle = 1.0` (uniformly failing). Per-rollout RNG is seeded deterministically via SHA-256 of `(run_seed, data_inst_id)` so every batch is reproducible.

### 3.2 Arms

**cert-binary (treatment).** `OperonCertificateAdapter` bound to `behavioral_stability_windowed`. Score  $\in \{0.0, 1.0\}$ ; reflection feedback is produced by a task-specific `stability_windowed_obligation_formatter` that reads the harness trajectory and emits, in order: a theorem framing line (`Theorem: behavioral_stability_w [FAILED/HOLDS]`), a conclusion line, per-window means with `[VIOLATING]` markers on windows that exceed  $\tau$ , and a final `Unmet obligation` line. The per-window text is deliberately content-matched to the active-control arm so that the only remaining differences between cert-binary and scalar-evidence are (a) binarization of the score and (b) the theorem framing line.

**scalar-reward (baseline).** `ScalarRewardAdapter`. Score is the graded reward

$$r(\mathbf{x}, \tau) = \begin{cases} 1.0 & \max \mathbf{x} \leq \tau, \\ \tau / \max \mathbf{x} & \text{otherwise,} \end{cases}$$

which peaks at 1.0 exactly when the theorem holds, ensuring cross-arm convergence detection uses the same predicate. Reflection feedback is the single line `Score: <value>`.

**scalar-with-evidence (active control).** `ScalarWithEvidenceAdapter`. Same graded reward; reflection feedback adds a list of per-window means annotated with a `[VIOLATING]` marker wherever the window exceeds  $\tau$ , plus a final line asking the LM to lower every window mean to  $\leq \tau$ . Crucially,

this arm does *not* name the theorem (no “Theorem:” line), so the only signal it adds over scalar-reward is evidence content, not theorem framing. The arm exists to separate two confounds in a positive cert-binary result: (a) binarization, and (b) obligation-text content.

### 3.3 Convergence criterion

A run converges at iteration  $k$  iff the best candidate’s GEPA validation aggregate score equals 1.0 for three consecutive iterations starting at  $k$ . For all three arms, the best score is aggregated over a held-out validation set of 16 rollouts. Because the scalar arms peak at 1.0 only when the theorem holds (see  $r(\cdot)$  above), the threshold is the same across arms.

### 3.4 Protocol

Each cell is one  $(arm, seed)$  pair. Ten seeds per arm, three arms,  $10 \times 3 = 30$  cells. GEPA is configured with `max_metric_calls = 50 · 16 · 2 = 1600`, batch size 16 for training and 16 for validation, reflection LM `ollama/gemma4` at default settings, and a `raise_on_exception=False` policy (GEPA failures are recorded and excluded from statistics with a note).

### 3.5 Statistics

Primary: mutations-to-convergence, reported as the GEPA iteration number (1-based; GEPA’s iteration 0 is the initial evaluation, not a mutation step) at the first event of the convergence streak. Non-converged runs are imputed at `budget_iterations` (50) for the central-tendency statistics (mean iterations, Mann-Whitney U); the CDF plot (Section 4) is built *without* imputation and treats non-converged runs as right-censored, so arm curves level off at the arm’s convergence rate rather than at 1.0. Runs that raised an exception inside GEPA (`gepa_error` non-null) are excluded from all statistics and reported separately in `summary.json` under `errored_runs_excluded`. Mann-Whitney U (`alternative="less"`) compares cert-binary against each baseline. Rank-biserial correlation is reported as effect size; by convention in this paper, positive rank-biserial means the treatment dominates (fewer iterations). Convergence rates are reported with Wilson 95% confidence intervals, matching the statistical conventions of papers [4, 3].

The predeclared success criterion, matching the companion memo’s decision gate, is:

$$\frac{\bar{n}_{\text{cert-binary}}}{\min(\bar{n}_{\text{scalar}}, \bar{n}_{\text{scalar-evidence}})} \leq 0.75 \quad \text{with} \quad p < 0.05.$$

Any smaller observed effect is reported as inconclusive rather than as a win.

### 3.6 Ablation

On seeds 0–4, we rerun cert-binary with a *minimal* obligation formatter that replaces `default_obligation_format` with a single-line variant: `"constraint violated: window k"` (no evidence numbers, no theorem framing). If the minimal-formatter arm matches the default-formatter arm, the signal is in the binarization; if it underperforms, the signal is in the evidence content.

