AlloyEval: Replication and Extension of "On the Effectiveness of LLMs in Writing Alloy Formulas"

Shmulik Cohen
*Advanced Topics in Specification Engineering, Spring 2026**

Abstract

We replicate and extend the study by Hong et al. [1], which evaluated LLMs on three Alloy specification tasks. The original study used OpenAI o3-mini and DeepSeek R1; we evaluate Google Gemini 2.5 Flash Lite and Gemini 2.5 Pro via Vertex AI. We extend with three new task variants (guided, agent, reflect) and expand the benchmark from 11 to 41 properties across four additional domains. Gemini 2.5 Pro achieves results comparable to the original study's reasoning models, with 100% on sketch completion and 95–100% on formula equivalence. The guided and agent variants improve the weaker Flash Lite model by up to 24 percentage points, while Pro is already near ceiling. Error analysis reveals a qualitative difference: Flash Lite fails on Alloy syntax, Pro fails on logic.

1 Introduction

Alloy [2] is a lightweight formal specification language based on first-order relational logic. Writing correct Alloy formulas requires precise quantifiers, relational operators, and set-theoretic constructs — making it a challenging target for LLM-based code generation.

Hong et al. [1] conducted the first systematic evaluation of LLMs on Alloy formula synthesis. They tested two reasoning-oriented models (OpenAI o3-mini and DeepSeek R1) on 11 properties using three tasks: generating Alloy formulas from natural language, producing equivalent formulas, and completing formula sketches. With 20 trials per property, DeepSeek R1 achieved 10+ correct formulas on 10/11 properties for the NL-to-Alloy task.

We replicate their protocol with Google Gemini models and extend it along three axes: (1) additional task variants that test the effect of domain documentation, compiler feedback, and self-critique; (2) an expanded benchmark with 30 additional properties across four new domains; and (3) structured output constraints that improve evaluation reliability.

*Project repository: `https://github.com/shmulc8/AlloyEval`

2 Background

2.1 Alloy

Alloy models consist of *signatures* (types with fields) and *predicates* expressed in first-order relational logic. The Alloy Analyzer performs bounded model checking: given a scope (maximum number of atoms per signature), it exhaustively searches for counterexamples to assertions. If no counterexample exists within the scope, the assertion is considered valid.

For example, the property that a binary relation is reflexive is expressed as:

```
sig S { r: set S }
pred Reflexive { all s: S | s->s in r }
```

2.2 Original Study

Hong et al. defined three tasks:

- **nl2alloy**: Given a natural language description and Alloy signatures, generate the formula body of a predicate.
- **alloy2alloy**: Given a canonical Alloy formula, generate a logically equivalent formula.
- **sketch2alloy**: Given a partial formula with holes (`_`), fill in the missing operators or subexpressions. One error-feedback retry is allowed on syntax or type errors.

They evaluated on 11 properties: 3 graph properties (DAG, Cycle, Circular) using `sig Node { link: set Node }`, and 8 relation properties (Connex, Reflexive, Symmetric, Transitive, Antisymmetric, Irreflexive, Functional, Function) using `sig S { r: set S }`.

3 Methodology

3.1 Models

We evaluate two models from Google's Gemini family, accessed via the Vertex AI REST API:

- **Gemini 2.5 Flash Lite** — a lightweight, low-latency model optimized for throughput and cost. Represents the class of cheap, fast models.
- **Gemini 2.5 Pro** — a high-capability model with extended reasoning. Represents models closer in capability to the o3-mini and DeepSeek R1 used in the original study.

3.2 Task Variants

We replicate the three base tasks from the original study and introduce three additional modes, yielding 12 task variants (3 base tasks x 4 modes).

Base tasks. Each base task sends a single prompt to the LLM with a system message establishing the Alloy expert role. The prompt includes the Alloy signatures and asks the model to output only the formula body. For example, the nl2alloy prompt for the Reflexive property is:

> *Implement the following Alloy predicate Reflexive as defined in the comments:* `sig S { r: set S }` / `pred Reflexive { - Every element in S is related to itself }` *Output only the complete formula that goes inside the predicate body. Include all quantifiers and operators.*

The alloy2alloy prompt instead shows the canonical formula and asks for an equivalent one. The sketch2alloy prompt shows the formula with holes and a hint.

