COMPOSITIONAL ANNOTATION SYNTHESIS FOR PROGRAM VERIFICATION WITH LLMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Formal verification can establish that LLM-generated code satisfies its specification, but constructing proof annotations remains costly. With interacting functions, locally provable annotations may fail to compose: a callee's guarantee may omit facts its callers need, or a generated requirement may exceed what its callers establish. Verifier feedback identifies failures but does not by itself determine how dependent annotations should change together. We present CoVeri, which combines bottom-up symbolic reasoning over implementations and current summaries with top-down analysis that traces failed obligations to candidate interfaces. These analyses guide the LLM to strengthen guarantees, relax generated requirements, and jointly revise dependent contracts while preserving fixed specifications. Across three LLM backbones, CoVeri outperforms all evaluated baselines on DafnyComp, VeriEquivBench, and SWE-Dafny. On DafnyComp, it matches the strongest baseline while using a smaller backbone. Ablations support the complementary benefits of bidirectional guidance and joint revision.

1 INTRODUCTION

Code generated by large language models (LLMs) is increasingly incorporated into software repositories (Murphy-Hill et al., 2026; Pinna et al., 2026). Establishing its correctness requires more than passing a finite test suite: stronger tests can expose errors in solutions previously judged correct (Liu et al., 2023). Formal verification offers a complementary guarantee by proving that code satisfies explicit specifications over all permitted inputs (Leino, 2010; Sun et al., 2024; Lattuada et al., 2023; de Moura & Ullrich, 2021). Such proofs depend on function contracts, loop invariants, and other annotations whose construction remains costly. *Annotation synthesis* automates the construction of these contracts and proof annotations to enable program verification. This task has motivated automated inference (Flanagan & Leino, 2001) and, more recently, LLM-based synthesis guided by verifier feedback (Chakraborty et al., 2023; Mugnier et al., 2025; Banerjee et al., 2026). We study this task with executable code and task-fixed requirements held unchanged, while designated annotations may be generated or revised.

As programs include more interacting functions and control-flow regions, annotation synthesis becomes a compositional problem (Leino, 2010; Cobleigh et al., 2003; Xu et al., 2025). A callee's contract is used to prove its callers; a loop invariant must supply facts needed after exit. Local provability therefore does not ensure that inferred summaries compose. Each summary must satisfy two constraints: it must be *feasible*, meaning that the implementation establishes it, and *sufficient* at its uses, meaning that it supplies the information needed by the surrounding proof. Composition can fail because a producer's guarantee is too weak or because a generated consumer requirement exceeds what the producer establishes. Verifier feedback identifies failed obligations, but does not by itself determine which dependent summaries should change or how their changes should be coordinated.

Consider constructing a histogram of the prefix $a_{<k} = (a_0, \ldots, a_{k-1})$ of an array of nonnegative integers, with $1 \leq k \leq |a|$ (Figure 1). The pipeline first calls `Max` to compute the prefix maximum m, then calls `Histogram` to allocate $m + 1$ zero-initialized counters and increment $h[a_i]$ for $0 \leq i < k$. These indexed accesses require $0 \leq a_i \leq m$ for each value counted. Yet `Max`'s initial contract guarantees only $m \in a_{<k}$, while `Histogram`'s generated precondition requires $a_i \leq m$ for every $0 \leq i < |a|$. Both methods verify locally, but their contracts do not establish the pipeline's call obligation.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

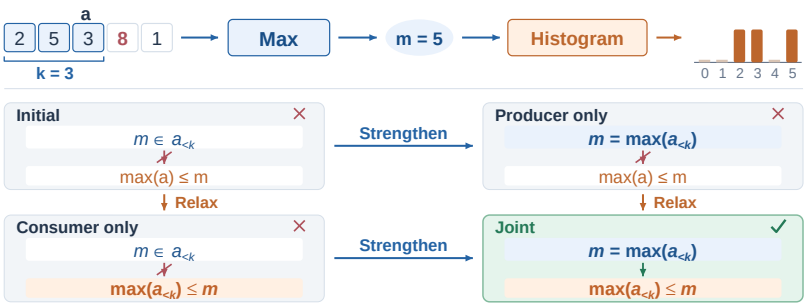

Figure 1: Why contracts must change together. The fixed pipeline constructs a histogram of a nonnegative array prefix. Blue edges strengthen `Max`'s guarantee; orange edges relax `Histogram`'s generated requirement. Neither edit alone makes the call verifiable. Together, they expose and require the same prefix bound; the unused suffix need not satisfy it.

Strengthening only `Max`'s guarantee to $m = \max(a_{<k})$ leaves the call unverifiable because the prefix bound does not cover the unused suffix in `Histogram`'s requirement. Relaxing only the consumer's requirement also fails because the producer has not exposed the prefix bound. Joint revision resolves this mismatch: `Max`'s loop invariant supports the stronger guarantee, while `Histogram`'s indexed accesses require the bound only over the prefix. The example illustrates why repair must consider both what implementations support and what their uses require. Generated internal requirements may be relaxed, but task-fixed requirements remain protected.

We introduce CoVeri, a compositional annotation-synthesis method that combines bottom-up symbolic reasoning with top-down demand analysis. Bottom-up reasoning composes code and current summaries into symbolic relations that provide implementation context. Top-down analysis traces failed obligations to candidate interfaces, retaining their calling contexts and distinguishing fixed goals from generated requirements. Call dependencies group these interfaces for joint revision. The LLM uses the relations and routed demands to propose compatible changes, including strengthening guarantees and relaxing generated requirements. The resulting program is then checked by Dafny. Beyond permitting multiple declarations to change, CoVeri provides explicit guidance on where to revise and what constraints the revisions must satisfy.

Our contributions are threefold. First, we formulate annotation synthesis in terms of feasibility and use-side sufficiency. Second, we develop a bidirectional search procedure for joint revision across dependent interfaces. Third, we introduce SWE-Dafny, a collection of 100 verification tasks derived from SWE-bench Verified (OpenAI, 2024; Jimenez et al., 2024). Across three LLMs, CoVeri achieves the highest final verification rate among the compared methods on DafnyComp (Xu et al., 2025), SWE-Dafny, and both VeriEquivBench settings (Zeng et al., 2026). Under the evaluated search budget on DafnyComp, restricting each LLM proposal to one declaration reduces the number of verified programs by 26 with Qwen3.5-122B-A10B and by 38 with Qwen3.5-35B-A3B, supporting the benefit of coordinated revision.

2 Related Work

Modular verification and annotation inference. Hoare-style reasoning relates commands to preconditions and postconditions (Hoare, 1969), while predicate transformers characterize the entry conditions needed for a desired result (Dijkstra, 1975). Dafny and Verus support modular verification through contracts, loop invariants, and proof annotations: each region is verified using summaries of its dependencies (Leino, 2010; Lattuada et al., 2023). Automated methods infer inductive invariants through abduction (Dillig et al., 2013), procedure specifications through bi-abduction (Calcagno et al., 2009), and recursive summaries through SMT-based refinement of over- and under-approximations (Komuravelli et al., 2014). CoVeri uses modular reasoning to guide LLM search for compatible annotations across dependent regions.

LLM-based annotation synthesis. LLM-based methods generate and refine annotations using verifier feedback. Existing approaches localize helper assertions and retrieve proofs (Mugnier et al., 2025), combine proof strategies with invariant pruning (Banerjee et al., 2026), and use staged re-

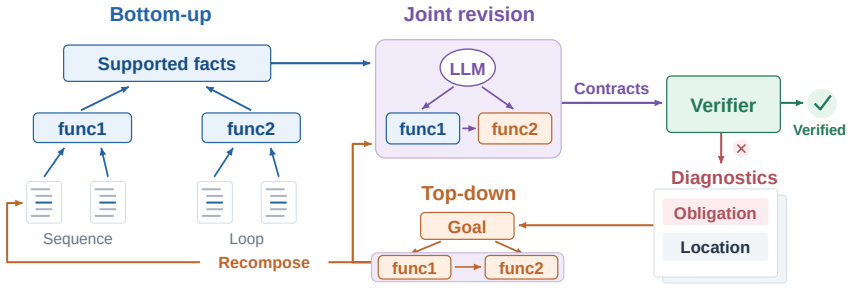

Figure 2: The CoVeri loop. Bottom-up symbolic reasoning composes code and current summaries into symbolic relations. Top-down demand analysis traces failed obligations to candidate interfaces for revision, illustrated by `func1` and `func2`. The relations and routed demands guide joint revision, followed by another whole-program check.

finement, tree search, or iterative repair of implementations and proofs (Yang et al., 2025; Aggarwal et al., 2025; Breen et al., 2026). Other work trains specialized models (Yan et al., 2025) or exploits program structure (Wen et al., 2024; Ding et al., 2026; Wang et al., 2025). AutoSpec retains validated specifications during synthesis (Wen et al., 2024), while Preguss uses staged host–callee synthesis with restrictions on callee preconditions (Wang et al., 2025). CoVeri combines bottom-up symbolic relations with top-down proof demands to guide joint revision of producer guarantees and generated consumer requirements, while preserving fixed specifications.