## 4 Results

All results are aggregated across 10 seeds per arm. Per-seed trajectories live at `eval/results/theorem_6/{arm}/seeds`. The analysis pipeline (`eval/convergence/analysis_theorem_6.py`) produces the machine-readable

summary at `eval/results/theorem_6/summary.json`; numbers reported here are copied verbatim from that file.

#### 4.1 Convergence rate per arm

Table 1 reports the fraction of runs that reached the predicate  $\max \mathbf{x} \leq \tau$  on the validation set within the 50-iteration budget.

Table 1: Convergence rate within 50 iterations (10 seeds per arm, Wilson 95% CIs).

Arm	Converged / N	Rate	95% CI
cert-binary	[c/10]	[0.xxx]	[[lo, hi]]
scalar	[c/10]	[0.xxx]	[[lo, hi]]
scalar-evidence	[c/10]	[0.xxx]	[[lo, hi]]

#### 4.2 Mutations-to-convergence

Table 2 reports mean iterations to convergence per arm; non-converged runs are imputed at the budget (50).

Table 2: Mean iterations to convergence per arm, with Mann-Whitney U comparison (`alternative="less"`) and rank-biserial effect size (positive = treatment dominates).

Arm	Mean iters	U	$p$	Effect size
cert-binary (treatment)	[m]	—	—	—
scalar	[m]	[u]	[p]	[rb]
scalar-evidence	[m]	[u]	[p]	[rb]

#### 4.3 Convergence CDF

Figure 1 shows the empirical cumulative distribution function of mutations-to-convergence across arms. At iteration  $k$ , each arm’s curve gives the probability that a run had converged by iteration  $k$  or earlier.

[CDF plot: `eval/results/theorem_6/cdf.png`]

Figure 1: Empirical CDF of iterations to convergence. Lines that reach 1.0 faster indicate arms that converge more reliably.

#### 4.4 Ablation: minimal vs default obligation formatter

On seeds 0–4, we compare cert-binary run with `default_obligation_formatter` against cert-binary run with a minimal formatter ("`constraint violated: window k`", no numbers, no theorem framing). Table 3 reports mean iterations for each.

Table 3: Formatter ablation on seeds 0–4 (cert-binary only).

Formatter	Mean iters
default (numbers + theorem framing)	[m]
minimal (window index only)	[m]

## 4.5 Honest verdict

[One of the three outcomes declared in the companion memo: *Strong positive* — mean-iters ratio  $\leq 0.75$  with  $p < 0.05$ ; the conjecture is accepted as Theorem 3 for this task. *Null* — ratio in  $(0.75, 1.0)$  or  $p \geq 0.05$ ; the conjecture is retained as a conjecture; ablation results inform which pipeline stage carries what signal. *Negative* — cert-binary is slower than the stronger baseline; the conjecture is refuted for this task and the memo is updated to report the refutation. ]

## 5 Discussion

### 5.1 What the arms hold constant and what they vary

The three arms share the same task, the same harness, the same reflection LM, the same GEPA configuration, and the same convergence predicate. What they vary is strictly the evaluator’s output channel:

- **cert-binary**:  $\{0, 1\}$  score plus structured obligation text derived from the theorem’s verify function’s evidence dictionary.
- **scalar**: graded score on  $[0, 1]$  peaking at theorem-holds, no feedback text beyond the score.
- **scalar-evidence**: graded score on  $[0, 1]$  peaking at theorem-holds, plus free-form text enumerating the violating windows and their means. No theorem framing.

Any observed performance difference, therefore, attaches to one of: (a) binarization of the score, (b) presence of obligation evidence text, or (c) presence of the theorem framing line that names the predicate explicitly. The active-control arm separates (a) from (b). Comparing cert-binary to scalar-evidence isolates (a) + (c): if the two tie, framing alone did not help; if cert-binary wins, (a) or (c) carries signal. The formatter ablation on seeds 0–4 further separates (b) from (b) + (c) by stripping the evidence numbers while keeping the window index, giving three grain levels of text structure.

### 5.2 Reading the outcome

[Interpretation of the observed table: which signal channel the reflection LM is most responsive to. Specifically, comment on the cert-binary vs scalar-evidence gap (isolates binarization and framing), and on the formatter-ablation gap (isolates numbers vs indices).]

