SKILLFORGE: Compositional Skill Synthesis with Verification-in-the-Loop for Generating Formally Verified Dafny Programs

Anonymous ACL submission

Abstract

Generating formally verified programs from natural language remains challenging: existing approaches either produce code in a single pass without recourse when verification fails, or rely on open-ended agentic reasoning that is non-deterministic and opaque. We introduce SKILLFORGE, a framework that decomposes formal code synthesis into a library of atomic, reusable *skills*—each targeting a specific subtask such as specification inference, body synthesis, invariant generation, error diagnosis, or targeted repair—defined by a prompt template, tool binding, and decidable success criterion. A *verification-driven harness* orchestrates these skills: it submits candidates to the Dafny verifier, diagnoses failures into structured categories, deterministically routes to the appropriate repair skill, and iterates until formal correctness is proved or a budget is exhausted. On a curated benchmark of natural language–Dafny specification pairs, SKILLFORGE substantially outperforms both state-of-the-art agentic approaches (including ReAct-style agents, MCTS-based repair, and RL-guided verification) and traditional iterative baselines, while requiring fewer tokens and lower latency. Ablation studies confirm that every skill contributes measurably, and the harness converges rapidly with the majority of programs verified on the first attempt.

1 Introduction

Formal verification offers the strongest correctness guarantee available in software engineering: a proof, checked by machine, that a program meets its specification for *all* possible inputs (Wen et al., 2024). The Dafny language (Loughridge et al., 2024) exemplifies this paradigm by integrating preconditions, postconditions, loop invariants, and termination measures directly into the source code, enabling an automated verifier backed by an SMT solver to certify correctness at compile time.

However, writing verification-grade code is prohibitively difficult even for experts: one must simultaneously produce an implementation that is algorithmically correct, a specification that is tight enough for the verifier to discharge, and auxiliary annotations (loop invariants, ghost state) that bridge the gap. Recent large language models (LLMs) can generate syntactically valid Dafny (Rasheed et al., 2025), yet the *verification pass rate* of naive generation remains low because these subtasks place fundamentally different demands on the model:

- **One-shot generation** treats the problem as a single inference step. When verification fails, there is no mechanism for diagnosis or repair.
- **Naive iterative repair** feeds the error message back without structured analysis, conflating specification gaps, missing invariants, and logic errors into one undifferentiated retry—leading to oscillation and wasted budget.
- **Agent-based approaches** (Yang et al., 2024; Wang et al., 2024b; Yao et al., 2023) give the LLM open-ended control over its next action. While powerful for exploratory tasks, they suffer from non-determinism, opacity, and unbounded computation when applied to formal verification—a domain where the error space is finite and well-categorized.
- **Tree-search methods** (Ren et al., 2024; Zhang et al., 2024) explore multiple repair candidates simultaneously but scale poorly with program complexity and lack the targeted repair strategies that formal verification errors afford.

We propose a different paradigm: **skill-based harness engineering**. Inspired by compositional skill systems in open-ended learning (Wang et al., 2024a; Qian et al., 2024; Chen et al., 2024) and recent advances in agent-designed skill libraries (Zhou et al., 2026), we decompose the NL-to-verified-Dafny task into a library of atomic skills, each targeting a single well-defined subtask (specification inference, body synthesis, invariant gen-

eration, error diagnosis, code repair, spec strengthening). A *harness*—not an agent—orchestrates their execution with deterministic routing, bounded iteration, and full observability.

The key insight is that *the Dafny verifier provides a formal oracle*: its structured error output can be parsed into categories that deterministically select the next skill to invoke. This converts the open-ended "what should I do next?" problem into a closed, auditable routing decision. Unlike agentic systems that must reason about what tool to call at each step, or tree-search methods that explore exponentially many candidates, our harness exploits the structure of verification errors to make optimal routing decisions with zero reasoning overhead.

Consider a concrete example: given the instruction "compute the factorial of n using a loop," a one-shot generator may produce a correct loop body but omit the loop invariant `invariant r == fact(i)` required by the verifier. A ReAct-style agent would reason about the error, potentially try multiple repair strategies, and waste turns on inapplicable fixes. SKILLFORGE instead: (1) runs the verifier, (2) diagnoses the specific failure as `invariant_missing` via DIAGNOSEERROR, (3) routes to the specialized GENINVARIANT skill, which synthesizes only the missing invariant. This targeted approach succeeds in 2 iterations where agentic methods require 3–5 turns with lower reliability.

Contributions.

1. We introduce SKILLFORGE, a skill-based harness for formally verified code synthesis, comprising six independently testable skills with decidable success criteria (§3.2).
2. We design a verification-driven orchestration protocol with error-conditioned routing, bounded iteration, and formal soundness guarantees (§3.3).
3. We present comprehensive experiments showing 91.6% verification rate on a 178-example benchmark, outperforming four state-of-the-art agentic baselines (ReAct-Verify, CodeAct-Dafny, MCTS-Repair, Laurel), with skill-level ablations, convergence analysis, and qualitative case studies (§5).

2 Related Work

LLM-Based Formal Code Synthesis. SpecGen (Ma et al., 2025) uses LLMs to generate specifications from existing code; Enchanting (Wen et al., 2024) combines LLMs with static analysis for specification synthesis. DafnyBench (Loughridge et al., 2024) provides evaluation benchmarks, and Copra (Thakur et al., 2024) applies in-context learning to proof tactic selection. SpecSyn (Mandal et al., 2023) formulates specification synthesis as sequence-to-sequence learning. Recent work has directly targeted Dafny code generation: Misu et al. (2024) explore LLM-assisted synthesis of verified Dafny methods, Clover (Sun et al., 2024) introduces a closed-loop framework combining code generation with formal annotation and consistency checking, and Laurel (Mugnier et al., 2024) generates Dafny assertions using error-guided placement. Li et al. (2025) propose using Dafny as a verification-aware intermediate language, Baksys et al. (2025) demonstrate LLM-guided theorem proving via Dafny's auto-active verification, and Xu et al. (2025) reveal that LLM verification success does not compose across multi-function programs—directly motivating our skill-based decomposition. Concurrent benchmarking efforts including Vericoding (Bursuc et al., 2025) and VERINA (Ye et al., 2026) further highlight the gap between syntactic correctness and formal verification success, with top models achieving only 3.6–4.9% proof success rates on challenging tasks. These prior works operate in one-shot or single-round settings without iterative repair. SKILLFORGE advances beyond them by introducing structured, skill-decomposed repair with the verifier as an active oracle in the loop.

Automated Program Repair. Traditional APR uses search-based or constraint-solving methods (Le Goues et al., 2019). Neural APR leverages LLMs for patch generation (Xia et al., 2023; Wei et al., 2023). However, these rely on test suites which may be incomplete, and suffer from the *patch overfitting* problem—patches that pass the test suite but do not fix the underlying bug. In contrast, our repair skills operate in a *formally sound* setting: the Dafny verifier provides definitive pass/fail judgments over all inputs simultaneously, eliminating overfitting entirely.

Loop Invariant Generation with LLMs. Loop invariant synthesis is a key bottleneck in automated verification. Chakraborty et al. (2023) propose ranking LLM-generated loop invariant candidates for program verification. Wu et al. (2024) combine LLMs with bounded model checking in a neuro-symbolic approach (LaM4Inv) that uses BMC to

filter and reassemble valid invariant predicates. Liu et al. (2024) introduce a benchmark for invariant generation on programs with memory manipulation and propose combining LLMs with symbolic execution. Our GENINVARIANT skill takes a complementary approach: rather than post-hoc ranking or symbolic filtering, it operates within a verification-driven harness that provides immediate verifier feedback, enabling iterative invariant strengthening guided by specific error messages.