Benchmarks and specification quality. Specification evaluation examines whether generated contracts capture intended behavior (Sun et al., 2024; Ye et al., 2026b; Zeng et al., 2026). Benchmarks cover proof-hint completion (Loughridge et al., 2024), repository-level implementation and proof synthesis (Ye et al., 2026a), and compositional contract generation across interacting functions (Xu et al., 2025). Our evaluation distinguishes verification under generated contracts from agreement with reference specifications, and also assesses proofs under fixed requirements. SWE-Dafny adds tasks derived from repaired software, with fixed implementations, entry-point requirements, and trusted external contracts.

3 Method

CoVeri synthesizes annotations by combining implementation-side reasoning with use-side proof demands. It first constructs annotations in callee-first order. When whole-program verification fails, it traces unresolved obligations to dependent interfaces and asks the LLM to revise their annotations together. Bottom-up symbolic relations provide implementation context; top-down analysis identifies where additional or revised proof information may be needed. Figure 2 summarizes the search loop. These analyses guide the search, while Dafny determines whether the resulting program verifies under the fixed task requirements.

3.1 Problem Statement

The input consists of a program C, task-fixed requirements F, and permitted annotation edits $\mathcal{A}$. We seek annotations α such that $C[\alpha]$ verifies while preserving executable code and protected annotations. Generated annotations may be revised only where $\mathcal{A}$ permits; task-fixed requirements remain mandatory. For tasks with a fixed entry domain, generated preconditions must hold throughout that domain. A *region* is a syntactic unit of the resolved Dafny AST, such as a statement, sequence, conditional, loop, or declaration. Regions are the units of symbolic composition. A *summary* is the proof information exposed by a region, including a declaration contract or loop invariant; its *boundary* is the location at which that information becomes available to the surrounding proof. The permissions $\mathcal{A}$ determine which boundaries admit edits.

Two conditions guide synthesis. First, a summary must be feasible to implement. For region code C_r, precondition P_r, and postcondition Q_r, this requires the Hoare triple (Hoare, 1969)

$$\{P_r\}\ C_r\ \{Q_r\}. \tag{1}$$

Establishing this condition with an automated verifier may require additional internal proof annotations. Second, the summary must suffice at its uses. Let U_r be the use sites of region r, and let P_r^u and Q_r^u denote the respective instantiations of P_r and Q_r at use $u \in U_r$. With entry and exit states s, s', let Γ_u^{pre} contain the facts available before execution, $\Gamma_u^{\mathrm{frame}}$ the contextual facts justified across execution, and D_u the subsequent proof obligation. Use-side sufficiency requires

$$\Gamma_u^{\mathrm{pre}}(s) \Rightarrow P_r^u(s), \qquad \Gamma_u^{\mathrm{frame}}(s, s') \wedge Q_r^u(s, s') \Rightarrow D_u(s, s'). \tag{2}$$

Facts about the modified state cannot simply be carried forward from the entry. We call the properties needed to discharge these obligations the *demand* on a boundary. A demand describes what the proof needs; it need not already follow from the current summary.

In the histogram example in Figure 1, `Max`'s membership guarantee is feasible but does not supply the prefix bound its consumer needs. Conversely, the consumer's generated whole-array bound cannot be guaranteed by a method that computes only a prefix maximum. The search must therefore consider both the producer's guarantee and the consumer's editable requirement.

3.2 Bottom-Up Symbolic Reasoning

Bottom-up reasoning follows the AST within each declaration and constructs a symbolic transition relation $B_r(s, s')$ for each region. Symbolic states consist of a variable environment, path constraints, and a shared heap for supported integer-array operations, encoded as Z3 expressions (de Moura & Bjørner, 2008). Each relation records the summaries and unresolved obligations on which it depends.

Assignments update symbolic values, sequences compose through intermediate states, and branches retain their guards. Calls use the callee's current contract without unfolding its body: the instantiated precondition becomes an obligation, and the postcondition constrains the returned state. Loops use their current invariants to summarize modified state at exit, subject to initiation and preservation obligations (Leino, 2010). Frame information records which facts remain valid across state changes.

These relations provide candidate consequences, not a correctness certificate. Consequences using unproved contracts or invariants remain conditional on those premises. SMT implication checks test consequences under current premises; unresolved checks and unsupported constructs remain gaps. Native verification must ultimately discharge the associated obligations.

To construct annotations, the analysis seeks consequences expressible using names visible at an editable boundary. It projects exit constraints onto inputs and results where possible, and uses fixed structural templates for simple invariants such as counter bounds. For example, eliminating the internal value t from $t = x + 1 \wedge y = 2t$ yields the boundary clause $y = 2x + 2$. Entry obligations likewise provide candidates for permitted preconditions. Mechanically generated clauses are installed in the current candidate program, and affected relations are recomputed; installation does not imply final acceptance.

In `Max`, the fixed loop invariant bounds every element of $a_{<i}$ by m. The index bound and exit guard establish $i = k$, yielding

$$(\forall j.\ 0 \le j < i \Rightarrow a[j] \le m) \wedge i = k \ \Rightarrow\ (\forall j.\ 0 \le j < k \Rightarrow a[j] \le m). \tag{3}$$

The conclusion uses only method parameters and the result, so it can be exposed as a postcondition. When mechanical generation cannot produce such a clause, the invariant and exit condition still provide context for the LLM. Appendix A specifies the composition rules, annotation construction, and supported fragment.

3.3 Top-Down Demand Analysis

Top-down analysis identifies candidate revision locations from failed proof obligations. Verifier diagnostics and related source locations first identify the failed assertion, contract clause, or safety obligation in the resolved AST. Starting from its variables, the analysis traces dependencies backward. An assignment replaces a tracked assigned variable with the variables read by its right-hand side. A call selects the callee's contract when it produces a tracked value or modifies relevant state. A loop selects its invariant when it modifies a tracked value or governs a relevant exit. At branches,

the analysis follows the diagnostic's branch or combines dependencies across continuing branches. This is dependency-based routing rather than a complete weakest-precondition computation.

Each routed demand retains its originating clause and whether that clause is fixed or generated. Its context includes argument and result substitutions, caller entry conditions, branch guards, and current loop invariants. Calling contexts remain separate, including uses that already verify, so a demand arising under one caller's guards is not imposed unconditionally elsewhere.

When a guarded demand can be expressed in the callee's vocabulary while preserving entry- and exit-state references, a bounded local proof probe adds it temporarily as a callee postcondition. The resulting diagnostics can expose further summary dependencies; the probe does not commit that clause to the candidate program. If the goal, dependency, or argument mapping cannot be resolved, routing falls back to the failed declaration and its directly called interfaces, marking these locations as uncertain.

In Figure 1, the failed call refers to `Histogram`'s requirement on a and m. Tracing m selects `Max`'s contract, while the generated consumer requirement is also eligible for revision. Routing exposes both interfaces; it does not prescribe the replacement clauses. The LLM uses the routed demand and implementation context to propose a stronger producer guarantee, a relaxed generated requirement, or coordinated changes to both.

3.4 Bidirectional Annotation Synthesis

Initial construction. CoVeri visits callees before callers and handles recursive declarations together. Within each dependency group, it composes available relations, installs permitted mechanically generated annotations, and checks the declarations. When an editable obligation or needed summary remains unresolved, the LLM receives the relevant code, current annotations, symbolic context, and available goals and use contexts. A whole-program check then identifies obligations left unresolved by the initial construction.

Demand-guided joint revision. Let E contain declarations with unresolved checks, P contain declarations owning interfaces selected by demand analysis, and $U \subseteq E$ contain those with unknown routing. With $\mathrm{Calls}(u)$ denoting the direct callees of u, the candidate revision set is

$$S = E \cup P \cup \bigcup_{u \in U} \mathrm{Calls}(u).$$

Revision groups are connected components of the call graph induced by S, ignoring edge directions. Groups are processed in source order and bound the declarations the LLM may edit under $\mathcal{A}$.

Before each proposal, CoVeri recomputes the group's symbolic relations and refreshes mechanically generated annotations. If verification still fails, the LLM receives the current code and interfaces, routed demands and their provenance, local proof status, native diagnostics, edit history, and permissions. It may revise one or several declarations in the group. Thus, bottom-up context and top-down demands guide annotation choices, while grouping makes dependent interfaces available for coordinated revision.

Candidate updates and acceptance. Proposals are applied within the permitted scope while preserving executable code and protected annotations. An omitted generated postcondition is dropped if it follows from the revised contract; otherwise, it is restored if a proof check confirms that it remains provable. An update may become the current search candidate even while some checks fail. This differs from accepting a solution, which requires successful whole-program verification under the fixed requirements. After each update, verification and routing refresh the remaining groups. Each group is visited once per revision iteration, and the search ends on verification success or budget exhaustion. Appendix A gives the full procedure.