### 5.3 What this does not show

The synthetic task is a *signal-channel* test, not a general-capability test. A cert-binary win on a one-knob task does not establish that certificate-based evaluators dominate on multi-component

DSPy programs, on non-enumerable failure modes, or on tasks where the reflection LM is weaker at parsing structured constraints than numeric gradients.

Our LM is Gemma 4, run locally via Ollama. Frontier models may behave differently, particularly on the scalar-evidence arm where the feedback is rich English. A cross-LM robustness follow-up is declared as out of scope here; see Section 7.1.

## 5.4 Relationship to Operon’s broader thesis

The Operon convergence layer already supports six external frameworks at the *topology* level (swarms, DeerFlow, AnimaWorks, Ralph, A-Evolve, Scion). This paper’s contribution is to extend convergence to the *evaluator* level: a single integration module (`operon_ai/convergence/gepa_adapter.py`) lets any of Operon’s eight registered theorems stand in for GEPA’s scalar reward. Whether that integration *helps* is the empirical question this paper answers. The theorem registry [5] is append-only: downstream users register a verify function once via `operon_ai.core.certificate.register_verify_fn` and the GEPA adapter picks it up automatically.

## 6 Limitations

**Single theorem.** Only `behavioral_stability_windowed` is tested. The conjecture (memo Theorem 3) applies to theorems with *enumerable* failure modes; generalization to, e.g., `behavioral_quality` (continuous-score theorems) or `no_false_activation` (structural theorems) is untested. If the primary result is positive, a follow-up sweep across at least `behavioral_quality` and `behavioral_no_anomaly` is the natural next experiment.

**Single LM.** The reflection model is fixed at Gemma 4. Frontier models may parse structured obligation text differently. A cross-LM re-run on the treatment and strongest baseline is declared as follow-up.

**Single-component candidate.** Our candidate has one mutable text slot (`policy_prompt`). GEPA’s component-selection machinery is therefore exercised trivially. Multi-component candidates — where different components fail different obligations — may change the picture.

**Synthetic task.** The harness is deliberately a one-knob task to isolate the evaluator-to-optimizer pipeline. A strong result here suggests but does not establish that real agent tasks will behave the same way; realistic tasks introduce retrieval noise, API flakiness, and agent-complexity variance that may interact with either arm.

**GEPA version coupling.** Results are reported for GEPA 0.1.1. GEPA’s adapter protocol may evolve; the `propose_new_texts` attribute, for instance, is a required member even when unused. Future GEPA versions may change the conditions under which certificate-based adapters are picked up.

**Mock-smoke and real-run separation.** The experiment driver supports a mock reflection LM for CI smoke testing. Mocked runs are flagged in the output JSON (`mock_reflection_lm: true`) and filtered by the analysis script by default. Measured results in Section 4 use real Gemma 4 only.

**No compute-adjusted comparison.** The arms use the same budget in GEPA iterations, but the reflection-LM token cost differs by arm (the cert-binary feedback is longer than the scalar arm’s single-line score). A compute-adjusted re-run, matching total reflection-LM token budget rather than iteration count, is a follow-up.

## 7 Conclusion

We tested Theorem 3 of Operon’s external-frameworks integration memo — the conjecture that certificate-based evaluators converge faster than scalar-reward evaluators on tasks with enumerable failure modes — by running GEPA with three evaluator shapes on a synthetic windowed-stability task. The cert-binary arm replaces the scalar reward with a pass/fail verdict plus structured obligation text drawn from an Operon theorem verifier. The scalar-reward and scalar-with-evidence arms form a minimal pair that isolates binarization from obligation-text content.

**[Final one-sentence verdict: accept, refine, or reject the conjecture. Mean iterations-to-convergence by arm. Effect size and p-value for cert-binary vs the stronger baseline. Formatter-ablation outcome.]**

### 7.1 Follow-ups unblocked

1. **Theorem-shape sweep:** rerun the protocol on `behavioral_quality` (continuous) and `behavioral_no_anomaly` (structural).
2. **Cross-LM robustness:** rerun cert-binary and the strongest baseline with a frontier LM.
3. **Operon–autocontext adapter:** wire Operon certificates into autocontext’s staged-validation loop [8], giving a second external framework that consumes certificates as evaluator output. A plan for this is logged separately; this is a paper-7 candidate.
4. **Lightweight DSPy reproducibility theorem:** ship `dspy_compile_pinned_inputs` as the cheap variant of memo Theorem 2.
5. **gepa\_candidate\_improvement theorem:** certify that an evolved candidate Pareto-dominates its parent on the validation set; close GEPA’s outer loop with a certificate.