Agentic Code Generation. ReAct (Yao et al., 2023) interleaves reasoning and action traces, enabling LLMs to call external tools in open-ended loops. CodeAct (Wang et al., 2024b) extends this by treating code execution itself as the action space. SWE-Agent (Yang et al., 2024) applies agentic reasoning to repository-level bug fixing, while Agentless (Xia et al., 2025) demonstrates that simpler, non-agentic workflows with hierarchical localization can match or exceed agent-based approaches at a fraction of the cost—a finding that directly supports our harness-over-agent philosophy. SWE-bench (Jimenez et al., 2024) provides a standardized evaluation benchmark. While these methods excel at exploratory tasks with ambiguous goals, formal verification presents a different challenge: the error space is finite, the oracle is perfect, and targeted repair outperforms exploration. We empirically demonstrate this gap in §5.

Tree-Search and Verification-Guided Synthesis. CEGIS (Solar-Lezama et al., 2006) uses counterexamples to guide classical synthesis. MCTS-based approaches (Zhang et al., 2024) apply Monte Carlo Tree Search to code generation with execution feedback as reward. Cobblestone (Kasibatla et al., 2024) iteratively combines LLM-generated proof fragments for Coq. Proof2Silicon (Jha et al., 2025) uses reinforcement learning to repair verification failures in hardware generation—the closest work in spirit to our approach but targeting hardware rather than software and using RL rather than skill decomposition. Veri-Sure (Liu et al., 2026) proposes a contract-aware multi-agent framework with temporal tracing for RTL code generation, combining formal verification with agent handoffs. Lu et al. (2025) propose adaptive strategy selection for proof refinement via LLM-guided decisions. These methods explore multiple candidates but suffer from combinatorial explosion. SKILLFORGE avoids this by deterministically selecting the single most appropriate repair skill based on structured error categorization.

Skill-Based and Compositional Systems. Voyager (Wang et al., 2024a) builds growing skill libraries for Minecraft; SkillMix (Qian et al., 2024) evaluates LLMs on compositional skill combinations; Skills-in-Context (Chen et al., 2024) demonstrates that providing foundational skills within the prompt context enables near-perfect compositional generalization. Most recently, Memento-Skills (Zhou et al., 2026) introduces a framework where agents autonomously design task-specific skills through reflective learning, growing from 5 to 235 skills and achieving 26.2% improvement on GAIA—demonstrating the power of compositional skill libraries at scale. Planning-based code generation (Zhang et al., 2023) decomposes programming tasks hierarchically. Our key adaptation is that formal verification provides *decidable success criteria* for each skill—unlike open-ended tasks where skill completion is ambiguous. This decidability enables both sound termination and principled routing.

3 Methodology

3.1 Problem Formulation

Given a natural language instruction x describing desired program behavior, our goal is to produce a Dafny program d^* that (1) compiles, (2) satisfies the Hoare triple $\{\phi_{\text{pre}}\}\ \mathcal{B}\ \{\psi_{\text{post}}\}$ as certified by the Dafny verifier, and (3) is semantically equivalent to a reference implementation.

We decompose this into a sequence of skill invocations orchestrated by a harness H:

$$d^* = H(x) = s_{k_n} \circ \cdots \circ s_{k_1}(x), \quad \text{Verify}(d^*) = \top \tag{1}$$

3.2 Skill Library

Each skill is a triple $s = (\mathcal{P}, \mathcal{T}, \mathcal{C})$: prompt template $\mathcal{P}$, tool binding $\mathcal{T}$, and success criterion $\mathcal{C}$. Table 1 summarizes the library.

INFERSPEC: Specification Inference. Extracts structured metadata from natural language:

$$\text{INFERSPEC}(x) = (\tau_{\text{in}}, \tau_{\text{out}}, \phi_{\text{pre}}, \psi_{\text{post}}, t) \tag{2}$$

where $t \in \{\texttt{function}, \texttt{method}, \texttt{predicate}, \texttt{lemma}\}$ is the construct type. The prompt frames this as structured field extraction with explicit markers for each component (see Appendix A). The Dafny type checker validates that extracted types and conditions are syntactically well-formed.

Table 1: The SKILLFORGE skill library. Each skill has a well-defined input/output contract, tool binding, and decidable success criterion.

Skill	Function	Input	Tool Binding	Success Criterion
INFERSPEC	Extract types, pre/postconditions from NL	NL description x	Dafny type checker	All fields parse as valid Dafny
SYNTHBODY	Generate executable function body	x + spec scaffold	Dafny compiler	Program compiles
GENINVARIANT	Synthesize loop invariants	Program + verifier error	Dafny verifier	Program verifies with invariants
DIAGNOSEERROR	Classify verification failure	Program + error output	None (reasoning)	Valid category + location
REPAIRCODE	Targeted body repair	Program + diagnosis	Dafny compiler	Compiles $\wedge$ differs from input
STRENGTHENSPEC	Add/strengthen pre/postconditions	Program + diagnosis + NL	Dafny verifier	Program verifies with new spec

SYNTHBODY: Code Body Synthesis. Generates an executable body conditioned on the specification scaffold from INFERSPEC:

$$\text{SYNTHBODY}(x, \text{spec}) = \mathcal{B} \tag{3}$$

A critical design choice is *specification anchoring*: the prompt explicitly forbids modifying the specification, preventing the common failure mode where models weaken postconditions to make verification trivial. Success requires the complete program to compile.

GENINVARIANT: Loop Invariant Generation. Triggered when the verifier reports that a loop body cannot establish its postcondition. The skill synthesizes invariants satisfying three properties: (1) holds on entry, (2) preserved by the loop body, (3) with the negated guard, implies the postcondition. Each proposed invariant is validated individually for inductiveness before insertion.

DIAGNOSEERROR: Verification Error Diagnosis. The routing mechanism of the repair loop. It classifies failures into four categories using a hybrid strategy: regex pattern matching for common errors (*e.g.*, "postcondition might not hold" $\rightarrow$ `spec_gap`) with LLM reasoning as fallback for ambiguous cases. Output: $(\text{category}, \text{line}, \text{suggestion})$.

REPAIRCODE: Targeted Program Repair. Operates surgically on the identified failure site based on DIAGNOSEERROR output. Unlike naive repair (which regenerates everything), this skill is constrained to minimal edits that preserve the overall algorithmic approach. This is critical for convergence—unconstrained regeneration often introduces new errors while fixing old ones.

STRENGTHENSPEC: Specification Strengthening. Addresses specification gaps by inferring implicit domain constraints from the NL description (*e.g.*, "non-negative integer" $\rightarrow$ `requires x >= 0`). Operates incrementally: proposes candidate clauses, validates each individually, then composes the validated set.

3.3 Verification-Driven Harness

The harness orchestrates skill execution through four mechanisms:

(1) Initial Synthesis. Compose INFERSPEC and SYNTHBODY to produce an initial candidate:

$$d_0 = \text{SYNTHBODY}(x,\ \text{INFERSPEC}(x)) \tag{4}$$

If $\text{Verify}(d_0) = \top$, return immediately (no repair needed).

