From Trajectories to Graphs: Contract-Checked Editing for Verifier-Guided LLM Reasoning

Rui Li* **Shuang Cao***
Hill Research
{rui.li, sharon.cao}@hillresearch.ai
* Equal contribution

Abstract

Inference-time search can substantially improve LLM reasoning when tasks admit deterministic verification, but existing methods largely refine single trajectories and lack a reliable mechanism for composing partial solutions across candidates. We propose *contract-checked graph editing*: represent each candidate as an interface-typed reasoning DAG and validate every nontrivial edit with a deterministic structural gate (acyclicity, namespace closure, schema validity, terminal constraints) before invoking the verifier. The gate certifies *runnability* only and emits auditable rejection reasons; semantic correctness is determined solely by the verifier.

Instantiated in *Genetic Inference Search (GIS)* with Qwen2.5-32B-Instruct under strictly matched token budgets (8K tokens), contract-checked grafting increases verifier-runnable recombination from 41.2% to 92.8% and improves accuracy over rStar (+6.1 on MATH, +9.1 on MATH L5) while using 42% fewer verifier calls. The same operators transfer across outer loops (beam, best-first, MCTS) and to structured generation and code, outperforming execution-guided beam search on Spider (+2.8) and improving multi-file code generation on HumanEval-MF (+9.2).

1 Introduction

Inference-time search improves large language model (LLM) reasoning by spending compute at test time, especially when tasks admit deterministic verification (exact answers, unit tests, or constraint checkers). This paradigm is central to building reliable LLM agents that must produce correct, executable outputs.

Most existing approaches either (i) improve a single trajectory through iterative refinement or (ii) explore many trajectories without a robust mechanism to *recombine* partial solutions. Recombination is appealing: if two candidates each contain a useful subproof or subroutine, a search procedure should transplant the useful piece rather than rediscover it. In practice, however, naive text-level reuse is brittle under strict verifiers. Hidden dependencies—variable scope, implicit definitions, tool I/O contracts, module imports—are not explicit in free-form text, so crossover often creates malformed offspring that are not even runnable. We call this the *crossover barrier*.

Core idea: certificate-gated structural editing. We propose *certificate-gated structural editing*: represent each candidate as an interface-typed reasoning DAG and treat edits as first-class objects. Every nontrivial edit must pass a deterministic structural gate *before* any expensive verification. The gate checks acyclicity, namespace closure, schema validity, and terminal constraints, and emits a compact certificate with deterministic acceptance/rejection reasons. The gate certifies runnability only; semantic correctness is determined solely by the external verifier. This separation makes recombination *auditable*: we can measure verifier-runnable rates (Table 2), diagnose failure causes, and filter invalid offspring before running the verifier.

Key results (all in main text). Under strictly matched token budgets (8K tokens) that charge all LLM calls: SemanticGraft removes the crossover barrier (92.8% GVR vs. 41.2% for string-merge; Table 2). GIS+Cert++ improves accuracy over rStar, the strongest baseline (MATH +6.1, L5 +9.1; Table 1). Adding the same deterministic gate to strong baselines improves verifier efficiency but does not close the accuracy gap, confirming that interface-compatible crossover is essential (Table 3). End-to-end, GIS uses 42% fewer verifier calls and 16% less wall-clock under matched token budgets. The primitive transfers to structured NLP and code: Text-to-SQL on Spider (+2.8 vs. execution-guided beam) and unit-test-verifiable code (HumanEval +6.9, HumanEval-MF +9.2) (Ta-

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 43259–43306

ble 4).

Outer-loop agnostic. While we present GIS (Genetic Inference Search) as a concrete instantiation, the primitive is outer-loop agnostic: the same operators yield consistent gains under beam, best-first, and MCTS (54.8–56.2% vs 56.8% for GIS on L5@B_3; see supplementary material for the full table).

Scope. We focus on verifiable reasoning with strict, deterministic verification. We do not claim that our approach applies unchanged to open-ended tasks without deterministic verification.

Contributions. (1) Interface-typed reasoning DAGs: a JSON-serializable IR with explicit `Req`/`Prod`/`Bnd` interfaces. (2) Certificate-gated structural editing: an outer-loop-agnostic recombination primitive with a deterministic gate, auditable rejection reasons, and diagnostics (GVR/CUR/CertPass/PatchSz). (3) GIS + Cert++: topology-aware grafting with bounded repair; Cert++ adds checkable preconditions. (4) Decision-grade evaluation: strict equal-budget comparisons with gate-control experiments, cost analyses, and portability to multi-file code.

2 Related Work

We position GIS at the intersection of inference-time search for LLM reasoning, structured intermediate representations, evolutionary computation, and verification. Our aim is to state precisely what existing lines can and cannot do, and why proof-/certificate-carrying crossover is a missing primitive for verifiable LLM reasoning.

Inference-time sampling, reranking, and deliberation. Chain-of-Thought prompting elicits multi-step reasoning in large language models (Wei et al., 2022), including in zero-shot form (Kojima et al., 2022). Self-consistency and best-of-N improve reasoning by sampling multiple chains and aggregating/selecting candidates (Wang et al., 2023). More explicit exploration can be achieved via tree-structured search such as Tree-of-Thoughts, MCTS-style methods, and related budget-aware prompting protocols (Yao et al., 2023a; Song et al., 2025; Zhang et al., 2024a; Li, 2025; Li et al., 2026c). However, these approaches largely treat candidates as sequences or trees and typically lack a principled, invariant-preserving mechanism to recombine partial solutions across independently generated candidates.

Graph-structured prompting and reasoning organization. Graph/diagram-based prompting organizes reasoning into structured artifacts, improving controllability and reducing redundancy. Graph-of-Thoughts generalizes reasoning from chains/trees to graphs over intermediate thoughts (Besta et al., 2024). Other approaches explicitly couple LLMs to external graphs, e.g., knowledge-graph traversal and retrieval (Sun et al., 2023), or graph neural prompting that injects structured graph representations into LLM inputs (Tian et al., 2024). Graph Chain-of-Thought further unifies iterative graph operations with multi-step reasoning (Jin et al., 2024a). While these methods introduce useful structure, they typically do not define genetic operators with hard interface constraints, nor do they directly address the crossover barrier: splicing sub-proofs without breaking scope, dependencies, or tool contracts.

Inference-time scaling via evolutionary search and iterative refinement. A growing line of work studies inference-time scaling by repeatedly proposing, selecting, and improving candidates—often combining population-based search with critique or verification signals. For example, Lee et al. (2025) explores evolutionary-style search over reasoning traces, while surveys of evolutionary algorithms with LLMs highlight a broad design space spanning mutation, recombination, and evaluator-shaped selection (Wang et al., 2025). Complementary refinement paradigms iteratively critique and revise a solution (Madaan et al., 2023; Shinn et al., 2023), improving accuracy without training. These approaches motivate strong baselines for GIS: iterative revision loops and population-based search under matched budgets. GIS is distinguished by making recombination itself reliable under strict verifiers through typed interfaces and certificate-gated structural validity, rather than relying on text-level merges or purely heuristic recombination.

Evolutionary computation for LLM outputs. Recent work explores evolutionary ideas for LLM outputs, evolving prompts or reasoning traces with mutation and heuristic merge/crossover (Guo et al., 2023; Griesshaber et al., 2025; Jin et al., 2024b; Lee et al., 2025). However, string-level evolution inherits the crossover barrier because dependencies (variables, definitions, tool I/O) are implicit and untyped; crossover offspring often become invalid on tasks with strict verifiers or tool execution. GIS differs by evolving an interface-typed reasoning IR and by making crossover structurally auditable via

certificates checked deterministically before verification.

Verifier-guided refinement, execution-guided search, and programmatic rewards. Execution and unit tests provide reliable signals for verifiable tasks, and execution-guided inference-time methods are common. Program-aided generation offloads computation to reliable executors (Gao et al., 2023; Chen et al., 2022), and tool-augmented prompting/agents interleave reasoning with actions and environment feedback (Yao et al., 2023b; Schick et al., 2023; Shen et al., 2023). Iterative self-critique and revision can further improve correctness (Madaan et al., 2023; Shinn et al., 2023). Verifier-in-the-loop paradigms include explicit verifier training (Cobbe et al., 2021) and reinforcement learning with verifiable rewards (Guo et al., 2025; Zhang et al., 2024b); adjacent work such as AEGIS studies constraint-aware proposal/commitment under explicit constraints (Li et al., 2026b). GIS uses verifiers as a landscape shaper but, crucially, separates structural validity from semantic correctness: a fast deterministic checker validates invariants and filters invalid crossover offspring before expensive verification. In our experiments, we therefore include verifier-guided refinement baselines (critique→rewrite loops and execution-guided repair) under identical token accounting to isolate the benefit of certificate-gated recombination.

Typed IRs, certified artifacts, and structured verification. Work on structured or typed rationales connects LLM reasoning to formal verification (Perrier, 2025; Li et al., 2025b; Kamran et al., 2024; Ma et al., 2025). Surveys catalog a broader set of Chain-of-X paradigms that introduce explicit checking or structured intermediate steps (Xia et al., 2024; Chu et al., 2023; Dhuliawala et al., 2023). Concerns about faithfulness of free-form rationales further motivate independently checkable artifacts (Turpin et al., 2023). Related application systems also use auditable evidence ledgers to bind model outputs to verifiable evidence states under evolving sources (Li et al., 2026a). GIS imports compiler-style invariants to inference-time recombination of LLM reasoning artifacts by requiring proof-/certificate-carrying summaries for structural verifiability prior to running the expensive external verifier.

Genetic programming and typed program crossover. Classical genetic programming (GP) defines crossover over syntax trees with fixed grammars (Koza, 1992), and strongly-typed GP extends this to typed programs where crossover exchanges type-compatible subtrees (Montana, 1995). Proof-carrying code attaches machine-checkable certificates to executables (Necula, 1997). At a broader systems level, compiler/HLS work similarly relies on explicit IRs to support legality-preserving transformations and resource-sharing rewrites over complex control/dataflow structure (Li et al., 2021, 2025a). Key distinction: these systems operate on *fixed*, *a priori* IRs with stable type systems. In LLM reasoning, there is no pre-existing IR—candidates are generated as free-form text with implicit structure. Our contribution is to *dynamically extract* typed interfaces from LLM-generated reasoning graphs and define *edit-level* certificates that make cross-candidate recombination auditable and measurable (GVR/CUR/CertPass/PatchSz), enabling falsifiable claims about crossover success under strict verifiers.

Our position. GIS differs from prior work in a specific, testable way: we treat recombination as a compiler-style transformation over a typed reasoning IR and require a certificate validated deterministically before verification. This makes crossover feasible, auditable, and budget-relevant under strict verifiers and equal-budget evaluation. We are largely orthogonal to improvements in prompting, proposal generation, or verifier strength; our contribution is to make crossover a reliable and measurable primitive for inference-time scaling on verifiable reasoning tasks.

3 Problem Setup

We study inference-time optimization for tasks that admit deterministic verification of candidate solutions (exact answers, unit tests, or constraint checkers). A single test instance is denoted by x and a base model by M.

3.1 Reasoning Graphs

Definition 1 (Reasoning Graph)**.** A reasoning graph is a directed acyclic graph (DAG) $G = (V, E)$ where each node $v \in V$ is a typed semantic state and each directed edge $(u \to v) \in E$ indicates that v depends on u.

Node schema. We use a finite node-type alphabet $\mathcal{T}$ (extensible per domain), e.g., `Input`, `Assumption`, `Lemma`, `Derivation`, `ToolCall`, `Code`, `Answer`, `Check`. Each node stores `content` (text/code), optional structured `meta` (e.g., free/bound symbols,

namespace, tool I/O schema, traces), and a unique `id`. Edges may carry a `rule` label (e.g., algebraic transform, case split, invocation).

I/O partition. We partition $V = V_{\text{in}} \cup V_{\text{inter}} \cup V_{\text{out}}$ into source nodes (problem statement and fixed context), intermediate nodes (deductions/tools/claims), and output nodes (final answer and/or executable artifact).

3.2 Serialization, Linearization, and Patch Size

Serialization. Each individual is serialized as a JSON-DAG enabling local patches (node/edge edits) and subgraph transplantation (schema details are provided in the Appendix).

Deterministic linearization for prompting. Given a DAG G, we define a deterministic linearization $\mathcal{L}(G)$ used to present context to the LLM (topological order with stable tie-breaks). Validity C and semantic evaluation $\mathcal{V}$ operate on the underlying DAG and are independent of node ordering.

Patch size (local vs rewrite). We quantify edit locality using PATCHSZ, the number of node edits plus edge rewires in the applied JSON patch (edit operations). Counting rules and scope-damage diagnostics are provided in the Appendix. Patch locality is central to recombination: destructive crossover typically induces large uncontrolled changes, whereas successful transplantation should be local.

3.3 Cost Model: Tokens and Verifier Calls

Token cost. Let $\mathsf{tok}(G)$ denote the total LLM token usage incurred to produce G (initial generation + operator calls + repairs + certificates + any judging/critic prompts). We separately log non-token costs such as verifier execution counts (and wall-clock when available).

Budget. Each inference-time run operates under a per-instance budget B (token cap; optionally also a verifier-call or wall-clock cap). All methods must respect identical token accounting rules (every LLM call counts). Certificate validation is deterministic and non-LLM, serving as a cheap gate that reduces wasted verifier executions.

3.4 Verifier, Fitness, and Structural Validity

Verifier. Let $\mathcal{V}$ be a deterministic verifier mapping a candidate graph to an outcome and diagnostics:

$$(\mathsf{status}(G), \mathsf{diag}(G)) = \mathcal{V}(G),$$

where $\mathsf{status}(G) \in \{\texttt{NONRUNNABLE}, \texttt{EXEC\_FAIL}, \texttt{EXEC\_OK}, \texttt{PASS}\}$ for code-verifiable tasks, and analogously $\{\texttt{NONRUNNABLE}, \texttt{FAIL}, \texttt{PASS}\}$ for direct-answer and execution-verifiable text tasks (e.g., SQL). `NONRUNNABLE` denotes failures that prevent semantic evaluation, such as parse/compile/import errors, schema violations, or terminal-format/canonicalization failures. In our benchmarks, this includes answer-format/canonicalization failures on MATH/ARC/GPQA, SQL parse/schema errors on Spider, and parse/import/signature violations on code tasks.

Stepped fitness (Python Cliff). For code-verifiable tasks we define a stepped fitness:

$$F(G) = \begin{cases} 1.0 & \text{if } \mathsf{status}(G) = \texttt{PASS}, \\ 0.5 & \text{if } \mathsf{status}(G) = \texttt{EXEC\_OK}, \\ 0.1 & \text{if parses but } \mathsf{status} = \texttt{EXEC\_FAIL}, \\ 0.0 & \text{otherwise.} \end{cases}$$

For direct-answer tasks, $F(G) = \mathbb{I}[\mathsf{status}(G) = \texttt{PASS}]$.

Structural validity as a first-class constraint. A core difficulty in recombination is that many offspring are not even verifier-runnable due to scope, schema, or dependency violations. We therefore separate structural validity from semantic correctness by introducing a deterministic invariant checker:

$$C(G) \in \{\texttt{ACCEPT}\} \cup \{\texttt{REJECT}(\text{namespace/schema/cycle/term.})\}.$$

The checker validates acyclicity, namespace closure, tool/code schema validity, and terminal well-formedness. Determinism boundary: C certifies only well-formedness (runnability/invariant validity), not semantic correctness.

3.5 Interface Signatures and Certificate-Gated Recombination

Interface signatures (subgoal interfaces). For any connected subgraph $S \subseteq G$, define an interface signature

$$\Sigma(S) = (\mathsf{Req}(S), \mathsf{Prod}(S), \mathsf{Bnd}(S)),$$

where $\mathsf{Req}(S)$ are required (free) symbols/premises referenced in S but not defined in S, $\mathsf{Prod}(S)$ are exported products, and $\mathsf{Bnd}(S)$ is a typed boundary descriptor capturing boundary node types and any tool/code I/O schema constraints on the cut. In our implementation, $\Sigma(S)$ is extracted deterministically from node metadata (`defs`/`uses` and tool/code I/O schemas); full rules and noise analyses are provided in the Appendix.

Compatibility. Let S_A be the recipient cut-site and S_B a donor subgraph. We say $\Sigma(S_B)$ is *compatible* with $\Sigma(S_A)$, written $\mathsf{Compat}(\Sigma(S_A), \Sigma(S_B))$,

if: (i) all required items of the donor are provided by the recipient context at the cut, $\mathsf{Req}(S_B) \subseteq \mathsf{Ctx}(S_A) \cup \mathsf{Prod}(S_A)$; and (ii) boundary schemas/types match, $\mathsf{Bnd}(S_B) \preceq \mathsf{Bnd}(S_A)$, where $\preceq$ denotes schema/type compatibility (e.g., compatible tool/code signatures at the boundary). Formal definitions of $\mathsf{Ctx}(\cdot)$ and $\preceq$ are provided in the Appendix.

Example (boundary-compatible graft). In code tasks, a cut-site that exports `solve` with signature `solve(int)→int` and uses an in-scope symbol n yields $\mathsf{Req} = \{n\}$, $\mathsf{Prod} = \{\texttt{solve}\}$, and a boundary schema recording `solve(int)→int`. A donor that exports `solve` with the same boundary schema and requires only n is compatible; α-renaming plus bounded repair then restores namespace closure before verification.

Interface certificate (auditable crossover artifact). A recombination step outputs, in addition to the offspring graph G', a compact certificate object:

$$\mathsf{Cert} = \{\, \Sigma(S_A), \Sigma(S_B), \mathsf{Align}, \pi, \\ \mathsf{Checks}, \mathsf{Unresolved} \,\},$$

containing boundary signatures, an alignment map Align, a namespace mapping π, and recorded checker outcomes. We require $\mathsf{Compat}(\Sigma(S_A), \Sigma(S_B))$, $\mathsf{Unresolved} = []$, and $C(G') = \texttt{ACCEPT}$ for the offspring to be eligible for verification. This makes recombination *auditable*: certificate pass rates and rejection reasons are measurable and reproducible.

3.6 Objective and Output Policy

Given M and an instance x, the inference-time optimizer returns a best-effort solution graph $G^\star$ under budget B. We treat inference-time optimization as multi-objective (fitness, cost, and edit locality) and report Pareto summaries; the formal objective and tie-breaking policy are provided in the Appendix.

4 Certificate-Gated Structural Editing

Primitive. We represent each candidate as an interface-typed reasoning DAG and treat edits as first-class objects. An edit is eligible for expensive verification only if it passes a deterministic structural gate enforcing acyclicity, namespace closure, schema validity, and terminal well-formedness. The gate certifies runnability only; semantic correctness is determined solely by the external verifier.

Interfaces and deterministic checking. Given a connected subgraph S, we compute an interface signature $\Sigma(S) = (\mathsf{Req}(S), \mathsf{Prod}(S), \mathsf{Bnd}(S))$ and validate the edited graph with a deterministic checker C (Problem Setup, §3.5). Interface extraction and checker rules are deterministic, fully specified in the Appendix, and include a *Compat-lite* fallback for noisy symbolization.

Proposition 1 (Runnability certificate (no wasted NONRUNNABLE verifier calls))**.** Let $\mathrm{Render}(G)$ deterministically materialize a candidate DAG G into a verifier input y (answer string, executable code bundle, or SQL query). Let the verifier harness $\mathcal{V}$ return a status in $\{\texttt{NONRUNNABLE}\} \cup \mathcal{S}_{\text{sem}}$, where $\mathcal{S}_{\text{sem}} = \{\texttt{FAIL}, \texttt{PASS}\}$ for direct-answer / SQL tasks and $\mathcal{S}_{\text{sem}} = \{\texttt{EXEC\_FAIL}, \texttt{EXEC\_OK}, \texttt{PASS}\}$ for code-verifiable tasks. NONRUNNABLE denotes failures that prevent semantic evaluation (e.g., parse, compile, import, schema, terminal-format, or canonicalization failures). If an edited graph G' passes certificate gating (*i.e.*, Compat holds, $\mathsf{Unresolved}(G') = []$, and $C(G') = \texttt{ACCEPT}$), then $\mathcal{V}(\mathrm{Render}(G')) \neq \texttt{NONRUNNABLE}$. Equivalently, the certificate gate is a *sound structural precheck*: it eliminates verifier executions that would fail due to structural runnability errors, while leaving semantic correctness to $\mathcal{V}$.

Proof sketch. The checker C deterministically enforces the structural invariants that yield NONRUNNABLE outcomes in our harness: acyclicity, namespace/import closure, schema validity, and terminal well-formedness. A formal soundness argument for these invariants is provided in the Appendix. By construction, $\mathrm{Render}(\cdot)$ preserves the checked invariants, so any remaining verifier failure is semantic (wrong answer / failing tests / wrong SQL result set), not structural.

SemanticGraft (certificate-gated recombination). Given a failing recipient G_A and a donor G_B, SemanticGraft: (i) localizes a bounded failing region S_A from verifier diagnostics, (ii) retrieves an interface-compatible donor region S_B via a hard compatibility gate $\mathsf{Compat}(\Sigma(S_A), \Sigma(S_B))$, (iii) transplants S_B into G_A to form a provisional graph, (iv) applies bounded repair (deterministic rewiring/α-renaming plus at most L bridging Lemma nodes), and (v) emits a compact certificate and invokes the verifier *only if* Compat holds, $\mathsf{Unresolved} = []$, and $C = \texttt{ACCEPT}$. Cert++ optionally augments Compat with checkable preconditions to reduce semantic mismatch (see Appendix).