Guided mode. Identical to the base tasks, but a concise Alloy language reference is appended to the system prompt. This reference covers relational operators (join, transpose, transitive closure), quantifiers (`all`, `some`, `no`, `one`, `lone`), set operations, special relations (`iden`, `univ`, `none`), and common idioms. This tests whether providing domain-specific documentation improves generation quality without changing the task itself.

Agent mode. After the initial generation, the formula is checked with the Alloy analyzer. If it fails, the error message (e.g., "Syntax Error," "Type Error," or "Counterexample") is appended to the conversation as a user message, and the model is asked to fix it. This is capped at one retry (2 LLM rounds total). The conversation looks like:

1. System: Alloy expert prompt
2. User: Task prompt (e.g., nl2alloy)
3. Assistant: Generated formula
4. *[Alloy check fails]*
5. User: "The Alloy checker reported: {error}. Please fix it."
6. Assistant: Corrected formula

Reflect mode. Same two-round structure as agent, but instead of compiler feedback, the model is asked to self-critique: "Please review your Alloy formula carefully. Check that it correctly captures the property described, paying attention to quantifiers, relational operators, and edge cases." This isolates the benefit of self-review from external tool use.

3.3 Dataset

Base (11 properties). We use the same 11 properties from the original study, preserving their signatures, canonical formulas, sketches, and natural language descriptions.

Extended (30 properties). We expand the benchmark with 30 additional properties drawn from Alloy4Fun [4], an online educational platform for learning Alloy. Alloy4Fun presents users with a predicate description and signature structure, then lets them write the formula body with automated feedback from the Alloy analyzer — a workflow closely analogous to our nl2alloy and agent tasks, but with human students instead of LLMs. The exercises span four domains:

- **Graph** (5): undirected, oriented, strongly connected, transitive, weakly connected
- **Social network** (7): invariants over users, followers, photos, and influencers
- **Production line** (9): factory workflow with workstations, human/robot workers, and components
- **Trash/filesystem** (9): file system operations with trash, protected files, and links

The extended properties have more complex signature structures (up to 6 signatures with inheritance) compared to the base set's single-signature models. They do not include sketches, so sketch-based tasks run only on the 11 base properties. Using Alloy4Fun exercises as our extended benchmark provides a natural comparison point: these are problems designed for human learners, and we can now ask how LLMs perform on the same challenges.

3.4 Experimental Setup

Sampling. Each task variant runs 3 independent trials per property with sampling temperature 1.0, following the original study's emphasis on solution diversity. The original study used 20 trials; we use fewer to manage cost across a larger task matrix.

Structured output. All LLM responses are constrained to a JSON schema with a single field, `formula`. We enforce this using Vertex AI's structured output mode. This eliminates the need for post-hoc parsing and prevents common failure modes we observed during development: models returning only the fill-in token for sketch tasks (e.g., `implies` instead of the full formula), or returning formula fragments without quantifiers for alloy2alloy tasks. This single change improved Flash Lite's pass rate from 28% to 59% on base tasks.

System prompt. All tasks use the same base system prompt:

> *You are an expert in formal methods and the Alloy specification language. When asked, output ONLY the Alloy formula (the predicate body), nothing else. Do not include the pred declaration, curly braces, or any explanation.*

For guided tasks, the Alloy language reference is appended to this prompt.

Infrastructure. 32 parallel threads for I/O-bound API calls. Results checkpointed every 10 trials. Alloy analyzer invoked with scope 3 and a 30-second timeout per check.

3.5 Validation

For each generated formula, we construct a complete `.als` file that checks logical equivalence with the canonical solution:

```
sig S { r: set S }

pred Reflexive {
    all s: S | s in s.r    /* LLM-generated formula */
}

check Reflexive {
    Reflexive iff (all s: S | s->s in r)    /* canonical formula */
} for 3
```

The Alloy analyzer returns:

- **UNSAT** (no counterexample in scope 3) = **pass** — the formulas are equivalent within the bounded scope.
- **SAT** (counterexample found) = **fail** (Counterexample) — the formulas are not logically equivalent.
- **Parse/type error** = **fail** (Syntax Error or Type Error) — the generated formula is not valid Alloy.