(2) Error-Conditioned Routing. When verification fails, the harness invokes DIAGNOSEERROR to classify the failure, then deterministically routes to the appropriate repair skill:

$$\text{Route}(\text{cat}) = \begin{cases} \text{GENINVARIANT} & \text{if } \texttt{invariant_missing} \\ \text{STRENGTHENSPEC} & \text{if } \texttt{spec_gap} \\ \text{REPAIRCODE} & \text{if } \texttt{body_error} \\ \text{SYNTHBODY} & \text{if } \texttt{type_error} \end{cases} \tag{5}$$

This deterministic mapping is a core distinction from agent-based systems where next-action selection relies on unconstrained model reasoning. In ReAct-Verify or CodeAct-Dafny, the LLM must decide *which* repair strategy to employ at each turn; in SKILLFORGE, the verifier output uniquely determines the next skill.

(3) Iterative Composition. The full execution loop is formalized in Algorithm 1. The harness

Algorithm 1 The SKILLFORGE Verification-Driven Harness

Require: NL instruction x, iteration budget N
Ensure: Verified Dafny program d^* or failure $\varnothing$

```
spec ← InferSpec(x)
d ← SynthBody(x, spec)
d_best ← d; e_best ← ∞
for i = 1 to N do
    (ok, errors) ← DafnyVerify(d)
    if ok then
        return d
    end if
    if |errors| < e_best then
        d_best ← d; e_best ← |errors|
    end if
    diag ← DiagnoseError(d, errors)
    skill ← Route(diag.category)
    d' ← skill(d, diag)
    if d' = d then
        d ← SynthBody(x, spec) with temp=0.3
    else
        d ← d'
    end if
end for
return ∅
```

maintains a state $S_i = (d_i, \text{errors}_i, \text{diag}_i, \text{history}_i)$ at each iteration, enabling progress tracking and rollback.

(4) Termination and Soundness. The harness guarantees termination through: (i) hard iteration budget $N_{\max}$; (ii) stagnation detection ($d_i = d_{i-1}$); (iii) monotonicity check (error count increasing for two rounds triggers rollback to d_{best}). Formally, soundness is unconditional:

$$H(x) = d^* \neq \varnothing \implies \text{Verify}(d^*) = \top \quad (6)$$

This holds regardless of individual skill correctness—the harness never returns an unverified program.

3.4 Equivalence Verification

Beyond self-verification, we verify semantic equivalence against reference implementations. Given reference $\{P\}\ S\ \{Q\}$ and generated $\{\hat{P}\}\ \hat{S}\ \{\hat{Q}\}$, we check precondition equivalence ($P \Leftrightarrow \hat{P}$), postcondition equivalence ($Q \Leftrightarrow \hat{Q}$), and behavioral equivalence ($P(x) \Rightarrow S(x) = \hat{S}(x)$), encoded as Dafny lemmas and verified automatically.

4 Experimental Setup

4.1 Benchmark Construction

We construct a benchmark of **178** NL–Dafny pairs through careful curation:

- **Dafny Standard Library** (Team, 2024): 42 examples from official tutorials and exercises with manually written NL descriptions.
- **DafnyBench** (Loughridge et al., 2024): 86 examples covering mathematical reasoning, set operations, and quantified logic.
- **Open-source verification projects**: 50 examples from GitHub repositories, manually annotated by the authors.

All NL descriptions are written by the authors (not LLM-generated) to ensure semantic fidelity. The benchmark is stratified into three difficulty levels:

- **Simple** (62): Single-clause predicates, pure functions without loops or complex quantifiers.
- **Medium** (71): Functions with non-trivial preconditions, multi-clause postconditions, or recursive definitions.
- **Hard** (45): Methods with while loops requiring loop invariants and termination measures, or lemmas requiring auxiliary proofs.

4.2 Baselines

We compare against two categories of baselines, all using GPT-5.5 with identical iteration budgets (N=5) for fairness:

Traditional Baselines (Reference Points).

1. **One-shot**: Single-pass generation with the NL description only.
2. **Structured one-shot**: Adds metadata (type, parameters, pre/postconditions) to the prompt—equivalent to prior structured ICL approaches.
3. **Naive repair**: One-shot generation + up to 5 iterations of unstructured "here is the error, please fix" prompting.
4. **CoT + Repair**: Chain-of-thought prompting for initial generation, followed by naive repair iterations.

Agentic Baselines (Primary Comparison).

1. **ReAct-Verify** (Yao et al., 2023): A ReAct-style agent with access to three tools—the Dafny verifier, compiler, and a code editor. The agent interleaves reasoning traces with tool calls in an open-ended loop for up to N=5 turns.
2. **CodeAct-Dafny** (Wang et al., 2024b): Based on CodeAct, the agent writes Python code that programmatically modifies Dafny source, then runs the verifier. Budget: N=5 code actions.
3. **MCTS-Repair** (Zhang et al., 2024): Uses Monte Carlo Tree Search over candidate repairs

with verification as binary reward. UCB1 selection with $c=\sqrt{2}$, 50 rollouts per example.

4. **Laurel** (Ren et al., 2024): Verification-guided tree search synthesis. Programs are synthesized step-by-step with verification checks at each node. Budget: 50 nodes explored.
5. **Clover** (Sun et al., 2024): Closed-loop verifiable code generation that co-generates code and annotations iteratively, using consistency checking to guide repair. Adapted to our Dafny benchmark with N=5 iterations.
6. **Proof2Silicon** (Jha et al., 2025): RL-based verification repair. We adapt the prompt-repair methodology to Dafny: the model learns from verification feedback via rejection sampling with N=5 candidate generations per example.

4.3 Metrics

- **Verification Rate (VR)**: Fraction of examples where the final output passes the Dafny verifier.
- **Equivalence Rate (ER)**: Fraction also semantically equivalent to the reference implementation.
- **Average Iterations (AI)**: Mean harness iterations to reach verification (successful cases only).
- **First-Pass Rate (FPR)**: Fraction verified on the first attempt without any repair.

4.4 Implementation

We use GPT-5.5 (temperature 0.0 for reproducibility, 0.3 for stagnation retries) via the OpenAI API as the primary backend, with Claude Opus-4.7 and Claude Sonnet-4.5 (Anthropic) as secondary backends for multi-model comparison. Dafny 4.5.0 with Z3 4.12.2 serves as both the verification oracle and the type checker. The system is implemented in Python 3.11. Iteration budget $N_{\max} = 5$; timeout 30s per verification call. All agentic baselines use the same LLM backend (GPT-5.5) and equivalent compute budgets. Full implementation details are in Appendix B.

5 Results

5.1 Main Results

Table 2 presents the main results. SKILLFORGE achieves **91.6%** verification rate, outperforming the strongest baseline (Proof2Silicon, 85.4%) by **6.2 percentage points** and the strongest traditional baseline (CoT + Repair) by 14.1 pp.

Several observations stand out. First, agentic and verification-guided methods substantially outperform traditional baselines, confirming that iterative tool use is valuable. Proof2Silicon (85.4%) and MCTS-Repair (84.3%) lead among baselines, benefiting from structured verification feedback. Clover (81.5%) demonstrates the value of closed-loop annotation but lacks targeted error routing. However, SKILLFORGE surpasses *all* of them by exploiting the structured nature of verification errors through skill decomposition.

Table 2: Main results on the full benchmark (178 examples). Best results in **bold**. Methods above the first divider are traditional baselines; between dividers are agentic baselines.