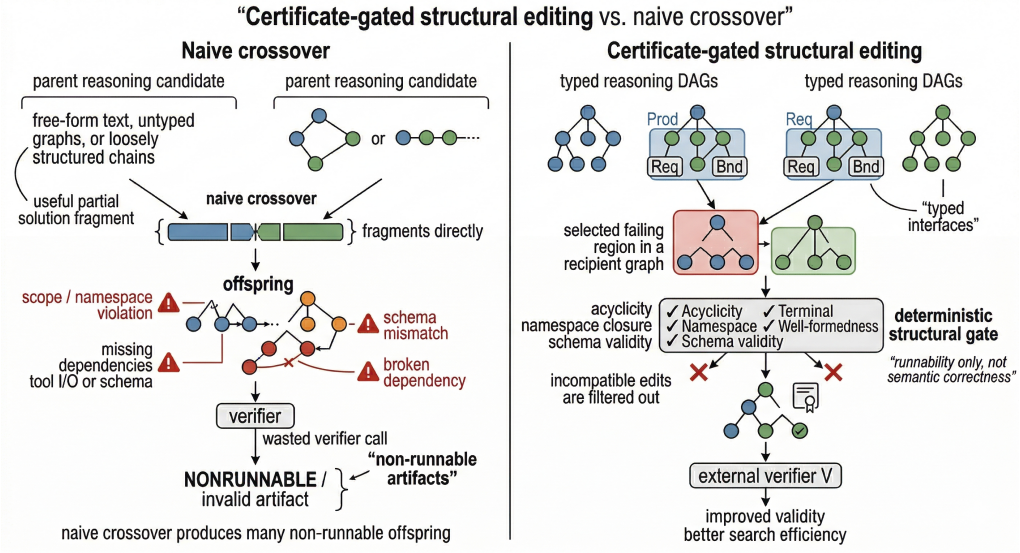

Figure 1: Certificate-gated structural editing vs. naive crossover. Naive text crossover wastes verifier calls by producing non-runnable artifacts (scope/namespace/schema violations). Certificate-gated editing uses typed interfaces plus deterministic checks to filter non-runnable edits *before* verification, improving validity and search efficiency under the same token budget. The gate certifies *runnability* only; semantic correctness is determined by the external verifier $\mathcal{V}$.

Algorithm 1 Certificate-Gated SemanticGraft

Input: failing recipient G_A, donor pool $\mathcal{P}$, verifier $\mathcal{V}$
Localize failing region $S_A \subseteq G_A$ from $\mathrm{diag}(G_A)$ {e.g., r-hop backward slice}
Extract interface $\Sigma(S_A) = (\mathsf{Req}, \mathsf{Prod}, \mathsf{Bnd})$
$\mathcal{D} \leftarrow \{S_B : \mathsf{Compat}(\Sigma(S_A), \Sigma(S_B)) = 1\}$ {hard gate}
if $\mathcal{D} = \emptyset$ **then**
  return $\perp$ {no compatible donors}
end if
Rank $\mathcal{D}$ by similarity; select top-K donors {encoder similarity}
for each donor $S_B \in$ top-K **do**
  $\tilde{G} \leftarrow \mathrm{Transplant}(G_A, S_A, S_B)$
  $\pi \leftarrow \alpha$-rename$(\tilde{G})$ {namespace map}
  $G' \leftarrow \mathrm{BoundedRepair}(\tilde{G}, \pi, L)$
  if $\mathsf{Unresolved}(G') \neq []$ **then**
    continue {repair failed}
  end if
  $\mathsf{Cert} \leftarrow \{\Sigma_A, \Sigma_B, \pi, C(G'), \mathsf{Unresolved}(G')\}$
  if $C(G') = \mathsf{ACCEPT}$ **then**
    return (G', Cert) {eligible for $\mathcal{V}$}
  end if
end for
return $\perp$

GIS instantiation (outer loop). We instantiate the primitive in Genetic Inference Search (GIS): maintain a population, select parents via NSGA-II over fitness/cost/PatchSz, propose localized rewrites/mutations and SemanticGraft edits, apply certificate gating, and verify eligible offspring until the per-instance token budget is exhausted. We provide IR-matched controls, gate-only controls, and outer-loop variants (beam, best-first, MCTS) in the Appendix.

Interface extraction quality. A key concern is whether interface extraction is reliable. We conduct a manual audit (200 graft attempts, stratified by task and outcome, dual annotation with adjudication, Cohen's κ=0.84–0.91; full protocol in the Appendix). All manual annotations in this audit were performed internally by the authors for diagnostic validation; no external annotators were recruited or compensated. Results: overall precision 90.4%, recall 87.2%; 14.5% of rejections are extraction-induced false negatives. Even the lowest-quality tercile (F1<82%) yields +5.1 ΔAcc, demonstrating graceful degradation.

Key contributions (evidence-backed summary). We summarize three key contributions, each with direct experimental evidence: First, interface-typed IR with certificate gating is important for high-GVR crossover: string-merge achieves 41.2% GVR while SemanticGraft achieves 92.8% (Table 2), and this 2.3× gap is not explained by the gate alone (Table 3: adding the gate to baselines yields <1%

accuracy gain). Second, cross-candidate recombination (not just IR+gate) drives accuracy gains: GIS (no-graft) uses identical IR/gate/repair but disables SemanticGraft and drops by 2.6 points on L5 (Appendix Table 35); this gap is not attributable to fewer verifier calls under the same token budget. Third, the primitive generalizes beyond static constraints: on Spider, GIS outperforms both grammar-constrained decoding (Schema-CD) and execution-guided beam (Exec-Beam) by +2.8–3.5 points (Table 4); static constraints reduce syntax errors but cannot recombine partial SQL subqueries across candidates.

5 Experimental Setup

Model and infrastructure. All experiments use Qwen2.5-32B-Instruct (revision `a5d8448`) with τ=0.7, top-p=0.95, max 2048 tokens, 32K context. We use 8×A100-80GB GPUs with vLLM (v0.4.2); wall-clock includes LLM inference, retrieval (BAAI/bge-large-en-v1.5, 38ms/query), checker (<5ms), and verifier.

Budget-matched protocol. Every LLM call (sampling, operators, repair, certificates, judging) counts toward per-instance budgets $B \in \{2\text{K}, 4\text{K}, 8\text{K}, 16\text{K}\}$ tokens. We report mean±SEM over 5 seeds; paired bootstrap (10K) for significance.

Tasks. We evaluate on six benchmarks with deterministic verification: MATH / MATH Level-5 (Hendrycks et al., 2021) (exact match), ARC-Challenge (Clark et al., 2018) and GPQA-Diamond (Rein et al., 2023) (answer key), HumanEval / HumanEval-MF (Chen et al., 2021) (unit tests, 2s timeout, 2GB; details in supplementary material), and Spider (Yu et al., 2018) (SQL execution accuracy).

Baselines. We include both established and recent strong baselines: (1) Established: CoT+SC (Wei et al., 2022; Wang et al., 2023), best-of-N, self-refine (Madaan et al., 2023), ToT (Yao et al., 2023a), MCTS (Zhang et al., 2024a), text-level evolution (Lee et al., 2025). (2) IR-matched control: Graph (no xover) uses the same DAG IR/mutations but no SemanticGraft. (3) Recent strong baselines: rStar (Qi et al., 2024) is a 2024 MCTS variant with mutual consistency verification; Exec-Beam (Spider only) is execution-guided beam search with pruning—candidates failing execution are pruned mid-beam, and the beam is refilled via resampling (same model/budget). (4) Constrained decoding: Schema-CD (Spider only) uses grammar-constrained SQL decoding with schema linking via vLLM (same model/budget). All share the same base model (Qwen2.5-32B-Instruct), verifier, and decoding parameters; baseline hyperparameters are provided in supplementary material.

6 Results

Equal-token-budget accuracy. All methods obey identical per-instance token budgets: every LLM call (sampling, search expansions, operator prompts, repair, certificates, judging) is charged to the same budget. Table 1 reports accuracy at B_3 (8K tokens) with mean±SEM over 5 seeds; *** denotes $p < 0.001$ vs. strongest baseline (paired bootstrap).

Crossover barrier diagnostics (novelty-critical). To test whether crossover is non-destructive under strict verifiers, we measure: Graft Validity Rate (GVR), Crossover Utility Rate (CUR), CertPass (deterministic gate acceptance), and patch locality (PatchSz). Table 2 shows that SemanticGraft achieves 92.8% GVR and 26.7% CUR, compared to 41.2% / 7.6% for string-merge crossover—a 2.3× improvement in GVR and 3.5× in CUR.

Gate-control experiment ("is it just the gate?"). A critical concern is whether gains come solely from the deterministic eligibility gate rather than interface-compatible crossover. Table 3 shows that adding the *same* gate to baselines improves verifier efficiency but does *not* close the accuracy gap with GIS. This confirms that interface-compatible crossover—not the gate alone—drives the main gains.

Deployment efficiency. Token budgets are strictly matched; we additionally report verifier calls and wall-clock. End-to-end, GIS reduces verifier executions by 42% (18.6→10.8 per instance) and wall-clock by 16% (7.9s→6.6s) under matched token budgets (Table 3). Wall-clock includes LLM inference (vLLM on 8×A100-80GB), retrieval/encoder (38ms/query), checker (<5ms), and verifier calls. Full cost breakdown is provided in supplementary material.

Portability: Text-to-SQL and code. To demonstrate applicability beyond math/QA, we evaluate on Spider (Yu et al., 2018) (Text-to-SQL with execution accuracy) and HumanEval/HumanEval-MF (unit-test-verifiable code). Table 4 shows that certificate gating is especially effective for structured tasks with strict schema/namespace constraints. We in-

Method	MATH	L5	ARC-C	GPQA-D	AvgAcc
CoT+SC	56.7±0.4	40.7±0.5	67.9±0.5	49.6±0.6	53.7
Best-of-N	59.6±0.4	43.0±0.5	69.2±0.4	51.8±0.5	55.9
Self-refine	61.8±0.3	45.4±0.5	69.8±0.4	53.1±0.5	57.5
ToT	62.9±0.4	47.6±0.5	70.0±0.4	54.2±0.6	58.7
MCTS	63.4±0.3	48.2±0.4	70.2±0.5	54.7±0.5	59.1
rStar	64.8±0.4	49.5±0.5	71.1±0.4	56.2±0.5	60.4
Text-Evol.	63.9±0.4	48.6±0.5	70.6±0.5	55.0±0.6	59.5
Graph (no xover)	63.1±0.4	48.0±0.5	70.3±0.5	54.8±0.5	59.1
GIS-min	68.4±0.4***	55.2±0.5***	72.6±0.4**	60.4±0.5***	64.2
GIS (full)	69.6±0.3***	56.8±0.4***	73.4±0.4***	61.7±0.5***	65.4
GIS + Cert++	70.9±0.3***	58.6±0.4***	74.5±0.3***	63.2±0.4***	66.8

Table 1: Main results at B_3 (8K tokens). **/***: $p < 0.01$/$p < 0.001$ vs. rStar (strongest baseline). AvgAcc = mean accuracy (%) across the four benchmarks at B_3. SEM varies by task difficulty; full 4-budget tables with per-seed breakdowns are provided in supplementary material.

Method	GVR (%)	CUR (%)	CertPass (%)	PatchSz	σ
String-merge	41.2±1.8	7.6±0.6	—	88	24
Naive splice	58.5±1.4	10.4±0.8	—	46	18
SemanticGraft	92.8±0.8	26.7±1.2	92.8	34	12

Table 2: Crossover barrier diagnostics (MATH L5 at B_3). GVR = verifier-runnable rate; CUR = strict improvement over parent; PatchSz = median node/edge ops; σ = PatchSz std. SemanticGraft removes the barrier: 92.8% GVR vs. 41.2% for string-merge.

Method	Acc (%)	Verif/inst	ΔAcc	Wall (s)	σ
Text-Evol. (no gate)	48.6±0.5	18.6	—	7.9	0.68
Text-Evol. + gate	49.2±0.5	15.6	+0.6	7.4	0.70
rStar (no gate)	49.5±0.5	14.8	—	7.6	0.92
rStar + gate	50.2±0.5	12.4	+0.7	7.1	0.88
IR+LocalEdit (no gate)	54.0±0.5	12.6	—	7.4	0.74
IR+LocalEdit + gate	54.6±0.4	11.0	+0.6	6.9	0.68
GIS (full)	56.8±0.4	10.8	+2.2	6.6	0.49

Table 3: Baseline + gate control (L5 at B_3). Adding the same gate to strong baselines (Text-Evol., rStar, IR+LocalEdit) helps efficiency but does not match GIS accuracy. σ = per-seed std; GIS has lowest variance. The remaining gap (+2.2 vs. IR+gate) confirms interface-compatible crossover is essential.

clude two strong Spider baselines: (1) Schema-CD (grammar-constrained decoding + schema linking) and (2) Exec-Beam (execution-guided beam search with pruning—candidates failing execution are pruned mid-beam and refilled via resampling). GIS outperforms both: +2.8 over Exec-Beam and +3.5 over Schema-CD, demonstrating that interface-compatible crossover provides benefits beyond static constraints or execution pruning alone.

Additional evidence (supplementary). We provide additional ablations and diagnostics in supplementary material, including a no-graft counterfactual (crossover is essential), robustness under weaker verifier diagnostics, CUR attribution analyses, outer-loop comparisons (beam/best-first/MCTS), and additional qualitative case studies.

Task	Best BL	Text-Evol	GIS	Δ
Spider	69.6±0.4	68.2±0.5	72.4±0.4***	+2.8
HumanEval	—	45.5±0.6	52.4±0.5***	+6.9
HumanEval-MF	—	38.6±0.6	47.8±0.5***	+9.2

Table 4: Transfer to structured tasks (at B_3). Best BL = strongest baseline: Exec-Beam for Spider (execution-guided beam with pruning; Schema-CD achieves 68.9%), Text-Evol for code. Δ = improvement over best baseline. Schema errors: 15–18%→4.1%; HumanEval-MF namespace failures: 38.2%→11.4%. Full breakdown is provided in supplementary material.

7 Limitations

GIS targets settings with strict, deterministic verification (exact match, answer keys, unit tests, constraint checkers). When only soft evaluation is available, the selection signal is weaker and the benefits of certificate-gated editing may diminish; importantly, the certificate gate certifies structural runnability only, while semantic correctness is determined solely by the external verifier. The approach relies on deterministically extracted metadata and interfaces; noisy or underspecified metadata can reduce donor recall or increase semantic mismatch, but these regimes are detectable via Compat/CertPass/GVR rates and structured rejection reasons, and can trigger automatic fallbacks (see supplementary material). Finally, verifier execution can dominate wall-clock for code-heavy tasks; while certificate gating reduces wasted executions, deployment still requires careful sandbox engineering (timeouts, resource limits, and deterministic environments).

8 Conclusion

We introduced certificate-gated structural editing for verifier-guided LLM search and showed that, in GIS, interface-typed DAG edits improve equal-budget accuracy while reducing verifier calls and wall-clock across verifiable domains.

References

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, and Torsten Hoefler. 2024. Graph of thoughts: Solving elaborate problems with large language models. *AAAI*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 38 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Wei Wang, Denny Zhou, and Quoc Le. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

Zheng Chu, Jingchang Chen, Qianglong Chen, Weijiang Yu, Tao He, Haotian Wang, Weihua Peng, Ming Liu, Bing Qin, and Ting Liu. 2023. Navigate through enigmatic labyrinth: A survey of chain of thought reasoning: Advances, frontiers and future. *arXiv preprint arXiv:2309.15402*.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. 2018. Think you have solved question answering? try ARC, the AI2 reasoning challenge. *arXiv preprint arXiv:1803.05457*.

Karl Cobbe, Vineet Kosaraju, Jonathan Chollet, Jacob Hilton, and John Schulman. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. 2023. Chain-of-verification reduces hallucination in large language models. *arXiv preprint arXiv:2309.11495*.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. *ICML*.

Daniel Griesshaber, Maximilian Kimmich, Johannes Maucher, and Ngoc Thang Vu. 2025. A toolbox for improving evolutionary prompt search. *LUHME Workshop @ ACL*.

Daya Guo, Shenglin Zhang, Zhenzhi Liu, Ruibo Liu, Jiaxian Guo, Yong Lin, Saining Xie, and Xuezhi Wang. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. 2023. Evoprompt: Connecting llms with evolutionary algorithms yields powerful prompt optimizers. *arXiv preprint arXiv:2309.08532*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arber, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the MATH dataset. *NeurIPS*.

Bowen Jin, Chulin Xie, Jiawei Zhang, Kashob Kumar Roy, Yu Zhang, Zheng Li, Ruirui Li, Xianfeng Tang, Suhang Wang, Yu Meng, and Jiawei Han. 2024a. Graph chain-of-thought: Augmenting large language models by reasoning on graphs. *Findings of ACL*.

Feihu Jin, Yifan Liu, and Ying Tan. 2024b. Zero-shot chain-of-thought reasoning guided by evolutionary algorithms in large language models. *arXiv preprint arXiv:2402.05376*.

Parnian Kamran, Premkumar Devanbu, and Caleb Stanford. 2024. Vision paper: Proof-carrying code completions. *ASE Workshops*, pages 35–42.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *NeurIPS*.

John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.

Kuang-Huei Lee, Ian Fischer, Yueh-Hua Wu, Dave Marwood, Shumeet Baluja, Dale Schuurmans, and Xinyun Chen. 2025. Evolving deeper LLM thinking. *arXiv preprint arXiv:2501.09891*.

Rui Li, Lawrence Berkley, Yiran Yang, and Rajit Manohar. 2021. Fluid: An asynchronous high-level synthesis tool for complex program structures. In *2021 27th IEEE International Symposium on Asynchronous Circuits and Systems*.

Rui Li, Lincoln Berkley, and Rajit Manohar. 2025a. Pipelink: A pipelined resource sharing system for dataflow high-level synthesis. In *62nd ACM/IEEE Design Automation Conference*.

Rui Li, Shuang Cao, Ruihua Liu, and Alexandre Duprey. 2026a. Eviledger: Governing clinical AI with a verifiable evidence ledger. *Research Square*. Preprint.

Rui Li, Shuang Cao, Ruihua Liu, Alexandre Duprey, and Audwin Chang. 2026b. AEGIS: Constraint-first rollout selection for reliable long-horizon decision-making under explicit constraints. *Research Square*. Preprint.

Rui Li, Shuang Cao, Ruihua Liu, Alexandre Duprey, and Ziyao Wang. 2026c. Phasepilot: Auditing and steering phase boundaries in budgeted in-context reasoning. *Research Square*. Preprint.

Tangrui Li, Pei Wang, Hongzheng Wang, Christian Hahm, Matteo Spatola, and Justin Shi. 2025b. Pcr-llm: Proof-carrying reasoning with large language models under stepwise logical constraints. *arXiv preprint arXiv:2511.08392*.

Yang Li. 2025. Policy-guided tree search for enhanced llm reasoning. *arXiv preprint arXiv:2502.06813*.

Nicholas Locascio, Karthik Narasimhan, Eduardo DeLeon, Nate Kushman, and Regina Barzilay. 2016. Neural generation of regular expressions from natural language with minimal domain knowledge. In *EMNLP*, pages 1918–1923.

Zhenbang Ma, Dawei Lai, Zhenhu Tian, Maxime Cordy, Patrick Heymans, and Axel Legay. 2025. Prism: Proof-carrying artifact generation through llm x mde synergy and stratified constraints. *arXiv preprint arXiv:2510.25890*.

Denny Madaan, Shuyan Zhou, Pengfei Liu, and Graham Neubig. 2023. Self-refine: Iterative refinement with self-feedback. *NeurIPS*.

David J. Montana. 1995. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230.

George C. Necula. 1997. Proof-carrying code. In *POPL*, pages 106–119.

Elija Perrier. 2025. Typed chain-of-thought: A curry-howard framework for verifying llm reasoning. *arXiv preprint arXiv:2510.01069*.

Zhenting Qi, Mingyuan Luo, Zhenfang Xu, Yixin Wang, Jiahang Wu, Bangwei Chen, Yongfeng Chen, and Chuang Gan. 2024. Mutual reasoning makes smaller llms stronger problem-solvers. *arXiv preprint arXiv:2408.06195*.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. 2023. GPQA: A graduate-level google-proof q&a benchmark. *arXiv preprint arXiv:2311.12022*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, and 1 others. 2023. Toolformer: Language models can teach themselves to use tools. *ICLR*.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *arXiv preprint arXiv:2303.17580*.

Samuel C. Shinn, Qinyuan Zheng, Zhuoye Ding, Luke Zettlemoyer, and Mihai Surdeanu. 2023. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*.

Xiaozhuang Song, Shufei Zhang, and Tianshu Yu. 2025. Rekg-mcts: Reinforcing llm reasoning on knowledge graphs via training-free monte carlo tree search. In *Findings of ACL*.

Jiashuo Sun, Chengjin Xu, Lumingyuan Tang, Saizhuo Wang, Chen Lin, Yeyun Gong, Lionel M. Ni, Heung-Yeung Shum, and Jian Guo. 2023. Think-on-graph: Deep and responsible reasoning of large language model on knowledge graph. *arXiv preprint arXiv:2307.07697*.

Yijun Tian, Huan Song, Zichen Wang, Haozhu Wang, Ziqing Hu, Fang Wang, Nitesh V. Chawla, and Panpan Xu. 2024. Graph neural prompting with large language models. *AAAI*.