For the histogram example, the revision group includes the failing caller and the `Max` and `Histogram` interfaces. A coordinated proposal adds the prefix bound to `Max`'s guarantee and replaces the consumer's whole-array requirement with that bound. The producer's fixed invariant supports the new guarantee. Together with input nonnegativity and $|h| = m + 1$, it supplies the bounds needed for the consumer's accesses to $h[a_i]$ for $0 \le i < k$, resolving the illustrated call and indexing obligations.

Table 1: Final verified programs (verification rate, ↑). Left: generated contracts (§4.2); right: fixed-specification proofs (§4.3). Bold: best; underline: second best within each backbone and task. Iterative methods use up to five iterations including initial construction; Base is single-shot.

		Contract generation (§4.2)		**Fixed-specification proofs** (§4.3)	
Model	Method	DafnyComp ($n = 300$)	SpecGen ($n = 186$)	SWE-Dafny ($n = 100$)	SpecRefine ($n = 186$)
Qwen3.5-122B-A10B	Base	5 (1.7%)	30 (16.1%)	20 (20.0%)	27 (14.5%)
	Base Refine	68 (22.7%)	56 (30.1%)	55 (55.0%)	52 (28.0%)
	DafnyPro	162 (54.0%)	61 (32.8%)	60 (60.0%)	45 (24.2%)
	AxDafny	164 (54.7%)	122 (65.6%)	69 (69.0%)	51 (27.4%)
	CoVeri	**175 (58.3%)**	**132 (71.0%)**	**73 (73.0%)**	**59 (31.7%)**
Qwen3.5-35B-A3B	Base	7 (2.3%)	17 (9.1%)	19 (19.0%)	19 (10.2%)
	Base Refine	32 (10.7%)	44 (23.7%)	53 (53.0%)	47 (25.3%)
	DafnyPro	112 (37.3%)	57 (30.6%)	52 (52.0%)	45 (24.2%)
	AxDafny	68 (22.7%)	73 (39.2%)	55 (55.0%)	46 (24.7%)
	CoVeri	**164 (54.7%)**	**110 (59.1%)**	**70 (70.0%)**	**53 (28.5%)**
ReForm-14B-RL	Base	45 (15.0%)	54 (29.0%)	34 (34.0%)	29 (15.6%)
	Base Refine	55 (18.3%)	64 (34.4%)	32 (32.0%)	32 (17.2%)
	DafnyPro	49 (16.3%)	62 (33.3%)	35 (35.0%)	36 (19.4%)
	AxDafny	31 (10.3%)	49 (26.3%)	36 (36.0%)	35 (18.8%)
	CoVeri	**67 (22.3%)**	**72 (38.7%)**	**40 (40.0%)**	**44 (23.7%)**

4 EXPERIMENTS

We evaluate CoVeri on two forms of annotation synthesis. Contract generation requires deciding which implementation facts to expose at declaration boundaries; proof construction under fixed specifications requires supplying the intermediate facts that connect an implementation to given requirements. We ask whether CoVeri improves verification in contract generation (RQ1) and yields more verified programs with Superior Specifications (RQ2), whether the gains carry over to fixed specifications (RQ3), which components account for them (RQ4), how they vary with compositional complexity (RQ5), and what the search costs (RQ6).

4.1 EXPERIMENT SETUP

We assess generated contracts through whole-program verification and reference-specification checks on **DafnyComp** (Xu et al., 2025) and the SpecGen split of **VeriEquivBench** (Zeng et al., 2026). For proof construction under fixed specifications, we use **SWE-Dafny** and the SpecRefine split of **VeriEquivBench** (Zeng et al., 2026). SWE-Dafny contains 100 Dafny verification tasks derived from repaired code in SWE-bench Verified (OpenAI, 2024). Each task supplies a fixed implementation and entry-point requirements; models complete internal contracts and proof annotations (Appendix B.1.2). SpecRefine likewise preserves the given contracts while allowing internal contracts and proof annotations to be added. All tasks preserve executable implementations and protected annotations. Across Qwen3.5-35B-A3B, Qwen3.5-122B-A10B (Qwen Team, 2026), and ReForm-14B-RL (Yan et al., 2025) (See Appendix B.4 for details), we compare CoVeri with Base, Base Refine, DafnyPro (Banerjee et al., 2026), and AxDafny (Breen et al., 2026). Iterative methods run for up to five iterations, with iteration 1 corresponding to initial construction; Base is single-shot. *Verified* counts programs that pass whole-program verification while preserving the fixed code and annotations. Programs without an `ensures` clause are counted as unverified.

Verified with Superior Specifications additionally requires that, for every evaluated declaration, the generated contract (P_c, Q_c) and reference contract $(P_{\text{ref}}, Q_{\text{ref}})$ satisfy

$$P_{\text{ref}} \Rightarrow P_c, \qquad P_{\text{ref}} \wedge Q_c \Rightarrow Q_{\text{ref}}. \tag{4}$$

The first implication preserves the reference input domain; the second establishes its guarantee on that domain, under the common state and frame semantics.

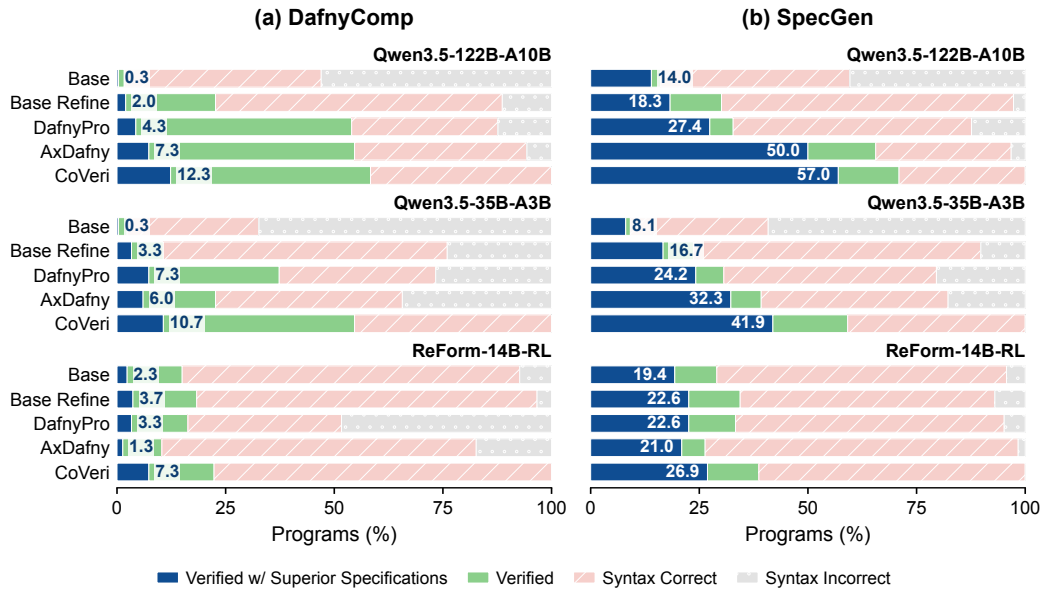

Figure 3: Final contract-generation outcomes. Blue: verified with Superior Specifications; green: otherwise verified; pink: syntax-correct but unverified; gray: syntax-incorrect.

4.2 Generating Verifiable and Superior Specifications

RQ1: Compositional guidance makes verification robust to backbone scale. DafnyComp and SpecGen require contracts for fixed implementations, so success depends on both local provability and compatibility across uses. CoVeri achieves the highest verification rate with every backbone on both benchmarks (Table 1, left). Its margin on DafnyComp is 11 programs over AxDafny with Qwen3.5-122B-A10B but 52 over DafnyPro with Qwen3.5-35B-A3B. The difference lies in how methods degrade: switching to the smaller backbone costs CoVeri 11 verified programs, compared with 50 for DafnyPro and 96 for AxDafny. The baselines leave it to the model to determine which contracts to revise and in which direction, whereas CoVeri supplies this information through symbolic support and routed demands. As a result, CoVeri with the smaller backbone matches AxDafny with the larger one. On ReForm-14B-RL, iterative baselines gain at most 10 programs over single-shot generation, and AxDafny falls below Base; CoVeri still adds 12 programs over the best baseline on DafnyComp and 8 on SpecGen, suggesting that its guidance complements Dafny-specific post-training.