3.6 Metrics

- **Pass rate**: fraction of trials that produce a correct (equivalent) formula.
- **Error breakdown**: distribution of failure types (Syntax Error, Type Error, Counterexample).

4 Results

We ran 2,424 trials across 2 models, 12 task variants, and 41 properties (3 solutions each). Overall, 1,833 trials passed (75.6%).

4.1 Overall Performance

Gemini 2.5 Pro achieves near-perfect results on alloy2alloy (94.8–100%) and sketch2alloy (100% across all variants), and strong results on nl2alloy (79.3–85.2%). This is comparable to the original study's reasoning models: they reported 10/11 sketch completions and high alloy2alloy rates for both o3-mini and DeepSeek R1.

Task	Variant	Flash Lite	Pro
nl2alloy	Base	31.9%	80.7%
	Guided	52.6%	79.3%
	Agent	45.9%	83.7%
	Reflect	30.4%	85.2%
alloy2alloy	Base	72.6%	94.8%
	Guided	74.8%	99.3%
	Agent	78.5%	100.0%
	Reflect	71.9%	95.6%
sketch2alloy	Base	72.7%	100.0%
	Guided	90.9%	100.0%
	Agent	97.0%	100.0%
	Reflect	78.8%	100.0%

Table 1: Pass rates across all task variants.

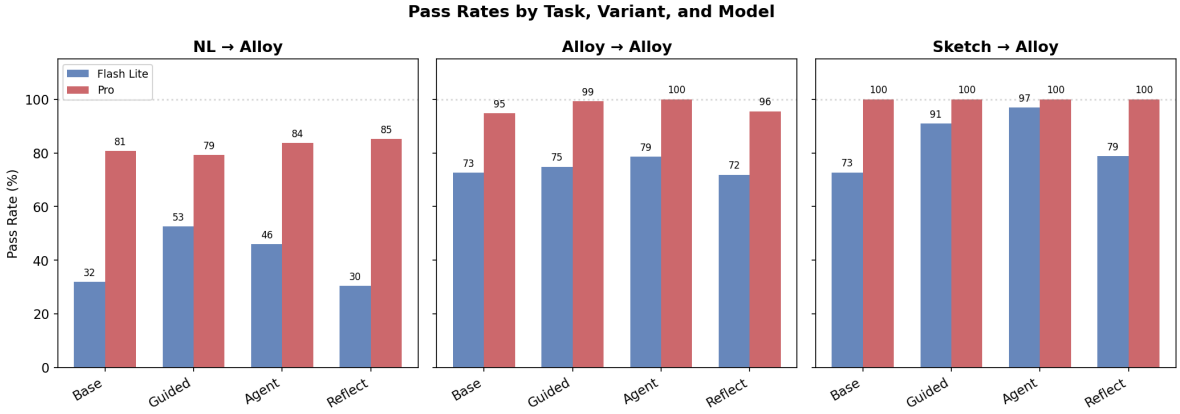


Figure 1: Pass rates by task, variant, and model.

Flash Lite shows a significant capability gap, particularly on nl2alloy (30.4–52.6%), where generating formulas from scratch requires deeper understanding of Alloy semantics.

4.2 Effect of Task Variants

The benefit of the extensions depends heavily on model capability:

Flash Lite benefits substantially from guided prompting (+20.7pp on nl2alloy, +18.2pp on sketch2alloy). It also improves with the agent variant (+14.0pp on nl2alloy, +24.3pp on sketch2alloy). Reflect provides no improvement and sometimes slightly hurts performance. This indicates that weaker models need either domain documentation or external tool feedback — self-critique alone is insufficient when the model's Alloy knowledge is limited.

Pro is already near ceiling, so variants produce small effects (+3–5pp from agent). Guided prompting slightly *decreases* nl2alloy performance for Pro (-1.4pp), possibly because the added context increases prompt length without providing information the model already has.