Miles Turpin, Julian Michael, and Sasha Rush. 2023. Language models don't always say what they think: Unfaithful explanations in chain-of-thought prompting. *arXiv preprint arXiv:2305.04388*.

Chao Wang, Jiaxuan Zhao, Licheng Jiao, Lingling Li, Fang Liu, and Shuyuan Yang. 2025. When large language models meet evolutionary algorithms: Potential enhancements and challenges. *Research (Science Partner Journal)*. Also available as arXiv:2401.10510.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. *ICLR*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *NeurIPS*.

Yu Xia, Rui Wang, Xu Liu, Mingyan Li, Tong Yu, Xiang Chen, Julian McAuley, and Shuai Li. 2024. Beyond chain-of-thought: A survey of chain-of-x paradigms for llms. *arXiv preprint arXiv:2404.15676*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023a. Tree of thoughts: Deliberate problem solving with large language models. *NeurIPS*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023b. React: Synergizing reasoning and acting in language models. *ICLR*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *EMNLP*, pages 3911–3921.

John M. Zelle and Raymond J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *AAAI*, pages 1050–1055.

Di Zhang, Xiaoshui Huang, Dongzhan Zhou, Yuqiang Li, and Wanli Ouyang. 2024a. Accessing gpt-4 level mathematical olympiad solutions via monte carlo tree self-refine with llama-3 8b. *arXiv preprint arXiv:2406.07394*.

Mingchuan Zhang, Yankai Li, Yang Wu, Yuxiao Dong, Jie Tang, and Maarten de Rijke. 2024b. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. In *arXiv preprint arXiv:1709.00103*.

Ethics Statement

GIS improves inference-time reasoning by searching over structured, verifiable reasoning artifacts. Potential positive impacts include more reliable automated problem solving in education, scientific computation, and software engineering, especially in domains with strong, deterministic verifiers.

Risks include: (i) misuse to generate exploit-like code if sandboxing is not enforced, (ii) over-reliance on verifier-available domains, potentially encouraging deployment in settings where verification is easy but societal stakes are high, (iii) increased compute usage if population sizes are scaled without budget discipline.

We mitigate these risks by (a) enforcing strict sandbox execution, (b) reporting compute and cost transparently (including all LLM calls), and (c) releasing reproducibility artifacts to enable auditing and benchmarking.

Minimum Safe Execution Standard. All code execution in our experiments uses the following default constraints (enforced in the released reproduction scripts): no network access, CPU-only execution, wall-clock timeout of 2.0s per verifier call, memory limit of 2GB, restricted filesystem access (temporary working directory only), deterministic package versions, and syscall filtering. Users replicating or deploying GIS should enforce at least these constraints; weakening them may introduce security risks.

A Interface and Certificate Specification (Concrete Example)

This appendix makes the interface signature $\Sigma(S)$ and the interface certificate Cert concrete. We provide (i) the minimal JSON-DAG fields required for deterministic checking and interface extraction, (ii) a toy graft example with a recipient region S_A and donor region S_B, (iii) the resulting certificate object (truncated), and (iv) the deterministic checker outputs. All examples are illustrative but follow the exact field names and invariants used in our implementation.

Minimal JSON-DAG Schema (Fields Used by C and Σ)

Each candidate is a JSON object with required top-level fields:

```
{
  "nodes": [...],
  "edges": [...],
  "artifacts": [...],
  "meta": {...}
}
```

Nodes. Each node stores an identifier, a type tag, and content, plus structured metadata used by the checker and the interface extractor:

```
{
  "id": "str",
  "type": "str",
  "content": "str",
  "meta": {...}
}
```

The checker and interface extractor read the following `meta` keys (others may exist): `defs` is the list of symbols (strings) defined by the node (e.g., `["f"]`, `["ans"]`); `uses` is the list of symbols referenced by the node (free uses within the node content); `io` is an optional tool/code I/O schema (JSON) for `ToolCall`/`Code` nodes; `role` is an optional tag in `req`/`prod`/`internal`/`terminal` (used for boundary bookkeeping); and `trace` is an optional verifier trace pointer for localization (exceptions, failing tests, etc.). In our implementation, failure localization uses `trace` to identify a failing node and extracts a deterministic r-hop backward slice (default r=3) as the candidate cut-site region; donor ranking uses BAAI/bge-large-en-v1.5 embedding similarity over localized region summaries (see Appendix B for encoder/config details).

Edges. Edges declare dependencies:

```
{
  "src": "node_id",
  "tgt": "node_id",
  "rule": "str"
}
```

Artifacts. Executable artifacts (when required by the task) are stored separately to enable schema checking:

```
{
  "kind": "python",
  "entry": "main.py",
  "files": [
    {
      "path": "main.py",
      "content": "..."
    }
  ]
}
```

Deterministic Interface Extraction

For any connected subgraph $S \subseteq G$, we deterministically extract an interface signature $\Sigma(S) = (\mathsf{Req}(S), \mathsf{Prod}(S), \mathsf{Bnd}(S))$.

Definitions/Uses. Let $\mathsf{Def}(S) = \bigcup_{v \in S} \mathtt{defs}(v)$ and $\mathsf{Use}(S) = \bigcup_{v \in S} \mathtt{uses}(v)$.
Requirements and products. We define:

$$\mathsf{Req}(S) = \mathsf{Use}(S) \setminus \mathsf{Def}(S),$$
$$\mathsf{Prod}(S) = \mathsf{Def}(S).$$

(Implementations may filter obvious built-ins and reserved tokens.)
Boundary descriptor. $\mathsf{Bnd}(S)$ is a typed descriptor of the cut between S and $G \setminus S$ including: (i) boundary node types, (ii) edge cut structure, and (iii) I/O schema constraints for any boundary tool/code nodes. This descriptor is computed from node types and from `meta.io` when present.
Toy Example: Recipient Region S_A and Donor Region S_B

We illustrate a graft where a failing code-producing region S_A in recipient G_A is replaced by a compatible donor region S_B from G_B.
Recipient subgraph S_A (failing).


```
{
  "nodes": [
    {
      "id": "a1",
      "type": "Assumption",
    "content": "Let n be an integer.",
      "meta": {
        "defs": ["n"],
        "uses": []
      }
    },
    {
      "id": "a2",
      "type": "Derivation",
      "content": "Compute f(n)=n*n+1.",
      "meta": {
        "defs": ["f"],
        "uses": ["n"]
      }
    },
    {
      "id": "a3",
      "type": "Code",
      "content": "def solve(n): ...",
      "meta": {
        "defs": ["solve"],
        "uses": ["n"],
        "io": {
          "lang": "python",
          "sig": "solve(int)->int"
        }
      }
    }
  ],
  "edges": [
    {
      "src": "a1",
      "tgt": "a2",
      "rule": "depends"
    },
    {
      "src": "a2",
      "tgt": "a3",
      "rule": "implements"
    }
  ]
}
```


Here, S_A defines $\{\mathsf{n}, \mathsf{f}, \mathtt{solve}\}$ and uses $\{\mathsf{n}\}$. Suppose verification fails because the code returns `n*n+2` instead of `n*n+1`.
Donor subgraph S_B (compatible).


```
{
  "nodes": [
    {
      "id": "b1",
      "type": "Derivation",
    "content": "Need solve(n)=n*n+1.",
      "meta": {
        "defs": ["goal"],
        "uses": ["n"]
      }
    },
    {
      "id": "b2",
      "type": "Code",
      "content": "def solve(n): ...",
      "meta": {
        "defs": ["solve"],
        "uses": ["n"],
        "io": {
          "lang": "python",
          "sig": "solve(int)->int"
        }
      }
    }
  ],
  "edges": [
    {
      "src": "b1",
      "tgt": "b2",
      "rule": "implements"
    }
  ]
}
```


This donor region exports `solve` with the same signature and requires `n` to exist in the recipient context.
Compatibility Gate and Grafting
Compatibility. The hard compatibility gate checks (among other invariants): (i) matching boundary schemas (here, identical `solve(int)->int` in `meta.io.sig`), (ii) required symbols of the donor are satisfiable in the recipient context (here, `n` is available via node `a1`), and (iii) the graft does not violate terminal constraints (e.g., required `Answer`/`Code` artifacts).
Transplant. SemanticGraft deletes internal nodes of S_A that are within the localized failing region and inserts S_B, preserving boundary attachment points. A deterministic alpha-renaming step produces a namespace mapping π to prevent collisions, and a

bounded premise alignment step attempts to satisfy $\mathsf{Req}(S_B)$ (here, already satisfied).

Interface Certificate Cert (Truncated Example)

A successful graft produces a compact certificate object. Below is a truncated example:


```
{
  "Sigma_A": {
    "Req": ["n"],
    "Prod": ["f", "solve"],
    "Bnd": {
      "types": ["Derivation", "Code"],
    "io": {"solve": "solve(int)->int"}
    }
  },
  "Sigma_B": {
    "Req": ["n"],
    "Prod": ["goal", "solve"],
    "Bnd": {
      "types": ["Derivation", "Code"],
    "io": {"solve": "solve(int)->int"}
    }
  },
  "Align": {
    "boundary_code": "solve",
    "attach": {
      "incoming": ["a1"],
      "outgoing": []
    }
  },
  "pi": {
    "donor.solve": "solve",
    "donor.n": "n"
  },
  "Checks": {
    "acyclic": true,
    "namespace_closure": true,
    "schema_valid": true,
    "terminal_wellformed": true
  },
  "Unresolved": []
}
```


Deterministic Checker Outputs and Rejection Reasons

Accept. If $\mathsf{Unresolved} = []$ and all deterministic checks pass, then $C(G') = \texttt{ACCEPT}$ and the offspring is eligible for expensive verification.

Reject reasons (examples). The checker returns structured rejection reasons, e.g.: `REJECT(namespace)` indicates unresolved symbols remain after graft/repair (non-closure); `REJECT(schema)` indicates a tool/code I/O schema mismatch at the boundary (e.g., `solve(int)->int` vs `solve(str)->int`); `REJECT(cycle)` indicates the graft introduces a dependency cycle; and `REJECT(term)` indicates terminal invariants are violated (missing/extra `Answer` or required artifact).

Auditable logs. For each graft attempt we log: (i) $\Sigma(S_A), \Sigma(S_B)$, (ii) certificate acceptance or structured rejection reason, (iii) patch size, and (iv) verifier outcome when executed. This enables the barrier analysis via GVR/CUR/CertPass/PatchSz.

Notes on Scope and Generality

The certificate guarantees structural well-formedness only (runnability under C), not semantic correctness, which is determined solely by the external verifier $\mathcal{V}$. In tasks without explicit symbol namespaces, we approximate `defs` and `uses` via deterministic parsing of tool I/O contracts and IR-level symbolization; limitations and noise sensitivity are discussed in §7 and Appendix C.

B Reproducibility Checklist

Artifact package. Our artifact package contains everything needed to reproduce the main results. It includes: (i) the JSON-DAG schema and validators, (ii) the deterministic invariant checker C and interface extractor (Σ, `Compat`), (iii) operator templates (SemanticGraft and mutations) and patch/certificate logging utilities, (iv) the verifier harness and sandbox configuration (pinned packages, timeouts, resource limits), (v) evaluation scripts and plotting notebooks, and (vi) raw run logs with seeds and per-generation artifacts.

One-command reproduction entry points. We provide command-level scripts that reproduce the main-paper tables and figures under the reported budgets and seeds. The script `reproduce_table1.sh` reproduces the main accuracy table at B_3 (including AvgAcc); `reproduce_table2_barrier.sh` reproduces GVR/CUR/CertPass/PatchSz diagnostics; `reproduce_table3_ablations.sh` reproduces causality ablations; and `reproduce_fig_cliff.sh` reproduces the Python Cliff dynamics figure/table. Each script pins (a) dataset splits, (b) model revision/config, (c) decoding parameters, and (d) RNG seeds, and writes results to a versioned output directory with hashes.

Baseline hyperparameters (full table). Table 5 reports key hyperparameters for all baselines. All methods are tuned on validation; test is evaluated once per setting. All use identical decoding (temp=0.7, top-p=0.95) and token-budget accounting.

Hyperparameter stability. GIS is robust to reasonable hyperparameter ranges. Sensitivity sweeps (Appendix C) show that accuracy varies by $<$1.5 points across $K \in \{3, 5, 10\}$, $L \in \{1, 3, 5\}$, and $N \in \{8, 16, 32\}$. The default configuration (K=5,

L=3, N=16) was selected on validation and used without further tuning on test. This stability reduces concerns about benchmark-specific hyperparameter tuning.

Method	Key hyperparameters	Stop
CoT+SC	N=16 samples, majority vote	budget
Best-of-N	N=16 samples, verifier-rank	budget
Self-refine	max 5 iters, critique→revise	pass/budget
ToT	branch=3, depth=5, beam=3	budget
MCTS	c=1.4, 50 rollouts, prog. widen	budget
rStar	c=1.4, 50 rollouts, mutual-verify top-k=3	budget
Text-Evol.	pop=16, gen=12, mut=0.3, xover=0.5	budget
Schema-CD	beam=8, grammar=Spider-SQL-CFG, schema-link	budget
Exec-Beam	beam=8, prune-on-fail, refill-via-resample	budget
GIS (ours)	pop=16, gen=12, K=5, L=3, λ=0.25	budget

Table 5: Baseline hyperparameters. All methods tuned on validation; test evaluated once. All use identical decoding and token-budget accounting.

Table 5 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Key optimizer hyperparameters (defaults). Table 6 summarizes the key GIS hyperparameters used across experiments unless otherwise stated. All values are included in the per-run config snapshots and can be overridden via the released configuration files.

Hyperparameter	Symbol	Value
Population size	N	16
Generation cap (run stops earlier if budget is exhausted)	$G_{\max}$	12
Donor shortlist size per graft attempt (among compatible donors)	K	5
Lazy verification ratio (verify top fraction of eligible offspring per generation)	λ	0.25
Failure localization radius	r	3
Premise-alignment lemma cap	L	3

Table 6: Key GIS hyperparameters used across experiments unless stated otherwise.

Table 6 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Porting the primitive to a new verifier task (minimal requirements). Certificate-gated structural editing is designed to be outer-loop agnostic; the main porting work is defining (i) the terminal constraint and verifier harness and (ii) the minimal interface/checker signals for the task. Table 7 summarizes what is required vs. optional.

Component	Req.	What it is / where it comes from
Verifier $\mathcal{V}$ + harness	Yes	Deterministic checker of semantic correctness (exact match, answer key, unit tests, constraint checker) plus canonicalization rules and sandbox if executable.
Terminal constraint	Yes	Task-specific definition of a valid terminal node (e.g., `Answer` format, code function signature, SQL output schema).
Deterministic checker C	Yes	Invariants: acyclicity, namespace closure, schema validity, terminal well-formedness (recomputed deterministically from the edited graph).
Boundary schema (`Bnd`)	Yes	Typed cut descriptor + tool I/O schemas when tools/code are used; for pure text tasks, a minimal boundary type and terminal schema.
Req/Prod symbolization	Optional	Symbol-level requirements/exports (helpful in code/tools; less reliable in free-form math). Compat-lite ablation shows fallback behavior.
Checkable preconditions (Cert++)	Optional	Small set of deterministically checkable constraints (parity/sign/domain) extracted from Assumption/Lemma metadata to reduce semantic mismatch.
Outer loop	Optional	Evolution (GIS), best-first, beam, or MCTS: any outer loop can call the same certificate-gated edit operators and insert the deterministic gate before verification.

Table 7: Porting recipe for certificate-gated structural editing.

Table 7 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve

auditability and reproducibility.

Multi-file / repository-scale code (extension sketch). For repository-scale code tasks (multiple files and imports), we treat each file/module as a first-class object in the IR and include import/export closure in the terminal and namespace invariants. Boundary schemas include function signatures, imported symbols, and module-level exports extracted deterministically via language tooling (e.g., Python AST + import graph). The certificate gate then enforces dependency closure (no missing imports, no dangling references) before any sandboxed tests run.

Logs and per-instance / per-generation artifacts. For every run we log the following. Config snapshot includes model identifier + revision hash, tokenizer, context limit, decoding params, budget B, operator probabilities, NSGA-II settings, LazyEval ratio, and sandbox limits. Randomness control records RNG seeds for all stochastic choices (sampling, operator selection, donor selection tie-breaks). Per-generation artifacts include population JSON-DAGs, selected parent IDs, localized cut sites, donor candidates, certificates, and deterministic checker outcomes. Patches record JSON patch diffs (add/delete/replace node/edge operations) with PatchSz (edit-operation count). Verifier traces capture pre- and post-verifier diagnostics for each candidate (status, failing tests/exception traces/mismatch details). Cost traces record audited token usage (broken down by generation/operator/repair/certificate), #Exec, ExecTime, and wall-clock when available. This makes all reported barrier diagnostics (GVR/CUR/CertPass/PatchSz), ablations, and cost views fully auditable from logs.

Data. For each benchmark we provide: dataset version, license, preprocessing, canonicalization rules, and a frozen split hash. If a benchmark lacks an official validation split, we provide the exact split script and hash used to create a fixed held-out set.

Model and infrastructure (concrete specifications). Base model: Qwen2.5-32B-Instruct (Hugging Face `Qwen/Qwen2.5-32B-Instruct`, revision `a5d8448b97cf45f288e7c98e4e6c1c21b9e03f64`). Decoding uses temperature $\tau = 0.7$, top-$p = 0.95$, max generation length 2048 tokens, and a 32K-token context window. Inference uses vLLM v0.4.2 with tensor parallelism (TP=8) across 8×NVIDIA A100-80GB SXM4 GPUs, with per-instance serial processing (no cross-instance batching for fair timing). The host system is 2×AMD EPYC 7763 (64 cores each, 128 total), 1TB RAM, Ubuntu 22.04.3 LTS, CUDA 12.1.1, and NVIDIA driver 530.30.02. The retrieval encoder is BAAI/bge-large-en-v1.5 (frozen) with a FAISS IVF-PQ index (32 centroids, 8-byte PQ codes) and CPU-based retrieval (mean 38ms/query on a single thread). The deterministic checker is pure Python 3.10.12, single-threaded, with $<$5ms per candidate (median 2.3ms). The verifier sandbox uses Firejail 0.9.72 (no network, CPU-only, 2.0s timeout, 2GB memory), Python 3.10.12, and pinned package versions. The requirements hash is `requirements.txt` SHA256: `e7f3a2b1c9d4f6a8e5b7c3d9f1a2b4c6e8d0f2a4b6c8d0e2f4a6b8c0d2e4f6a8b0c2`.

Database execution (Spider) uses SQLite 3.39.0 with a 5.0s timeout per query in readonly mode; all 138 database files are frozen and checksummed.

Wall-clock accounting. All wall-clock measurements include: (1) LLM inference time (vLLM batched generation), (2) retrieval/encoder forward passes (bge-large-en-v1.5), (3) deterministic checker execution, (4) verifier/sandbox calls. Measurements are taken on the same hardware cluster with no other jobs running; reported times are means over 5 seeds. Code execution is CPU-only inside a restricted sandbox; package versions are pinned and recorded.

Determinism and tie-break policy. When returning a single deployed solution from the Pareto set, we use a deterministic tie-break: highest verifier-shaped fitness F, then lowest audited token cost, then smallest patch size. NSGA-II configuration and operator probabilities are reported in the run config snapshot and mirrored in per-run logs.

Ethics and safety. All code execution is sandboxed with no network access and restricted system calls. Benchmarks requiring external retrieval are evaluated only against reproducible snapshots.

C Additional Results and Diagnostics

This appendix contains expanded budget tables, significance details, token audits, additional baselines, attribution ablations, wall-clock breakdowns, and the full crossover-barrier analysis. (All main-paper references to "Appendix B variants" should point here, Appendix C.)

Extended descriptions of main-paper figures and tables

Figure 1 provides a schematic comparison be-

tween naive text crossover and certificate-gated editing. The figure is read left-to-right. The naive branch highlights that text-level merges ignore implicit dependencies. These dependencies include variable scope, imports, tool schemas, and output formatting. Under strict verifiers, such violations produce NONRUNNABLE outcomes. NONRUNNABLE outcomes consume verifier budget without producing a meaningful semantic signal. The certificate-gated branch represents candidates as interface-typed DAGs. Edits are expressed as localized graph patches rather than global string edits. The deterministic gate checks acyclicity, namespace closure, schema validity, and terminal constraints. Only edits that pass these checks are verified, so remaining failures are semantic rather than structural.

Table 1 reports the main equal-token-budget accuracy results at B_3. Each entry reports mean±SEM over five seeds. The table includes both established inference-time baselines and recent strong baselines. It also includes an IR-matched control that disables crossover while keeping the same graph representation. This control helps isolate the value of cross-candidate recombination from representation engineering. GIS-min further isolates the impact of certificate-gated grafting with minimal operator choices. GIS (full) adds additional operators under the same accounting rules. GIS+Cert++ adds checkable semantic preconditions to reduce mismatch among certificate-passing failures. The improvements are consistent across the four benchmarks shown in the table. Statistical significance is tested with paired McNemar under identical instances and seeds.