RQ2: CoVeri improves verifiability without simply trading away specification quality. To assess the quality of generated contracts beyond verification alone, we apply the Superior Specifications criterion, which requires each generated contract to admit the reference input domain and establish the reference guarantee (Equation 4). CoVeri yields the most programs meeting the Superior Specifications criterion for every backbone on both benchmarks (Figure 3). With Qwen3.5-122B-A10B on DafnyComp, it reaches 37 programs, compared with 22 for AxDafny, and a larger share of its verified programs meet the criterion (21.1% vs. 13.4%). The two revision directions match the two conditions in Equation 4: relaxing an overly strong generated precondition moves it toward admitting the reference domain, while strengthening a postcondition with implementation-supported facts moves it toward the reference guarantee. Thus, CoVeri's verification improvements are accompanied by an increase in programs whose generated contracts satisfy the stronger reference-based criterion, showing that the gains are not solely attributable to specification weakening.

4.3 Proof Construction under Fixed Specifications

RQ3: The gains carry over to fixed specifications. SWE-Dafny and SpecRefine fix the entry-point requirements, so synthesis must connect a given implementation to given requirements through internal contracts and proof annotations. CoVeri achieves the highest verification rate with every backbone on both tasks (Table 1, right). On SWE-Dafny, the pattern from RQ1 recurs: the margin

Table 2: Local proof completion on DafnyBench (Qwen3.5-122B-A10B).

Method	Base	Base Refine	DafnyPro	AxDafny	**CoVeri**
Verified ($n = 782$)	471 (60.2%)	534 (68.3%)	567 (72.5%)	586 (74.9%)	**606 (77.5%)**

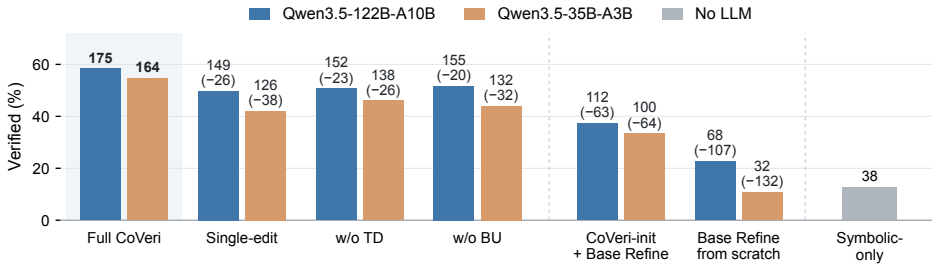

Figure 4: Ablation results on DafnyComp ($N = 300$). Labels show verified-program counts; parentheses indicate changes relative to full CoVeri with the same backbone. Symbolic-only uses no LLM and is shared across backbones. TD: top-down demand analysis; BU: bottom-up symbolic reasoning. Settings are given in Appendix B.3.

over the strongest baseline grows from 4 programs with Qwen3.5-122B-A10B to 15 with Qwen3.5-35B-A3B, and the smaller backbone verifies nearly as many programs as the larger one.

Local proof completion. On DafnyBench (Loughridge et al., 2024), where specifications are given and loop invariants and assertions must be supplied, CoVeri verifies 606 of 782 programs (77.5%), compared with 586 (74.9%) for AxDafny (Table 2). This shows that CoVeri also helps propagate facts within a method, so the analysis is not limited to contracts between functions.

4.4 Ablating Guidance, Joint Revision, and LLM Generation

RQ4: Guidance matters more than edit scope, and the smaller backbone relies on it more. We evaluate five variants on the same 300 DafnyComp programs with Qwen3.5-122B-A10B and Qwen3.5-35B-A3B (Figure 4; Appendix B.3). Single-edit restricts each LLM proposal to one declaration. The w/o TD variant disables top-down demand analysis during repair, and w/o BU disables bottom-up symbolic reasoning and constructs its own initialization. CoVeri-init + Base Refine retains CoVeri's initialization but replaces guided repair with base refinement that may edit multiple declarations. Symbolic-only runs without an LLM.

Removing any analysis component reduces verification, and each removal hurts the smaller backbone more than the larger one. This mirrors RQ1, suggesting that the analyses supply information the smaller model cannot reliably infer from verifier feedback alone. CoVeri-init + Base Refine permits multi-declaration edits yet falls below even Single-edit on both backbones: knowing which contracts to change and in which direction matters more than the freedom to change several at once, and combining both performs best. It also improves substantially over Base Refine from scratch (Table 1) but remains well below the full method, so initialization and guided repair both contribute. Symbolic-only verifies only a small fraction of the programs that CoVeri verifies, indicating that most gains require LLM generation guided by the analysis rather than the symbolic procedure alone.

4.5 Performance by Compositional Complexity

RQ5: The gains grow with the number of call sites. We use call-site count as a measure of compositional complexity, since each call site requires a callee's contract to supply the facts its caller needs. Table 3 stratifies CoVeri's gains over AxDafny and DafnyPro on DafnyComp with Qwen3.5-35B-A3B. The margin over AxDafny grows steadily across

Table 3: Verification-rate gain (percentage points) over baselines on DafnyComp (Qwen3.5-35B-A3B), grouped by call-site count.

Baseline	≤ 5 $N = 76$	6–8 $N = 81$	≥ 9 $N = 143$
AxDafny	+27.63	+32.10	**+34.27**
DafnyPro	+13.16	+13.58	**+21.68**

groups, and the margin over DafnyPro stays roughly constant in the first two groups before widening in the third; both are largest for programs with nine or more call sites. With more call sites, each contract must satisfy more callers, so mismatches are more likely and more complex to fix. In CoVeri, demand analysis locates these contracts, and joint revision changes them together.

4.6 Higher Verification Rate with Fewer Tokens

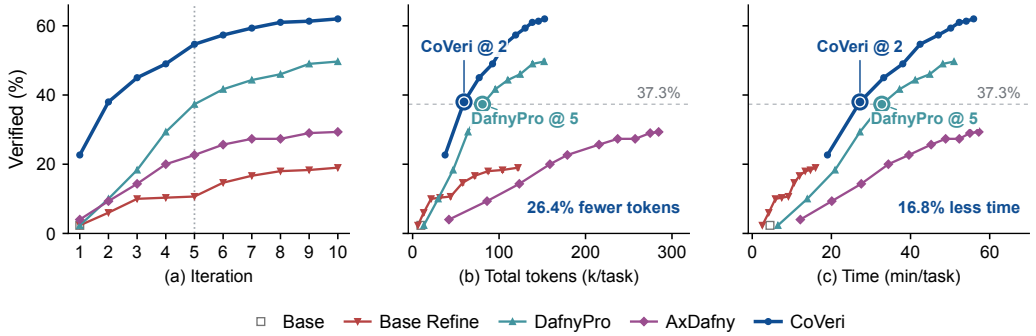

Figure 5: Qwen3.5-35B-A3B search trajectories on DafnyComp. Iteration 1 denotes initial construction; the vertical line marks the main budget at iteration 5.

RQ6: CoVeri reaches baseline coverage at lower cost. Figure 5 traces verification against iterations, tokens, and time on DafnyComp with Qwen3.5-35B-A3B. CoVeri leads from the start: initial construction alone verifies substantially more programs than any baseline's first iteration. By iteration 2, CoVeri exceeds what DafnyPro, the strongest baseline in this setting, reaches at iteration 5, while using 26.4% fewer tokens and 16.8% less time. The savings arise because symbolic analysis, rather than the model, locates what needs repair, so the model edits only the implicated interfaces instead of rereading and regenerating the complete program at every iteration. The advantage is not only faster convergence: CoVeri remains ahead of every baseline when the budget is extended to ten iterations, at a total token cost comparable to DafnyPro's.

5 Discussion and Limitations

CoVeri treats annotation synthesis as a compositional search problem: each summary must be supported by its implementation and must supply the information needed at its uses. Bottom-up compositional reasoning and top-down demand analysis together guide the direction of annotation changes, while joint revision coordinates edits across dependent interfaces. Across the evaluated benchmarks and LLM backbones, this approach improves whole-program verification. The ablations separate the guidance supplied to synthesis from the changes a proposal is allowed to make, and show that both matter. Single-edit retains CoVeri's guidance but verifies fewer programs when each LLM proposal is restricted to one declaration. Base refinement permits joint edits but still falls short despite using the same initialization. Bidirectional guidance and coordinated revision thus play complementary roles.

Limitations and future work. Because CoVeri relies on LLM proposals, it provides no theoretical guarantee of finding the strongest expressible annotations. Our implementation currently targets Dafny; the formulation of feasibility and use-side sufficiency is not specific to Dafny, and we plan to extend CoVeri to other verification frameworks such as Verus (Lattuada et al., 2023) and Lean (de Moura & Ullrich, 2021) in the future. Building on our fixed-specification results, we also aim to scale CoVeri to larger programs and broader workflows for verifying independently specified functional requirements, where synthesized contracts, invariants, and assertions can serve as proof interfaces connecting implementation details to user-specified correctness goals.

AI USE STATEMENT

We used Codex with GPT-6 Astra to translate Python sources into Dafny when constructing SWE-Dafny, as described in the appendix. We also used Codex and Claude Code to assist with coding, running experiments, searching for related work, and polishing the writing. We reviewed all AI-assisted work and take responsibility for the final content and artifacts.