### 7.2 Artifacts

- Experiment driver: `eval/convergence/theorem_6_experiment.py`
- Harness: `eval/convergence/synthetic_signal_harness.py`
- Baseline adapters: `scalar_reward_adapter.py`, `scalar_with_evidence_adapter.py`
- Analysis: `eval/convergence/analysis_theorem_6.py`
- Per-arm trajectories: `eval/results/theorem_6/{arm}/seed_*.json`
- Summary: `eval/results/theorem_6/summary.json`
- Unit tests: `tests/unit/convergence/test_paper6_*.py`



## Reproducibility.

```
pip install gepa matplotlib scipy
ollama pull gemma4 # if not already local
python -m eval.convergence.theorem_6_experiment --sweep
python -m eval.convergence.analysis_theorem_6
cd article/paper6 && tectonic main.tex
```

## A Factor-graph framing

This appendix re-expresses the gates and verifiers of Section 3 in the language of *factor graphs* [7], the formalism that underpins fixed-lag smoothing in robotics state estimation. The point is not novelty: the Operon code described here already runs and has been the subject of this paper. The point is *lineage*. A recent exposition by Dellaert frames factor graphs as a concrete, structured implementation of the energy-based “world models” currently discussed in the JEPA literature, and calls the resulting rolling-past / rolling-future loop STAG<sup>1</sup> [6]. In that vocabulary, Operon’s runtime guards are a discrete-state STAG loop over symbolic agent trajectories, and `behavioral_stability_windowed` (Section 3.1) is its past-graph smoother.

Making this correspondence explicit does two things. First, it gives the wedge artefacts of Operon — `StagnationGate`, `IntegrityGate`, and the certificate codec — a citable prior art: the analogues in SLAM have been used in production robotics for two decades. Second, it clarifies what Operon is and is not doing. It is *not* training an energy function. It is running the classical fixed-structure variant of the same loop — factor topology and residual predicates fixed in advance, solver replaced by a verifier over a discrete symbolic state. Section A.6 records where the analogy stops.

### A.1 Factor graphs in one paragraph

A factor graph is a bipartite graph with variable nodes  $x_i$  and factor nodes  $\varphi_j$ , where each factor  $\varphi_j$  is a non-negative function of its adjacent variables. The joint distribution factors as  $p(\mathbf{x} \mid \mathbf{z}) \propto \prod_j \varphi_j(\mathbf{x}_{\partial j})$  and, in energy form,  $E(\mathbf{x}) = -\log p(\mathbf{x} \mid \mathbf{z}) = \sum_j E_j(\mathbf{x}_{\partial j})$ . In the STAG presentation two graphs share a common variable  $x_t$  at the current time and a common dynamics model: a *past graph* anchored at  $x_{t-w}, \dots, x_t$  with measurement factors  $\varphi^{\text{obs}}$  from sensors and dynamics factors  $\varphi^{\text{dyn}}$  between consecutive states (fixed-lag smoothing), and a *future graph* anchored at  $x_t, \dots, x_{t+h}$  with the same dynamics and goal / objective factors (predictive control) [6]. Perception is smoothing; planning is control; both are energy minimisation.

### A.2 Operon gates as a discrete STAG loop

We translate the ingredients of Section 3 term by term. Each paragraph tags its claim as either **[implemented]** — the cited code already discharges the mapping directly, **[derived]** — the mapping follows from existing behaviour but is not surfaced at the named line, or **[analogy]** — the structure lines up but no single code path instantiates it; this is a description layered on top. The validation surface per claim is summarised in Section A.3.

---

<sup>1</sup>Sense-Think-Act with Graphs.

**Variables.** [analogy] The latent variable  $x_t$  is the *abstract agent state at stage  $t$*  — the tuple (genome, expression, short-term memory) tracked by the Operon runtime between stages. This is not the LangGraph GuardedState dict (operon\_ai/convergence/guarded\_graph.py:59), which also carries compile-time plumbing (messages, stage\_outputs, retry\_count, watcher\_state); those fields are part of the runtime envelope, not of the factor-graph reading.  $x_t$  is symbolic, not Euclidean.