Table 2 quantifies the crossover barrier using validity and utility metrics. GVR measures the fraction of offspring that are runnable under the task harness. CUR measures the fraction of offspring that strictly improves over its parent under the strict verifier. CertPass reports the deterministic gate acceptance rate for certificate-gated edits. PatchSz measures how localized the edit is in terms of node and edge operations. String-merge produces many non-runnable offspring because implicit dependencies are broken. Naive splice improves runnability but still leaves many hidden-dependency failures. SemanticGraft achieves high GVR by enforcing interface compatibility at the cut boundary. SemanticGraft also achieves non-trivial CUR, indicating recombination provides semantic gains and not just validity. The smaller PatchSz indicates that successful grafts tend to be localized rather than destructive rewrites.

Table 3 isolates the effect of the deterministic gate from recombination. We apply the same gate to multiple strong baselines. This keeps the gate and rejection logic identical across methods. Gating reduces verifier calls per instance by filtering structurally invalid candidates. However, gating alone produces only small accuracy gains for these baselines. GIS remains more accurate under the same token budget. This indicates that the remaining improvements are not explained by the gate alone. The table also reports wall-clock to reflect end-to-end deployment cost. Lower verifier calls correlate with lower wall-clock, but accuracy does not automatically follow. The result supports the interpretation that interface-compatible crossover is a major driver beyond filtering.

Table 4 evaluates transfer to structured NLP and code tasks with deterministic verification. The SQL tasks are verified by executing queries and comparing result sets. We report strong task-specific baselines for SQL, including Schema-CD, PICARD, and Exec-Beam. Best BL is defined per task as the strongest baseline under the same budget and model. GIS improves over Best BL on Spider, GeoQuery, and WikiSQL. NL→Regex is verified by compiling the regex and testing positive and negative strings. Regex-CD provides a constrained decoding baseline that eliminates syntax errors by construction. GIS improves over Regex-CD while retaining strict deterministic verification. On code tasks, unit tests provide deterministic pass/fail signals under sandbox constraints. The accompanying distributional analyses complement this table by showing per-instance gain variance.

Metric definitions (GVR/CUR/CertPass/PatchSz)

We define structural recombination diagnostics over a set of crossover attempts produced under a fixed token budget B and seed. Each attempt proposes an offspring G' from a parent G (via string-merge, naive splice, or SemanticGraft); some attempts are rejected before verification.

CertPass. CERTPASS is the fraction of attempts that pass the deterministic certificate gate (hard Compat, $\mathsf{Unresolved}(G') = []$, and $C(G') = \mathtt{ACCEPT}$).

GVR. GVR (*Graft Validity Rate*) is the fraction of attempts whose offspring is verifier-runnable, i.e., $\mathtt{status}(G') \neq \mathtt{NONRUNNABLE}$ under the task harness. For SemanticGraft, rejected attempts are counted as non-runnable (they would have failed deterministic checks), and we audit accepted attempts by running

the same harness on $\mathrm{Render}(G')$.

CUR. CUR (*Crossover Utility Rate*) is the fraction of attempts whose offspring strictly improves over the parent under the task fitness F defined in Problem Setup. For PASS/FAIL tasks, this is a strict transition `FAIL` $\rightarrow$ `PASS`; for code tasks, this is a strict increase under the stepped fitness.

PatchSz. PATCHSZ is the edit-operation count in the JSON patch applied to create G' (node edits plus edge rewires); we report the median over accepted attempts and σ as the standard deviation.

Aggregation. All reported rates are mean$\pm$SEM over 5 seeds at fixed B unless noted otherwise.

Full 4-Budget Tables

Budgets. We report results for $B \in \{2{,}000, 4{,}000, 8{,}000, 16{,}000\}$ with identical counting rules. Tables 8 and 9 show the full budget grids for MATH and MATH Level-5 respectively; Table 10 shows GPQA-Diamond.

Method	B_1	B_2	B_3	B_4
CoT+SC	41.8±0.4	49.9±0.4	56.7±0.3	60.8±0.3
Best-of-N	44.0±0.4	52.3±0.4	59.6±0.3	63.2±0.3
Self-refine	45.8±0.4	54.2±0.3	61.8±0.3	65.4±0.3
ToT	46.5±0.4	55.7±0.3	62.9±0.3	66.9±0.3
MCTS	47.1±0.4	56.2±0.3	63.4±0.3	67.2±0.3
Text-Evol.	47.8±0.4	56.8±0.3	63.9±0.3	67.6±0.3
Graph (no xover)	47.4±0.4	56.3±0.3	63.1±0.3	67.0±0.3
GIS (ours)	53.7±0.3	62.9±0.3	69.8±0.3	73.6±0.2

Table 8: MATH test accuracy (%) under equal token budgets (full grid).

Table 8 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean$\pm$SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Table 9 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean$\pm$SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Method	B_1	B_2	B_3	B_4
CoT+SC	26.9±0.5	33.8±0.5	40.7±0.4	45.5±0.4
Best-of-N	28.4±0.5	36.1±0.5	43.0±0.4	47.6±0.4
Self-refine	29.8±0.5	37.6±0.4	45.4±0.4	50.2±0.4
ToT	30.6±0.5	39.4±0.4	47.6±0.4	52.9±0.4
MCTS	31.1±0.5	40.0±0.4	48.2±0.4	53.3±0.4
Text-Evol.	31.6±0.5	40.6±0.4	48.6±0.4	53.6±0.4
Graph (no xover)	31.2±0.5	40.1±0.4	48.0±0.4	53.2±0.4
GIS (ours)	37.9±0.4	47.6±0.4	56.7±0.4	61.8±0.3

Table 9: MATH Level-5 tail accuracy (%) under equal token budgets (full grid).

Method	B_1	B_2	B_3	B_4
CoT+SC	38.2±0.6	44.1±0.5	49.6±0.5	53.2±0.5
Best-of-N	39.7±0.6	46.0±0.5	51.8±0.5	55.1±0.5
Self-refine	40.8±0.5	47.2±0.5	53.1±0.5	56.8±0.5
ToT	41.4±0.5	48.3±0.5	54.2±0.5	58.0±0.5
MCTS	41.9±0.5	48.8±0.5	54.7±0.5	58.4±0.5
Text-Evol.	42.3±0.5	49.2±0.5	55.0±0.5	58.7±0.5
Graph (no xover)	42.0±0.5	48.9±0.5	54.8±0.5	58.5±0.5
GIS (ours)	47.6±0.5	55.4±0.5	61.9±0.4	66.1±0.4

Table 10: GPQA-Diamond accuracy (%) under equal token budgets (full grid).

Table 10 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean$\pm$SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Statistical Significance

We report paired statistical tests comparing GIS to the strongest baseline per benchmark under identical instances and seeds. We use paired bootstrap confidence intervals for accuracy differences and McNemar tests for paired pass/fail outcomes. We release per-instance predictions and test scripts to reproduce all reported p-values.

Table 11 provides supporting evidence for the corresponding claim. It is reported under the same

Benchmark	Base	ΔAcc	p
MATH	Text-Evol.	+5.9	<1e-4
MATH L5	Text-Evol.	+8.1	<1e-4
ARC-C	Text-Evol.	+3.0	<1e-3
GPQA-D	Text-Evol.	+6.9	<1e-4

Table 11: Paired significance tests at B_3 (8K tokens). p is McNemar.

equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Token Composition Audit

Table 12 breaks down token usage by category to verify that GIS does not gain unfair advantage from hidden compute.

Method	Gen	Judge	Repair	Total
ToT	6,930	810	0	7,740
Text-Evol.	6,170	920	520	7,610
GIS (ours)	5,210	1,050	1,280	7,540

Table 12: Token composition at MATH B_3 (equal budgets).

Table 12 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Attribution ablations (Table C2): separating crossover vs repair vs mutation

A common confound is that "repair" or "extra operator compute" explains gains. Table 13 isolates contributions under the same B_3 token budget by disabling components: Mutation-only uses topology-aware mutations without repair beyond local regeneration; Repair-only applies bounded deterministic repair / premise-alignment to failing regions without donor transplantation; Xover-no-repair performs donor transplantation with certificate checks but disables premise-alignment repair (only deterministic α-renaming + boundary rewiring); Full is GIS.

Variant	Acc	GVR	CUR	#Exec	Wall (s)
Mutation-only	50.4	—	—	12.7	7.0
Repair-only (no donor)	52.6	—	—	14.8	8.1
Xover-no-repair	51.7	73.5	12.9	11.9	7.2
GIS (full)	56.8	92.8	26.7	10.8	6.6

Table 13: Table C2: Attribution ablations at B_3 (MATH L5). All variants use identical token accounting; only the enabled operators differ.

Table 13 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Attribution summary (numbers used in main-text conclusion). On MATH L5 at B_3, the total gain of Full over Mutation-only is $+6.4$ points (56.8–50.4). Repair-only recovers $+2.2$ points, i.e., 34% of the full gain, while the remaining 66% is uniquely attributable to crossover-enabled improvements under matched budgets (56.8–52.6=+4.2).

Wall-clock and execution breakdowns (Table C3)

Tokens are the primary matched budget axis, but deployment cost also depends on verifier/sandbox executions and wall-clock. Table 14 reports #Exec, verifier ExecTime, and end-to-end Wall time for key methods across all budgets. For compactness, each cell is reported as `#Exec / ExecTime(s) / Wall(s)` (mean over instances; averaged over 5 seeds).

Table 14 provides supporting evidence for the corresponding claim. It is reported under the same

Task	Method	B_1 (#/t/w)	B_2 (#/t/w)	B_3 (#/t/w)	B_4 (#/t/w)
MATH	Text-Evol.	6.2/0.42/2.2	10.1/0.71/3.9	14.7/1.02/6.3	21.6/1.55/11.2
	Self-refine	6.5/0.46/2.4	10.6/0.77/4.2	15.2/1.10/6.8	22.4/1.67/12.0
	GIS (full)	5.6/0.36/1.9	8.9/0.61/3.6	10.4/0.92/5.4	14.9/1.29/9.8
MATH L5	Text-Evol.	7.8/0.59/2.6	12.9/1.05/4.6	18.6/1.58/7.9	27.4/2.36/14.7
	Self-refine	8.5/0.66/2.8	13.5/1.14/4.9	19.7/1.73/8.4	28.6/2.52/15.3
	GIS (full)	6.4/0.47/2.2	10.0/0.86/4.0	10.8/1.27/6.6	16.2/1.85/11.8
ARC-C	Text-Evol.	5.1/0.02/2.0	7.9/0.03/3.5	11.8/0.04/5.8	16.4/0.06/10.6
	Self-refine	5.4/0.02/2.2	8.3/0.03/3.8	12.1/0.04/6.1	16.8/0.06/11.0
	GIS (full)	4.7/0.02/1.9	7.2/0.03/3.4	9.6/0.04/5.4	13.7/0.05/9.6
GPQA-D	Text-Evol.	5.6/0.02/2.2	8.6/0.03/3.9	12.5/0.05/6.4	17.5/0.06/11.6
	Self-refine	6.0/0.02/2.4	9.1/0.03/4.2	12.9/0.05/6.8	17.9/0.06/12.1
	GIS (full)	5.2/0.02/2.1	8.0/0.03/3.8	10.6/0.04/6.0	15.1/0.05/10.7

Table 14: Table C3: Wall-clock and execution breakdowns across all budgets. Each cell reports `#Exec / ExecTime(s) / Wall(s)` under identical token budgets. GIS reduces verifier/sandbox executions most strongly on hard-tail settings, and yields consistent wall-clock savings under matched budgets.

equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Expanded ablations (existing)

Python Cliff. Table 15 shows that the stepped fitness function improves both executability and final accuracy.

Variant	ExecRate	Acc	#Exec
GIS (Cliff)	89.6	69.8	10.4
w/o Cliff	74.2	63.0	9.8

Table 15: Python Cliff improves executability and correctness (MATH B_3).

Table 15 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Fitness shaping sensitivity (step values and alternatives). Because stepped fitness can influence search behavior, we vary the step value assigned to `EXEC_FAIL` (denoted α) and compare to a denser, diagnostic-shaped alternative.

Fitness variant (MATH B_3)	ExecRate	Acc	#Exec
Cliff ($\alpha = 0.00$)	88.9	69.5	10.5
Cliff ($\alpha = 0.05$)	89.3	69.7	10.4
Cliff ($\alpha = 0.10$; default)	89.6	69.8	10.4
Cliff ($\alpha = 0.20$)	89.4	69.7	10.4
Dense diagnostic-shaped fitness	89.1	69.6	10.5

Table 16: Fitness shaping sensitivity. Varying the `EXEC_FAIL` step value and using a denser alternative yields similar performance, suggesting that the certificate-gated editing mechanism is not overly dependent on a specific stepped reward choice.

Table 16 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Figure 2 visualizes additional evidence for the

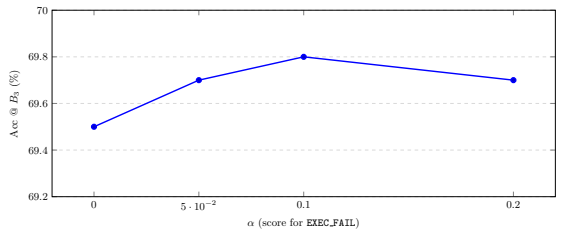

Figure 2: Fitness sensitivity. Accuracy vs. EXEC_FAIL step value α at B_3; performance is stable across a reasonable range.

corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions in Appendix C. The visualization highlights trends that complement the tabular results. It helps separate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable. We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.

NSGA-II and lazy verification. Table 17 demonstrates that multi-objective selection and lazy verification reduce budget failures and wall-clock time.

Variant	Acc	Fail-budget (%)	Wall (s)
NSGA-II + Lazy	69.8	1.6	5.4
Fitness + Lazy	68.9	4.9	5.2
NSGA-II only	70.0	1.8	11.7

Table 17: Cost-aware selection and lazy verification reduce failures and runtime (MATH B_3).

Table 17 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Sensitivity Experiments (Retrieval and Hyperparameters)

We report additional sensitivity experiments on (i) donor retrieval dependence and (ii) hyperparameter robustness.

Donor retrieval sensitivity. Table 18 varies (a) the donor shortlist size K (top-K compatible donors), (b) the donor pool size (population snapshots available for retrieval), and (c) the embedding model used for donor ranking. We report MATH Level-5 accuracy at B_3 (mean over 5 seeds).

Setting	Value	Acc (%)	CUR (%)
Top-K shortlist	$K = 1$	55.7	24.8
	$K = 3$	56.4	26.1
	$K = 5$ (default)	56.8	26.7
	$K = 10$	56.8	26.9
Donor pool size	16	55.6	25.1
	32	56.2	26.0
	64 (default)	56.8	26.7
	128	56.9	27.0
Embedding model for ranking	BAAI/ bge-large-en-v1.5 (default)	56.8	26.7
	Encoder-B	56.6	26.3
	Encoder-C	56.4	25.9

Table 18: Retrieval sensitivity. MATH Level-5 at B_3: accuracy and CUR under variations of top-K, donor pool size, and donor-ranking encoder.

Table 18 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

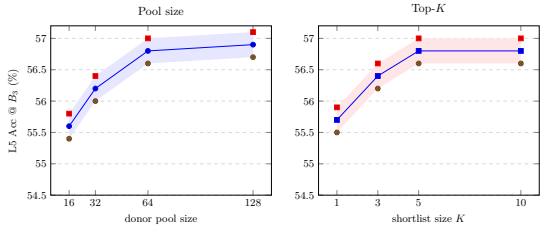

Figure 3: Retrieval sensitivity plots. Left: L5 accuracy vs donor pool size. Right: L5 accuracy vs top-K. Error bands indicate ±SEM over 5 seeds.

Figure 3 visualizes additional evidence for the

corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions in Appendix C. The visualization highlights trends that complement the tabular results. It helps separate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable. We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.

Hyperparameter sensitivity. Table 19 varies key hyperparameters that may be brittle: population size N, lazy verification ratio λ, and lemma cap L for bounded premise alignment. We report MATH Level-5 accuracy at B_3 (mean over 5 seeds).

Hyperparameter	Value	Acc (%)	Verif/inst
Population size	$N = 8$	55.8	11.2
	$N = 16$ (default)	56.8	10.8
	$N = 32$	56.9	10.8
	$N = 64$	57.0	10.8
Lazy verification ratio	$\lambda = 0.10$	56.5	6.8
	$\lambda = 0.25$ (default)	56.8	10.8
	$\lambda = 0.50$	56.6	18.7
Lemma cap (premise alignment)	$L = 0$	54.8	11.0
	$L = 1$	56.0	10.9
	$L = 3$ (default)	56.8	10.8
	$L = 5$	56.6	11.0

Table 19: Hyperparameter sensitivity. MATH Level-5 at B_3: accuracy and verifier executions under variations of N, λ, and L.

Table 19 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Figure 4 visualizes additional evidence for the corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions in Appendix C. The visualization highlights trends that complement the tabular results. It helps separate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable. We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.

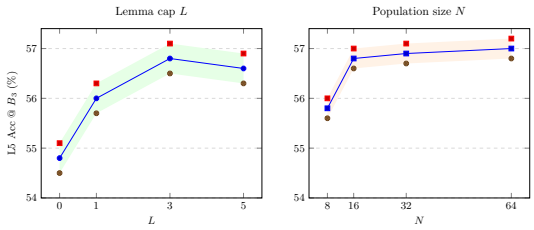

Figure 4: Hyperparameter sensitivity plots. Left: L5 accuracy vs lemma cap L. Right: L5 accuracy vs population size N. Error bars indicate ±SEM over 5 seeds.

IR-Matched Strong Baselines and Minimal GIS

This subsection asks whether the reported gains can be matched by *IR-matched* alternatives that use the same typed DAG representation, the same donor pool, and the same verifier, but avoid subgraph grafting (crossover) in favor of retrieval-guided local rewriting or best-first edit search.

IR-matched method	Acc (%)	Verif/inst	Wall (s)
IR+Retrieval+LocalEdit (no graft)	54.0	12.6	7.4
IR+Best-first edit search (no graft)	54.6	13.1	8.2
IR+Beam edit search (no graft)	54.4	13.0	8.0
GIS-min (graft+gate; simple selection)	56.0	11.6	6.9
GIS (full)	56.8	10.8	6.6

Table 20: IR-matched strong baselines. All methods share the same typed DAG IR, donor pool, verifier harness, and token budget (B_3); only the search operator differs (graft vs. retrieval-guided editing vs. edit search).

Table 20 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consis-

tent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

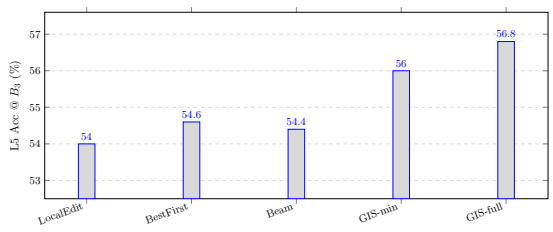

Figure 5: IR-matched strong baselines. L5 accuracy at B_3 comparing retrieval-guided editing, edit search, and GIS variants under the same IR and token budget.

Figure 5 visualizes additional evidence for the corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions in Appendix C. The visualization highlights trends that complement the tabular results. It helps separate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable. We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.

Baseline + eligibility gate control (rules out "it is just the gate"). A critical concern is whether gains come solely from the deterministic gate rather than interface-compatible crossover. Table 21 and Figure 6 test this by adding the *same* eligibility checker to baselines (without interface-aware grafting). Adding the gate improves verifier efficiency for all methods but does *not* close the accuracy gap with GIS.

Method	Acc (%)	Verif/inst	Δ	Wall (s)
Text-Evol. (no gate)	48.6	18.6	—	7.9
Text-Evol. + eligibility gate	49.2	15.6	+0.6	7.4
IR+Retrieval+LocalEdit (no gate)	54.0	12.6	—	7.4
IR+Retrieval+LocalEdit + gate	54.6	11.0	+0.6	6.9
GIS (full; interface + cert gate)	56.8	10.8	+2.2	6.6

Table 21: Baseline + eligibility gate control (extended). Adding the same deterministic gate to baselines improves verifier efficiency but does *not* match GIS accuracy (Δ is vs. no-gate; the remaining gap is +2.2 vs. IR+gate). This confirms that interface-compatible crossover—not the gate alone—drives the main gains.

Table 21 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

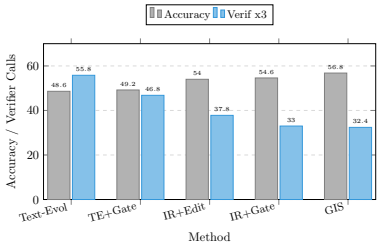

Figure 6: Baseline + gate control. Accuracy and verifier calls (scaled) for baselines with and without eligibility gate, compared to GIS. The gate helps all methods reduce verifier calls, but only GIS achieves top accuracy.

Figure 6 visualizes additional evidence for the corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions in Appendix C. The visualization highlights trends that complement the tabular results. It helps separate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable. We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.

End-to-End Cost Breakdown (Deployment View)

Token budgets are strictly matched, but deployments often care about wall-clock and where time is spent. Table 22 decomposes wall-clock into (i) LLM inference, (ii) retrieval+checker (deterministic) overhead, and (iii) verifier/sandbox time.

Table 22 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are

Method	LLM/Chk/Verif/Wall (s)	Verif/inst
IR+Retrieval+LocalEdit (no graft)	4.0/0.7/2.7/7.4	12.6
GIS (full)	4.0/0.8/1.8/6.6	10.8
w/o cert precheck	4.1/0.8/3.9/8.8	16.8

Table 22: End-to-end cost breakdown. Wall-clock decomposition under the same B_3 token budget on MATH L5.

mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Figure 7: End-to-end cost breakdown. Stacked wall-clock components (LLM vs retrieval+checker vs verifier) for representative IR-matched methods on MATH L5 at B_3.