REPRODUCIBILITY STATEMENT

We will release CoVeri and SWE-Dafny on GitHub upon acceptance.

REFERENCES

Pranjal Aggarwal, Bryan Parno, and Sean Welleck. AlphaVerus: Bootstrapping Formally Verified Code Generation through Self-Improving Translation and Treefinement. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 587–615. PMLR, 2025. URL https://proceedings.mlr.press/v267/aggarwal25a.html.

Debangshu Banerjee, Olivier Bouissou, and Stefan Zetzsche. DafnyPro: LLM-Assisted Automated Verification for Dafny Programs. *arXiv preprint arXiv:2601.05385*, 2026. URL https://arxiv.org/abs/2601.05385.

Benjamin Breen, Austin Letson, Borja Requena Pozo, and Leopoldo Sarra. AxDafny: Agentic Verified Code Generation in Dafny. *arXiv preprint arXiv:2606.32007*, 2026. URL https://arxiv.org/abs/2606.32007.

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 289–300. ACM, 2009. doi: 10.1145/1480881.1480917.

Saikat Chakraborty, Shuvendu K. Lahiri, Sarah Fakhoury, Akash Lal, Madanlal Musuvathi, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. Ranking LLM-generated loop invariants for program verification. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 9164–9175, 2023. doi: 10.18653/v1/2023.findings-emnlp.614.

Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning assumptions for compositional verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pp. 331–346. Springer, 2003. doi: 10.1007/3-540-36577-X_24.

Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pp. 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24.

Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, pp. 625–635, Berlin, Heidelberg, 2021. Springer-Verlag. ISBN 978-3-030-79875-8. doi: 10.1007/978-3-030-79876-5_37. URL https://doi.org/10.1007/978-3-030-79876-5_37.

Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975. doi: 10.1145/360933.360975.

Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. Inductive invariant generation via abductive inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, pp. 443–456. ACM, 2013. doi: 10.1145/2509136.2509511.

Haoran Ding, Zhaoguo Wang, and Haibo Chen. FM-Agent: Scaling formal methods to large systems via LLM-based hoare-style reasoning, 2026. URL https://arxiv.org/abs/2604.11556.

Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pp. 500–517. Springer, 2001. doi: 10.1007/3-540-45251-6_29.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12 (10):576–580, 1969. doi: 10.1145/363235.363259.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.06770.

Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-based model checking for recursive programs. In *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pp. 17–34. Springer, 2014. doi: 10.1007/978-3-319-08867-9_2.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023. URL https://arxiv.org/abs/2309.06180.

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1), 2023. doi: 10.1145/3586037.

K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355 of *Lecture Notes in Computer Science*, pp. 348–370. Springer, 2010. doi: 10.1007/978-3-642-17511-4_20.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/43e9d647ccd3e4b7b5baab53f0368686-Paper-Conference.pdf.

Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. DafnyBench: A Benchmark for Formal Software Verification. *arXiv preprint arXiv:2406.08467*, 2024. URL https://arxiv.org/abs/2406.08467.

Eric Mugnier, Emmanuel Anaya Gonzalez, Ranjit Jhala, Nadia Polikarpova, and Yuanyuan Zhou. Laurel: Unblocking Automated Verification with Large Language Models. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA1), 2025. doi: 10.1145/3720499.

Emerson Murphy-Hill, Jenna Butler, and Alexandra Savelieva. Adoption and Impact of Command-Line AI Coding Agents: A Study of Microsoft's Early 2026 Rollout of Claude Code and GitHub Copilot CLI. *arXiv preprint arXiv:2607.01418*, 2026. URL https://arxiv.org/abs/2607.01418.

OpenAI. Introducing SWE-bench Verified, August 2024. URL https://openai.com/index/introducing-swe-bench-verified/.

OpenAI. Codex. https://openai.com/codex/, 2026a. Accessed: 2026-09-25.

OpenAI. GPT-6 Astra: A New Generation of Intelligence. https://openai.com/index/gpt-6-astra/, 2026b. Accessed: 2026-09-25.

Giovanni Pinna, Jingzhi Gong, David Williams, and Federica Sarro. Comparing AI Coding Agents: A Task-Stratified Analysis of Pull Request Acceptance. In *Proceedings of the International Conference on Mining Software Repositories*, 2026. URL https://arxiv.org/abs/2602.08915.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-Loop Verifiable Code Generation. In *Proceedings of the First International Symposium on AI Verification*, volume 14846 of *Lecture Notes in Computer Science*, pp. 134–155. Springer, 2024. doi: 10.1007/978-3-031-65112-0_7.

Zhongyi Wang, Tengjie Lin, Mingshuai Chen, Mingqi Yang, Haokun Li, Xiao Yi, Shengchao Qin, and Jianwei Yin. Preguss: It analyzes, it specifies, it verifies. In *Proceedings of the Workshop on Language Models and Programming Languages*, 2025. doi: 10.1145/3759425.3763394.

Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. Enchanting program specification synthesis by large language models using static analysis and program verification. In *Computer Aided Verification*, pp. 302–328, 2024. doi: 10.1007/978-3-031-65630-9_16.

Xu Xu, Xin Li, Xingwei Qu, Jie Fu, and Binhang Yuan. Local Success Does Not Compose: Benchmarking Large Language Models for Compositional Formal Verification. *arXiv preprint arXiv:2509.23061*, 2025. URL https://arxiv.org/abs/2509.23061.

Chuanhao Yan, Fengdi Che, Xuhan Huang, Xu Xu, Xin Li, Yizhi Li, Xingwei Qu, Jingzhe Shi, Chenghua Lin, Yaodong Yang, Binhang Yuan, Hang Zhao, Yu Qiao, Bowen Zhou, and Jie Fu. Re:Form—Reducing Human Annotations in Scalable Formal Software Verification with RL in LLMs: A Preliminary Study on Dafny. *arXiv preprint arXiv:2507.16331*, 2025. URL https://arxiv.org/abs/2507.16331.

Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu K. Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. AutoVerus: Automated Proof Generation for Rust Code. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA2):3454–3482, 2025. doi: 10.1145/3763174.

Zhe Ye, Hantao Lou, Yuechun Sun, Peiyang Song, Zhengxu Yan, Timothe Kasriel, Qingyang Zhang, Kaiyu Yang, Soonho Kong, Jingxuan He, and Dawn Song. Vero: Can AI Agents Build Formally Verified Software Repositories? *arXiv preprint arXiv:2608.13522*, 2026a. URL https://arxiv.org/abs/2608.13522.

Zhe Ye, Aidan Z. H. Yang, Huangyuan Su, Zhenyu Liao, Samuel Tenka, Zhizhen Qin, Udaya Ghai, Dawn Song, and Soonho Kong. Intent-aligned Formal Specification Synthesis via Traceable Refinement. *arXiv preprint arXiv:2604.10392*, 2026b. URL https://arxiv.org/abs/2604.10392.

Lingfei Zeng, Fengdi Che, Xuhan Huang, Fei Ye, Xu Xu, Binhang Yuan, and Jie Fu. VeriEquivBench: An Equivalence Score for Ground-Truth-Free Evaluation of Formally Verifiable Code. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/ebfa4297cd6419f64efe86f657ba49d0-Abstract-Conference.html.

A Implementation details

A.1 Compositional Search

Initial construction follows call dependencies between declarations and AST nesting within declarations: callees precede callers, and needed inner summaries precede enclosing boundaries. It constructs loop invariants as well as declaration contracts, handling recursive declarations together. Revision selects related interfaces from failed obligations for a coordinated proposal (Algorithm 1).

Algorithm 1 CoVeri: initial construction and joint revision

Input: Program C, fixed requirements F, edit permissions $\mathcal{A}$, and search budget. All operations use the current program $C[\alpha]$; L_g denotes its local proof feedback for group g.

Initial Construction

$\alpha \leftarrow$ supplied annotations
for each declaration group g in callee-first order **do**
  while an unconsidered editable boundary needs annotations **do**
    $(\alpha, B_g) \leftarrow \mathrm{Compose}(g, \alpha)$
    $r \leftarrow$ next needed region in AST dependency order; **break** if none
    $\alpha \leftarrow \mathrm{Install}(\alpha, \mathrm{InitLLM}(r, B_g, L_g))$
  end while
end for
$V \leftarrow \mathrm{VerifyWhole}(\alpha, F)$

Demand-Guided Revision

while V is unresolved and budget remains **do**
  $(D, \mathcal{G}) \leftarrow \mathrm{RouteAndGroup}(\alpha, V)$
  for each remaining group g in $\mathcal{G}$ **do**
    $(\alpha, B_g) \leftarrow \mathrm{Compose}(g, \alpha)$
    check the current program; stop if it verifies
    $\alpha \leftarrow \mathrm{Install}(\alpha, \mathrm{JointLLM}(g, D, L_g))$
    $V \leftarrow \mathrm{VerifyWhole}(\alpha, F)$
    update demands D and remaining groups $\mathcal{G}$
  end for
end while
return (α, V)

Compose combines region relations and installs permitted mechanically generated annotations. Install applies the proposed annotation edits within the permitted scope, preserving executable code and fixed requirements. For each omitted generated postcondition, it checks whether the clause follows from the revised contract; otherwise, it restores the clause if a proof check confirms that it remains provable. Initial proposals use these relations; joint proposals use the resulting code and interfaces, routed demands, and local proof feedback. A group defines the allowed edit scope. Each revision iteration visits a group once. The search stops on successful whole-program verification or budget exhaustion.