**Measurement factors.** [implemented] A measurement factor is a non-negative function of  $x_t$  and an observation  $z_t$ ; high values indicate consistency. In Operon:

- For `behavioral_stability_windowed`, the observation at stage  $t$  is the per-window mean  $x_t^{(\text{obs})}$  computed by the verifier at `operon_ai/core/certificate.py:261`, and the residual is  $r_t = \max(0, x_t^{(\text{obs})} - \tau)$  where  $\tau$  is the stability threshold. The factor is  $\varphi_t^{\text{obs}} = \mathbb{K}[r_t = 0]$ . A violating window contributes a zero factor; the smoothed belief — “the system is stable over the last  $W$  windows” — is the product  $\prod_{i=t-W+1}^t \varphi_i^{\text{obs}}$ , which equals  $\mathbb{K}[\max_i x_i \leq \tau]$  and recovers exactly the predicate of Section 3.1.
- For `DNARepair.scan()` (`operon_ai/state/dna_repair.py:268`), the observation is the current (hash, gene\_count, expression, required\_genes) and the residual is the number of damage reports emitted. Each of the five scans — checksum (§291), gene-value (§307), gene-count (§335), expression (§351), and required-gene presence (§381) — corresponds to one measurement-factor component; their conjunction is the integrity certificate.

**Dynamics factors.** [implemented] A dynamics factor  $\varphi_t^{\text{dyn}}$  is a non-negative function of  $(x_{t-1}, x_t)$  that encodes which transitions are admissible. In Operon, the admissible transition from  $x_{t-1}$  to  $x_t$  is recorded by the checkpoint written in `StateCheckpoint`. The dynamics-residual at stage  $t$  is the mismatch between what the checkpoint predicts and what `DNARepair.scan` sees at  $t$ ; this is the “DSB analog” (checksum mismatch) in `dna_repair.py:292–305`. Dynamics factors are thus not learned; they are deterministic consistency factors driven by the genome-hash invariant.

**Pre-guard = repeated invariant re-validation.** [implemented] `compile_guarded_graph` (`operon_ai/convergence/guarded_graph.py:93`) installs a `pre_guard` node before every LLM stage. The pre-guard runs `DNARepair.scan(genome, checkpoint)` and halts the run if any damage is detected (lines 187–199). In factor-graph language: each stage evaluates the measurement-factor conjunction at a single  $x_t$  against a *static* checkpoint baseline and routes to HALT if any residual is positive. This is deliberately *not* a rolling past-graph smoother: there is no per-stage checkpoint evolution, no  $W$ -step window over evolving states, and no MAP re-estimate across stages. The caller-supplied `StateCheckpoint` is plumbed once into `compile_guarded_graph` (lines 524–535) and closed over inside `make_pre_guard`; the same object is then re-checked by the pre-guard before every stage, and retry routing sends control back through the pre-guard (lines 186–199, 448). The factor-graph reading of the pre-guard is therefore “repeated invariant re-validation against a fixed baseline,” and a genuine past-graph smoother — with per-stage checkpoint advancement and a bounded-lag window — is a deliberate non-goal at this scope (see Section A.6). The windowed *verifier* (Section A.2, Measurement factors) is what plays the smoother role; the pre-guard is its one-step invariant counterpart in the compiled runtime.

**Future graph = post-guard.** [implemented] The `post_guard` in `compile_guarded_graph` evaluates the stage output against a rubric (`_evaluate_quality`, line 248), threads a simplified

watcher state, and routes `RETRY` / `ESCALATE` / `HALT` via `_route_intervention` (line 300). The  $h$ -step future graph collapses to a single goal factor “quality  $\geq$  rubric-threshold”; if it fails, the conditional router fires. This is the degenerate but operationally honest version of a STAG future graph: horizon  $h = 1$ , goal expressed as a discrete rubric predicate, no cross-step control variables. The rubric is called as `rubric(output, stage_name)` — it does not read the standing checkpoint.

**Certificates are a separate channel.** **[implemented]** Behavioural and structural certificates are carried in the *compiled organism artefact*, not emitted at run time by the post-guard. `run_guarded_graph` iterates `compiled.get("certificates", [])` after the graph finishes and re-verifies each one via `certificate_from_dict` (line 557–572), populating `GuardedGraphResult.certificates_verified`. The factor-graph reading therefore has two distinct surfaces: the compile-path post-guard that produces the routing decision (above), and a post-run replay of compile-time certificates that attests preservation-under-compilation rather than attaching a fresh behavioural factor. This split is the right place to pressure-test the “future graph” framing: the goal factor realised in the compiled graph is a rubric predicate, while the certificate layer is a compile-time artefact that the runtime only *replays*.