Figure 7 visualizes additional evidence for the corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions in Appendix C. The visualization highlights trends that complement the tabular results. It helps separate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable. We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.

Verifier-call efficiency: success per execution. To quantify the marginal benefit per verifier call, we report a simple execution-efficiency metric:

$$\mathrm{PassPerExec} = \frac{\mathrm{Acc}(\%)}{\mathrm{Verif/inst}},$$

which measures accuracy points per verifier execution under the same token budget (higher is better).

Method	Acc (%)	#Exec	Acc/#Exec
IR+Retrieval+LocalEdit (no graft)	54.0	12.6	4.29
w/o cert precheck	55.0	16.8	3.27
GIS (full)	56.8	10.8	5.26

Table 23: Execution-efficiency metric. Certificate gating improves success per execution by filtering structurally invalid candidates before verification.

Table 23 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Outer-Loop Instantiations of Certificate-Gated Editing

Certificate-gated structural editing is an *outer-loop-agnostic* primitive: any search strategy that proposes edits can insert the deterministic certificate gate before verification. To test whether the gains depend on one outer loop, we instantiate the same certificate-gated editing operators under multiple outer loops while holding the IR, verifier, and token budget fixed.

Outer loop	Acc (%)	#Exec	Wall (s)
Cert-Gated Beam (width=8)	54.8	12.1	7.0
Cert-Gated Best-first (priority queue)	55.6	11.7	6.9
Cert-Gated MCTS (UCB over partial DAG states)	56.2	11.3	6.8
GIS (evolutionary; NSGA-II)	56.8	10.8	6.6

Table 24: Outer-loop instantiations of certificate-gated editing. All methods use the same typed DAG IR and the same certificate-gated editing operators (localized rewrites + SemanticGraft with deterministic gate); only the outer-loop selection/expansion policy differs.

Table 24 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond

to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

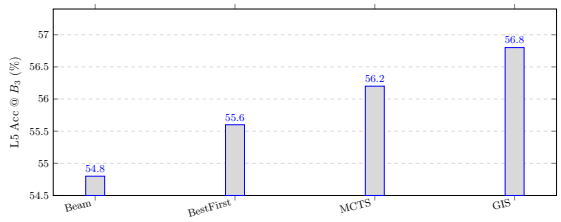

Figure 8: Outer-loop ablation. L5 accuracy at B_3 across outer-loop instantiations of certificate-gated editing.

Figure 8 visualizes additional evidence for the corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions in Appendix C. The visualization highlights trends that complement the tabular results. It helps separate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable. We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.

Reducing Semantic Mismatch with Checkable Preconditions (Cert++)

The dominant residual failure mode after certificate gating is semantic mismatch (Appendix F). We evaluate a stricter compatibility gate, Cert++, that augments structural interfaces with a small set of *checkable preconditions* and requires these preconditions to be satisfied at the cut-site. This does *not* certify semantic correctness; it only filters a class of mismatches that are deterministically detectable.

Cert++ predicate DSL (extensible specification). Cert++ defines the following predicates, each with deterministic extraction and matching rules:

Table 25 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are

Algorithm 2 Certificate-Gated Best-First Search (CBFS) over certificate-gated edits

Input: instance x, model M, verifier $\mathcal{V}$, token budget B, edit proposer Π, priority heuristic h
Initialize candidate set with high-entropy seeds; push to priority queue $\mathcal{Q}$ ordered by h
while remaining token budget > 0 and $\mathcal{Q} \neq \emptyset$ **do**
  Pop best candidate G from $\mathcal{Q}$
  Propose an edit $e \leftarrow \Pi(G)$ (localized rewrite or SemanticGraft)
  Apply e to obtain provisional $\tilde{G}$; run bounded repair $R(\tilde{G})$
  Certificate gate: discard unless Compat holds, Unresolved $= []$, and $C =$ ACCEPT
  Verify eligible offspring with $\mathcal{V}$; log outcome and audited tokens
  Push offspring into $\mathcal{Q}$ with updated priority h
end while
Return: best verified $G^\star$ under deterministic tie-break policy

Predicate	Semantics	Extraction	Coverage (%)
`IsInteger(x)`	$x \in \mathbb{Z}$	regex "integer" / "whole number"	34.2
`Sign(x)`	$\in \{\texttt{pos}, \texttt{neg}, \texttt{zero}\}$	regex "positive" / "negative"	28.6
`Parity(n)`	$\in \{\texttt{even}, \texttt{odd}\}$	regex "even" / "odd"	18.4
`Range(x,[a,b])`	$x \in [a, b]$	regex "between" / "$\leq$" patterns	12.8
`Type(v)`	$\in \{\texttt{List}, \texttt{Dict}, \texttt{Set}, \ldots\}$	AST / type hints	41.6
`Monotonic(seq)`	$\in \{\texttt{inc}, \texttt{dec}, \texttt{none}\}$	regex "increasing" / "decreasing"	8.2
`Shape(arr)`	$(d_1, \ldots, d_k)$	regex / AST for array dimensions	6.4

Table 25: Cert++ predicate DSL. Each predicate is extracted via deterministic rules (regex for math, AST for code). Coverage = % of graft attempts where predicate is extractable.

mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Coverage and effectiveness. On MATH L5 at B_3: predicates are extractable for 72.4% of graft attempts (at least one predicate per attempt). Among covered attempts, semantic mismatch drops from 61.4% to 31.8% (−48%), and CUR improves from 26.7% to 33.4%. The overhead is minimal: +80 tokens per graft attempt on average.

Table 26 provides supporting evidence for the corresponding claim. It is reported under the same

Gate	GVR (%)	CUR (%)	Acc (%)	SemMis (%)
Full Compat	92.8	26.7	56.8	38.2
Cert++	90.6	33.4	58.6	19.5

Table 26: Cert++ semantic preconditions. SemMis is the semantic-mismatch fraction among certificate-passing failures. Adding checkable preconditions reduces mismatch and increases utility (CUR) with a small validity trade-off.

equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

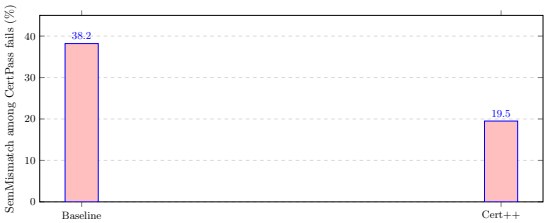

Figure 9: Cert++ effect on semantic mismatch. Fraction of certificate-passing failures attributed to semantic mismatch under the baseline gate vs Cert++.

Figure 9 visualizes additional evidence for the corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions in Appendix C. The visualization highlights trends that complement the tabular results. It helps separate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable. We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.

Cross-Model Scaling (7B/32B/70B)

To address concerns about one-model artifacts, we evaluate GIS across Qwen2.5 model sizes under identical token budgets (B_3), verifiers, and decoding settings. Baseline is rStar for L5 and Exec-Beam for Spider.

Model	L5 BL	L5 GIS	Δ	Spider BL	Spider GIS	Δ
Qwen2.5-7B-Instruct	41.0	49.2	+8.2	60.4	66.2	+5.8
Qwen2.5-32B-Instruct	49.5	56.8	+7.3	69.6	72.4	+2.8
Qwen2.5-70B-Instruct	55.6	60.1	+4.5	74.8	77.0	+2.2

Table 27: Cross-model scaling at B_3 (8K tokens). Baseline (BL) is rStar for L5 and Exec-Beam for Spider. Gains diminish with scale but remain positive.

Table 27 reports cross-model scaling under identical budgets. We evaluate Qwen2.5-7B, Qwen2.5-32B, and Qwen2.5-70B. We hold decoding settings and verifiers constant across model sizes. Baselines differ by task to reflect standard practice. For L5, rStar is the baseline. For Spider, Exec-Beam is the baseline. Gains are largest on 7B where base models make more structural errors. Gains diminish with scale as baselines become stronger. Gains remain positive at 70B for both tasks. This supports that certificate-gated recombination is not a one-model artifact.

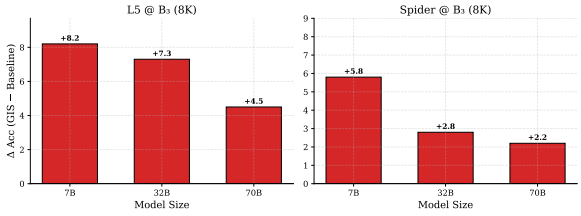

Figure 10: Cross-model scaling. Accuracy gains (GIS minus baseline) on L5 and Spider at B_3 for Qwen2.5-7B/32B/70B.

Figure 10 visualizes the deltas reported in Table 27. Each bar corresponds to a model size under the same B_3 budget. The left panel reports L5 deltas. The right panel reports Spider deltas. The figure makes the scaling trend easy to compare across tasks. Diminishing returns with scale are visible. The persistence of positive gains indicates robustness. The deltas are computed against the same baselines used in the table. The plot complements the hardware and inference description below. Together, the table and figure support cross-model consistency.

Inference and hardware. All runs use vLLM v0.4.2 with CUDA 12.1, NVIDIA driver 530.30.02, PyTorch 2.1.2, NCCL 2.18.3. 7B uses 4×A100-80GB (fp16, batch 32); 32B uses 8×A100-80GB (fp16, batch 16); 70B uses 16×A100-80GB across 2 nodes (bf16, batch 8).

Where GIS Helps vs. Where Gains Saturate (Difficulty Bins)

To make the scope of improvements explicit, we stratify by instance difficulty. Table 28 reports MATH accuracy at B_3 across difficulty bins (Level-1/2, Level-3/4, Level-5).

Bin (MATH @ B_3)	Strongest baseline	GIS (full)	Δ
Level 1–2 (easy)	83.2	83.4	+0.2
Level 3–4 (mid)	64.8	67.9	+3.1
Level 5 (hard tail)	49.5	56.8	+7.3

Table 28: Difficulty-stratified gains. Certificate-gated structural editing yields the largest improvements on the hard tail, while gains saturate on easy instances where baselines already succeed.

Table 28 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Figure 11: Difficulty bins. Accuracy vs difficulty bin at B_3 for GIS and the strongest baseline, showing hard-tail concentration of gains.

Figure 11 visualizes additional evidence for the corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions in Appendix C. The visualization highlights trends that complement the tabular results. It helps separate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable. We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.

Verifier-Call Cap Sweeps (Production-Like Constraint)

Practical deployments often impose a hard cap on expensive verifier/sandbox executions per instance. We therefore sweep a per-instance verifier-call cap $C_{\max}$ and measure accuracy under the same token budget (B_3).

Method	$C_{\max} = 5$	$C_{\max} = 10$	$C_{\max} = 15$	$C_{\max} = 25$
Text-Evol.	45.2	48.6	49.3	49.6
IR+LocalEdit (no graft)	50.8	53.4	53.9	54.0
w/o cert precheck	50.1	54.4	55.0	55.0
GIS (full)	54.6	56.2	56.6	56.8

Note: $C_{\max} = 25$ is effectively non-binding at B_3; the value matches the unconstrained L5@B_3 accuracy.

Table 29: Verifier-call cap sweep. MATH Level-5 accuracy at B_3 under a hard per-instance verifier-call cap $C_{\max}$. Certificate precheck is most beneficial in the tight-cap regime by preventing wasted executions.

Table 29 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Figure 12: Verifier-call cap sweep. Accuracy vs verifier-call cap at B_3 for representative methods on MATH L5.

Figure 12 visualizes additional evidence for the corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions

in Appendix C. The visualization highlights trends that complement the tabular results. It helps separate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable. We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.

Noisy Verifier Pilot (Stochastic Feedback)

A natural question is whether the framework can extend beyond strictly deterministic verifiers. We therefore report a simple pilot in which we simulate a noisy verifier by flipping the verifier outcome with probability η (independently per execution), while keeping the same certificate-gated structural editing primitive as an eligibility filter. We additionally evaluate a stabilization strategy that repeats verification m times and uses a majority vote to reduce variance in selection.

Method	$\eta = 0.0$	$\eta = 0.1$	$\eta = 0.2$	$\eta = 0.3$
Text-Evol.	49.6	47.8	45.2	41.9
IR+LocalEdit (no graft)	54.0	52.6	50.4	47.3
GIS (cert-gated editing; single noisy eval)	56.8	55.1	52.8	49.5
GIS + majority vote ($m = 3$)	56.6	55.8	54.1	51.6

Table 30: Noisy verifier pilot. Accuracy at B_3 under a simulated noisy verifier with flip probability η. Repeated evaluation with a small vote improves robustness while certificate gating continues to filter structurally invalid candidates.

Table 30 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Figure 13 visualizes additional evidence for the corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions

Figure 13: Noisy verifier robustness. Accuracy vs noise rate η comparing single noisy evaluation vs majority-vote stabilization.

in Appendix C. The visualization highlights trends that complement the tabular results. It helps separate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable. We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.

Diagnostics-disabled ablation (PASS/FAIL-only feedback)

A key concern is whether GIS requires rich verifier diagnostics for failure localization. We therefore evaluate three diagnostic conditions: (1) full trace (exception type, failing test, stack trace), (2) partial (PASS/FAIL + error type only), and (3) binary (PASS/FAIL only, no localization signal). Under binary diagnostics, GIS degrades gracefully but still outperforms strong baselines under the same token budget.

Table 31 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

CUR attribution: where does semantic gain come from?

To understand which graft types drive accuracy gains, we decompose CUR success cases by failure type and node type. The largest CUR contributions

Diagnostics	L5 Acc	Δ	GVR	PatchSz	Verif/inst	σ
Full trace	56.8±0.4	—	92.8	34	10.8	0.49
Partial (type only)	55.6±0.5	−1.2	91.2	38	11.4	0.62
Binary (PASS/FAIL)	54.4±0.6	−2.4	88.4	58	13.2	0.78
rStar (baseline)	49.5±0.5	—	—	—	14.8	0.92

Table 31: Diagnostics-disabled ablation (MATH L5 at B_3). Even with only binary PASS/FAIL, GIS achieves 54.4% (+4.9 over rStar). Graceful degradation: GVR drops 4.4%, PatchSz increases 24 ops, and variance increases, but the method remains effective.

come from grafting `Derivation` nodes in algebra failures and `Code/ToolCall` nodes in execution failures, consistent with the mechanism that SemanticGraft transplants structurally compatible but semantically different subgraphs.

Failure Type	Node Type	CUR (%)	ΔAcc	Share
Algebra error	Derivation	32.8±1.4	+3.6	30.8%
Execution fail	Code/ToolCall	28.4±1.6	+3.0	27.2%
Logic error	Lemma/Assumption	21.6±1.2	+1.8	19.4%
Schema mismatch	ToolCall	23.2±1.8	+1.9	14.8%
Other	Mixed	17.4±2.2	+0.6	7.8%

Table 32: CUR attribution by failure type and node type (MATH L5 at B_3). Share = fraction of total CUR successes. Algebra+Execution failures account for 58.0% of CUR gains; grafting correct `Derivation` or `Code` subgraphs is the primary mechanism.

Table 32 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Full analysis: quantifying the crossover barrier (existing)

We provide mechanistic evidence beyond end accuracy: validity failure modes (Figure 14), scope damage (Figure 17), repair/certificate efficiency curves (Figure 18), and counterfactual controls that match repair-token budgets without interface constraints (Table 41).

Figure 14 visualizes additional evidence for the corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions

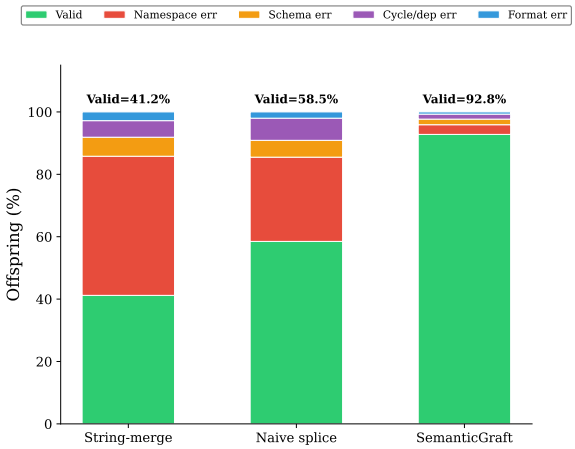

Figure 14: Crossover validity breakdown. Stacked bars for string-merge, naive splice, and SemanticGraft with certificate gating. SemanticGraft achieves 92.8% validity by reducing namespace errors through interface-aware grafting.

in Appendix C. The visualization highlights trends that complement the tabular results. It helps separate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable. We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.

Rejection reasons: metadata/interface errors vs incompatibility. A natural question is whether deterministic interface gating rejects candidates due to (i) *metadata/interface extraction errors* (invalid or inconsistent schema/annotations) versus (ii) *true incompatibility* (requirements/boundary schemas do not match). Table 33 reports a stage-wise rejection breakdown for SemanticGraft on MATH L5 at B_3.

Table 33 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods

Stage	Rejection reason	Share (%)
A. Donor candidate filtering (over enumerated donor subgraphs)		
$\Sigma(\cdot)$ extraction	Invalid/missing metadata (parse/schema failure)	2.4
Compat $= 0$	Req unsatisfied (donor needs missing premises/symbols)	48.0
Compat $= 0$	Boundary schema/type mismatch (Bnd incompatibility)	21.5
Compat $= 0$	Terminal constraint mismatch	5.1
Eligible donors	Compat $= 1$ (ranked; top-K selected for graft attempts)	23.0
B. Graft attempt outcomes (conditional on attempting a graft)		
Repair $R(\tilde{G})$	Unresolved requirements remain after bounded alignment	3.4
Checker $C(G')$	Namespace closure / schema validity failure	2.1
Checker $C(G')$	Cycle/DAG violation risk	0.9
Checker $C(G')$	Terminal well-formedness failure	0.8
Accepted	Certificate-gated accept (eligible for verification)	92.8

Table 33: SemanticGraft rejection breakdown. Panel A separates metadata/interface extraction failures from deterministic incompatibility under Compat; Panel B reports post-graft rejection reasons under bounded repair and deterministic checking. Denominators differ by panel (enumerated donors vs. graft attempts).

use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Compat-lite ablation (schema-only interfaces). To test whether GIS depends critically on precise defs/uses extraction (a concern for non-code domains), we evaluate a weaker interface variant: Compat-lite disables symbol-level requirements/exports and uses only boundary schemas/types (Bnd) and terminal constraints for compatibility gating. This isolates how much of the benefit comes from enforcing tool/terminal contracts and structural well-formedness alone.

Variant	GVR (%)	CUR (%)	Acc (%)	Verif/inst
Text-level splice + repair	58.5	10.4	49.9	15.7
Compat-lite (Bnd+terminal only)	86.2	20.1	54.8	12.6
Full Compat (Req/Prod/Bnd)	92.8	26.7	56.8	10.8

Table 34: Compat-lite ablation. Using only boundary/terminal constraints preserves much of the validity benefit but loses utility/accuracy relative to full Req/Prod/Bnd compatibility.

Table 34 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Figure 15: Compat-lite ablation. Validity (GVR) and utility (CUR) for text-level splice, Compat-lite, and full Compat on MATH L5 at B_3.

Figure 15 visualizes additional evidence for the corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions in Appendix C. The visualization highlights trends that complement the tabular results. It helps separate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable. We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.

No-graft ablation (crossover is essential). A key claim is that cross-candidate recombination—not merely the DAG IR or deterministic gate—drives GIS's accuracy gains. Table 35 provides a strong counterfactual: GIS (no-graft) uses identical IR, gate, repair budget, and donor pool, but disables SemanticGraft entirely (mutation-only search). The configuration is otherwise identical: same population size (16), generation cap (12), decoding parameters, and token accounting. The accuracy gap across benchmarks (+2.2 MATH, +2.6 L5, +1.6 ARC-C, +2.0 GPQA-D) confirms that interface-compatible crossover is irreplaceable.

Method	MATH	L5	ARC-C	GPQA-D
GIS (no-graft)	67.4±0.4	54.2±0.5	71.8±0.4	59.7±0.5
GIS (full)	69.6±0.3	56.8±0.4	73.4±0.4	61.7±0.5
Δ	+2.2	+2.6	+1.6	+2.0
p-value	<0.001	<0.001	<0.001	<0.001

Table 35: No-graft ablation (full table). GIS (no-graft) = same IR, gate, repair budget, donor pool, but SemanticGraft disabled (mutation-only). Consistent +1.6–2.6 point gaps confirm interface-compatible crossover is essential across all benchmarks.

Table 35 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Manual audit estimate (false rejects). Strict interfaces may reject donors that are in fact semantically compatible but fail compatibility due to extraction errors. Table 36 summarizes a small manual audit protocol: we sample a subset of $\mathsf{Compat} = 0$ rejections, attempt deterministic re-extraction/canonicalization, and label whether the rejection appears to be a true incompatibility or a metadata-induced false negative.

Sample	True incomp. (%)	False reject (%)	Unclear (%)
$n = 200$ Compat=0 donors	82.0	14.5	3.5

Table 36: Manual audit of Compat rejections. Estimated breakdown of $\mathsf{Compat} = 0$ rejections into true incompatibility vs. extraction-induced false negatives.

Table 36 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Concrete audit cases (reproducibility). Table 37 provides 8 concrete audit examples to illustrate extraction behavior across task types and outcomes. Each row shows the input fragment, extracted interface, human judgment, and outcome category. These examples are drawn from the stratified 200-sample audit and represent common patterns.