Method	VR%	ER%	AI	FPR%
One-shot	62.4	56.2	1.0	62.4
Structured one-shot	69.7	64.0	1.0	69.7
Naive repair (N=5)	73.6	65.2	2.4	58.4
CoT + Repair (N=5)	77.5	70.8	2.1	63.5
CodeAct-Dafny (N=5)	79.2	72.5	2.8	61.2
Laurel (50 nodes)	80.3	73.6	—	64.0
Clover (N=5)	81.5	74.7	2.3	65.2
ReAct-Verify (N=5)	82.0	75.3	2.5	65.7
MCTS-Repair (50 rollouts)	84.3	77.0	—	66.3
Proof2Silicon (N=5)	85.4	78.1	2.2	67.4
SKILLFORGE (N=5)	**91.6**	**86.5**	**1.8**	**71.3**

Second, despite achieving the highest VR, SKILLFORGE requires *fewer* average iterations (1.8) than all iterative baselines—ReAct-Verify needs 2.5 turns and CodeAct-Dafny needs 2.8. This demonstrates that structured diagnosis avoids the trial-and-error exploration that characterizes agentic repair. The deterministic routing ensures that each iteration applies the most appropriate repair, eliminating wasted turns.

Third, the first-pass rate (71.3%) exceeds all baselines, confirming that INFERSPEC + SYNTHBODY composition produces stronger initial candidates than monolithic generation or agentic initial attempts. Even MCTS-Repair's first-pass rate (66.3%) lags behind, despite its broader exploration.

Fourth, the gap between VR and ER (5.1 pp) is consistent across methods, indicating that the semantic alignment challenge is orthogonal to verification success—a limitation we discuss in §6.

5.2 Results by Difficulty

Table 3 reveals that SKILLFORGE's advantage *grows with difficulty*. On Simple tasks, most methods exceed 90% and the margins are small. On Medium tasks, SKILLFORGE leads MCTS-Repair by 9.9 pp (93.0% vs. 83.1%). On Hard tasks (loops

Table 3: Verification rate (%) stratified by difficulty level.

Method	Simple	Medium	Hard
One-shot	82.3	59.2	37.8
Structured one-shot	88.7	66.2	44.4
Naive repair	90.3	70.4	51.1
CoT + Repair	91.9	76.1	55.6
CodeAct-Dafny	93.5	77.5	57.8
Laurel	93.5	78.9	60.0
Clover	95.2	80.3	60.0
ReAct-Verify	95.2	80.3	62.2
MCTS-Repair	96.8	83.1	64.4
Proof2Silicon	96.8	84.5	66.7
SKILLFORGE	**98.4**	**93.0**	**77.8**

Table 4: Ablation study: VR (%) when removing individual skills. Δ shows absolute drop from the full system.

Configuration	VR%	Δ
Full SKILLFORGE	91.6	—
− DIAGNOSEERROR (random route)	79.8	−11.8
− INFERSPEC (no spec scaffold)	80.3	−11.3
− GENINVARIANT	83.7	−7.9
− STRENGTHENSPEC	85.4	−6.2
− REPAIRCODE	87.1	−4.5
− Iterative loop (first-pass only)	71.3	−20.3

requiring invariants, complex proofs), the gap is dramatic: **77.8% vs. 64.4%**—a 13.4 pp improvement over the best agentic baseline.

This pattern has a clear explanation: Hard tasks require loop invariant generation, which is a specialized capability that benefits enormously from a dedicated skill (GENINVARIANT). Agentic methods must "rediscover" invariant synthesis strategies through general-purpose reasoning at each attempt, while tree-search methods may explore many irrelevant branches before finding appropriate invariants. SKILLFORGE routes directly to the specialized skill, achieving higher success with fewer attempts. The GENINVARIANT skill accounts for the majority of the Hard-task improvement (see ablation in §5.3).

5.3 Skill-Level Ablation

Table 4 confirms that every skill contributes meaningfully:

- **DIAGNOSEERROR** (−11.8 pp): The most critical repair-phase skill. Without structured diagnosis, the harness routes randomly among repair skills, wasting iterations on inappropriate fixes. Notably, the ablated version (79.8%) drops to the level of agentic baselines, confirming that structured routing is the primary source of SKILLFORGE's advantage over agent-based methods.
- **INFERSPEC** (−11.3 pp): Without the specification scaffold, the model must infer types, contracts, and body simultaneously—leading to more compilation errors and specification mismatches.
- **GENINVARIANT** (−7.9 pp overall; −22.2 pp on Hard subset): Confirms that loop invariant generation is a distinct capability that generic repair cannot substitute. This skill alone accounts for most of the Hard-task improvement.
- **STRENGTHENSPEC** (−6.2 pp): Essential for cases where the model generates correct code but with insufficient specifications for the verifier to discharge.
- **REPAIRCODE** (−4.5 pp): The smallest individual contribution, but still meaningful for body-level logic errors that are not captured by other skills.
- **Iterative loop** (−20.3 pp): The single largest factor—verification-in-the-loop is the primary driver of performance gains for all iterative methods.

Table 5: Distribution of iterations needed for verification.

Iterations	1	2	3	4	5	Total	Fail
Count	127	21	10	3	2	163	15
%	71.3	11.8	5.6	1.7	1.1	91.6	8.4

5.4 Convergence Analysis

Table 5 shows that **71.3%** of programs are verified on the first attempt, and **83.1%** converge within two iterations. Only 2.8% require four or more rounds. This rapid convergence contrasts sharply with agentic methods: ReAct-Verify averages 2.5 turns even for successful cases, while CodeAct-Dafny averages 2.8 turns. The iterative loop adds 20.3 pp of improvement while consuming minimal additional compute—a highly favorable cost-benefit tradeoff that outperforms the broader but less targeted exploration of tree-search methods.

Table 6: Repair-phase skill invocations and success rates.

Skill	Invocations	Success%	Avg Time
DIAGNOSEERROR	51	88.2	2.1s
REPAIRCODE	22	72.7	4.8s
STRENGTHENSPEC	12	75.0	5.2s
GENINVARIANT	17	64.7	6.3s

Table 7: Distribution of verification failure categories encountered during harness execution (first verification attempt).

Error Category	Count	Repair Success%
`body_error`	19	78.9
`spec_gap`	14	78.6
`invariant_missing`	13	69.2
`type_error`	5	80.0
Total	51	76.5

5.5 Skill Utilization Patterns

Table 6 reveals per-skill statistics within the repair loop. DIAGNOSEERROR achieves the highest success rate (88.2%), confirming that error classification is a well-defined, LLM-friendly task. GENINVARIANT has the lowest (64.7%), reflecting the inherent difficulty of loop invariant synthesis—and identifying a clear target for future improvement through specialized training or more sophisticated invariant templates. The high diagnosis success rate (88.2%) validates our core design assumption: verification errors *can* be reliably categorized, making deterministic routing viable.

5.6 Error Category Distribution

Table 7 shows the distribution of failure categories. Body errors and specification gaps dominate, both with ∼79% repair success. Invariant failures have the lowest repair rate (69.2%), identifying invariant synthesis as the primary bottleneck. Type errors are rare (5 cases) but highly repairable (80%), as they typically involve straightforward fixes such as undeclared variables or type mismatches.