**Shared dynamics.** **[analogy]** STAG’s past- and future-graph share a dynamics model. In Operon, the object that plays the closest role is the standing `StateCheckpoint`, but only for the pre-guard: the pre-guard consults it for integrity, the post-guard does not (the rubric call does not take `checkpoint` as an argument). The category-theoretic machinery in `operon_ai/core/coalgebra.py` formalises discrete dynamics for Operon in the abstract — readout / update / bisimulation — and the compile path does not import it. So this mapping is a loose conceptual anchor, not a shared runtime object: both guards live in a world whose dynamics model is discrete and deterministic, but the compile path does not thread a single dynamics object through both. Flagging this so downstream work does not build on a shared-object or coalgebra/compile coupling that does not exist.

**Joining factor graphs across agents.** **[analogy]** In the continuous STAG loop, two robots with overlapping trajectories can join their graphs by sharing the variable nodes their observations agree on. The Operon code that plays a superficially similar role is the A2A certificate codec (`operon_ai/convergence/a2a_certificate.py`): it encodes certificates as `DataPart` payloads, decodes them on receipt, and implements a graceful-degradation rule for unknown theorems (§3.3 of [docs/site/external-frameworks.md](#)). What the code does *not* do is maintain an internal factor graph, attach received factors to any shared-variable state, or perform graph-level composition or joint inference. A receiver either verifies a single certificate locally or forwards it; there is no “my graph, your graph, joined along a shared variable” surface anywhere in the codebase. So the mapping here is analogy only — transport plus schema agreement, not graph joining. Downgrading to **[analogy]** so readers do not treat the transport layer as evidence of implemented cross-agent graph composition semantics. The monoidal-category structure Operon’s optimiser already exposes (`operon_ai/core/optimizer.py`, endofunctor composition) is the category-theoretic statement that certificate joining is associative.

### A.3 Claim-validation table

Each mapping in Section A.2 is tagged **[implemented]**, **[derived]**, or **[analogy]**. The table below names, for each **[implemented]** and **[derived]** mapping, the owning code path and the artefact that validates it. The **[analogy]**-tagged mappings are descriptive and have no validation artefact

by design; they are included for completeness and flagged so downstream work does not treat them as code-level contracts.

Mapping	Owning code path	Validation artefact
Variables	N/A (abstract)	— <i>[analogy]</i>
Measurement factors (behavioral_stability_windowed)	core/certificate.py:261	Unit test tests/unit/convergence/test_paper6_*.py; Section A.4
Measurement factors (DNARepair.scan)	state/dna_repair.py:268 (5 scans, §291, 307, 335, 351, 381)	Unit tests in tests/unit/state/test_dna_repair*
Dynamics factors	state/dna_repair.py:292–305 + StateCheckpoint	Integrity certificate on pre- guard (Paper 4 §4.1)
Pre-guard = repeated invariant re-validation (static checkpoint; no rolling window)	convergence/guarded_graph.py:93, 186–199, 448, 524–535	Pre-guard halts on damage; Paper 4 §4.1
Future graph = post- guard (rubric routing)	convergence/guarded_graph.py:248, 300	Conditional-router decision trace in the compiled graph
Certificates are a sepa- rate channel (post-run replay of compile-time certificates)	convergence/guarded_graph.py:554– 572 (compiled.get("certificates") + certificate_from_dict)	GuardedGraphResult.certificates_verified; this attests preservation- under-compilation, not runtime emission
Shared dynamics (pre-guard only via StateCheckpoint; no coalgebra coupling; rubric call (output, stage_name) ignores checkpoint)	—	— <i>[analogy]</i>
A2A transport of cer- tificates (not graph joining)	convergence/a2a_certificate.py	A2A DataPart round-trip + graceful-degradation (§3.3, external-frameworks.md); <i>[analogy]</i> — no internal- graph join is implemented