A.2 From Regions to Boundary Summaries

A symbolic state (ρ, Φ, H) contains a variable environment, path constraints, and a shared heap mapping integer-array references to sequences. These are Z3 expressions derived from the resolved Dafny AST. $B_r(s, s')$ is a symbolic transition relation between a region's entry and exit states, retaining the obligations on which the relation depends.

Assignments and control flow. An assignment $x := e$ gives $x' = e(s)$ and preserves other state components; simultaneous assignments evaluate all right-hand sides in the entry state. Sequences compose through an intermediate state, $B_{r_1;r_2}(s, s') = \exists t.\, B_{r_1}(s, t) \wedge B_{r_2}(t, s')$. A branch retains its guard: $(b(s) \wedge B_{\mathrm{then}}(s, s')) \vee (\neg b(s) \wedge B_{\mathrm{else}}(s, s'))$. Returns contribute exits to the enclosing declaration.

Heap updates. For $a[i] := e$, the heap becomes $H' = H[\rho(a) \mapsto H[\rho(a)][i(s) \mapsto e(s)]]$, where the inner update replaces one sequence element. Other cells and array lengths are preserved, and aliases observe the same write. Nullness and index bounds become proof obligations.

Calls and loops. A call instantiates its precondition P_f^u as an obligation and assumes its postcondition Q_f^u over fresh results. It preserves the heap when no modification is allowed; otherwise, the analysis introduces a fresh heap constrained by the postconditions. A loop with guard b and invariant I abstracts modified values and yields $I(s') \wedge \neg b(s')$, together with preserved frame facts. Its obligations are initiation and preservation: the entry condition implies I, and

$I(t) \wedge b(t) \wedge B_{\text{body}}(t, t') \Rightarrow I(t')$. Both rules remain conditional on the corresponding summaries being proved; Dafny checks these obligations and applicable termination requirements.

Boundary annotation generation. Let $R_r(\mathbf{x}, \mathbf{y}, \mathbf{z})$ collect exit constraints over inputs $\mathbf{x}$, results $\mathbf{y}$, and internal values $\mathbf{z}$. Projection seeks

$$\widehat{Q}_r(\mathbf{x}, \mathbf{y}) = \exists \mathbf{z}.\, R_r(\mathbf{x}, \mathbf{y}, \mathbf{z}). \tag{5}$$

For example, eliminating t from $t = x + 1 \wedge y = 2t$ gives $y = 2x + 2$. Program terms and their consequences supply postcondition candidates; fixed structural templates supply simple invariants such as counter bounds. Entry obligations likewise provide premises for precondition generation. A candidate is emitted only when expressible at an editable Dafny boundary. SMT implication checks test consequences under the current premises, including any unproved annotations used in constructing the relation. When projection cannot produce a clause, the relation can still guide the LLM.

The symbolic fragment covers integer and Boolean reasoning, integer arrays, selected character and sequence operations, and quantification over supported types. Transparent functions can be summarized; recursive calls use contracts. General datatype and object-heap reasoning and precise arbitrary `reads`/`modifies` footprints are outside this fragment.

A.3 From Proof Demands to Joint Revision

Routing traces failed obligations backward through assignments, calls, and loops, collecting possible summary boundaries together with caller assumptions and branch guards. The selected interfaces include editable generated consumer requirements alongside provider contracts.

Algorithm 2 Route demands and select joint revision groups

Input: Current annotations α and verification feedback V.

$E \leftarrow$ declarations with unresolved checks
$D, P \leftarrow \varnothing$; $U \leftarrow$ declarations with unlocated goals
for each unmet obligation e in V **do**
  $(d, \Gamma, \mathcal{B}) \leftarrow \text{TraceAST}(e, \alpha)$
  $D \leftarrow D \cup \{(d, \Gamma, \mathcal{B})\}$
  if routing is known **then** add declarations owning selected interfaces for d to P
  else $U \leftarrow U \cup \{\text{owner}(e)\}$
end for
extend D, P with bounded local probes; retain other caller contexts
$S \leftarrow E \cup P \cup \bigcup_{u \in U} \text{Calls}(u)$
$\mathcal{G} \leftarrow$ connected components of the call graph induced by S, ignoring edge directions
return $(D, \mathcal{G})$ in source order

In Algorithm 2, d retains its source clause and whether it is fixed or generated, Γ is its caller context, and $\mathcal{B}$ contains the traced boundaries. The revision set includes declarations whose current contracts do not establish the demand and declarations with editable consumer requirements. For a goal q expressible in the callee's vocabulary through a substitution θ that preserves entry- and exit-state references, the guarded demand is $\theta(\Gamma) \Rightarrow \theta(q)$. Different calling contexts remain separate, including uses that already verify. Bounded probes temporarily check such demands as callee post-conditions, extending routing by one call layer under a fixed allowance. Groups are refreshed after each proposal, excluding declarations already considered in that iteration.

B Experiment Details

B.1 Benchmarks

B.1.1 DafnyComp

DafnyComp (Xu et al., 2025) evaluates contract generation for programs containing interacting declarations. Its central challenge is to produce specifications that both follow from individual implementations and provide sufficient information for their callers. We use all 300 programs in the test

split from the released DafnyComp dataset distributed with Re:Form (Yan et al., 2025). We also corrected a few minor line-break formatting errors in the released DafnyComp inputs without changing their executable logic.

In our evaluation, executable code, declaration signatures, and existing body annotations remain fixed. Models complete the permitted declaration-level contracts, including preconditions, postconditions, and frame conditions. Existing loop invariants, assertions, and loop termination annotations are preserved. This setting isolates the construction of compatible interfaces across program components.

B.1.2 SWE-DAFNY

SWE-Dafny is our collection of 100 Dafny verification tasks derived from SWE-bench Verified (OpenAI, 2024). It focuses on establishing properties of repaired software through internal contracts and proof annotations. Each task models selected functionality from a real repository, together with explicit assumptions about dependencies outside that scope.

Selecting the 100 instances. We first screened all 500 SWE-bench Verified instances for functionality involving multiple methods or interfaces. An instance was eligible when its identified scope contained at least one implemented method and at least three distinct members in total, counting implemented methods and identifiable external interfaces. Repeated uses of the same member were counted once; ordinary logging, exception constructors, and Python built-ins were excluded from this count.

The screening produced 413 eligible instances. We sorted their instance identifiers lexicographically and sampled 100 instances without replacement using Python's `random.Random(0).sample`. (See Table 4)

Recovering the source version. For each selected instance, we used the repository and `base_commit` specified by SWE-bench together with its official repair patch to identify the repaired source. Changed files were taken in their post-patch form; required unchanged dependencies were obtained from the same base revision. The translated functionality therefore corresponds to the officially repaired code, rather than a later repository revision.

Constructing Dafny references and verification targets. We used Codex (OpenAI, 2026a) with GPT-6 Astra (OpenAI, 2026b) to translate the selected Python source into Dafny and construct the corresponding verification targets from the original issue requirements and repaired source. For each task, the target specifies the selected entry points and their required behavior, including the input domain, relevant object state, outputs, exceptional behavior, and properties to establish. These requirements are encoded as fixed entry-point contracts consisting of preconditions, postconditions, and frame conditions; because SWE-bench does not provide Dafny specifications, they are derived from the issue requirements and their interpretation in the repaired source. The construction protocol preserves the relevant failure scenario within the modeled input domain rather than excluding it through stronger preconditions, while dependencies outside the selected implementation are represented by interfaces with explicit contracts over their arguments, results, and state effects. Codex translated the complete selected methods, including branches, loops, calls, state updates, and returns, and makes language-specific behavior explicit within each task's modeling scope. We then reviewed the resulting Dafny programs for correspondence with the selected Python methods and corrected discrepancies where necessary. The corrected references are checked with Dafny 4.11.0, and all 100 references verify within their documented specification and external-contract boundaries. Source-correspondence review and Dafny verification provide complementary guarantees: the former checks the modeling choices, whereas the latter proves properties of the resulting Dafny program.