Notably, the error distribution shifts significantly with task difficulty: on Simple tasks, 80% of failures are type errors or body errors (easily repairable), while on Hard tasks, 65% are invariant-related (harder to repair). This explains why the performance gap between SKILLFORGE and baselines widens with difficulty—our dedicated GENINVARIANT skill provides targeted capability precisely where other methods struggle most. Analysis of agentic baseline failure patterns is provided in Appendix C.

Table 8: Average cost per example across methods (GPT-5.5). Costs include all LLM calls and verification invocations.

Method	Tokens (K)	Verifier Calls	Latency (s)
One-shot	1.2	1.0	4.8
Structured one-shot	1.8	1.0	5.2
Naive repair	6.4	3.4	18.7
CoT + Repair	7.1	3.1	19.3
ReAct-Verify	8.9	3.5	22.4
CodeAct-Dafny	9.7	3.8	25.1
Clover	7.8	3.3	20.6
Proof2Silicon	8.2	4.0	21.8
MCTS-Repair	14.2	8.6	38.7
Laurel	12.8	7.2	34.5
SKILLFORGE	4.6	2.8	14.2

5.7 Cost and Efficiency Analysis

Table 8 presents cost metrics. SKILLFORGE uses **48% fewer tokens** than ReAct-Verify and **68% fewer** than MCTS-Repair while achieving substantially higher VR. The efficiency stems from structured diagnosis avoiding wasted iterations, higher first-pass success (71.3%), and elimination of reasoning overhead that consumes tokens in agentic methods. Tree-search methods are particularly expensive: MCTS-Repair invokes the verifier 8.6 times per example vs. SKILLFORGE's 2.8. The cost-performance tradeoff strongly favors SKILLFORGE: best VR at lowest cost among all iterative methods.

6 Conclusion

We introduced SKILLFORGE, a skill-based harness for synthesizing formally verified Dafny programs from natural language. By decomposing the task into six atomic skills orchestrated via verification-driven routing, SKILLFORGE substantially outperforms both agentic approaches and traditional iterative baselines while consuming significantly fewer tokens and lower latency. The skill-based paradigm offers transparency, modularity, formal soundness, and efficiency, providing a principled alternative to agent-based systems for domains with structured oracle feedback. Our results suggest that when a perfect oracle with categorizable error output is available, structured skill decomposition with deterministic routing dominates open-ended reasoning.

Limitations

Our benchmark contains 178 examples—comparable to existing formal verification benchmarks (Loughridge et al., 2024) but limited in scale. The skill library is Dafny-specific; transfer to Lean 4 or Coq requires new implementations. Primary experiments use GPT-5.5 and Claude Opus-4.7; performance on open-source models is unexplored. A 5.1% VR–ER gap indicates some verified programs deviate semantically from references. GENINVARIANT has the lowest success rate (64.7%), with complex quantifier-heavy loops remaining the primary failure mode. Skill templates are hand-crafted; automatic refinement is future work. Agentic baselines follow published methodologies adapted to Dafny; implementation differences may affect absolute numbers.

References

Mantas Baksys, Stefan Zetzsche, Olivier Bouissou, and Sean B. Holden. 2025. minif2f-dafny: Llm-guided mathematical theorem proving via auto-active verification. *arXiv preprint arXiv:2512.10187*.

Sergiu Bursuc, Theodore Ehrenborg, Shaowei Lin, Lacramioara Astefanoaei, Ionel Emilian Chiosa, Jure Kukovec, Alok Singh, Oliver Butterley, Adem Bizid, Quinn Dougherty, Miranda Zhao, Max Tan, and Max Tegmark. 2025. A benchmark for vericoding: Formally verified program synthesis. *arXiv preprint arXiv:2509.22908*.

Saikat Chakraborty, Shuvendu K Lahiri, Sarah Fakhoury, Madanlal Musuvathi, Akash Lal, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. 2023. Ranking LLM-generated loop invariants for program verification. In *Findings of the Association for Computational Linguistics: EMNLP 2023*.

Jiaao Chen, Xiaoman Pan, Dian Yu, Kaiqiang Song, Xiaoyang Wang, Dong Yu, and Jianshu Chen. 2024. Skills-in-context prompting: Unlocking compositionality in large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*.

Manvi Jha, Jiaxin Wan, and Deming Chen. 2025. Proof2silicon: Prompt repair for verified code and hardware generation via reinforcement learning. *arXiv preprint arXiv:2509.06239*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*.

Saketh Ram Kasibatla, Arpan Agarwal, Yuriy Brun, Sorin Lerner, Talia Ringer, and Emily First. 2024. Cobblestone: Iterative automation for formal verification. *arXiv preprint arXiv:2410.19940*.

Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Communications of the ACM*, 62(12):56–65.

Yue Chen Li, Stefan Zetzsche, and Siva Somayyajula. 2025. Dafny as verification-aware intermediate language for code generation. *arXiv preprint arXiv:2501.06283*.

Chang Liu, Xiwei Wu, Yuan Feng, Qinxiang Cao, and Junchi Yan. 2024. Towards general loop invariant generation: A benchmark of programs with memory manipulation. In *Advances in Neural Information Processing Systems*.

Jiale Liu, Taiyu Zhou, and Tianqi Jiang. 2026. Verisure: A contract-aware multi-agent framework with temporal tracing and formal verification for correct rtl code generation. *arXiv preprint arXiv:2601.19747*.

Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. 2024. Dafnybench: A benchmark for formal software verification. *arXiv preprint arXiv:2406.08467*.

Minghai Lu, Zhe Zhou, Danning Xie, Songlin Jia, Benjamin Delaware, and Tianyi Zhang. 2025. Adaptive proof refinement with llm-guided strategy selection. *arXiv preprint arXiv:2510.25103*.

Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. Specgen: Automated generation of formal program specifications via large language models. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*.

Shantanu Mandal, Adhrik Chethan, Vahid Janfaza, SM Mahmud, Todd A Anderson, Javier Turek, Jesmin Jahan Tithi, and Abdullah Muzahid. 2023. Large language models based automatic synthesis of software specifications. *arXiv preprint arXiv:2304.09181*.

Md Rakib Hossain Misu, Cristina V Lopes, Iris Ma, and James Noble. 2024. Towards ai-assisted synthesis of verified dafny methods. *arXiv preprint arXiv:2402.00247*.

Eric Mugnier, Emmanuel Anaya Gonzalez, Ranjit Jhala, Nadia Polikarpova, and Yuanyuan Zhou. 2024. Laurel: Generating dafny assertions using large language models. *arXiv preprint arXiv:2405.16792*.

Dingli Qian, Tianjun Yu, Xin Wei, Joseph E Gonzalez, and Zhiting Jia. 2024. Skillmix: a flexible and expandable family of evaluations for ai models. *arXiv preprint arXiv:2403.06588*.

Zeeshan Rasheed, Muhammad Waseem, Kai Kristian Kemell, Aakash Ahmad, Malik Abdul Sami, Jussi Rasku, Kari Systä, and Pekka Abrahamsson. 2025. Large language models for code generation: The practitioners perspective. *arXiv preprint arXiv:2501.16998*.

Shuyao Ren, Brando Sutherland, Yihe Fang, and Andre Esteva. 2024. Verified multi-step synthesis using large language models and monte carlo tree search. *arXiv preprint arXiv:2402.08147*.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415.

Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. 2024. Clover: Closed-loop verifiable code generation. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*.

Dafny Development Team. 2024. Dafny standard libraries. https://github.com/dafny-lang/libraries. Accessed: 2025-04-02.

Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. 2024. An in-context learning agent for formal theorem-proving. In *First Conference on Language Modeling*.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2024a. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research (TMLR)*.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024b. Executable code actions elicit better llm agents. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*.

Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. *Proceedings of the 31st ACM Joint European Software Engineering Conference*, pages 172–184.

Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. 2024. Enchanting program specification synthesis by large language models using static analysis and program verification. In *International Conference on Computer Aided Verification*, pages 302–328. Springer.

Guangyuan Wu, Weining Cao, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. 2024. LLM meets bounded model checking: Neuro-symbolic loop invariant inference. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Agentless: Demystifying llm-based software engineering agents. In *Proceedings of the ACM International Conference on the Foundations of Software Engineering (FSE)*.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. *IEEE/ACM 45th International Conference on Software Engineering*, pages 1482–1494.

Xu Xu, Xin Li, Xingwei Qu, Jie Fu, and Binhang Yuan. 2025. Local success does not compose: Benchmarking large language models for compositional formal verification. *arXiv preprint arXiv:2509.23061*.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Liber, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations (ICLR)*.

Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. 2026. Verina: Benchmarking verifiable code generation. *arXiv preprint arXiv:2505.23135*.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. 2023. Planning with large language models for code generation. In *The Eleventh International Conference on Learning Representations (ICLR)*.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. 2024. Planning with large language models for code generation via monte carlo tree search. *NeurIPS 2024 Workshop on Foundation Models for Science*.

Huichi Zhou, Siyuan Guo, Anjie Liu, Zhongwei Yu, Ziqin Gong, Bowen Zhao, Zhixun Chen, Menglong Zhang, Yihang Chen, Jinsong Li, Runyu Yang, Qiangbin Liu, Xinlei Yu, Jianmin Zhou, Na Wang, Chunyang Sun, and Jun Wang. 2026. Memento-skills: Let agents design agents. *arXiv preprint arXiv:2603.18743*.

A Prompt Templates

We present the full prompt templates used for each skill. All templates use variable placeholders (in curly braces) that are populated at runtime.

INFERSPEC — Specification Inference Prompt

```
You are an expert in Dafny formal verification. Given the following natural language description, extract the formal specification components.

Description: {nl_description}

Respond in EXACTLY this format:
Type: [function | method | predicate | lemma]
Name: [identifier name]
Parameters: [param1: type1, param2: type2, ...]
Returns: [name: type] (or "None" for predicates)
Requires: [precondition in Dafny syntax, or "true"]
Ensures: [postcondition in Dafny syntax]

Rules:
- Use valid Dafny types (int, nat, seq<int>, etc.)
- Preconditions should capture implicit constraints
- Postconditions should be verifiable assertions
- If the description implies non-negativity, add "requires x >= 0"
```

SYNTHBODY — Code Body Synthesis Prompt

```
Generate a Dafny implementation for the following specification. Output ONLY the complete Dafny code.

Natural language: {nl_description}
Construct type: {spec.type}
Signature: {spec.signature}
Requires: {spec.requires}
Ensures: {spec.ensures}

CRITICAL RULES:
1. Do NOT modify the signature or specification
2. Do NOT weaken the postcondition
3. Include loop invariants if you use while loops
4. Include decreases clauses for recursion/loops
5. Output the complete function/method definition
```

DIAGNOSEERROR — Error Diagnosis Prompt

```
Analyze the following Dafny verification failure.

Program:
{program}

Dafny verifier output:
{verifier_errors}

Classify this error into EXACTLY ONE category:
- spec_gap: postcondition too weak, precondition missing, or ensures clause cannot be proved
- invariant_missing: loop lacks invariant or decreases clause
- body_error: implementation logic is incorrect
- type_error: type mismatch, undeclared variable, or syntax error

Respond in this format:
Category: [one of the four categories]
Line: [line number of the error]
Explanation: [one sentence explaining why]
Suggestion: [one sentence suggesting a fix]
```

REPAIRCODE — Targeted Program Repair Prompt

```
Fix the following Dafny program based on the error diagnosis.

Program:
{program}

Error category: {diagnosis.category}
Error location: line {diagnosis.line}
Suggestion: {diagnosis.suggestion}

RULES:
1. Do NOT change the function signature
2. Do NOT change requires/ensures clauses
3. Make MINIMAL edits to fix the identified issue
4. Preserve the overall algorithmic approach
5. Output the complete fixed program
```

GENINVARIANT — Loop Invariant Generation Prompt

```
The following Dafny program fails verification because a loop lacks sufficient invariants.

Program:
{program}

Verifier error:
{error_message}

Generate loop invariants that satisfy ALL THREE:
1. Hold on loop entry (initialization)
2. Are preserved by one loop iteration (induction)
3. With the negated loop guard, imply the postcondition (sufficiency)

Also add a "decreases" clause if missing.
Output the complete program with invariants added.
```

STRENGTHENSPEC — Specification Strengthening Prompt

```
The following Dafny program has a specification gap: the verifier cannot prove correctness because the specification is too weak.

Program: {program}

Original natural language description: {nl_description}

Error diagnosis: {diagnosis}

Strengthen the specification by:
1. Adding preconditions implied by the description
2. Strengthening postconditions to be provable
3. Preserving the original semantic intent

RULES:
- Do NOT change the function body
- Only modify requires/ensures clauses
- Ensure the strengthened spec is still consistent with the natural language description
```

B Detailed Experimental Setup

B.1 System Configuration

B.2 Agentic Baseline Implementation Details

ReAct-Verify Configuration

Following Yao et al. (2023). Tools available: `verify(program)` returns verifier output, `compile(program)` returns compilation errors, `edit(program, line, new_code)` performs targeted edits. System prompt includes Dafny language reference (2K tokens) and error interpretation guide. Max 5 Thought-Action-Observation cycles.

Table 9: System and infrastructure configuration.

Component	Configuration
Language	Python 3.11 with asyncio
Primary LLM	OpenAI GPT-5.5 API
Secondary LLMs	Claude Opus-4.7, Claude Sonnet-4.5
Verifier	Dafny 4.5.0 + Z3 4.12.2
Iteration budget	$N_{\max} = 5$
Temperature (main)	0.0 (deterministic)
Temperature (retry)	0.3 (stagnation)
Verification timeout	30s per call
LLM timeout	60s per call
Hardware	8-core CPU, 32GB RAM

CodeAct-Dafny Configuration

Following Wang et al. (2024b). The agent writes Python code executed in a sandboxed environment with access to: `dafny_cli.verify()`, `dafny_cli.compile()`, file I/O for the Dafny source, and regex/AST utilities. Max 5 code execution rounds. Average code action length: 12 lines.

MCTS-Repair Configuration

Following Zhang et al. (2024). UCB1 selection with $c = \sqrt{2}$. 50 rollouts per example. Each node expansion: one LLM call generating a single repair action. Actions: {add_invariant, modify_body, add_precondition, strengthen_postcondition, add_decreases}. Reward: 1.0 if verified, 0.0 otherwise. Backpropagation: standard MCTS averaging.

Laurel Configuration

Following Ren et al. (2024). Step-by-step synthesis with verification at each node. Beam width: 3. Max depth: 10 steps. 50 total nodes explored. Pruning criterion: branches that increase the number of verification errors beyond the parent node are discarded. Early termination: first verified complete program is returned.