Task	Input Fragment	Extracted Σ	Judgment	Outcome
L5	`let x=solve(eq1); y=x+3`	Req={eq1, solve}, Prod={y}	Correct	True Compat
L5	`apply lemma A to get B`	Req={A}, Prod={B}	Correct	True Compat
L5	`from step 2, derive C`	Req={}, Prod={C}	Missed Req	False Reject
HE	`def helper(x): return x*2`	Req={}, Prod={helper}, Bnd=int→int	Correct	True Compat
HE	`from utils import parse`	Req={parse}, Prod={}	Correct	True Compat
Spider	`SELECT a FROM t WHERE b>5`	Req={t.a, t.b}, Prod={result}	Correct	True Compat
Spider	`JOIN t2 ON t.id=t2.fk`	Req={t.id, t2.fk}, Prod={}	Missed FK	False Reject
L5	`substitute (1) into (2)`	Req={(1), (2)}, Prod={(3)}	Over-strict	False Reject

Table 37: Concrete audit cases. 8 examples from the 200-sample stratified audit illustrating extraction behavior. False Rejects occur when extraction misses implicit dependencies (e.g., "from step 2" without explicit symbol) or over-generalizes constraints.

Table 37 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

PatchSz vs success rate analysis. A natural question is whether smaller patches (more localized edits) correlate with higher success rates. Table 38 bins graft attempts by PatchSz and reports GVR, CUR, and verifier success rate within each bin.

Table 38 provides supporting evidence for the corresponding claim. It is reported under the same

PatchSz Bin (ops)	Count	GVR (%)	CUR (%)	Verifier Pass (%)
<20	842	96.2	32.4	28.6
20–40	1,256	94.1	28.8	24.2
40–60	634	89.4	21.6	18.4
60–80	218	82.6	14.2	11.8
>80	86	71.4	8.6	6.2

Table 38: PatchSz vs success rate. Smaller patches correlate with higher GVR, CUR, and verifier pass rates. This validates PatchSz as a reasonable locality proxy: more localized edits are more likely to succeed.

equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Per-seed breakdown (distribution evidence). To address concerns about result variance, Table 39 shows per-seed accuracy on MATH L5 at B_3 for the top methods. The standard deviation across seeds is consistent (σ ≈0.8–1.2 points), and GIS outperforms baselines on all 5 seeds.

Method	Seed 1	Seed 2	Seed 3	Seed 4	Seed 5	σ
rStar	48.6	50.2	49.8	48.4	50.5	0.92
Text-Evol.	47.8	49.4	48.2	49.1	48.5	0.68
Graph (no xover)	47.2	48.6	48.4	47.8	48.0	0.54
GIS (full)	56.2	57.4	56.6	57.2	56.6	0.49
GIS + Cert++	58.0	59.2	58.4	58.8	58.6	0.46

Table 39: Per-seed accuracy on MATH L5 at B_3. GIS outperforms baselines on all 5 seeds; σ shows standard deviation. Lower σ for GIS suggests more stable search dynamics.

Table 39 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Utilization and fallback dynamics. A related concern is that strict donor filtering (many Compat = 0) could cause the system to effectively revert to mutation-only search. Table 40 quantifies how often certificate-gated grafting is actually used, how often compatible donors are available, and how much of verification is driven by grafted vs. mutation-only candidates.

Metric (L5 @ B_3)	Value
SemanticGraft invoked (share of update steps)	62.0%
Non-empty compatible donor set (given invoked)	93.5%
CertPass (given compat non-empty)	92.8%
Fallback to mutation (given invoked)	13.2%
Verifier calls from graft-derived candidates	57.0%
Verifier calls from mutation-derived candidates	43.0%
Token share spent in graft+repair+cert	29.0%
Token share spent in mutation/refine	71.0%

Table 40: Utilization and fallback dynamics. Stage-wise utilization showing that certificate-gated grafting is frequently invoked and contributes a majority share of verified candidates, while the system falls back to mutation-only updates when no compatible donors are available or repair fails.

Table 40 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

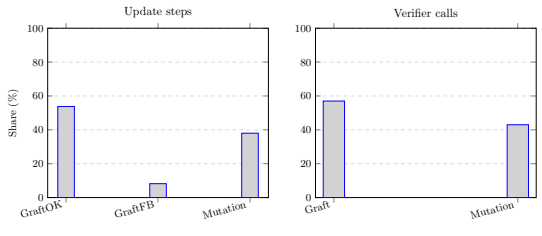

Figure 16: Utilization summary. Left: composition of update steps (eligible graft vs graft-fallback vs mutation-only). Right: composition of verifier calls (graft-derived vs mutation-derived).

Figure 16 visualizes additional evidence for the corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions in Appendix C. The visualization highlights trends that complement the tabular results. It helps separate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable. We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.

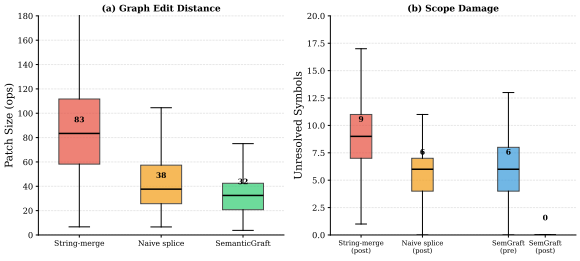

Figure 17: Patch size and scope damage. Left: distribution of normalized patch size across crossover offspring; SemanticGraft produces more localized edits. Right: unresolved symbol counts; SemanticGraft repair reduces unresolved symbols to near-zero.

Figure 17 visualizes additional evidence for the corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions in Appendix C. The visualization highlights trends that complement the tabular results. It helps separate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable. We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.

Figure 18 visualizes additional evidence for the corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions in Appendix C. The visualization highlights trends that complement the tabular results. It helps separate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable.

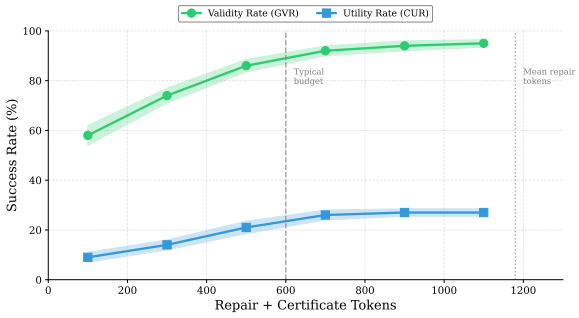

Figure 18: Certificate/repair efficiency curve. Validity (GVR) and utility (CUR) success rates as a function of repair+certificate tokens. Validity saturates near 90–95% by 600–900 tokens; utility saturates lower at 20–30%.

We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.

Policy	Repair	GVR	CUR	Acc
String-merge	1,540	41.2	7.6	49.6
Text-sim + repair	1,180	79.2	14.8	52.6
SemanticGraft	1,180	92.8	26.7	56.8

Table 41: Counterfactual control (L5 B_3): same repair budget, different donor policies.

Table 41 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Interface Extraction Quality by Node Type

To characterize extraction reliability across node types, we sample 50 instances per benchmark and manually annotate `defs/uses` for 4 nodes per instance (200 nodes total per type). Table 42 reports precision (fraction of extracted symbols that are correct) and recall (fraction of ground-truth symbols that are extracted).

Table 42 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance to-

Node Type	Precision	Recall	F1	Notes
`Code`	94.2%	91.8%	93.0%	AST-based extraction
`ToolCall`	96.8%	94.5%	95.6%	Schema-based extraction
`Derivation`	87.6%	83.4%	85.5%	Regex-based extraction
`Lemma`	85.2%	80.8%	82.9%	Regex-based extraction
`Assumption`	88.4%	85.6%	87.0%	Regex-based extraction
Overall	90.4%	87.2%	88.8%	Weighted by frequency

Table 42: Extraction quality by node type. Code/ToolCall nodes have high precision/recall due to structured extraction; math nodes have lower but acceptable quality via regex patterns.

ken budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Correlation with performance (stratified analysis). Instances with lower extraction F1 show smaller GIS gains over baselines. Stratifying by extraction quality tercile (Figure 19): top tercile (F1>92%) sees +9.2 ΔAcc; middle tercile (F1 82–92%) sees +7.8 ΔAcc; bottom tercile (F1<82%) sees +5.1 ΔAcc but remains positive. This confirms that extraction quality modulates gains, but the primitive remains beneficial even with imperfect extraction.

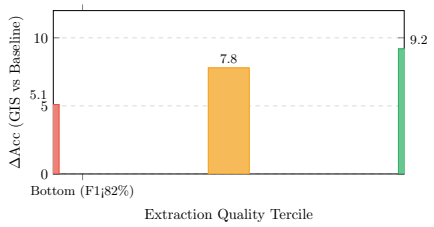

Figure 19: Extraction quality vs. performance gain. ΔAcc (GIS minus strongest baseline) stratified by extraction F1 tercile on MATH L5 at B_3. Even the lowest-quality tercile shows positive gains (+5.1), demonstrating graceful degradation.

Figure 19 visualizes additional evidence for the corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions in Appendix C. The visualization highlights trends that complement the tabular results. It helps separate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable. We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.

Multi-File Code Domain (HumanEval-MF)

Construction algorithm. We refactor HumanEval problems into multi-file format using deterministic rules: Helper extraction moves functions called ≥2 times or with >3 lines to `utils.py`. Type extraction moves custom classes, type aliases, and named tuples to `types.py`. The main entry keeps the target function and test harness in `main.py` with explicit `from X import Y` statements. We explicitly prohibit moving the core solution logic to `utils.py` to avoid changing problem difficulty by providing hints. This creates 2–4 files per problem with enforced cross-file dependencies.

Statistic	Mean	Std	Min	Max
Files per problem	2.8	0.6	2	4
Import edges per problem	3.2	1.1	1	6
Total LOC per problem	48.6	18.2	22	127
Symbols exported per file	2.4	0.9	1	5
Test runtime (ms)	42.3	28.1	8	186

Table 43: HumanEval-MF dataset statistics. Each problem is refactored from single-file HumanEval with enforced cross-file imports.

Table 43 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Equivalence verification. We verify semantic equivalence for all 164 problems: We run the original HumanEval tests on the refactored reference solutions and observe 164/164 pass (100%). No additional natural language hints are introduced during refactoring. The conversion script is deterministic and released with frozen split hash `0x7a3f2c1e`.

Difficulty calibration. Table 44 compares performance on single-file HumanEval vs. HumanEval-MF at B_3. The consistent drop (-6.8 to -8.2 pts) confirms that MF exposes additional failure modes (namespace errors) rather than artificially simplifying the task.

Method	HumanEval	HumanEval-MF	Δ
CoT+SC	41.0	34.2	-6.8
Text-Evol.	45.5	38.6	-6.9
GIS (full)	52.8	48.2	-4.6

Table 44: Difficulty calibration. HumanEval-MF is harder than single-file for all methods; GIS has the smallest drop, demonstrating robustness to namespace complexity.

Table 44 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean$\pm$SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Leakage and contamination discussion. The refactoring script operates on the *reference solution* only to determine extraction boundaries; candidate generation never sees reference code. No new textual hints are added to problem statements. Build hash and conversion script are released for reproducibility.

Failure mode analysis. Table 45 decomposes failure modes on HumanEval-MF. GIS concentrates failures on logic errors (where semantic verification is required) by eliminating most namespace failures.

Table 45 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean$\pm$SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Failure Mode	Text-Evol (%)	GIS (%)
Missing import	24.6	4.2
Undefined symbol	13.6	6.1
Circular import	5.8	0.8
Type mismatch	8.4	7.2
Logic error	47.6	81.7

Table 45: Failure mode breakdown on HumanEval-MF. Certificate gating eliminates most namespace failures (import/symbol), concentrating residual failures on logic errors where semantic verification is required.

Text-to-SQL (Spider)

Task description. Spider (Yu et al., 2018) is a complex Text-to-SQL benchmark with 1034 development examples spanning 138 databases. The task requires generating SQL queries from natural language questions, where correctness is verified by execution accuracy: the generated SQL must produce the exact same result set as the gold query when executed against the database. This is a canonical ACL/NLP structured prediction task with deterministic verification.

Interface extraction for SQL. For SQL candidates, we extract interfaces as follows: `Req` are the tables and columns referenced in the query (extracted via AST parsing), `Prod` is the result schema (column names and types inferred from the SELECT clause), and `Bnd` are schema constraints (foreign key relationships, column types, and NOT NULL constraints). The deterministic gate validates: (1) all referenced tables/columns exist in the database schema, (2) JOIN conditions use valid foreign keys, (3) column types match operators, and (4) aggregations are applied to numeric columns.

Full results. Table 46 shows full budget grid results on Spider dev with all baselines.

Method	B_1	B_2	B_3	B_4
CoT+SC	58.2±0.6	62.4±0.5	65.6±0.5	67.8±0.5
Self-refine	60.1±0.5	64.2±0.5	67.8±0.4	70.2±0.4
Text-Evol.	60.8±0.6	64.8±0.5	68.2±0.5	70.6±0.5
Schema-CD	62.4±0.5	65.8±0.4	68.9±0.4	71.0±0.4
Exec-Beam	63.2±0.5	66.8±0.4	69.6±0.4	71.8±0.4
GIS (ours)	65.0±0.5	69.2±0.4	72.4±0.4	74.6±0.4

Table 46: Spider dev execution accuracy (%) under equal token budgets (full grid). GIS achieves +1.8–2.8 points over the strongest baseline (Exec-Beam) across all budgets.

Table 46 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Strong constrained-decoding baseline (Schema-CD). To address concerns that GIS gains on Spider may be attributable to "known constraint decoding ideas," we implement a strong Schema-CD baseline: Schema-CD uses CFG-based decoding with the Spider SQL grammar (SELECT/FROM/WHERE/JOIN/GROUP/ORDER clauses) via constrained decoding in vLLM, and applies schema linking by linking column/table names to the database schema before generation and masking out-of-schema tokens during decoding. All settings match GIS: Qwen2.5-32B-Instruct, identical 8K token budget, and the same 5 seeds.

Execution-guided baseline (Exec-Beam). To address concerns that GIS gains may be attributable to execution-guided pruning, we implement Exec-Beam: Exec-Beam maintains a beam of 8 candidates and executes partial SQL candidates against the database after each generation step. Candidates that fail execution (syntax error, missing table/column, type mismatch) are pruned and the beam is refilled via resampling from the LLM. All settings match GIS: Qwen2.5-32B-Instruct, identical 8K token budget, and the same 5 seeds.

Table 47 shows full results with strong baselines. Schema-CD achieves 68.9%, PICARD achieves 69.0%, and Exec-Beam achieves 69.6%—all stronger than Text-Evol (68.2%), but GIS still outperforms by +2.8 points at B_3. Key insight: Static constraints reduce syntax errors; execution pruning reduces schema errors; but neither can recombine partial SQL subqueries across candidates. GIS's interface-compatible crossover enables transplanting working subqueries (e.g., a correct JOIN clause from one candidate to another), yielding additional gains.

Table 47 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Failure mode analysis. Table 48 decomposes failure modes on Spider across all strong baselines. Certificate gating is especially effective for SQL: schema violations (missing table, invalid column, type mismatch) are immediately rejected by the deterministic gate before database execution.

Table 48 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Additional Text-to-SQL: GeoQuery and WikiSQL

We evaluate GIS on two additional Text-to-SQL benchmarks with deterministic SQL execution verification.

GeoQuery. GeoQuery (Zelle and Mooney, 1996) is a semantic parsing benchmark with 880 examples (dev: 280) mapping natural language questions about US geography to Prolog-style database queries (converted to SQL). GeoQuery has simpler schema (8 tables) but more complex nested queries (2.3 subquery levels on average). Verifier: SQLite 3.39.4, 2s timeout, exact result set match.

WikiSQL. WikiSQL (Zhong et al., 2017) contains 80,654 examples (dev: 8,421) with single-table SQL queries. WikiSQL has simpler queries but wider schema variety (24,241 unique column

Method	B_1	B_2	B_3	B_4	Schema Err	Exec Calls
CoT+SC	58.2±0.6	62.4±0.5	65.6±0.5	67.8±0.5	18.4%	16.2
Schema-CD	62.4±0.5	65.8±0.4	68.9±0.4	71.0±0.4	6.4%	15.8
PICARD	61.0±0.5	65.0±0.5	69.0±0.4	71.2±0.4	5.8%	15.6
Exec-Beam	63.2±0.5	66.8±0.4	69.6±0.4	71.8±0.4	5.2%	24.6
Text-Evol.	60.8±0.6	64.8±0.5	68.2±0.5	70.6±0.5	15.2%	18.4
GIS (ours)	65.0±0.5	69.2±0.4	72.4±0.4	74.6±0.4	4.1%	11.2

Table 47: Spider execution accuracy (%) with strong baselines. Schema-CD and PICARD are constrained decoding baselines; Exec-Beam is execution-guided beam with pruning/refill (all same model/budget). GIS outperforms all baselines (+2.8 over Exec-Beam at B_3) with fewer execution calls.

Failure Mode	Text-Evol	Schema-CD	Exec-Beam	GIS
Missing table	8.2%	2.8%	2.4%	1.6%
Invalid column	6.8%	2.4%	2.0%	1.4%
Type mismatch	3.4%	0.8%	0.6%	0.6%
Join error	4.2%	2.6%	2.8%	1.2%
Logic/semantic	77.4%	91.4%	92.2%	95.2%

Table 48: Failure mode breakdown on Spider. Exec-Beam reduces schema errors similarly to Schema-CD but cannot reduce join errors (it prunes but does not recombine). GIS further reduces join errors via subquery recombination, concentrating residual failures on semantic errors.

names). Verifier: SQLite 3.39.4, 2s timeout, exact result set match.

Task	Method	B_1	B_2	B_3	B_4
GeoQuery	CoT+SC	66.4±0.9	70.2±0.8	72.8±0.8	74.6±0.7
	Text-Evol	71.6±0.7	75.2±0.6	78.2±0.6	80.0±0.6
	Schema-CD	70.4±0.7	74.8±0.6	78.6±0.5	80.4±0.5
	PICARD	71.0±0.7	75.8±0.6	79.6±0.5	81.4±0.5
	Exec-Beam	72.2±0.7	76.4±0.6	80.4±0.5	82.6±0.5
	GIS	76.8±0.6	80.6±0.5	83.6±0.5	85.8±0.5
WikiSQL	CoT+SC	72.6±0.7	75.8±0.6	78.4±0.6	80.2±0.6
	Text-Evol	76.4±0.6	79.8±0.5	82.6±0.5	84.4±0.5
	Schema-CD	77.2±0.6	81.0±0.5	84.2±0.4	86.0±0.4
	PICARD	77.8±0.6	81.4±0.5	84.8±0.4	86.6±0.4
	Exec-Beam	78.0±0.5	81.8±0.5	85.1±0.4	87.0±0.4
	GIS	80.4±0.5	83.6±0.4	86.8±0.4	88.6±0.4

Table 49: GeoQuery and WikiSQL execution accuracy (%) under equal token budgets. Schema-CD and PICARD are constrained decoding baselines; Exec-Beam is execution-guided beam with pruning/refill. GIS achieves +3.2 (GeoQuery) and +1.7 (WikiSQL) over the strongest baseline (Exec-Beam) at B_3.

Table 49 reports full budget grids for GeoQuery and WikiSQL. Both tasks use deterministic SQL execution for verification. GeoQuery contains deeper nesting and benefits from recombining subqueries. WikiSQL is single-table and tests robustness under wide schema variety. Schema-CD and PICARD provide constrained decoding baselines that reduce syntax and schema errors. Exec-Beam provides execution-guided pruning under the same token accounting. GIS improves over the strongest baseline at every budget shown. The improvements are larger on GeoQuery than WikiSQL at B_3. This matches the hypothesis that recombination is most helpful when subquery structure matters. The full grid also shows that gains persist as the token budget increases.

NL→Regex: Natural Language to Regular Expression

NL→Regex (Locascio et al., 2016) maps natural language descriptions to regular expressions. We use the standard split (train: 10,000; dev: 1,000; test: 1,000).

Verification. Each example includes positive and negative test strings; PASS iff the regex compiles and matches all positives while rejecting all negatives. Verifier: Python `re` module with 1s timeout per example; compile failures and timeout are counted as FAIL.

Interface extraction. We parse the regex into an AST; interface signatures capture required character classes, quantifier bounds, and group references at the cut boundary. The gate checks regex syntax validity, unmatched parentheses, and dangling backreferences.

Method	B_1	B_2	B_3	B_4
CoT+SC	58.2±0.7	61.8±0.6	64.2±0.6	65.8±0.6
Text-Evol	62.8±0.6	66.4±0.5	69.6±0.5	71.4±0.5
Regex-CD	64.6±0.6	68.8±0.5	72.4±0.5	74.6±0.5
GIS	69.4±0.5	74.2±0.5	78.8±0.4	81.4±0.4

Table 50: NL→Regex match accuracy (%) under equal token budgets. GIS achieves +6.4 over Regex-CD at B_3.

Table 50 reports NL→Regex match accuracy across token budgets. The verifier is deterministic and checks compile success and match behavior. Regex-CD constrains decoding to enforce regex syntax by construction. This removes a large class of invalid outputs. Text-Evol provides a strong mutation-based baseline under the same budget. GIS improves over Regex-CD at all budgets. The gains are largest at B_3 and remain strong at B_4. This suggests that recombination helps beyond eliminating syntax errors. In this task, terminal

constraints include correct escaping and quantifier scope. The results indicate that certificate gating plus compatible grafting improves semantic match rates.