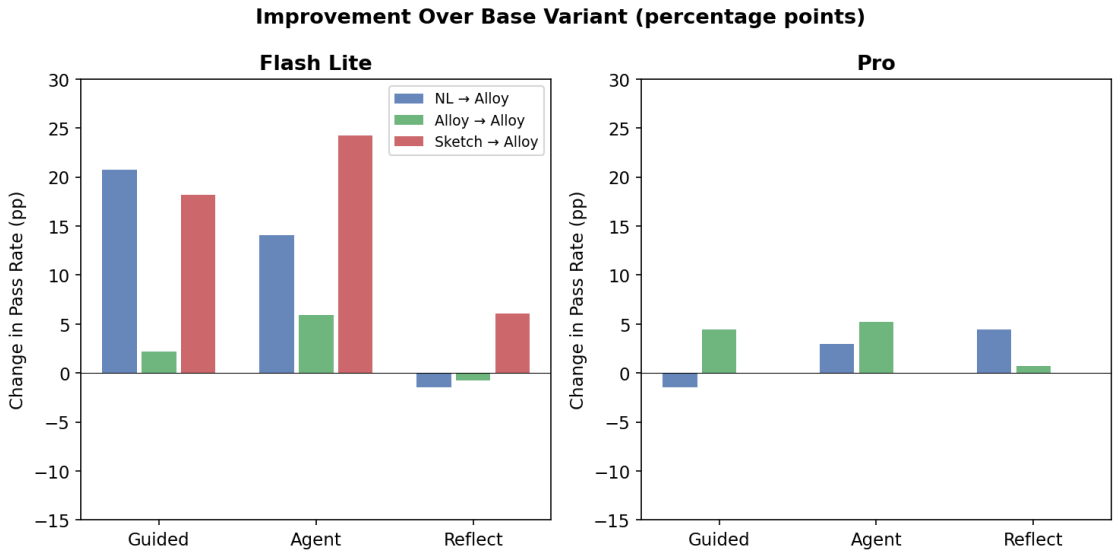


Figure 2: Improvement over base variant.

Extended properties are generally harder than the base set, with Flash Lite dropping from 60% (base) to 31–63% across extended domains. Pro remains consistent at 78–97%. The production line domain is hardest for both models, likely due to its complex signature structure (6 sigs with inheritance).

4.3 Error Analysis

The error profiles differ qualitatively between models:

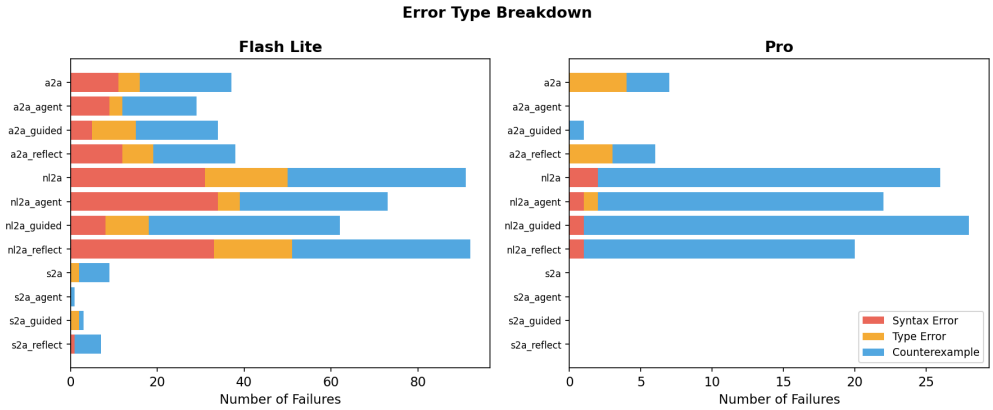


Figure 3: Error type breakdown.

- **Flash Lite**: 144 syntax errors, 81 type errors, 251 counterexamples. Nearly half of all failures (47%) are syntactic or type errors — the model frequently uses wrong quantifier forms (`all x in S` instead of `all x: S`), incorrect operator precedence, or nonexistent constructs.

- **Pro**: 5 syntax errors, 8 type errors, 97 counterexamples. Nearly all failures are logically incorrect but syntactically valid formulas. Pro has mastered Alloy syntax but occasionally gets the semantics wrong.

This distinction explains why the guided and agent variants help Flash Lite so much: the Alloy reference helps with syntax, and compiler feedback catches syntactic mistakes. For Pro, where failures are logical rather than syntactic, these interventions have less to offer.