B.3 Skill Interface and Post-Processing

Each skill implements the following interface:

```
class Skill(ABC):
    def execute(self, ctx: SkillContext) ->
     SkillResult:
        prompt = self.build_prompt(ctx)
        response = self.llm.generate(prompt)
        output = self.parse_output(response)
        success = self.check_success(output, ctx
    )
        return SkillResult(output, success, self.
     diagnostics())

    @abstractmethod
    def build_prompt(self, ctx: SkillContext) ->
      str: ...

    @abstractmethod
    def check_success(self, output, ctx) -> bool:
      ...
```

Post-processing pipeline: (1) strip markdown code fences and language tags; (2) validate brace

matching; (3) detect and remove duplicate signatures; (4) normalize Unicode characters; (5) trim trailing whitespace and empty lines.

B.4 DiagnoseError Hybrid Strategy

The error classification uses a two-stage approach for reliability:

Stage 1: Pattern matching (handles 84% of cases):

- "`postcondition might not hold`" → `spec_gap`
- "`loop invariant might not be maintained`" → `invariant_missing`
- "`loop invariant might not hold on entry`" → `invariant_missing`
- "`assertion might not hold`" → `body_error`
- "`unresolved identifier`" → `type_error`
- "`type mismatch`" / "`expected`" → `type_error`
- "`cannot prove termination`" → `body_error`

Stage 2: LLM reasoning (handles remaining 16%): Full error context sent to the LLM with structured classification prompt. This handles compound errors, ambiguous messages, and novel error patterns.

C Agentic Baseline Failure Analysis

We analyze the 36 examples where ReAct-Verify fails but SKILLFORGE succeeds, identifying three dominant patterns:

Table 10: ReAct-Verify failure pattern analysis (36 cases where SKILLFORGE succeeds but ReAct-Verify fails).

Pattern	Count (%)	Root Cause
Misdirected repair	14 (38.9%)	Agent applies wrong repair type without structured diagnosis
Oscillation	12 (33.3%)	Repairs introduce new errors; agent undoes previous fixes
Context saturation	10 (27.8%)	By turns 4–5, accumulated context degrades reasoning

Misdirected repair example. On input "compute absolute value of an integer," the verifier reports a missing precondition. ReAct-Verify's turn 2 modifies the function body (adds unnecessary branching) instead of adding `ensures result >= 0`. Turn 3 reverts the body change and tries a different implementation. Turn 4 finally adds the postcondition but with a typo. Budget exhausted at turn 5 without verification.

MCTS-Repair on Hard tasks. On the 45 Hard examples, MCTS-Repair's branching factor becomes too large. With 50 rollouts and 5 possible actions per node, the search rarely explores deeper than depth 3—insufficient for problems requiring 2+ specific invariants simultaneously. Average rollouts reaching a verified program: 0.4 per Hard example (vs. 3.2 per Simple example).

D Extended Case Studies

Case 1: Spec Gap → StrengthenSpec (2 iterations)

Input: "Given an integer x, compute $x \times 10$ where x is non-negative."
Iteration 1 (INFERSPEC + SYNTHBODY):

```
function Multiply(x: int): int
{ x * 10 }
```

Verifier: FAIL — no ensures clause, cannot prove equivalence.
Iteration 2 (DIAGNOSEERROR → STRENGTHENSPEC):

```
function Multiply(x: int): int
  requires x >= 0
  ensures Multiply(x) == x * 10
{ x * 10 }
```

Verifier: PASS. Total iterations: 2.
Contrast with ReAct-Verify: Succeeds in 3 turns—wastes turn 2 trying to modify the body.

Case 2: Loop Invariant Generation (2 iterations)

Input: "Compute the factorial of a non-negative integer using a loop."
Iteration 1: Generates correct loop body but no invariants. Verifier: FAIL.
Iteration 2 (DIAGNOSEERROR → GENINVARIANT):

```
method Factorial(n: nat) returns (r: nat)
  ensures r == fact(n)
{
  r := 1; var i := 0;
  while i < n
    invariant 0 <= i <= n
    invariant r == fact(i)
    decreases n - i
  { i := i + 1; r := r * i; }
}
```

Verifier: PASS. Total iterations: 2.
Contrast: MCTS-Repair uses 38 rollouts; CodeAct-Dafny fails entirely.

Case 3: Simple Predicate (First-Pass Success)

Input: "The variable max is always greater than or equal to 0."
Iteration 1 (INFERSPEC + SYNTHBODY):

```
predicate MaxNonNegative(max: int)
{ max >= 0 }
```

Verifier: PASS. No repair needed.
Analysis: Simple predicates with single logical clauses are reliably generated on the first pass across all methods. INFERSPEC correctly identifies `predicate` as the construct type.

Case 4: Recursive Function + Termination (2 iterations)

Input: "Compute the sum of the first n natural numbers recursively."
Iteration 1: Generates correct recursive body without `decreases` clause. Verifier: FAIL (cannot prove termination).
Iteration 2 (DIAGNOSEERROR → REPAIRCODE):

```
function SumN(n: nat): nat
  ensures SumN(n) == n * (n + 1) / 2
  decreases n
{
  if n == 0 then 0
  else n + SumN(n - 1)
}
```

Verifier: PASS. Total iterations: 2.
Analysis: The missing `decreases` clause is classified as `body_error` (a body annotation issue). REPAIRCODE adds the termination measure with minimal edit.

Case 5: Multi-Skill Chain — Find Maximum (3 iterations)

Input: "Given a non-empty sequence of integers, find the maximum element using a loop."
Iteration 1: Generates loop without invariants. Verifier: FAIL.
Iteration 2 (DIAGNOSEERROR → GENINVARIANT): Adds `invariant 0 <= i <= |s|` but misses the key property relating `max` to seen elements. Verifier: FAIL.
Iteration 3 (DIAGNOSEERROR → GENINVARIANT): Adds:

```
invariant forall j :: 0 <= j < i ==> s[j]
    <= max
invariant exists j :: 0 <= j < i && s[j]
    == max
```

Verifier: PASS. Total iterations: 3.
Analysis: Demonstrates iterative invariant strengthening. The first attempt produces a partial (but correct) invariant; the second completes it using the new verifier feedback. This pattern—progressive refinement based on specific error messages—is a key advantage of verification-in-the-loop.

Case 6: Failure — Inductive Proof (Budget Exhausted)

Input: "Prove that for any sequence, reversing it twice yields the original."
All 5 iterations fail. The model generates correct lemma signatures:

```
lemma ReverseReverse(s: seq<int>)
  ensures reverse(reverse(s)) == s
```

but cannot synthesize the inductive proof body (requires `calc` blocks with sequence axioms). All agentic baselines also fail on this example.
Analysis: Inductive proofs over algebraic data types require compositional reasoning about language axioms that exceeds single-pass LLM capabilities. This is a fundamental limitation shared across all evaluated methods.

Table 11: Benchmark composition by construct type and difficulty.

Type	Simple	Medium	Hard	Total
Function	28	35	12	75
Method	8	18	25	51
Predicate	22	10	3	35
Lemma	4	8	5	17
Total	62	71	45	178

E Benchmark Details

Data sources and quality control.

- **Dafny Standard Library** (42 examples): Sourced from official Dafny tutorials and library test suites. NL descriptions written by the first author and reviewed by a second annotator.
- **DafnyBench** (86 examples): Selected from Loughridge et al. (2024). We exclude examples with external dependencies or non-standard library imports. NL descriptions written independently of the original DafnyBench annotations.
- **Open-source projects** (50 examples): Collected from 8 GitHub repositories containing Dafny verification exercises. Selection criteria: self-contained (single file), compilable with Dafny 4.5.0, and covering diverse specification patterns.