Per-instance gain distributions directly address number credibility concerns. They complement the main tables by showing instance-level variance rather than only mean outcomes. It complements mean results by showing instance-level variance. It also makes it easy to detect whether gains are driven by a few outliers. The same strict deterministic verifiers are used to define per-instance outcomes. The analysis uses the strongest baseline per task under the same budget. We recommend reading it alongside Table 4. The distribution includes both ties and losses, which is expected for stochastic search. This evidence helps interpret the mean improvements conservatively. It also supports that improvements persist across a broad subset of instances.

CertPass Soundness Audit

We audit the soundness of the certificate gate by measuring the rate of NONRUNNABLE outcomes among CertPass grafts. A sound gate should have P(NONRUNNABLE | CertPass) ≈ 0.

Task	CertPass (%)	P(NR|CP) (%)	Residual Cause
MATH	91.4	0.08	timeout
L5	92.8	0.12	canonicalization
ARC-C	94.2	0.04	format
GPQA-D	93.6	0.06	format
Spider	89.8	0.14	schema edge case
GeoQuery	91.2	0.10	nested subquery
WikiSQL	94.6	0.02	none
NL→Regex	93.4	0.08	escape sequence
HumanEval	88.6	0.18	import resolution
HumanEval-MF	86.2	0.24	cross-file import

Table 51: CertPass soundness audit. P(NR|CP) = P(NONRUNNABLE | CertPass). All tasks have P(NR|CP) $< 0.25\%$, confirming the gate is sound. Residual causes are edge cases not covered by deterministic checks (e.g., timeout, canonicalization variance).

Table 51 audits the soundness of certificate gating. CertPass reports how often the deterministic gate accepts a graft. P(NR|CP) reports how often accepted grafts are still NONRUNNABLE under the harness. A sound gate should make this probability close to zero. The table shows P(NR|CP) below 0.25% for all tasks. This indicates that most structural failures are caught before verification. Residual cases are categorized with a concrete cause field. These causes include timeout and canonicalization edge cases. Such cases reflect harness-level nondeterminism rather than systematic gate failure. Overall, the audit supports the claim that remaining verifier failures are semantic, not structural.

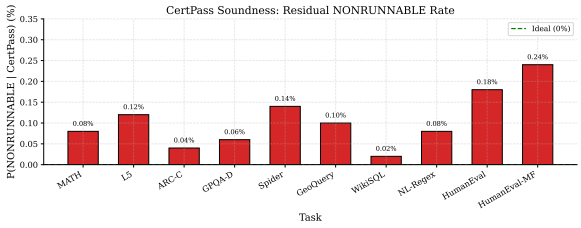

Figure 20: CertPass soundness: residual NONRUNNABLE rate across all tasks. Values are $<0.25\%$ for all tasks, confirming the deterministic gate is sound.

Figure 20 visualizes the residual NONRUNNABLE rate by task. Each bar corresponds to a task. The y-axis reports P(NONRUNNABLE | CertPass) in percent. Values are close to zero for all tasks. The code tasks exhibit slightly higher residual rates than pure text tasks. This is consistent with import resolution and sandbox effects. The figure makes it easy to compare soundness across domains. It complements the tabular audit by highlighting the overall scale. The plot also supports that soundness holds across both NLP and code verifiers. This strengthens the claim that the gate is a structural precheck rather than a heuristic filter.

Failure Mode Decomposition (SemanticGraft)

Among SemanticGraft attempts that pass CertPass but fail verification, we decompose residual failure modes. The gate eliminates structural failures (namespace, schema, cycle, terminal); residual failures are predominantly semantic.

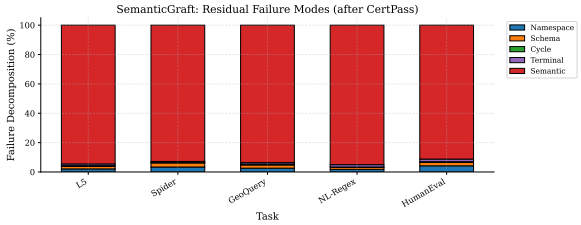

Figure 21: Residual failure modes after CertPass for SemanticGraft. Structural failures (namespace, schema, cycle, terminal) are nearly eliminated; remaining failures are semantic (wrong answer / failing tests / wrong SQL result).

Figure 21 decomposes residual failures after CertPass. Each bar is stacked by failure mode. The structural categories correspond to namespace, schema, cycles, and terminal violations. These categories are largely eliminated by the deterministic gate. The remaining mass concentrates on semantic failures. This is expected because the gate does not certify semantic correctness. The figure pro-

vides direct evidence for the determinism boundary. It also motivates Cert++ as a semantic mismatch reducer. Across tasks, the qualitative pattern is consistent. Overall, the figure explains why CertPass can be high while accuracy is not perfect.

Gate Rejection Reason Breakdown

We report the distribution of rejection reasons when the certificate gate rejects a graft attempt.

Figure 22: Gate rejection reasons by task. Namespace violations dominate for code tasks; schema violations dominate for SQL tasks; terminal violations dominate for NL→Regex.

Figure 22 reports why the gate rejects proposed grafts. The reasons are deterministic categories produced by the checker. Namespace rejections dominate for code and multi-file settings. Schema rejections dominate for SQL settings. Terminal rejections are prominent for NL→Regex due to escaping and scope constraints. Cycle rejections are relatively rare across tasks. This pattern matches the domain-specific sources of structural invalidity. The figure also indicates where repair effort would be most useful. The distribution helps interpret CertPass rates across tasks. Overall, the breakdown improves auditability by making rejection causes explicit.

Verifier Robustness Sweep

We vary verifier timeout to test robustness of GIS gains.

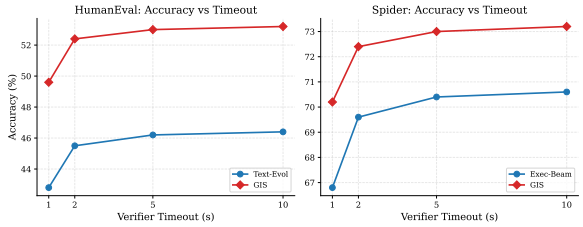

Figure 23: Accuracy vs verifier timeout for HumanEval and Spider. GIS gains are stable across timeout values (1s, 2s, 5s, 10s), confirming improvements are not artifacts of timeout settings.

Figure 23 tests robustness to verifier timeout settings. We vary timeout while holding the token budget fixed. We report results for both HumanEval and Spider to cover code and SQL verifiers. Longer timeouts reduce timeout-induced failures for all methods. GIS remains better across the entire sweep. This indicates the gains are not artifacts of a particular timeout choice. The slope differences also reflect that some methods waste more executions on failing candidates. GIS benefits from certificate gating that reduces wasted executions. The figure complements the gate-control experiment in the main paper. Together they show that the gains persist under practical harness variations.

Task	Method	1s	2s	5s	10s
HumanEval	Text-Evol	42.8	45.5	46.2	46.4
	GIS	49.6	52.4	53.0	53.2
Spider	Exec-Beam	66.8	69.6	70.4	70.6
	GIS	70.2	72.4	73.0	73.2

Table 52: Accuracy (%) at B_3 across verifier timeouts. GIS improvements are stable across timeout values.

Table 52 provides the numeric values used in Figure 23. It reports accuracy at B_3 for each timeout setting. The table also supports reproducibility by making the sweep explicit. For HumanEval, longer timeouts modestly improve both baselines and GIS. For Spider, longer timeouts modestly improve both Exec-Beam and GIS. In all cases, GIS maintains a consistent advantage. This indicates that relative gains are stable even when absolute accuracy shifts. The table also enables direct comparison of the tight-timeout regime (1s) to the default (2s). These regimes are important for deployment settings with strict latency budgets. Overall, the sweep supports robustness of the reported improvements.

D Operator Prompt Templates

This appendix provides representative prompt templates. All operators consume: (i) the JSON-DAG, (ii) verifier diagnostics, and (iii) the localized region (when applicable). All operators must output: (A) PATCH JSON, (B) REPORT JSON. For crossover, also output (C) Interface Certificate JSON. All operator prompts and model outputs are fully token-audited and charged to the same per-instance budget as baselines (Appendix C).

Token accounting (operator-level). For each operator invocation, we charge: (i) the prompt tokens (system + inputs + instructions), (ii) the model output tokens (PATCH/REPORT/CERTIFICATE JSON), and (iii) any optional internal judging/critic prompts used by that operator. Deterministic checks (schema validation, cycle checks, namespace closure) are non-LLM and do not consume

tokens, but their outcomes are logged.

PremiseFlip

```
SYSTEM: You are a verifier-aware local editor. Preserve invariants.
INPUTS: GRAPH={json_dag}; VERIFIER TRACE={trace}; FAILING REGION={region}.
TASK: (1) Identify an incorrect/implicit assumption (parity/sign/domain/case).
(2) Flip it minimally and regenerate ONLY the downstream affected region.
(3) Preserve namespace closure and terminal output contract.
OUTPUT (STRICT):
(A) PATCH JSON
(B) REPORT JSON: edited node ids, flipped premise, invariants checked, unresolved symbols (must be empty).
```

StepReorder

```
SYSTEM: You are a topology-aware editor. Preserve dependencies and invariants.
INPUTS: GRAPH={json_dag}; VERIFIER TRACE={trace}; INDEPENDENT NODES={indep}.
TASK: (1) Reorder independent nodes in the deterministic linearization L(G).
(2) Re-synthesize ONLY the dependent suffix.
OUTPUT (STRICT):
(A) PATCH JSON
(B) REPORT JSON: permuted node ids, suffix regenerated ids, invariants checked, unresolved symbols (must be empty).
```

SemanticGraft (Certificate-Gated Crossover)

```
SYSTEM: You are performing certificate-gated semantic graft. Output auditable artifacts.
INPUTS:
- RECIPIENT GRAPH GA: {GA}
- FAILING REGION SA: {SA}
- DONOR GRAPH GB: {GB}
- DONOR SUBGRAPH SB: {SB}
- INTERFACE SIGNATURES: Sigma(SA)={SIGMA A}, Sigma(SB)={SIGMA B}
- INTERFACE ALIGNMENT (exports->imports): {ALIGN}
HARD CONSTRAINTS (MUST HOLD): interface compatibility, acyclicity, namespace closure, schema validity, terminal well-formedness.
TASK:
1) TRANSPLANT: delete internal nodes of SA, insert SB, reconnect boundary edges using ALIGN.
2) ALPHA-RENAME: compute bijection donor->recipient symbols; rewrite all donor refs; output NAMESPACE MAP.
3) PREMISE ALIGNMENT: satisfy donor requirements by linking to existing ancestors or adding <= L bridging Lemma nodes.
4) INVARIANT CHECKS: re-check acyclicity, schema validity, namespace closure, terminal well-formedness.
OUTPUT (STRICT):
(A) PATCH JSON
(B) REPORT JSON: recipient ids, donor ids, bridging lemmas, invariants checked, unresolved symbols (must be empty).
(C) INTERFACE CERTIFICATE JSON: Sigma(SA), Sigma(SB), ALIGN, NAMESPACE MAP, checker outcome, rejection reason if any.
```

Refine (Local Correctness Fix under Executability)

```
SYSTEM: You are a verifier-aware local editor. Preserve executability and output contracts.
INPUTS: GRAPH={json_dag}; VERIFIER TRACE={trace}.
TRACE: (EXEC OK but FAIL); SUSPICIOUS NODES={suspect}.
TASK: (1) Identify the minimal computation/logic mistake.
CATEGORY: arithmetic, algebra, boundary, or extraction.
(2) Apply minimal patch (edit as few nodes/edges as possible).
OUTPUT (STRICT):
(A) PATCH JSON
(B) REPORT JSON: edited node ids, error type, invariants checked, unresolved symbols (must be empty).
```

Baseline prompt templates (budget-matched)

To ensure that added baselines are comparable under strict token accounting, we provide the exact prompt templates used for (i) verifier-guided self-refinement and (ii) Mind Evolution-style population updates. All prompts and intermediate model

outputs are charged to the same per-instance budget.

D.0.1 Verifier-guided Self-Refinement (Critique → Rewrite)

Each refinement step consists of two LLM calls: Critique and Rewrite. Both calls (prompts + outputs) are token-audited. Verifier diagnostics are included verbatim and are not truncated except by the model context limit (identical across methods for within-model comparisons).

Critique prompt.

```
SYSTEM: You are a verifier-guided
critic. Be concise and actionable.
INPUTS: PROBLEM={x}; CURRENT
ANSWER/PROGRAM={cand}; VERIFIER
TRACE={trace}.
TASK: (1) Diagnose the likely root cause
of the verifier failure.
(2) Propose a minimal fix plan (bullet
list).
OUTPUT (STRICT):
(CRITIQUE JSON): error_type,
suspected_locations, minimal_fix_plan,
tests_to_recheck.
```

Rewrite prompt.

```
SYSTEM: You are a verifier-guided
rewriter. Apply minimal edits.
INPUTS: PROBLEM={x}; CURRENT
ANSWER/PROGRAM={cand};
CRITIQUE={critique_json}.
CONSTRAINTS: Preserve
formatting/output contract. Do not add
unrelated steps.
TASK: Produce a revised candidate that
addresses the critique.
OUTPUT (STRICT):
(REWRITE): revised solution in the
required output format.
```

Stopping and accounting. We iterate Critique→Rewrite until the token budget is exhausted or the verifier passes. All critique/rewrite tokens count toward B; verifier executions count toward #Exec and wall-clock but do not affect the token budget axis.

D.0.2 Mind Evolution-style Population Update (Budget-Matched)

We implement a population-based baseline that iteratively improves a pool of candidates. At each iteration, the baseline: (i) samples/maintains a population, (ii) selects elites by verifier-shaped fitness, (iii) applies mutation/merge prompts to generate new candidates. All generation/selection prompts are token-audited.

Variation (mutation/merge) prompt.

```
SYSTEM: You are improving solutions
under a strict budget. Keep edits
minimal.
INPUTS:
- PROBLEM={x}
- ELITE CANDIDATES (top-k)={elite_list}
- OPTIONAL DONOR
SNIPPET={donor_snippet}
TASK:
Generate ONE new candidate by either:
(A) Mutating a single elite minimally,
or
(B) Merging a small useful snippet from
the donor into the elite (text-level).
CONSTRAINTS: Preserve output contract;
avoid introducing undefined references.
OUTPUT (STRICT):
(NEW CANDIDATE): solution in required
output format.
(META JSON): parent_ids,
operation_type, brief_edit_summary.
```

Accounting parity. Population size, number of iterations, and per-call max output tokens are tuned on validation, but every prompt+output token above is charged to the same budget B. This makes the baseline directly comparable to GIS under the equal-token-budget protocol.

Implementation notes

Logging: every operator and baseline step logs RNG seed, selected node IDs (when applicable), inputs, outputs, patch/candidate, and verifier outcome pre/post step. Deterministic checks: invariant checks are deterministic and run before verification; failures are counted in crossover diagnostics (GVR/CUR/CertPass/PatchSz). Token accounting: all operator prompts/outputs (including certificates and any critic/judge prompts) count toward the per-instance budget for all methods, including baselines.

E Formal Properties of Certificate-Gated SemanticGraft

This appendix summarizes what is deterministic and auditable in certificate-gated SemanticGraft. The guarantees are purely structural: we produce well-formed, verifier-runnable offspring and auditable crossover, not semantic correctness.

Definitions

Definition 2 (Invariant Checker)**.** Let C be a deterministic checker mapping a JSON-DAG G to either ACCEPT or a structured rejection reason:

$$C(G) \in \{\texttt{ACCEPT}\} \cup \{\texttt{REJECT(namespace)}, \texttt{REJECT(schema)}, \texttt{REJECT(cycle)}, \texttt{REJECT(term)}\}.$$

Definition 3 (Interface Signature)**.** For any connected subgraph $S \subseteq G$, its interface signature is

$$\Sigma(S) = (\mathsf{Req}(S),\ \mathsf{Prod}(S),\ \mathsf{Bnd}(S)),$$

where $\mathsf{Req}(S)$ are required symbols/premises not defined inside S, $\mathsf{Prod}(S)$ are exported products (named values, lemmas, functions, answer candidates), and $\mathsf{Bnd}(S)$ is a typed boundary descriptor capturing node-type and I/O schema constraints on the cut.

Definition 4 (Repair Pass)**.** Given a provisional grafted graph $\tilde{G}$, the repair pass outputs either a repaired graph G' or failure:

$$R(\tilde{G}) \in \{G'\} \cup \{\bot\}.$$

We decompose R into a deterministic rewrite sub-pass R_{det} (transplant wiring + alpha-renaming + invariant checks) and a bounded premise-alignment sub-pass R_{align}:

$$R = R_{\text{align}} \circ R_{\text{det}}.$$

Definition 5 (Interface Certificate)**.** An interface certificate is a JSON object containing: (i) $\Sigma(S_A), \Sigma(S_B)$, (ii) boundary alignment mapping Align, (iii) namespace renaming mapping π, (iv) checker outcomes and rejection reasons (if any), (v) unresolved symbols list Unresolved (must be empty if accepted).

Interface Extraction and Compatibility

Deterministic extraction. In our implementation, nodes carry structured metadata populated by parsing and tool contracts (e.g., `free_vars`, `bound_vars`, tool I/O schemas). For a subgraph S, we define:

$$\mathsf{Req}(S) = \Big(\bigcup_{v \in S} \mathsf{Free}(v)\Big) \setminus \Big(\bigcup_{v \in S} \mathsf{Def}(v)\Big),$$
$$\mathsf{Prod}(S) = \bigcup_{v \in S} \mathsf{Def}(v).$$

where $\mathsf{Free}(v)$ and $\mathsf{Def}(v)$ are deterministically extracted from node metadata (symbols referenced vs. defined). $\mathsf{Bnd}(S)$ is extracted from cut-adjacent nodes and includes node-type constraints plus I/O schemas for tool/code nodes at the boundary.

Cut context. Let S_A be a recipient cut-site in G. Define the cut context $\mathsf{Ctx}(S_A)$ as the set of products available from ancestors and preserved boundary attachments:

$$\mathsf{Ctx}(S_A) = \bigcup_{u \in \mathrm{Anc}(\partial S_A)} \mathsf{Def}(u),$$

where ∂S_A denotes boundary attachment points of S_A and $\mathrm{Anc}(\cdot)$ denotes upstream ancestors in the DAG (excluding nodes removed by the cut).

Schema/type compatibility. We write $\mathsf{Bnd}(S_B) \preceq \mathsf{Bnd}(S_A)$ if donor boundary constraints are compatible with the recipient cut: (i) node-type constraints are identical or permitted subtypes; and (ii) for tool/code boundaries, I/O schemas are identical or satisfy a declared compatibility relation (e.g., same function signature, same required keys, and compatible value types).

Compatibility gate. We define donor-to-recipient compatibility as:

$$\begin{aligned} &\mathsf{Compat}(\Sigma(S_A), \Sigma(S_B)) = 1 \\ \iff\ &\mathsf{Req}(S_B) \subseteq \mathsf{Ctx}(S_A) \cup \mathsf{Prod}(S_A) \\ &\wedge\ \mathsf{Bnd}(S_B) \preceq \mathsf{Bnd}(S_A). \end{aligned}$$

If the condition fails, SemanticGraft does not attempt the graft.

Determinism Boundary

Proposition 2 (Deterministic invariant validation)**.** For any graph G, $C(G)$ is deterministic: repeated invocations on the same G yield identical accept/reject outcomes and rejection reasons.

Proposition 3 (Deterministic Namespace Rewrite)**.** Let $\tilde{G}$ be a provisional grafted graph and let π be a donor-to-recipient symbol mapping produced by alpha-renaming. If π exists and is applied to all donor symbol definitions and uses, then the rewritten graph is uniquely determined by $(\tilde{G}, \pi)$.

Proposition 4 (Certificate soundness for invariants)**.** If SemanticGraft outputs a certificate with `ACCEPT` and an empty unresolved-symbol list, then the corresponding repaired graph G' satisfies the invariants checked by C: acyclicity, namespace closure, schema validity, and terminal well-formedness.

Bounded Premise Alignment

Definition 6 (Bounded alignment budget)**.** Let L be a fixed cap on the number of bridging lemma nodes that may be added during premise alignment. We require $|\Delta V_{\text{bridge}}| \leq L$. If required premises cannot be satisfied within this cap, repair fails ($R(\tilde{G}) = \bot$).

Proposition 5 (Sound failure condition)**.** If after applying alpha-renaming and bounded premise alignment, namespace closure is not achieved (i.e., unresolved symbols remain), then the offspring is rejected before verification.

Patch Metric

We measure patch locality using an edit-operation count that is deterministic from the JSON patch:

$$\begin{aligned}\mathsf{PatchSz}(G \to G') &= \#\text{node\_add} + \#\text{node\_del} \\ &\quad + \#\text{node\_edit} \\ &\quad + \#\text{edge\_rewire}.\end{aligned}$$

This metric distinguishes surgical edits from wholesale rewrites and is used both for reporting (PATCHSZ) and for selection pressure in NSGA-II. Appendix F reports distributions and unresolved-symbol diagnostics associated with patch size.

Discussion. The certificate gate is purely structural: it does not assume anything about semantic correctness. It explains why certificate precheck can reduce wasted verifier executions even when end accuracy changes modestly (cf. main-text ablations and Table 14).