**Reproducibility prerequisites.** Claims split into three reproducibility tiers:

- **No dependencies.** The worked example in Section A.4 runs purely from the harness `run_rollout` function (`eval/convergence/synthetic_signal_harness.py:192`). No LM, no LangGraph, no model server. Standalone verification of a constructed `Certificate` via `certificate_from_dict` + `verify()` also lives in this tier.
- **LangGraph optional-import.** The [implemented] guarded-graph claims require `pip install operon-ai[langgraph]`: `compile_guarded_graph` raises `ImportError` at lines 133–140 without it, and `run_guarded_graph` does the same at line 517–522. Consequently the post-run certificate-replay claim (“Certificates are a separate channel”) is in *this* tier, not the no-dependencies tier: it is reached via `run_guarded_graph.certificates_verified` (lines 554–572), and therefore needs LangGraph.
- **LangGraph + LM.** End-to-end execution of the pre- / post-guard routing needs a live or mock LM (Ollama or equivalent) in addition to LangGraph. The appendix does not depend on any claim in this tier.

**Maintenance checklist (documentation-drift contract).** This appendix cites exact line numbers. On any change to the files below, a reviewer must verify or retag:

- `operon_ai/core/certificate.py`:261 — verifier signature and `signal_values` contract; re-build PDF if signature changes.
- `operon_ai/state/dna_repair.py`:268, 291, 307, 335, 351, 381 — five scan components; retag row as `[analogy]` if any component is removed.
- `operon_ai/convergence/guarded_graph.py`:93, 186–199, 248, 255, 300, 448, 524–535, 554–572 — pre-guard, rubric call signature, retry routing, checkpoint plumbing, post-run certificate replay. If `run_guarded_graph` begins emitting fresh certificates from the post-guard, the “certificates are a separate channel” paragraph must be rewritten.
- `operon_ai/convergence/a2a_certificate.py` — if any internal graph-join API is added, the “Joining factor graphs across agents” row may be re-tagged from `[analogy]` to `[derived]` or `[implemented]`.

The contract is *retag-or-rewrite*: a drifted anchor does not block the build, but the corresponding row or paragraph must be re-tagged as `[analogy]` until a reviewer has re-verified the code.

## A.4 Worked example

We replay the harness of Section 3.1 deterministically on (`data_inst_id=0`, `run_seed=0`) at default calibration ( $W = 4$ ,  $\tau = 0.5$ ,  $\sigma = 0.08$ ,  $O = 8$ ). The harness seed is SHA-256 of (`run_seed`, `data_inst_id`), so these values are reproducible with no LM in the loop (`eval/convergence/synthetic`).

**Before convergence** — `policy_throttle = 0.55`. The harness draws per-window means

$$\mathbf{x}^{(\text{obs})} = (0.5769, 0.5653, 0.4934, 0.5172).$$

The past-graph smoother of this appendix attaches four measurement factors

$$\varphi_i^{\text{obs}} = \mathbb{K}[x_i^{(\text{obs})} \leq \tau], \quad i = 0, 1, 2, 3,$$

with residuals  $r_i = \max(0, x_i^{(\text{obs})} - \tau) = (0.0769, 0.0653, 0, 0.0172)$ . Three factors are zero, so the smoothed predicate  $\prod_i \varphi_i^{\text{obs}} = 0$ : the certificate does not hold. The obligation formatter of Section 3.2 serialises exactly the windows  $i \in \{0, 1, 3\}$  with positive residuals — these are the “[VIOLATING]” entries in the reflection feedback handed to GEPA. The `scalar-with-evidence` arm sees the same per-window evidence line but without the theorem-framing prefix.

**After a reducing mutation** — `policy_throttle = 0.42`. Replaying with the same instance under a smaller throttle gives

$$\mathbf{x}^{(\text{obs})} = (0.4469, 0.4353, 0.3634, 0.3872),$$

residuals  $(0, 0, 0, 0)$ , and  $\prod_i \varphi_i^{\text{obs}} = 1$ . The certificate holds on this rollout. Convergence (Section 3.3) is the first iteration whose best candidate has `holds = 1` for three consecutive aggregate-over-16 validation evaluations. Seed 0 of the `cert-binary` arm reaches that streak at iteration 4; the first three iterations have aggregate score 0.0 and iterations 4, 5, 6 all score 1.0 (`eval/results/theorem_6/cert-binary`).