Constructing evaluation inputs. From each verified reference, we derive an annotation-synthesis evaluation input that preserves the executable implementation, entry-point requirements, and trusted external contracts. Entry points are identified as roots of the direct-call graph among the selected source declarations, with multiple roots retained when present. We remove designated internal preconditions, postconditions, and frame conditions, together with invariants and assertions in imple-

mented method-like declarations, while keeping signatures, function and predicate definitions, and explicit termination clauses fixed. Models must reconstruct the permitted annotations so that the complete program verifies under the original entry-point requirements. Deletion uses compiler-derived annotation spans, with checks that executable tokens and protected definitions are preserved. Reference programs and removed annotations are withheld from generation.

Example evaluation input. Example 1 presents `django__django-11149`, which models permission checks for inline administration. Four entry points share the helper `HasAny`. This helper first identifies the relevant model and then checks whether any requested permission is granted. The entry-point specifications are fixed, while the helper's contract and loop invariants must be synthesized. The example requires proof information to connect the two loops, the helper interface, and its four callers.

Example 1: A SWE-Dafny verification task for inline permission checks (instance 060).

```
module InlinePermissionsGT {
  datatype Field = NoRemote | Remote(model:int)
  datatype Options =
    Opts(identity:int,appLabel:string,
         autoCreated:bool,fields:seq<Field>)

  trait Django {
    function Meta(model:int):Options
    function Codename(permission:string,opts:Options):string
    function HasPerm(request:int,permission:string):bool
    function SuperAdd(admin:int,request:int):bool
    function SuperChange(admin:int,request:int):bool
    function SuperDelete(admin:int,request:int,obj:int):bool
    function SuperView(admin:int,request:int):bool
  }

  function Target(opts:Options,parent:int,
                  fields:seq<Field>,d:Django):Options
    decreases |fields|
  {
    if |fields|==0 then opts
    else if fields[0].Remote? && fields[0].model!=parent
      then d.Meta(fields[0].model)
    else Target(opts,parent,fields[1..],d)
  }

  function Allowed(opts:Options,parent:int,request:int,
                   perm:string,d:Django):bool
  {
    var t:=Target(opts,parent,opts.fields,d);
    d.HasPerm(request,t.appLabel+"."+d.Codename(perm,t))
  }

  method HasAny(opts:Options,parent:int,request:int,
                perms:seq<string>,d:Django)
    returns(ok:bool)
  {
    var target:=opts;
    var i:=0;
    while i<|opts.fields|
    {
      var field:=opts.fields[i];
      if field.Remote? && field.model!=parent {
        target:=d.Meta(field.model);
        break;
      }
      i:=i+1;
    }

    i:=0;
    while i<|perms|
    {
      if d.HasPerm(request,
          target.appLabel+"."+d.Codename(perms[i],target)) {
        ok:=true;
        return;
      }
      i:=i+1;
    }
    ok:=false;
  }

```

```
  method Add(admin:int,opts:Options,parent:int,
             request:int,obj:int,d:Django)
    returns(ok:bool)
    ensures ok==(if opts.autoCreated
      then Allowed(opts,parent,request,"change",d)
      else d.SuperAdd(admin,request))
  {
    if opts.autoCreated {
      ok:=HasAny(opts,parent,request,["change"],d);
    } else {
      ok:=d.SuperAdd(admin,request);
    }
  }

  method Change(admin:int,opts:Options,parent:int,
                request:int,obj:int,d:Django)
    returns(ok:bool)
    ensures ok==(if opts.autoCreated
      then Allowed(opts,parent,request,"change",d)
      else d.SuperChange(admin,request))
  {
    if opts.autoCreated {
      ok:=HasAny(opts,parent,request,["change"],d);
    } else {
      ok:=d.SuperChange(admin,request);
    }
  }

  method Delete(admin:int,opts:Options,parent:int,
                request:int,obj:int,d:Django)
    returns(ok:bool)
    ensures ok==(if opts.autoCreated
      then Allowed(opts,parent,request,"change",d)
      else d.SuperDelete(admin,request,obj))
  {
    if opts.autoCreated {
      ok:=HasAny(opts,parent,request,["change"],d);
    } else {
      ok:=d.SuperDelete(admin,request,obj);
    }
  }

  method View(admin:int,opts:Options,parent:int,
              request:int,obj:int,d:Django)
    returns(ok:bool)
    ensures ok==(if opts.autoCreated
      then Allowed(opts,parent,request,"view",d)
        || Allowed(opts,parent,request,"change",d)
      else d.SuperView(admin,request))
  {
    if opts.autoCreated {
      ok:=HasAny(opts,parent,request,["view","change"],d);
    } else {
      ok:=d.SuperView(admin,request);
    }
  }
}
```

Table 4: The 100 SWE-bench Verified instances used to construct SWE-Dafny, listed in the frozen sampling order (seed 0).

No.	Instance ID	No.	Instance ID
1	matplotlib__matplotlib-20488	51	matplotlib__matplotlib-23299
2	sympy__sympy-19346	52	sphinx-doc__sphinx-9230
3	matplotlib__matplotlib-24970	53	sphinx-doc__sphinx-7454
4	django__django-10097	54	astropy__astropy-12907
5	django__django-15098	55	sphinx-doc__sphinx-10449
6	pylint-dev__pylint-6528	56	pydata__xarray-6721
7	pydata__xarray-4695	57	django__django-16255
8	matplotlib__matplotlib-23476	58	django__django-14725
9	sympy__sympy-22714	59	django__django-16100
10	django__django-15731	60	django__django-11149
11	pydata__xarray-4094	61	django__django-13794
12	django__django-16642	62	scikit-learn__scikit-learn-13142
13	scikit-learn__scikit-learn-14496	63	django__django-14349
14	django__django-14170	64	django__django-14608

Continued on next page

Table 4 (continued)

No.	Instance ID	No.	Instance ID
15	pylint-dev__pylint-4661	65	django__django-13028
16	django__django-13023	66	pytest-dev__pytest-7521
17	django__django-15380	67	psf__requests-1142
18	sympy__sympy-21612	68	django__django-11740
19	sympy__sympy-18698	69	django__django-11333
20	django__django-11820	70	django__django-15987
21	sphinx-doc__sphinx-10673	71	pylint-dev__pylint-6386
22	django__django-14855	72	pydata__xarray-6461
23	pytest-dev__pytest-6197	73	django__django-12143
24	sympy__sympy-13615	74	django__django-15629
25	scikit-learn__scikit-learn-25973	75	sympy__sympy-15017
26	django__django-13112	76	django__django-15554
27	django__django-15814	77	django__django-12663
28	django__django-11880	78	sympy__sympy-13852
29	sympy__sympy-15349	79	sympy__sympy-13031
30	django__django-11265	80	pytest-dev__pytest-7324
31	sphinx-doc__sphinx-9711	81	sympy__sympy-15345
32	django__django-16145	82	sympy__sympy-24562
33	pydata__xarray-3677	83	sphinx-doc__sphinx-8551
34	scikit-learn__scikit-learn-12682	84	scikit-learn__scikit-learn-14710
35	django__django-11885	85	django__django-15503
36	django__django-16612	86	mwaskom__seaborn-3069
37	matplotlib__matplotlib-25960	87	sphinx-doc__sphinx-9461
38	django__django-15930	88	scikit-learn__scikit-learn-25232
39	sphinx-doc__sphinx-10323	89	sympy__sympy-24661
40	sphinx-doc__sphinx-8035	90	django__django-15957
41	django__django-13933	91	scikit-learn__scikit-learn-13779
42	scikit-learn__scikit-learn-10908	92	django__django-14631
43	sympy__sympy-23262	93	django__django-15525
44	matplotlib__matplotlib-26466	94	django__django-13568
45	pytest-dev__pytest-10051	95	django__django-13786
46	django__django-15103	96	django__django-13658
47	django__django-11141	97	astropy__astropy-7606
48	pytest-dev__pytest-8399	98	sympy__sympy-13372
49	astropy__astropy-14182	99	sympy__sympy-13878
50	django__django-11749	100	pydata__xarray-4075

B.1.3 VERIEQUIVBENCH

VeriEquivBench (Zeng et al., 2026) provides complementary Dafny tasks for proof completion and specification generation. We use its released test splits for **SpecRefine** and **SpecGen**, containing 186 instances each.

SpecRefine. SpecRefine focuses on completing proofs under supplied specifications. We preserve the implementation, declaration signatures, and given contracts, while allowing additional internal contracts and proof annotations to complete verification. This setting tests whether an approach can establish an existing requirement through suitable proof support.

SpecGen. SpecGen focuses on generating contracts for a supplied implementation. We adapt the released data by using its `spec_baseline` program as the implementation source, removing formatting fences, and deleting method preconditions and postconditions using compiler-derived spans. Frames, termination clauses, and body annotations are retained.

B.2 BASELINES

We compare CoVeri with four baselines covering single-shot generation, verifier-guided refinement, strategy-guided proof repair, and agentic annotation synthesis. All methods receive the same benchmark inputs and must respect the task-specific editing permissions described above. Executable implementations and protected annotations remain fixed.