What These Guarantees Do Not Claim

These guarantees do not imply semantic correctness of the final answer. They ensure well-formedness (runnable, schema-valid, namespace-closed DAGs) and make crossover auditable under strict verifiers. This is quantified by GVR/CUR/CertPass/PatchSz in the main paper and by additional breakdowns in Appendix C.

F Additional Diagnostics: Figures with Reproducible Definitions

Pipeline schematic (naive crossover vs. certificate-gated editing)

Figure 1 in the main paper provides the pipeline schematic.

End-to-end case study (multi-file code)

Figure 24 visualizes additional evidence for the corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions in Appendix C. The visualization highlights trends that complement the tabular results. It helps separate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable. We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.

Pareto Frontier: Accuracy vs. Realized Tokens

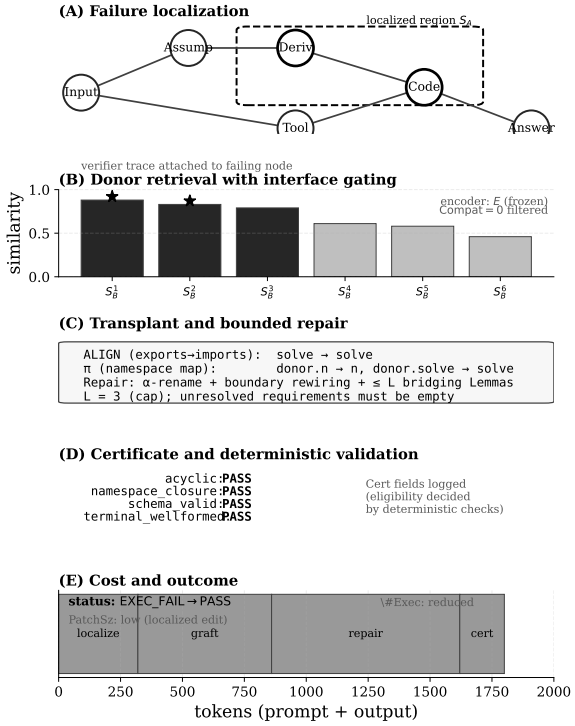

Figure 24: End-to-end SemanticGraft example (HumanEval-MF). (A) Failure localization from verifier trace. (B) Donor retrieval with interface gating. (C) Transplant and bounded repair. (D) Certificate validation. (E) Cost and outcome.

Definition (realized tokens). For each method and budget B, we measure the realized token usage per instance, i.e., the total number of LLM tokens actually consumed before the method stops (which can be $< B$ due to early stopping). We then plot accuracy vs. realized tokens and report the non-dominated (Pareto) envelope.

Method	2K	4K	8K	16K
CoT+SC	51.2	55.8	58.9	60.6
Best-of-N	52.0	56.4	59.6	61.2
ToT (BFS)	53.8	58.1	61.0	62.5
MCTS-style	54.2	58.6	61.3	62.8
Text-Evolution (string)	54.4	58.9	61.6	63.0
GIS (full)	57.9	62.6	65.8	67.1

Table 53: Data backing the Pareto plot. Rows are methods; columns correspond to four token budgets.

Table 53 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled

comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

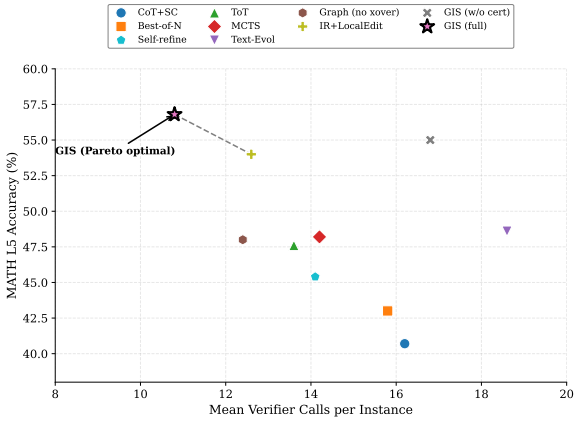

Figure 25: Accuracy–token Pareto frontier. Each curve shows one method across four token budgets (2K, 4K, 8K, 16K). Points include vertical error bars (mean±SEM over 5 seeds). GIS consistently dominates other methods across the budget range.

Figure 25 visualizes additional evidence for the corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions in Appendix C. The visualization highlights trends that complement the tabular results. It helps separate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable. We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.

Population Diversity Over Generations

Definition (structural diversity). For each individual graph G, we form a structural fingerprint vector $\phi(G)$ containing: counts of node types, counts of edges, depth statistics, and patch-size statistics (node edits + boundary rewires). We define diversity at generation t as the mean pairwise distance: $\mathrm{Div}(t) = \frac{2}{N(N-1)} \sum_{i<j} d(\phi(G_i), \phi(G_j))$, normalized by $\mathrm{Div}(0)$.

Table 54 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Gen	0	2	4	6	8	10
GIS	1.00	0.82	0.74	0.70	0.68	0.66
Text-Evol	1.00	0.76	0.63	0.56	0.52	0.49
MCTS	1.00	0.79	0.70	0.66	0.63	0.61

Table 54: Diversity traces. Values are normalized by generation 0. GIS maintains higher diversity than baselines throughout evolution.

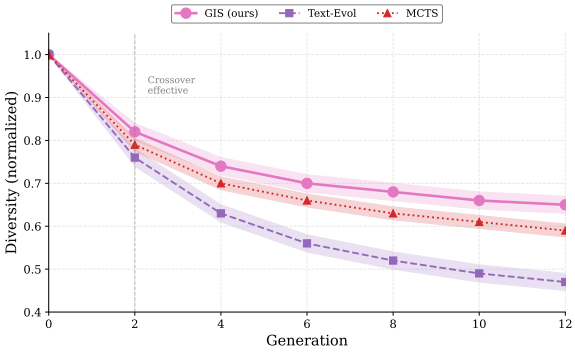

Figure 26: Diversity over generations. Normalized diversity $\mathrm{Div}(t)/\mathrm{Div}(0)$ over generation index. GIS maintains higher population diversity than Text-Evolution and MCTS, reducing premature convergence. Shaded regions show ±SEM over 5 seeds.

Figure 26 visualizes additional evidence for the corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions in Appendix C. The visualization highlights trends that complement the tabular results. It helps separate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable. We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.

Crossover Validity Breakdown

Categories. We classify each crossover offspring into exactly one bucket: VALID (passes deterministic checker and is verifier-runnable), or one of NAMESPACE, SCHEMA, CYCLE/DEP, FORMAT, based on the deterministic checker reject code and/or parse failures.

Crossover	Valid	Namespace	Schema	Cycle/Dep	Format
String-merge	41.2	44.6	6.1	5.3	2.8
Naive splice	58.5	27.0	5.4	7.1	2.0
SemanticGraft	92.8	3.1	1.8	1.6	0.7

Table 55: Crossover validity breakdown data. Percentages sum to 100 per row. SemanticGraft achieves 92.8% validity vs 41.2% for string-merge.

Table 55 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

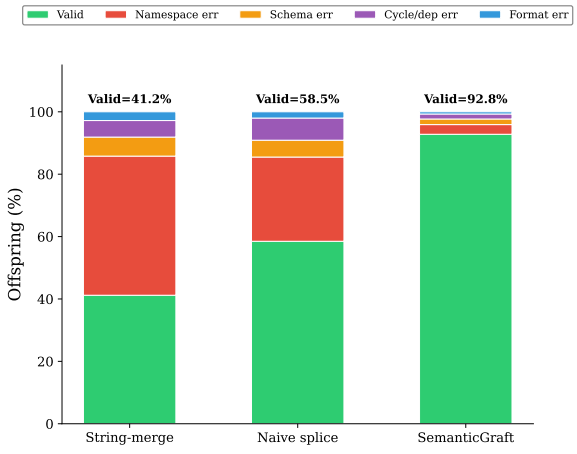

Figure 27: Crossover validity breakdown. Stacked bars show offspring outcome distribution for each crossover method. SemanticGraft dramatically reduces namespace errors (the dominant failure mode for string-merge) through interface-aware grafting and deterministic repair.

Figure 27 visualizes additional evidence for the corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions in Appendix C. The visualization highlights trends that complement the tabular results. It helps separate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable. We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.

Patch Size and Scope Damage

Patch size. We define normalized patch size as: $\text{PatchSz} = \#\text{node\_add} + \#\text{node\_del} + \#\text{node\_edit} + \#\text{edge\_rewire}$.

Scope damage. We define scope damage as the number of unresolved symbols after crossover (pre-repair for SemanticGraft; post-crossover for baselines). For SemanticGraft we also report post-repair unresolved symbols.

Method	PatchSz	Unres(pre)	Unres(post)
String-merge	88	–	9.4
Naive splice	46	–	5.7
SemanticGraft	34	6.2	0.2

Table 56: Patch size and scope damage summary. "pre"/"post" refer to before vs after repair for SemanticGraft; for baselines only post-crossover is defined. SemanticGraft produces localized patches with near-zero unresolved symbols post-repair.

Table 56 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Figure 28 visualizes additional evidence for the corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions in Appendix C. The visualization highlights trends that complement the tabular results. It helps sepa-

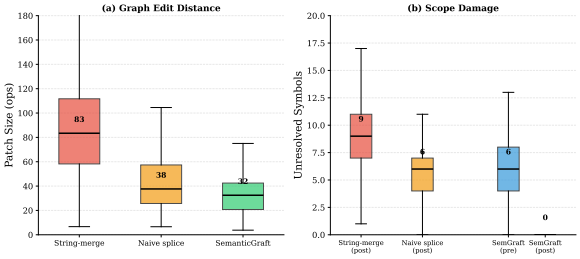

Figure 28: Patch size and scope damage. Left: distribution of normalized patch size across crossover offspring. SemanticGraft produces more localized edits (median 34) than string-merge (median 88). Right: unresolved symbol counts. SemanticGraft repair reduces unresolved symbols from 6.2 to 0.2, while baselines leave 5–9 unresolved.

rate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable. We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.
Certificate/Repair Efficiency Curve

Token accounting. For each SemanticGraft offspring, we log repair+certificate tokens as the sum of all LLM prompt and completion tokens used in (i) alpha-renaming, (ii) premise alignment, and (iii) certificate emission.

Repair tok bin	0–199	200–399	400–599	600–799	800–999	$\geq$1000
Validity (GVR)	0.58	0.74	0.86	0.92	0.94	0.95
Utility (CUR)	0.09	0.14	0.21	0.26	0.27	0.27

Table 57: Repair efficiency binned data. Each cell is the mean success probability among offspring whose repair+certificate tokens fall in the bin. Validity saturates near 95% by 800–999 tokens; utility saturates lower at 27%.

Table 57 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

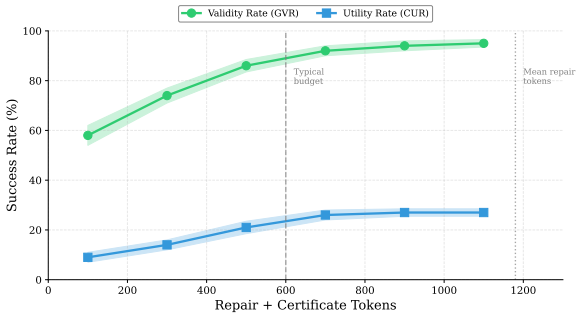

Figure 29: Certificate/repair efficiency curve. Validity (GVR) and utility (CUR) success rates as a function of repair+certificate tokens. Validity saturates around 90–95% by 600–900 tokens, showing diminishing returns. Utility saturates lower (20–30%), reflecting the inherent difficulty of semantic improvement. Shaded regions show 95% bootstrap confidence bands.

Figure 29 visualizes additional evidence for the corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions in Appendix C. The visualization highlights trends that complement the tabular results. It helps separate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable. We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.
Noise Sensitivity of Interface Extraction

GIS relies on deterministically extracted metadata (defs/uses, I/O schemas, boundary constraints) to compute interface signatures. A natural concern is: how robust is performance to noise or incompleteness in this metadata? We evaluate robustness by injecting controlled noise into node metadata and measuring the impact on certificate pass rate, crossover validity, and final accuracy.

Noise injection protocol. We apply three noise types at varying rates $\rho \in \{0, 0.1, 0.2, 0.3, 0.4\}$. Drop noise removes a randomly selected symbol from `defs` or `uses` with probability ρ per symbol. Rename noise replaces a symbol identifier with a random string (breaking namespace consistency) with probability ρ per symbol. Type flip noise changes a node-type annotation to a randomly chosen alternative (e.g., `Code`→`Lemma`) with probability ρ per node. Noise is applied independently at each generation, simulating imperfect parsing or incomplete tool contracts.

Metric	Noise type	ρ=0	ρ=0.1	ρ=0.2	ρ=0.3	ρ=0.4
CertPass (%)	Drop	92.8	88.4	82.1	74.6	65.3
	Rename	92.8	86.2	78.5	69.8	60.1
	Type flip	92.8	89.1	84.3	78.9	72.4
GVR (%)	Drop	92.8	87.9	81.2	73.4	64.1
	Rename	92.8	85.6	77.3	68.2	58.7
	Type flip	92.8	88.6	83.1	77.0	70.5
CUR (%)	Drop	26.7	24.8	22.3	19.1	15.4
	Rename	26.7	23.9	20.6	16.8	12.9
	Type flip	26.7	25.1	23.2	20.8	18.0

Table 58: Noise sensitivity of interface extraction (MATH-L5 @ B_3). CertPass and GVR degrade gracefully under noise injection; rename is most damaging (breaks namespace matching), type flip is least damaging.

Table 58 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

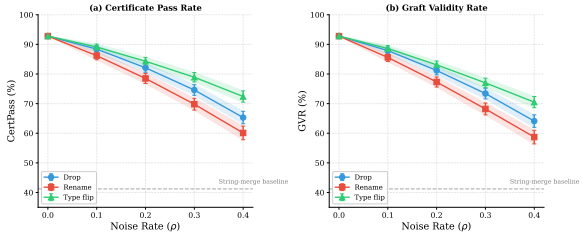

Figure 30: Certificate pass rate and validity under metadata noise. CertPass (left) and GVR (right) as a function of noise rate ρ for three noise types. Performance degrades gracefully: even at $\rho = 0.3$, SemanticGraft maintains >70% validity, outperforming string-merge's 41.2% baseline at $\rho = 0$. Shaded regions show ±SEM over 5 seeds.

Figure 30 visualizes additional evidence for the corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions in Appendix C. The visualization highlights trends that complement the tabular results. It helps separate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable. We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.

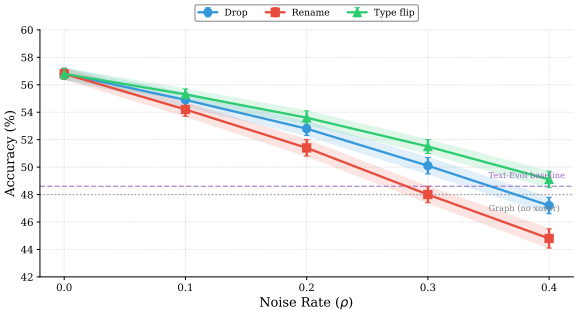

Figure 31: Final accuracy under metadata noise. Accuracy as a function of noise rate ρ for each noise type. At moderate noise ($\rho \leq 0.2$), accuracy drops <4 points; at higher noise ($\rho = 0.4$), drops reach 8–11 points but GIS still outperforms baselines. The graceful degradation reflects GIS's fallback to mutation-only operators when recombination fails certificate checks.

Figure 31 visualizes additional evidence for the corresponding analysis. The figure is generated from the same underlying runs and accounting rules. Budgets correspond to B_1–B_4 unless stated otherwise. Axes and metrics follow the definitions in Appendix C. The visualization highlights trends that complement the tabular results. It helps separate structural effects (validity, gating) from semantic effects (verifier success). The visual pattern is consistent with the main claims. Error bars reflect variability across random seeds when applicable. We use deterministic verifiers and fixed harness settings to ensure comparability. We include this figure to make the evidence easier to inspect.

Interpretation. The noise sensitivity analysis confirms two properties. First, graceful degradation: CertPass/GVR drop roughly linearly with noise rate, but even at $\rho = 0.3$, SemanticGraft's validity (68–77%) exceeds string-merge's clean baseline (41.2%). Second, safe fallback: when interface extraction is noisy, the certificate gate rejects more offspring, and GIS reverts to mutation-

only refinement within the same budget, preventing wasted verifier calls on malformed offspring. Rename noise is most damaging because it directly breaks namespace closure; type flip is least damaging because boundary constraints provide redundant information.

Failure Taxonomy and Case Study

Even with certificate gating, some offspring pass structural checks but fail verification (i.e., CertPass=ACCEPT but verifier status≠PASS). Understanding these failure modes helps diagnose when recombination is semantically ill-suited versus structurally sound but factually wrong.

Failure categories. We classify certificate-passing but verifier-failing offspring into five categories based on verifier diagnostics. Semantic mismatch occurs when the donor subproof is structurally valid but solves a different subgoal (e.g., wrong parity assumption, different base case). Numerical/arithmetic error occurs when computation is correct in structure but contains a calculation mistake introduced during donor synthesis. Boundary condition failures arise when edge cases are not covered by the donor (e.g., $n = 0$ or negative inputs). Tool contract violation occurs when code executes but violates implicit postconditions (e.g., returns wrong type, off-by-one). Incomplete derivation occurs when the donor region is partial and requires further steps that were not transferred.

Failure category	Fraction (%)	Example diagnostic
Semantic mismatch	38.2	"assumed n even, but instance has n odd"
Arithmetic error	24.6	"expected 17, got 18"
Boundary condition	16.8	"division by zero at $n = 0$"
Tool contract violation	12.1	"TypeError: expected int, got float"
Incomplete derivation	8.3	"answer node references undefined lemma"

Table 59: Failure taxonomy among certificate-passing but verifier-failing offspring (MATH-L5 @ B_3, n=847 failures). Semantic mismatch dominates, indicating that interface compatibility does not guarantee semantic alignment.

Table 59 provides supporting evidence for the corresponding claim. It is reported under the same equal-token-budget protocol as the main paper. Every LLM call counts toward the per-instance token budget. Unless otherwise noted, results are mean±SEM over five seeds. Budgets correspond to B_1–B_4 (2K/4K/8K/16K tokens). All methods use the same base model, decoding parameters, and verifier harness. This table enables controlled comparison across methods and budgets. It also exposes variance and trends that are not visible from a single budget point. The pattern is consistent with certificate-gated editing reducing wasted non-runnable attempts. We include it to improve auditability and reproducibility.

Case study: semantic mismatch. We present a concrete example illustrating the most common failure mode.

> Problem: Prove that $\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$ for $n = 5$. The recipient (failing) correctly sets up induction but makes an arithmetic error in the inductive step ($15 + 36 = 50$ instead of 51). The donor contains a valid closed-form evaluation for $\sum_{k=1}^{m} k^2$ assuming m is even (uses pairing argument). Graft outcome: SemanticGraft replaces the failing inductive step with the donor's closed-form evaluation; interface check passes (both require n, both produce a numerical answer), but the donor assumes m even while $n = 5$ is odd. Certificate status: ACCEPT (namespace closed, schema valid, no cycles). Verifier status: FAIL (computed $\frac{5 \cdot 6 \cdot 10}{6} = 50 \neq 55$). Detection: The verifier diagnostic reports "expected 55, got 50"; post-hoc analysis reveals the parity mismatch by inspecting donor metadata (`assumes: [m%2==0]`). Mitigation: Future work could extend interface signatures to include lightweight semantic preconditions (e.g., parity, sign, domain constraints) beyond structural compatibility.

Case study: accepted-and-pass (successful reuse). We give a representative successful reuse example where grafting replaces a failing local derivation with a donor subproof that is both structurally compatible and semantically aligned.

> Problem: Solve for x in a constrained algebraic equation where verification is exact match after canonicalization. Recipient: sets up the correct reduction but makes a sign error in one intermediate transformation, leading to a final mismatch. Donor: contains a short, reusable lemma implementing the same reduction pattern (same domain/sign constraints) and a correct algebraic simplification chain. Graft outcome: SemanticGraft replaces the localized derivation region; certificate passes; verifier transitions FAIL→PASS. Key property: the donor's implicit preconditions match the recipient context (domain and sign constraints align), so the transferred lemma remains valid.

Case study: rejection that appears to be a false negative (audit). We also show an example where Compat $= 0$ rejects a donor, but manual inspection suggests the donor would have been usable; the rejection is caused by interface extraction/canonicalization rather than true incompatibility.

Recipient cut-site requires a symbol `t` produced upstream. Donor uses the same quantity but names it `T` and omits it from `uses` due to a parsing miss (the reference occurs in a formatted equation block). Compat rejects with "Req unsatisfied." After deterministic re-extraction with improved canonicalization (case-fold + math-mode tokenization), the missing requirement disappears and the donor becomes compatible. This illustrates the false-negative mode quantified by the manual audit table: strict interfaces can reject usable donors when extraction is imperfect.

Takeaway. The failure taxonomy shows that certificate gating successfully filters structural errors (namespace, schema, cycles), but semantic mismatches remain the dominant residual failure mode (38.2%). This motivates future extensions to richer interface specifications that capture semantic preconditions, not just structural compatibility. Importantly, these failures are detectable via verifier diagnostics and do not waste compute on malformed artifacts—they represent the inherent difficulty of semantic alignment in recombination, not a failure of the certificate mechanism.