## A.5 What this framing buys

1. **Citation lineage.** The pre-/post-guard pattern is no longer a novel construct; it is the symbolic-state analogue of the rolling-past / rolling-future loop that has been standard in robotics state estimation since [9]. The LangGraph-gates wedge’s theoretical-basis paragraph can cite this directly.
2. **Certificate composition has the *shape* of factor joining.** The A2A codec is transport plus graceful degradation; under this framing it reads as the transport-layer analogue of graph joining along shared theorem variables, not an implemented graph-join operation (see Section A.2, “Joining factor graphs across agents”, tagged [analogy]). What the framing gives us is a common vocabulary for cross-agent reliability — model-agnostic agents still share a factor-structured certificate shape — not a semantic claim about joint inference.
3. **A vocabulary for future scope.** Any future extension — longer pre-guard windows,  $h > 1$  future graphs, learned residuals — can be stated and bounded in this vocabulary. The scope-discipline rule we adopt here is: *factors and topology fixed; only the set of theorems grows*.

## A.6 Where the analogy stops

We record the load-bearing disanalogies so later work does not overclaim.

- **No metric on state.** Symbolic agent state has no inner product. “Energy” in our usage is a residual count, not a differentiable potential; the gradient step behind MAP smoothing in SLAM has no counterpart. Solving is replaced by verification.
- **No learned factors.** Every factor in the mapping above is a deterministic predicate defined in code. The moment a factor is fit to data, the guarantee flips from structural to empirical. This is the Paper 3 lesson ([3] §“evolution did not beat random mutation”): the structure held, the learning did not.
- **Future graph is degenerate.** Operon’s post-guard has horizon  $h = 1$ . A non-degenerate future graph would plan multiple stages ahead and fold goal factors into the run, which is the direction of energy-based agent planning we deliberately do not pursue.
- **No variable elimination.** SLAM’s speed comes from eliminating variables in topological order. Operon’s guards verify one certificate at a time; there is no dual benefit from the structure, only a dual *description*.

The framing is load-bearing for *how we talk about* the gates, not for how they are computed. That is the correct line for a structural-guarantees paper, and the line this appendix commits to.

## References

- [1] A2A Project. Agent2agent (A2A) protocol. <https://github.com/a2aproject/A2A>, 2025. Open cross-vendor agent-communication protocol; JSON-RPC 2.0 over HTTP+SSE.
- [2] Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziemis, et al. GEPA: Reflective prompt evolution can outperform reinforcement learning. <https://github.com/gepa-ai/gepa>, 2025. Open-source reflective prompt optimizer, v0.1.1.
- [3] Bogdan Banu. Convergence by composition: A structured adapter architecture for multi-agent system integration, 2026. Preprint.
- [4] Bogdan Banu. Do biological structural guarantees earn their complexity? empirical benchmarks for biologically-inspired agent reliability, 2026. Preprint.
- [5] Bogdan Banu. Operon: Biomimetic wiring diagrams for robust agentic systems. <https://github.com/coredipper/operon>, 2026. Open-source framework for biology-inspired agent control patterns.
- [6] Frank Dellaert. Factor graphs and world models. <https://gtsam.org/2026/04/21/factor-graphs-and-world-models.html>, 2026. Frames factor graphs as a concrete structured instance of energy-based world models; introduces the STAG past-graph / future-graph rolling loop.
- [7] Frank Dellaert and Michael Kaess. *Factor Graphs for Robot Perception*, volume 6 of *Foundations and Trends in Robotics*. now Publishers, 2017. Standard reference for factor-graph formulations of SLAM, smoothing, and MAP estimation.
- [8] greyhaven-ai. Autocontext: Multi-agent knowledge-accumulation runtime. <https://github.com/greyhaven-ai/autocontext>, 2025. Staged-validation agent loops with persistent playbooks across generations.
- [9] Michael Kaess, Hordur Johannsson, Richard Roberts, Viorela Ila, John J. Leonard, and Frank Dellaert. iSAM2: Incremental smoothing and mapping using the Bayes tree. *International Journal of Robotics Research*, 31(2):216–235, 2012. Incremental fixed-lag smoothing over factor graphs; production state-estimation baseline.
- [10] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into self-improving pipelines. <https://github.com/stanfordnlp/dspy>, 2024. Programming-not-prompting framework for LM pipelines.
- [11] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers, 2024. OPRO: prompt optimization via LLM-generated gradient-like updates.
- [12] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers, 2022. APE: automatic prompt generation via LLM candidate pool + scoring.