5 Discussion

5.1 Structured Output as a Reliability Technique

A significant engineering finding was the impact of structured output on evaluation reliability. During development, we used free-form text generation with post-hoc parsing. This led to two classes of spurious failures: (1) for sketch2alloy, the model returned only the fill-in token (e.g., `implies`) instead of the complete formula, causing 60% of sketch failures; and (2) for alloy2alloy, the model returned formula fragments without quantifiers (e.g., `s in s.r` instead of `all s: S | s in s.r`), causing 36% of failures. We also discovered a formula-cleaning bug where an aggressive `rstrip("}")` was stripping set literal braces (e.g., `{z}`) from valid Alloy expressions.

Switching to Vertex AI's structured JSON output mode eliminated both failure classes and raised Flash Lite's pass rate from 28% to 59%. This suggests that for LLM-based formal methods evaluation, structured output constraints and careful prompt engineering are at least as important as model capability.

5.2 Compiler Feedback vs Self-Critique

The agent and reflect variants provide a controlled comparison of two iterative refinement strategies. The agent variant (one round of Alloy compiler feedback) was consistently the strongest performer: 100% on alloy2alloy for Pro, 97% on sketch2alloy for Flash Lite. The reflect variant (one round of self-critique without compiler access) provided little benefit and sometimes hurt performance.

The gap between them quantifies the value of tool integration in LLM-based formal methods. Compiler feedback provides precise, actionable error information, whereas self-critique relies on the model's own (potentially incomplete) understanding of Alloy. This finding supports the broader trend toward tool-augmented LLM workflows.

6 Threats to Validity

Construct validity. Equivalence checking at scope 3 may miss semantic differences that only manifest at larger scopes. This limitation is shared with the original study.

Internal validity. LLM outputs are non-deterministic. We run 3 trials per configuration (fewer than the paper's 20), which limits statistical power. Structured JSON output eliminates formula extraction errors that affected early experiments. The agent and reflect variants are limited to a single retry round; increasing the feedback budget could yield higher pass rates.

External validity. We evaluate only Google Gemini models; a direct comparison with o3-mini and DeepSeek R1 on our extended tasks would strengthen the findings. The extended dataset, while broader, still consists of relatively simple Alloy specifications.

7 Related Work

LLM code generation. Chen et al. [3] introduced the HumanEval benchmark for evaluating LLM code generation. Our work applies a similar evaluation framework to a formal specification language rather than a general-purpose programming language.

LLMs for formal methods. Hong et al. [1], which we replicate, is the first study targeting Alloy specifically. Broader work on LLMs for formal specification and verification — including proof generation for theorem provers and temporal logic synthesis from natural language — has shown that models struggle with the precise logical reasoning these domains require.

Tool-augmented generation. Our agent variant, which feeds Alloy compiler errors back to the model, relates to compiler-in-the-loop approaches for code generation, where iterative refinement with tool feedback outperforms single-shot generation. The contrast between our agent and reflect variants provides a controlled comparison of external feedback vs self-critique.

8 Conclusion

We replicated and extended the study by Hong et al. on LLM-based Alloy formula synthesis. Using Google Gemini models (2.5 Flash Lite and 2.5 Pro), we evaluated 12 task variants across 41 properties spanning five domains. Gemini 2.5 Pro matched or exceeded the original study's results on alloy2alloy and sketch2alloy. The guided and agent variants provide substantial improvements for weaker models (up to +24pp), while compiler feedback consistently outperforms self-critique. We also found that structured output constraints and prompt engineering are critical for reliable evaluation, eliminating spurious failures that inflated error rates by over 35% in early experiments. Our code and results are publicly available.

References

1. Hong, F., Jiang, M., Fu, C., & Khurshid, S. (2025). *On the Effectiveness of LLMs in Writing Alloy Formulas. arXiv:2502.15441.*

2. Jackson, D. (2012). *Software Abstractions: Logic, Language, and Analysis.* MIT Press, revised edition.

3. Chen, M., Tworek, J., Jun, H., Yuan, Q., et al. (2021). *Evaluating Large Language Models Trained on Code. arXiv:2107.03374.*

4. Macedo, N., Cunha, A., Pereira, J., Carvalho, R., Silva, R., Paiva, A.C.R., Ramalho, M.S., & Silva, D. (2019). *Sharing and Learning Alloy on the Web. arXiv:1907.02275.*