Base. Base prompts the model to complete the permitted annotations in a single request and return the complete Dafny program. The resulting program is checked for preservation and verification. It receives no subsequent opportunity to revise its output. This baseline measures the underlying model's ability to synthesize annotations directly from the supplied program and task constraints.

Base Refine. Base Refine extends single-shot generation with iterative whole-program revision. After an unsuccessful attempt, the model receives its previous outputs and the corresponding verifier or edit-policy feedback and produces a revised complete program. Generated annotations remain editable, including revision and withdrawal, while the original implementation and protected annotations remain fixed. The procedure provides no additional proof-strategy hints or automatic invariant pruning.

DafnyPro. DafnyPro (Banerjee et al., 2026) combines verifier-guided revision with proof-strategy guidance and invariant pruning. We implement a task-specific adaptation of these mechanisms. Because our benchmarks expose different annotation kinds, we use separate configurations for contract generation and proof-annotation completion.

DafnyPro for contract generation. On DafnyComp and VeriEquivBench SpecGen, the search is restricted to the declaration-level annotations permitted by the benchmark. Implementations and existing body annotations remain fixed. Following an unsuccessful attempt, the model receives its attempt history, verification feedback, and four general contract-writing hints: expressing entry conditions, expressing return guarantees, relating inputs and outputs or old and new states, and using well-defined conditional or quantified formulas.

These hints are part of our contract-generation adaptation; they are not presented as the original DafnyPro prompts. We disable invariant pruning in this configuration because loop invariants are outside the editable region. The resulting baseline retains strategy-guided iterative revision while respecting the contract-only task boundary.

DafnyPro for proof-annotation completion. On VeriEquivBench SpecRefine and SWE-Dafny, the adaptation permits internal contracts and proof annotations authorized by each task. SpecRefine preserves the given contracts while allowing additional internal contracts and proof annotations. SWE-Dafny keeps entry-point requirements and trusted external contracts fixed while allowing completion of designated internal contracts and proof annotations.

For revision, we supplement the general contract hints with eight proof-strategy hints paraphrased from the DafnyPro appendix (Banerjee et al., 2026). They cover slice and sequence relationships, recursive extensions, invariant simplification, termination reasoning, invariant initialization, filtered-set cardinalities, and boundary cases. Hints guide the search but do not expand editing permissions; for example, a termination hint does not authorize changing a protected termination clause.

This configuration also retains invariant pruning. When Dafny reports a failing generated invariant, the procedure attempts to remove the corresponding complete invariant clause and verifies the resulting program again. If pruning fails to produce an acceptable verified program, the original program and its diagnostics are retained for subsequent model revision. Thus, pruning operates only on generated proof support within the benchmark's declared edit boundary.

AxDafny. AxDafny (Breen et al., 2026) motivates an agentic baseline with proposer, reviewer, and rolling-memory roles. Our implementation uses the proposer to generate a complete annotated program from the original input, the preceding attempt and feedback, and a compact memory of earlier attempts. Each generated program first undergoes deterministic preservation checks and Dafny verification. A program that passes these checks is then assessed by a reviewer for proof bypasses or changes to the intended task. The reviewer provides an acceptance decision and explanation without editing the program. After an unsuccessful iteration, the memory role summarizes feedback-supported lessons for the next proposal.

We retain this interaction structure while restricting generation and review to each benchmark's annotation permissions. In particular, newly generated contracts are permitted only where the task allows contract synthesis.

B.3 Ablation Settings

We evaluate five variants to examine the roles of joint revision, top-down demand analysis, bottom-up symbolic reasoning, and LLM generation.

Single-edit. Single-edit uses the same CoVeri initialization and retains bottom-up symbolic reasoning, mechanical refresh, top-down demand routing, and the original rules for selecting and grouping declarations for revision. Each LLM proposal modifies at most one declaration, selected by the model from the allowed group.

w/o TD. The w/o TD uses the same CoVeri initialization but disables demand propagation and routing during repair. Revision groups contain declarations identified by native verifier errors and their directly called editable declarations, grouped by call-graph connectivity. Bottom-up symbolic reasoning, mechanical refresh, and joint revision remain enabled.

w/o BU. The w/o BU starts from the original input and constructs its own initial annotations. It retains callee-first initialization, recursive-component handling, top-down demand propagation, and joint revision, while disabling bottom-up symbolic relations, mechanical annotation generation, and symbolic refresh.

CoVeri-init + Base Refine. CoVeri-init + Base Refine starts from the same CoVeri initial annotations and replaces CoVeri's repair stage with generic whole-program LLM refinement. It receives the current annotated program, revision history, and raw verifier diagnostics, with permission to revise annotations across multiple declarations. CoVeri's symbolic analysis, demand routing, and mechanical refresh are disabled during repair.

Symbolic-only. Symbolic-only starts from the original input and uses mechanical analysis and annotation construction without LLM calls or model-generated initialization. It measures the coverage of the implemented symbolic procedure alone.

All variants use the same 300 DafnyComp programs and the same editing and verification criteria. The four LLM-based variants are evaluated with Qwen3.5-122B-A10B and Qwen3.5-35B-A3B, using up to five iterations, with iteration 1 devoted to initial construction. Shared initialization is included in the reported cost. An iteration allows multiple LLM requests; actual request counts vary across variants.

B.4 vLLM Deployment and Request Parameters

All three models are deployed using vLLM 0.19.1 (Kwon et al., 2023). Table 5 summarizes serving and decoding settings.

Table 5: Serving and decoding settings.

Parameter	Qwen3.5-35B-A3B	Qwen3.5-122B-A10B	ReForm-14B-RL
GPU	8×L40S	8×H200	8×L40S
TP / DP	2 / 4	2 / 4	1 / 8
Context tokens	150,000	150,000	32,768
`max-num-seqs`	64	64	10
`max-num-batched-tokens`	32,768	16,384	9,000
GPU memory utilization	0.95	0.94	0.90
Temperature	0.6	0.6	1.2
Top-p	0.95	0.95	0.95
Top-k	20	20	50
Thinking	enabled	enabled	disabled
Max output tokens	25,000	25,000	2,048

C CoVeri Prompts

We present condensed prompt templates for local construction and joint revision. Placeholders in angle brackets are filled with task-specific code, analysis context, and verifier feedback.

Local construction.

```
Construct candidate Dafny annotations for the supplied code
and proof obligations.
Preserve executable code, signatures, and fixed annotations.
Treat unproved candidate relationships as proof dependencies,
not established facts.

Use the available relationships to address the stated proof
obligation. Keep entry conditions separate from return guarantees.
Symbolic deductions are conditional on candidate premises and
do not override a native Dafny failure.

Task and edit permissions:
<Allowed owners, regions, and annotation kinds>

Visible names and types:
<Names available at the selected annotation site>

Selected declaration and focused region:
<Current code and selected declaration or loop>

Available local relationships and child proof status:
<Analysis context>

Fixed annotations:
<Protected specifications and annotations>

Related interfaces, definitions, and caller obligations:
<Relevant program context>

Verifier feedback and previous attempts, when applicable:
<Diagnostics and edit history>

Required response:
For header-only editing, return only the complete generated
annotation clauses, excluding fixed clauses.
Otherwise, return the complete selected declaration.
Apply only permitted edits.
Do not include explanations or Markdown fences.
```

Joint revision.

```
Revise candidate Dafny annotations to complete the whole-program
proof within the listed editing permissions.
Preserve executable code, signatures, and fixed annotations.
Treat unproved candidate relationships as proof dependencies,
not established facts. A local PASS alone is insufficient.

Fixed goals cannot be deleted. Generated guarantees may be
revised or withdrawn when unnecessary. Preserve still-needed
entry conditions and termination clauses.

Use call arguments, path guards, and code after each call
to decide whether to revise a generated caller goal, the
provider interface, or both.
Do not copy a guarded caller goal unconditionally to a callee.
UNKNOWN routing does not establish a dependency.
Symbolic sufficiency is conditional on candidate premises.

Task and edit permissions:
<Allowed owners, regions, and annotation kinds>

For each allowed owner:
<Exact identifier, visible names, fixed annotations,
```

```
 generated annotations, and required response field>

Current local checks:
<Proof status and dependencies>

Versioned goals and routing:
<Goals, callers, candidate providers, and routing uncertainty>

Current native diagnostics:
<Verifier feedback>

Read-only source:
<Current program or selected dependency context>

Failure/edit history:
<Previous proposals and outcomes>

Return ONLY JSON:
{
  "edits": [
    {
      "owner": "<exact owner identifier>",
      "clauses": "<complete generated header clauses>"
    }
  ]
}

For owners with editable body annotations, replace "clauses"
with "declaration" and return the complete selected declaration.
You may edit one or several allowed owners.
Omitted owners remain unchanged; empty clauses withdraw the
generated header; an empty edits array makes no change.
All edits apply to the same current source.
```