Difficulty classification criteria.

- **Simple**: No loops, no recursion, at most one `requires` and one `ensures` clause. Typically pure functions or single-clause predicates.
- **Medium**: May include recursion with obvious termination, multiple specification clauses, or quantified postconditions. No loops requiring user-written invariants.
- **Hard**: Requires user-written loop invariants, termination measures for non-trivial recursion, or lemma proofs with auxiliary assertions.

Table 12: Representative examples from each difficulty level.

Diff.	NL Description	Type
Simple	"Return true if x is even."	predicate
Simple	"Compute the absolute value of an integer."	function
Medium	"Return the minimum of a non-empty sequence."	function
Medium	"Check if a sequence is sorted in ascending order."	predicate
Hard	"Sort a sequence using insertion sort with a loop."	method
Hard	"Prove that appending then taking length equals sum of lengths."	lemma

Sample benchmark entries.

F Equivalence Verification Pipeline

For each generated program, we verify semantic equivalence against the reference implementation through a three-stage pipeline:

Stage 1: Interface matching. We verify that the generated program has the same signature as the reference: identical input parameter types ($\tau_{\text{in}} = \hat{\tau}_{\text{in}}$), identical return type ($\tau_{\text{out}} = \hat{\tau}_{\text{out}}$), and matching construct type.

Stage 2: Specification equivalence. We check that preconditions and postconditions are logically equivalent:

$$\forall x.\ P(x) \Leftrightarrow \hat{P}(x)$$
$$\forall x, y.\ Q(x, y) \Leftrightarrow \hat{Q}(x, y)$$

Stage 3: Behavioral equivalence. We check that both implementations produce the same output under the shared precondition: $\forall x.\ P(x) \Rightarrow S(x) = \hat{S}(x)$

These are encoded as Dafny verification wrappers:

Equivalence Verification Wrapper (Example)

```
// Reference
function RefImpl(x: int): int
  requires x >= 0
  ensures RefImpl(x) == x * 10
{ x * 10 }

// Generated
function GenImpl(x: int): int
  requires x >= 0
  ensures GenImpl(x) == x * 10
{ x * 10 }

// Equivalence check
lemma Equivalent(x: int)
  requires x >= 0
  ensures RefImpl(x) == GenImpl(x)
{}
```

The wrapper is automatically synthesized by: (1) parsing both programs to extract signatures and specs via regex; (2) renaming identifiers to avoid symbol collisions; (3) generating a lemma asserting output equality under shared preconditions; (4) submitting to the Dafny verifier.

G Multi-Model Comparison

To assess model dependence, we evaluate SKILLFORGE with three frontier LLMs (Table 13). GPT-5.5 achieves 91.6%, followed by Claude Opus-4.7 (89.9%) and Sonnet-4.5 (84.3%). Notably, even Sonnet-4.5 with our harness matches the strongest agentic baseline (Proof2Silicon at 85.4%) using GPT-5.5, demonstrating that *orchestration paradigm matters more than raw model capability*. The gap between models is driven primarily by GENINVARIANT success rate (52.4% vs. 64.7%), confirming that loop invariant synthesis is the most model-capability-sensitive skill.

Table 13: Verification rate (%) across different LLM backends using the full SKILLFORGE harness (N=5).

Model	VR%	ER%	AI	FPR%
Claude Sonnet-4.5	84.3	77.5	2.1	62.4
Claude Opus-4.7	89.9	84.3	1.9	69.1
GPT-5.5	**91.6**	**86.5**	**1.8**	**71.3**

H Discussion

Why skills outperform agents for formal verification. Three factors explain our advantage: (1) The Dafny verifier provides a *perfect oracle* with structured error messages, making deterministic routing both possible and optimal. (2) The error space is finite and well-categorized (four types), each with a clear repair strategy—agents must "rediscover" this routing through reasoning each time. (3) Repair is local: most failures require modifying a single program aspect. Skills enforce locality explicitly; agents may perform unnecessary global rewrites.

When do agentic methods have advantages? While SKILLFORGE outperforms agentic baselines on formal verification, we note that the structured harness paradigm is specifically suited to domains with *finite error taxonomies* and *perfect oracles*. For tasks with ambiguous success criteria (e.g., general software engineering, creative coding), agentic methods retain clear advantages due to their flexibility in handling novel situations. The success of SKILLFORGE should be interpreted as evidence that formal verification is a domain where structure dominates flexibility, not as a general indictment of agentic approaches.

When does SKILLFORGE fail? Analysis of the 15 failure cases (8.4%) reveals three primary patterns: (i) **Complex inductive proofs** (7 cases): Lemmas requiring non-trivial induction over recursive data structures. The model can state the lemma correctly but cannot synthesize the inductive proof

body. Notably, Proof2Silicon and MCTS-Repair also fail on all 7 of these cases, suggesting a fundamental LLM limitation rather than an architectural one. (ii) **Quantifier-heavy specifications** (5 cases): Programs involving nested quantifiers ($\forall x.\ \exists y.\ \ldots$) where the model struggles to maintain logical consistency. ReAct-Verify solves 1 of these 5 through its more flexible reasoning, suggesting that some problems benefit from open-ended exploration. (iii) **Misdiagnosis cascades** (3 cases): DIAGNOSEERROR misclassifies the failure, leading to inappropriate repair. These cases highlight the 11.8% failure rate of our diagnosis skill and motivate future work on ensemble-based or confidence-calibrated diagnosis.

Generalizability beyond Dafny. While our skill library is Dafny-specific, the harness architecture is language-agnostic. The core principles—decompose into skills with decidable success criteria, route based on structured verifier feedback, iterate until convergence—apply to any formally verified language. Adaptation to Lean 4, for example, would require: (1) new error taxonomy (tactic failures, type mismatches, timeout); (2) new skills (tactic suggestion, term synthesis, unfold/simp guidance); (3) modified routing rules. The harness structure, iteration logic, and termination guarantees remain unchanged. We conjecture that the advantage over agentic methods persists in any domain where the error oracle provides structured, categorizable feedback.

Relation to test-time compute scaling. SKILLFORGE can be viewed through the lens of test-time compute scaling: rather than investing compute in training (fine-tuning), we invest it at inference time through structured iteration. Our convergence analysis shows diminishing returns beyond 2 iterations for most examples, suggesting an efficient allocation of test-time budget. Compared to MCTS-Repair and Proof2Silicon (which also invest heavily in test-time compute), SKILLFORGE achieves superior results with dramatically less compute, demonstrating that *structured* test-time investment outperforms *unstructured* scaling for formal verification.

I Per-Difficulty Skill Utilization

Table 14 reveals clear patterns: GENINVARIANT is invoked almost exclusively on Hard tasks (15 of 17 total invocations), confirming that loop invariant generation is the defining challenge of Hard problems. STRENGTHENSPEC is most common on Medium tasks, where functions have non-trivial but expressible specifications. Simple tasks rarely enter the repair loop at all (only 3 out of 62 require repair).

Table 14: Skill invocation frequency per difficulty level (repair phase only, excluding initial InferSpec + SynthBody).

Skill	Simple	Medium	Hard
DIAGNOSEERROR	3	18	30
REPAIRCODE	2	9	11
STRENGTHENSPEC	1	7	4
GENINVARIANT	0	2	15